

网络基础

协议的概念

什么是协议

从应用的角度出发，协议可理解为“规则”，是数据传输和数据的解释的规则。

假设，A、B 双方欲传输文件。规定：

第一次，传输文件名，接收方接收到文件名，应答 OK 给传输方；

第二次，发送文件的尺寸，接收方接收到该数据再次应答一个 OK；

第三次，传输文件内容。同样，接收方接收数据完成后应答 OK 表示文件内容接收成功。

由此，无论 A、B 之间传递何种文件，都是通过三次数据传输来完成。A、B 之间形成了一个最简单的数据传输规则。双方都按此规则发送、接收数据。A、B 之间达成的这个相互遵守的规则即为协议。

这种仅在 A、B 之间被遵守的协议称之为**原始协议**。当此协议被更多的人采用，不断的增加、改进、维护、完善。最终形成一个稳定的、完整的文件传输协议，被广泛应用于各种文件传输过程中。该协议就成为一个**标准协议**。最早的 ftp 协议就是由此衍生而来。

TCP 协议注重数据的传输。http 协议着重于数据的解释。

典型协议

传输层 常见协议有 TCP/UDP 协议。

应用层 常见的协议有 HTTP 协议，FTP 协议。

网络层 常见协议有 IP 协议、ICMP 协议、IGMP 协议。

网络接口层 常见协议有 ARP 协议、RARP 协议。

TCP 传输控制协议 (Transmission Control Protocol) 是一种面向连接的、可靠的、基于字节流的传输层通信协议。

UDP 用户数据报协议 (User Datagram Protocol) 是 OSI 参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

HTTP 超文本传输协议 (Hyper Text Transfer Protocol) 是互联网上应用最为广泛的一种网络协议。

FTP 文件传输协议 (File Transfer Protocol)

IP 协议是因特网互联协议 (Internet Protocol)

ICMP 协议是 Internet 控制报文协议 (Internet Control Message Protocol) 它是 TCP/IP 协议族的一个子协议，用于在 IP 主机、路由器之间传递控制消息。

IGMP 协议是 Internet 组管理协议 (Internet Group Management Protocol)，是因特网协议家族中的一个组播协议。该协议运行在主机和组播路由器之间。

ARP 协议是正向地址解析协议 (Address Resolution Protocol)，通过已知的 IP，寻找对应主机的 MAC 地址。

RARP 是反向地址转换协议，通过 MAC 地址确定 IP 地址。

网络应用程序设计模式

C/S 模式

传统的网络应用设计模式，客户机(client)/服务器(server)模式。需要在通讯两端各自部署客户机和服务器来完成数据通信。

B/S 模式

浏览器()/服务器(server)模式。只需在一端部署服务器，而另外一端使用每台 PC 都默认配置的浏览器即可完

成数据的传输。

优缺点

对于 C/S 模式来说，其优点明显。客户端位于目标主机上可以保证性能，将数据缓存至客户端本地，从而**提高数据传输效率**。且，一般来说客户端和服务程序由一个开发团队创作，所以他们之间**所采用的协议相对灵活**。可以在标准协议的基础上根据需求裁剪及定制。例如，腾讯公司所采用的通信协议，即为 ftp 协议的修改剪裁版。

因此，传统的网络应用程序及较大型的网络应用程序都首选 C/S 模式进行开发。如，知名的网络游戏魔兽世界。3D 画面，数据量庞大，使用 C/S 模式可以提前在本地进行大量数据的缓存处理，从而提高观感。

C/S 模式的缺点也较突出。由于客户端和服务端都需要有一个开发团队来完成开发。**工作量**将成倍提升，开发周期较长。另外，从用户角度出发，需要将客户端安插至用户主机上，对用户主机的**安全性构成威胁**。这也是很多用户不愿使用 C/S 模式应用程序的重要原因。

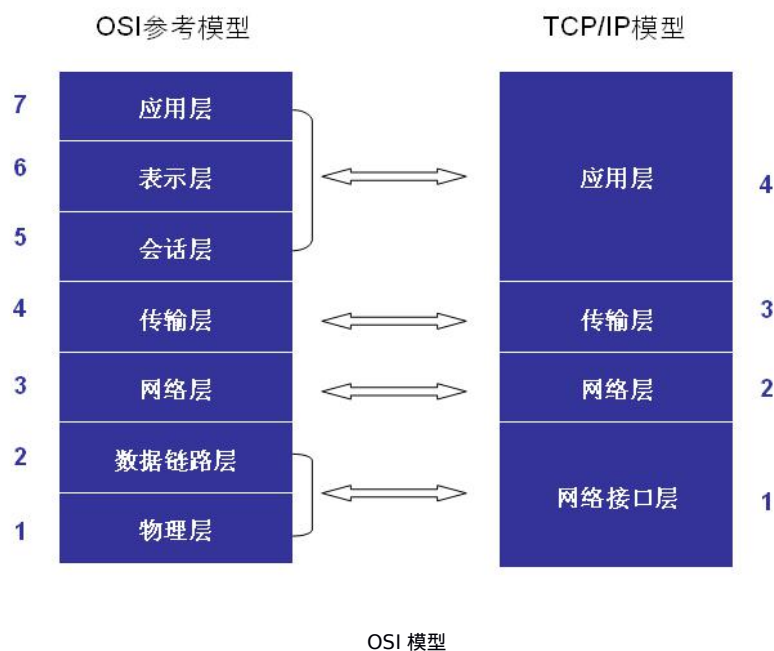
B/S 模式相比 C/S 模式而言，由于它没有独立的客户端，使用标准浏览器作为客户端，其**工作开发量较小**。只需开发服务器端即可。另外由于其采用浏览器显示数据，因此移植性非常好，**不受平台限制**。如早期的偷菜游戏，在各个平台上都可以完美运行。

B/S 模式的缺点也较明显。由于使用第三方浏览器，因此**网络应用支持受限**。另外，没有客户端放到对方主机上，**缓存数据不尽如人意**，从而传输数据量受到限制。应用的观感大打折扣。第三，必须与浏览器一样，采用标准 http 协议进行通信，**协议选择不灵活**。

因此在开发过程中，模式的选择由上述各自的特点决定。根据实际需求选择应用程序设计模式。

分层模型

OSI 七层模型



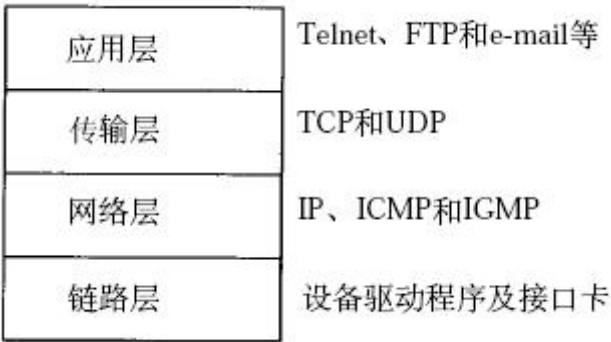
1. **物理层**：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由 1、0 转化为电流强弱来进行传输，到达目的地后再转化为 1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫做比特。
2. **数据链路层**：定义了如何让格式化数据以帧为单位进行传输，以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。如：串口通信中使用到的 115200、8、N、1
3. **网络层**：在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。Internet 的发展使得从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。
4. **传输层**：定义了一些传输数据的协议和端口号（WWW 端口 80 等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与 TCP 特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如 QQ 聊天数据就是通过这种方式传输的）。主要是将从下层接收的数据进行分段和传输，到达目的地后再进行重组。常常把这一层数据叫做段。
5. **会话层**：通过传输层(端口号：传输端口与接收端口)建立数据传输的通路。主要在你的系统之间发起会话或者

接受会话请求（设备之间需要互相认识可以是 IP 也可以是 MAC 或者是主机名）。

- 6. **表示层**：可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。例如，PC 程序与另一台计算机进行通信，其中一台计算机使用扩展二一十进制交换码(EBCDIC)，而另一台则使用美国信息交换标准码（ASCII）来表示相同的字符。如有必要，表示层会通过使用一种通格式来实现多种数据格式之间的转换。
- 7. **应用层**：是最靠近用户的 OSI 层。这一层为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。

TCP/IP 四层模型

TCP/IP 网络协议栈分为应用层（Application）、传输层（Transport）、网络层（Network）和链路层（Link）四层。如下图所示：

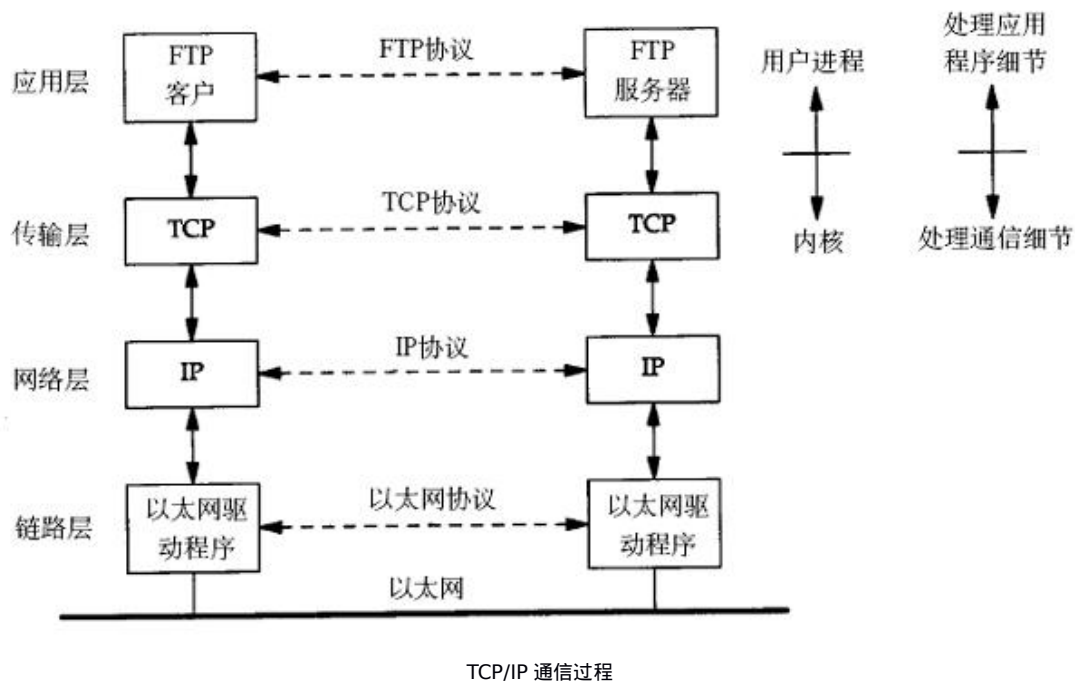


TCP/IP 模型

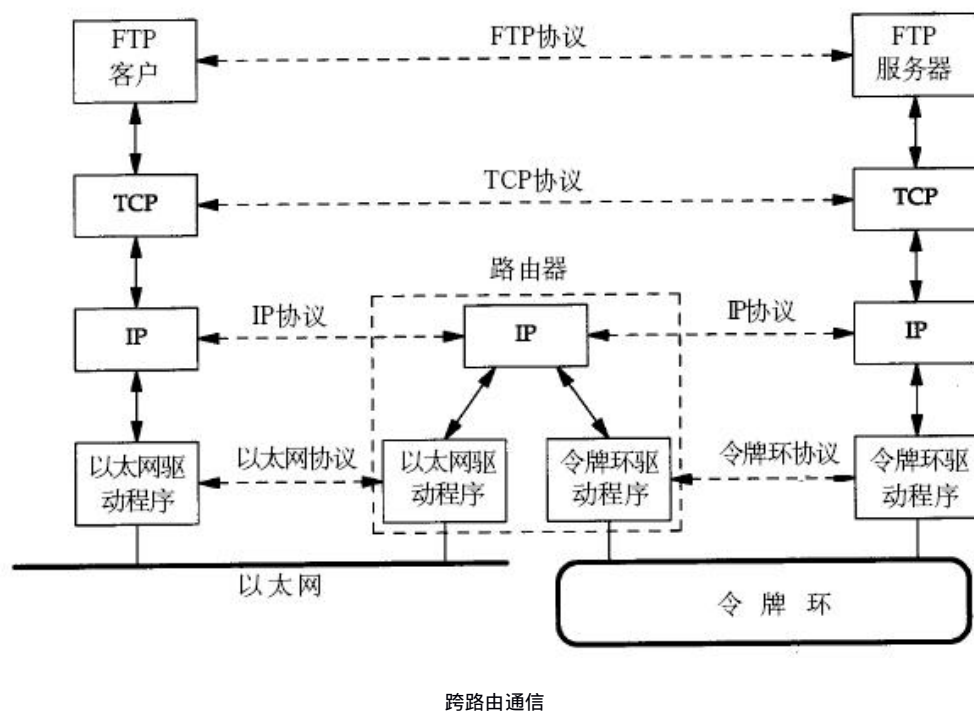
一般在应用开发过程中，讨论最多的是 TCP/IP 模型。

通信过程

两台计算机通过 TCP/IP 协议通讯的过程如下所示：



上图对应两台计算机在同一网段中的情况，如果两台计算机在不同的网段中，那么数据从一台计算机到另一台计算机传输过程中要经过一个或多个路由器，如下图所示：



链路层有以太网、令牌环网等标准，链路层负责网卡设备的驱动、帧同步（即从网线上检测到什么信号算作新帧的开始）、冲突检测（如果检测到冲突就自动重发）、数据差错校验等工作。交换机是工作在链路层的网络设备，可以在不同的链路层网络之间转发数据帧（比如十兆以太网和百兆以太网之间、以太网和令牌环网之间），由于不同链路层的帧格式不同，交换机要将进来的数据包拆掉链路层首部重新封装之后再转发。

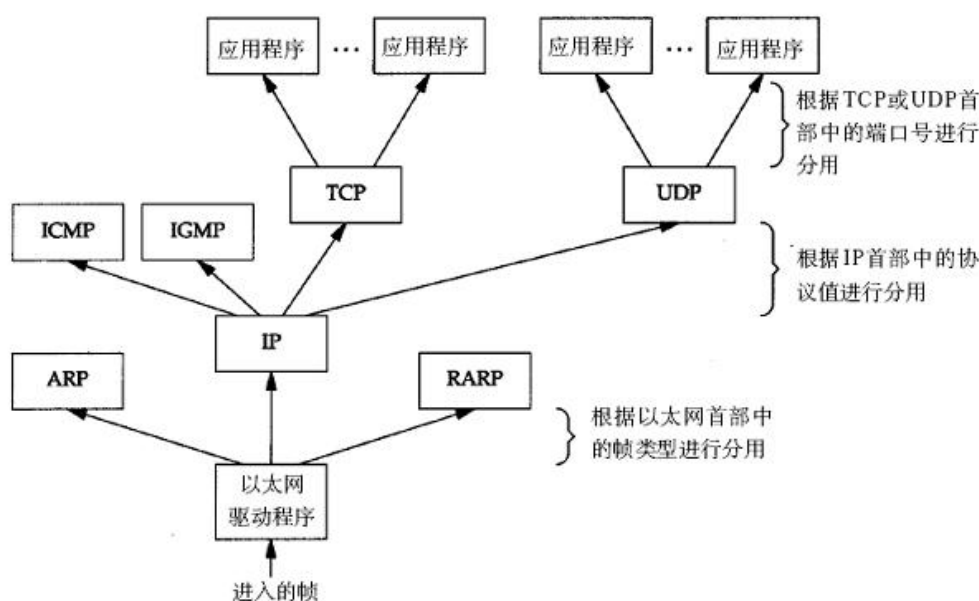
网络层的 IP 协议是构成 Internet 的基础。Internet 上的主机通过 IP 地址来标识，Internet 上有大量路由器负责根据 IP 地址选择合适的路径转发数据包，数据包从 Internet 上的源主机到目的主机往往要经过十多个路由器。路由器是工作在第三层的网络设备，同时兼有交换机的功能，可以在不同的链路层接口之间转发数据包，因此路由器需要将进来的数据包拆掉网络层和链路层两层首部并重新封装。IP 协议不保证传输的可靠性，数据包在传输过程中可能丢失，可靠性可以在上层协议或应用程序中提供支持。

网络层负责点到点 (point-to-point) 的传输 (这里的“点”指主机或路由器)，而传输层负责端到端 (end-to-end) 的传输 (这里的“端”指源主机和目的主机)。传输层可选择 TCP 或 UDP 协议。

TCP 是一种面向连接的、可靠的协议，有点像打电话，双方拿起电话互通身份之后就建立了连接，然后说话就行了，这边说的话那边保证听得到，并且是按说话的顺序听到的，说完话挂机断开连接。也就是说 TCP 传输的双方需要首先建立连接，之后由 TCP 协议保证数据收发的可靠性，丢失的数据包自动重发，上层应用程序收到的总是可靠的数据流，通讯之后关闭连接。

UDP 是无连接的传输协议，不保证可靠性，有点像寄信，信写好放到邮筒里，既不能保证信件在邮递过程中不会丢失，也不能保证信件寄送顺序。使用 UDP 协议的应用程序需要自己完成丢包重发、消息排序等工作。

目的主机收到数据包后，如何经过各层协议栈最后到达应用程序呢？其过程如下图所示：



以太网驱动程序首先根据以太网首部中的“上层协议”字段确定该数据帧的有效载荷 (payload，指除去协议首

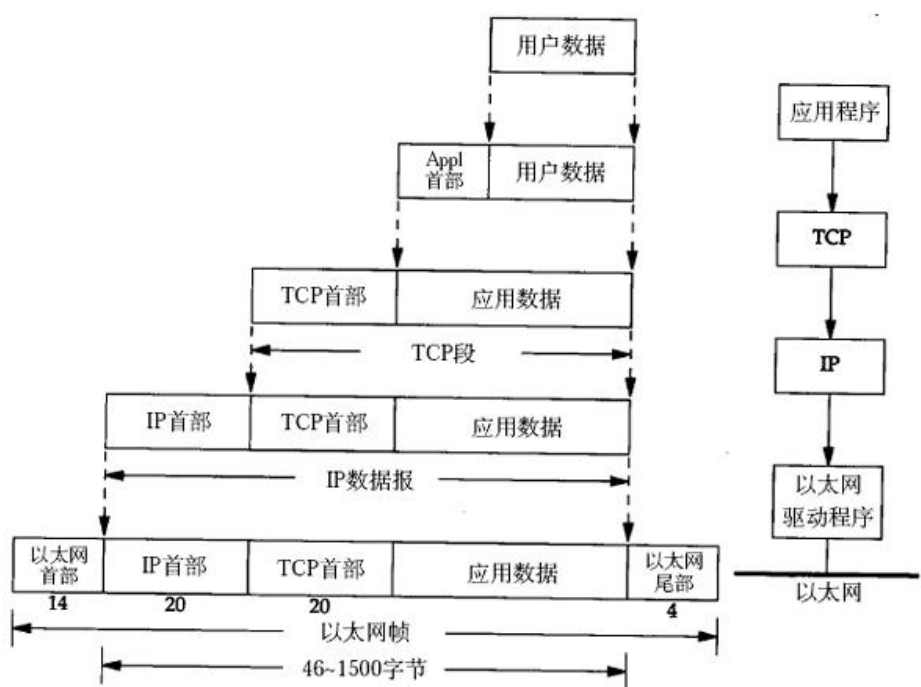
部之外实际传输的数据)是IP、ARP还是RARP协议的数据报,然后交给相应的协议处理。假如是IP数据报,IP协议再根据IP首部中的“上层协议”字段确定该数据报的有效载荷是TCP、UDP、ICMP还是IGMP,然后交给相应的协议处理。假如是TCP段或UDP段,TCP或UDP协议再根据TCP首部或UDP首部的“端口号”字段确定应该将应用层数据交给哪个用户进程。IP地址是标识网络中不同主机的地址,而端口号就是同一台主机上标识不同进程的地址,IP地址和端口号合起来标识网络中唯一的进程。

虽然IP、ARP和RARP数据报都需要以太网驱动程序来封装成帧,但是从功能上划分,ARP和RARP属于链路层,IP属于网络层。虽然ICMP、IGMP、TCP、UDP的数据都需要IP协议来封装成数据报,但是从功能上划分,ICMP、IGMP与IP同属于网络层,TCP和UDP属于传输层。

协议格式

数据包封装

传输层及其以下的机制由内核提供,应用层由用户进程提供(后面将介绍如何使用socket API编写应用程序),应用程序对通讯数据的含义进行解释,而传输层及其以下处理通讯的细节,将数据从一台计算机通过一定的路径发送到另一台计算机。应用层数据通过协议栈发到网络上时,每层协议都要加上一个数据首部(header),称为封装(Encapsulation),如下图所示:

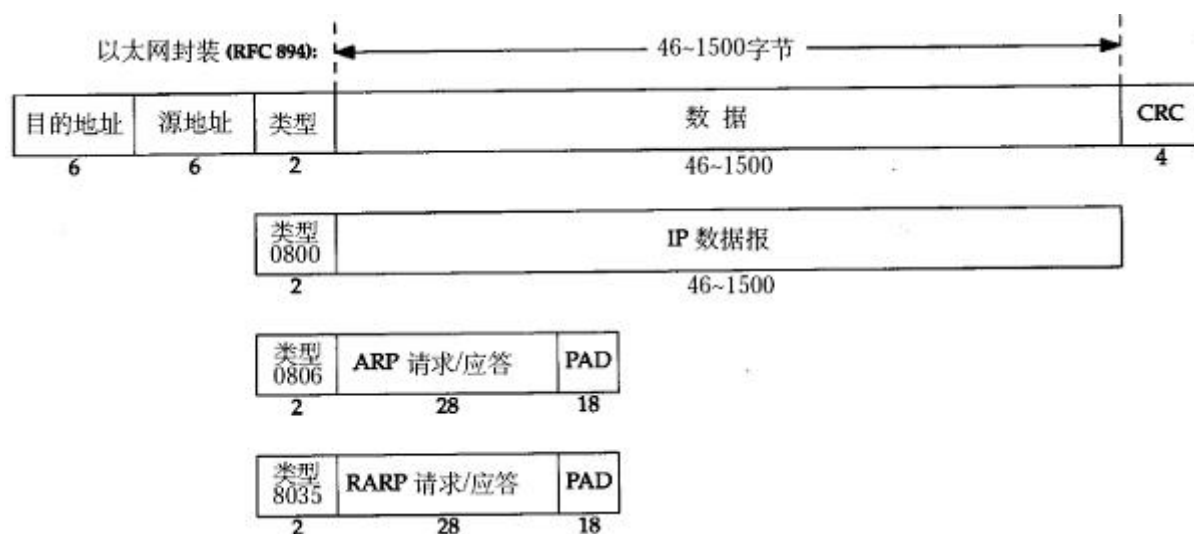


TCP/IP 数据包封装

不同的协议层对数据包有不同的称谓，在传输层叫做段（segment），在网络层叫做数据报（datagram），在链路层叫做帧（frame）。数据封装成帧后发到传输介质上，到达目的主机后每层协议再剥掉相应的首部，最后将应用层数据交给应用程序处理。

以太网帧格式

以太网的帧格式如下所示：



以太网帧格式

其中的源地址和目的地址是指网卡的硬件地址（也叫 MAC 地址），长度是 48 位，是在网卡出厂时固化的。可在 shell 中使用 ifconfig 命令查看，“HWaddr 00:15:F2:14:9E:3F”部分就是硬件地址。协议字段有三种值，分别对应 IP、ARP、RARP。帧尾是 CRC 校验码。

以太网帧中的数据长度规定最小 46 字节，最大 1500 字节，ARP 和 RARP 数据包的长度不够 46 字节，要在后面补填充位。**最大值 1500 称为以太网的最大传输单元（MTU）**，不同的网络类型有不同的 MTU，如果一个数据包从以太网路由到拨号链路上，数据包长度大于拨号链路的 MTU，则需要对数据包进行分片（fragmentation）。ifconfig 命令输出中也有“MTU:1500”。注意，MTU 这个概念指数据帧中有效载荷的最大长度，不包括帧头长度。

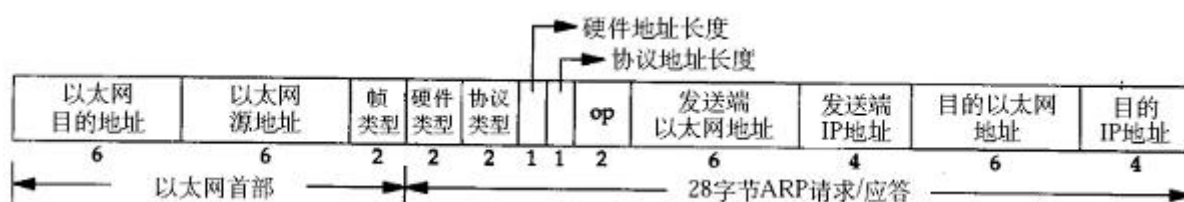
ARP 数据报格式

在网络通讯时，源主机的应用程序知道目的主机的 IP 地址和端口号，却不知道目的主机的硬件地址，而数据包首先是被网卡接收到再去处理上层协议的，如果接收到的数据包的硬件地址与本机不符，则直接丢弃。因此在通

讯前必须获得目的主机的硬件地址。ARP 协议就起到这个作用。源主机发出 ARP 请求 ,询问“IP 地址是 192.168.0.1 的主机的硬件地址是多少”,并将这个请求广播到本地网段（以太网帧首部的硬件地址填 FF:FF:FF:FF:FF:FF 表示广播），目的主机接收到广播的 ARP 请求，发现其中的 IP 地址与本机相符，则发送一个 ARP 应答数据包给源主机，将自己的硬件地址填写在应答包中。

每台主机都维护一个 ARP 缓存表，可以用 arp -a 命令查看。缓存表中的表项有过期时间（一般为 20 分钟），如果 20 分钟内没有再次使用某个表项，则该表项失效，下次还要发 ARP 请求来获得目的主机的硬件地址。想一想，为什么表项要有过期时间而不是一直有效？

ARP 数据报的格式如下所示：



ARP 数据报格式

源 MAC 地址、目的 MAC 地址在以太网首部和 ARP 请求中各出现一次，对于链路层为以太网的情况是多余的，但如果链路层是其它类型的网络则有可能是必要的。硬件类型指链路层网络类型，1 为以太网，协议类型指要转换的地址类型，0x0800 为 IP 地址，后面两个地址长度对于以太网地址和 IP 地址分别为 6 和 4（字节），op 字段为 1 表示 ARP 请求，op 字段为 2 表示 ARP 应答。

看一个具体的例子。

请求帧如下（为了清晰在每行的前面加了字节计数，每行 16 个字节）：

以太网首部（14 字节）

0000: ff ff ff ff ff ff 00 05 5d 61 58 a8 08 06

ARP 帧（28 字节）

0000: 00 01

0010: 08 00 06 04 00 01 00 05 5d 61 58 a8 c0 a8 00 37

0020: 00 00 00 00 00 00 c0 a8 00 02

填充位（18 字节）

0020: 00 77 31 d2 50 10

0030: fd 78 41 d3 00 00 00 00 00 00 00

以太网首部：目的主机采用广播地址，源主机的 MAC 地址是 00:05:5d:61:58:a8，上层协议类型 0x0806 表示 ARP。

ARP 帧：硬件类型 0x0001 表示以太网，协议类型 0x0800 表示 IP 协议，硬件地址（MAC 地址）长度为 6，协议地址（IP 地址）长度为 4，op 为 0x0001 表示请求目的主机的 MAC 地址，源主机 MAC 地址为 00:05:5d:61:58:a8，源主机 IP 地址为 c0 a8 00 37（192.168.0.55），目的主机 MAC 地址全 0 待填写，目的主机 IP 地址为 c0 a8 00 02（192.168.0.2）。

由于以太网规定最小数据长度为 46 字节，ARP 帧长度只有 28 字节，因此有 18 字节填充位，填充位的内容没有定义，与具体实现相关。

应答帧如下：

以太网首部

0000: 00 05 5d 61 58 a8 00 05 5d a1 b8 40 08 06

ARP 帧

0000: 00 01

0010: 08 00 06 04 00 02 00 05 5d a1 b8 40 c0 a8 00 02

0020: 00 05 5d 61 58 a8 c0 a8 00 37

填充位

0020: 00 77 31 d2 50 10

0030: fd 78 41 d3 00 00 00 00 00 00 00

以太网首部：目的主机的 MAC 地址是 00:05:5d:61:58:a8，源主机的 MAC 地址是 00:05:5d:a1:b8:40，上层协议类型 0x0806 表示 ARP。

ARP 帧：硬件类型 0x0001 表示以太网，协议类型 0x0800 表示 IP 协议，硬件地址（MAC 地址）长度为 6，协议地址（IP 地址）长度为 4，op 为 0x0002 表示应答，源主机 MAC 地址为 00:05:5d:a1:b8:40，源主机 IP 地址为 c0 a8 00 02（192.168.0.2），目的主机 MAC 地址为 00:05:5d:61:58:a8，目的主机 IP 地址为 c0 a8 00 37（192.168.0.55）。

思考题：如果源主机和目的主机不在同一网段，ARP 请求的广播帧无法穿过路由器，源主机如何与目的主机通

信？

IP 段格式

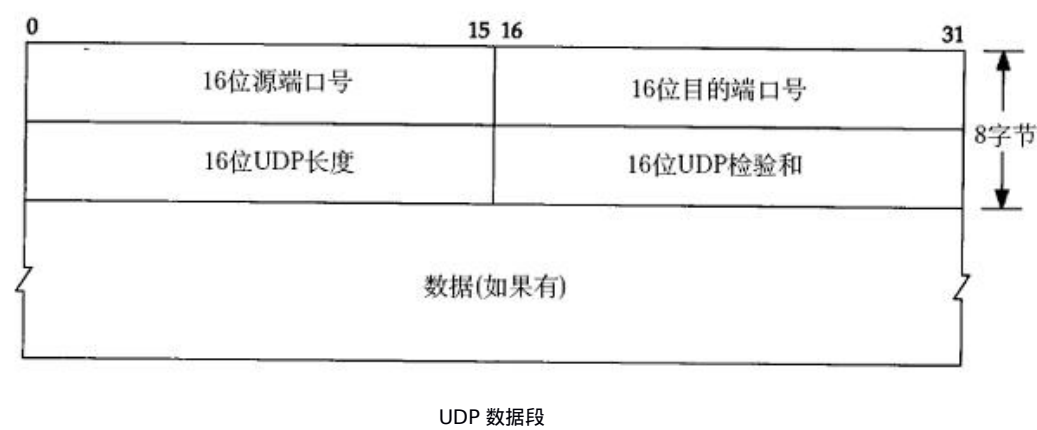


IP 数据报格式

IP 数据报的首部长度和数据长度都是可变长的，但总是 4 字节的整数倍。对于 IPv4，4 位版本字段是 4。4 位首部长度的数值是以 4 字节为单位的，最小值为 5，也就是说首部长度最小是 $4 \times 5 = 20$ 字节，也就是不带任何选项的 IP 首部，4 位能表示的最大值是 15，也就是说首部长度最大是 60 字节。8 位 TOS 字段有 3 个位用来指定 IP 数据报的优先级（目前已经废弃不用），还有 4 个位表示可选的服务类型（最小延迟、最大吞吐量、最大可靠性、最小成本），还有一个位总是 0。总长度是整个数据报（包括 IP 首部和 IP 层 payload）的字节数。每传一个 IP 数据报，16 位的标识加 1，可用于分片和重新组装数据报。3 位标志和 13 位片偏移用于分片。TTL (Time to live) 是这样用的：源主机为数据包设定一个生存时间，比如 64，每过一个路由器就把该值减 1，如果减到 0 就表示路由已经太长了仍然找不到目的主机的网络，就丢弃该包，因此这个生存时间的单位不是秒，而是跳（hop）。协议字段指示上层协议是 TCP、UDP、ICMP 还是 IGMP。然后是校验和，只校验 IP 首部，数据的校验由更高层协议负责。IPv4 的 IP 地址长度为 32 位。

想一想，前面讲了以太网帧中的最小数据长度为 46 字节，不足 46 字节的要求用填充字节补上，那么如何界定这 46 字节里前多少个字节是 IP、ARP 或 RARP 数据报而后面是填充字节？

UDP 数据报格式



下面分析一帧基于 UDP 的 TFTP 协议帧。

以太网首部

0000: 00 05 5d 67 d0 b1 00 05 5d 61 58 a8 08 00

IP 首部

0000: 45 00
0010: 00 53 93 25 00 00 80 11 25 ec c0 a8 00 37 c0 a8
0020: 00 01

UDP 首部

0020 : 05 d4 00 45 00 3f ac 40

TFTP 协议

0020: 00 01 'c':\"'q'
0030: 'w'e'r'q'.'q'w'e'00 'n'e't'a's'c'i'
0040: 'i'00 'b'l'k's'i'z'e'00 '5'1'2'00 't'i'
0050: 'm'e'o'u't'00 '1'0'00 't's'i'z'e'00 '0'

0060: 00 以太网首部 : 源 MAC 地址是 00:05:5d:61:58:a8 , 目的 MAC 地址是 00:05:5d:67:d0:b1 , 上层

协议类型 0x0800 表示 IP。

IP 首部 : 每一个字节 0x45 包含 4 位版本号和 4 位首部长度 , 版本号为 4 , 即 IPv4 , 首部长度为 5 , 说明 IP 首部不带有选项字段。服务类型为 0 , 没有使用服务。16 位总长度字段 (包括 IP 首部和 IP 层 payload 的长度) 为 0x0053 , 即 83 字节 , 加上以太网首部 14 字节可知整个帧长度是 97 字节。IP 报标识是 0x9325 , 标志字段和片偏移字段设置为 0x0000 , 就是 DF=0 允许分片 , MF=0 此数据报没有更多分片 , 没有分片偏移。TTL 是 0x80 , 也就是 128。上层协议 0x11 表示 UDP 协议。IP 首部校验和为 0x25ec , 源主机 IP 是 c0 a8 00 37 (192.168.0.55) ,

目的主机 IP 是 c0 a8 00 01 (192.168.0.1) 。

UDP 首部：源端口号 0x05d4 (1492) 是客户端的端口号，目的端口号 0x0045 (69) 是 TFTP 服务的 well-known 端口号。UDP 报长度为 0x003f，即 63 字节，包括 UDP 首部和 UDP 层 payload 的长度。UDP 首部和 UDP 层 payload 的校验和为 0xac40。

TFTP 是基于文本的协议，各字段之间用字节 0 分隔，开头的 00 01 表示请求读取一个文件，接下来的各字段是：

```
c:\qwerq.qwe
netascii
blksize 512
timeout 10
tsize 0
```

一般的网络通信都是像 TFTP 协议这样，通信的双方分别是客户端和服务端，客户端主动发起请求（上面的例子就是客户端发起的请求帧），而服务端被动地等待、接收和应答请求。客户端的 IP 地址和端口号唯一标识了该主机上的 TFTP 客户端进程，服务端的 IP 地址和端口号唯一标识了该主机上的 TFTP 服务进程，由于客户端是主动发起请求的一方，它必须知道服务端的 IP 地址和 TFTP 服务进程的端口号，所以，一些常见的网络协议有默认的服务端口，例如 HTTP 服务默认 TCP 协议的 80 端口，FTP 服务默认 TCP 协议的 21 端口，TFTP 服务默认 UDP 协议的 69 端口（如上例所示）。在使用客户端程序时，必须指定服务端的主机名或 IP 地址，如果不明确指定端口号则采用默认端口，请读者查阅 ftp、tftp 等程序的 man page 了解如何指定端口号。/etc/services 中列出了所有 well-known 的服务端口和对应的传输层协议，这是由 IANA (Internet Assigned Numbers Authority) 规定的，其中有些服务既可以用 TCP 也可以用 UDP，为了清晰，IANA 规定这样的服务采用相同的 TCP 或 UDP 默认端口号，而另外一些 TCP 和 UDP 的相同端口号却对应不同的服务。

很多服务有 well-known 的端口号，然而客户端程序的端口号却不一定是 well-known 的，往往是每次运行客户端程序时由系统自动分配一个空闲的端口号，用完就释放掉，称为 ephemeral 的端口号，想想这是为什么？

前面提过，UDP 协议不面向连接，也不保证传输的可靠性，例如：

发送端的 UDP 协议层只管把应用层传来的数据封装成段交给 IP 协议层就算完成任务了，如果因为网络故障该

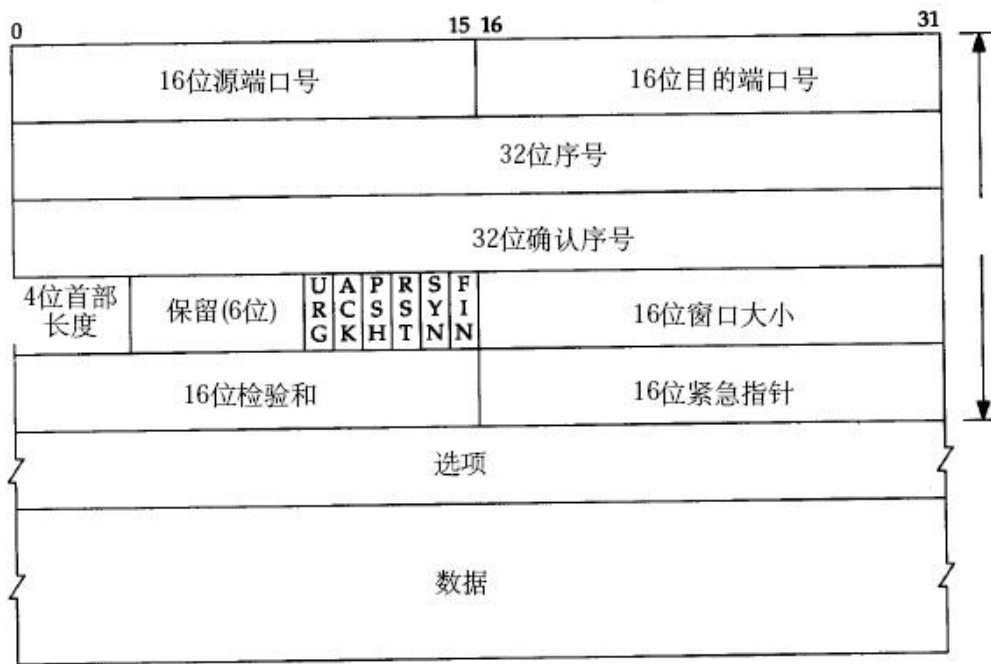
段无法发到对方，UDP 协议层也不会给应用层返回任何错误信息。

接收端的 UDP 协议层只管把收到的数据根据端口号交给相应的应用程序就算完成任务了，如果发送端发来多个数据包并且在网络上经过不同的路由，到达接收端时顺序已经错乱了，UDP 协议层也不保证按发送时的顺序交给应用层。

通常接收端的 UDP 协议层将收到的数据放在一个固定大小的缓冲区中等待应用程序来提取和处理，如果应用程序提取和处理的速度很慢，而发送端发送的速度很快，就会丢失数据包，UDP 协议层并不报告这种错误。

因此，使用 UDP 协议的应用程序必须考虑到这些可能的问题并实现适当的解决方案，例如等待应答、超时重发、为数据包编号、流量控制等。一般使用 UDP 协议的应用程序实现都比较简单，只是发送一些对可靠性要求不高的消息，而不发送大量的数据。例如，基于 UDP 的 TFTP 协议一般只用于传送小文件（所以才叫 trivial 的 ftp），而基于 TCP 的 FTP 协议适用于各种文件的传输。TCP 协议又是如何用面向连接的服务来代替应用程序解决传输的可靠性问题呢。

TCP 数据报格式



TCP 数据段

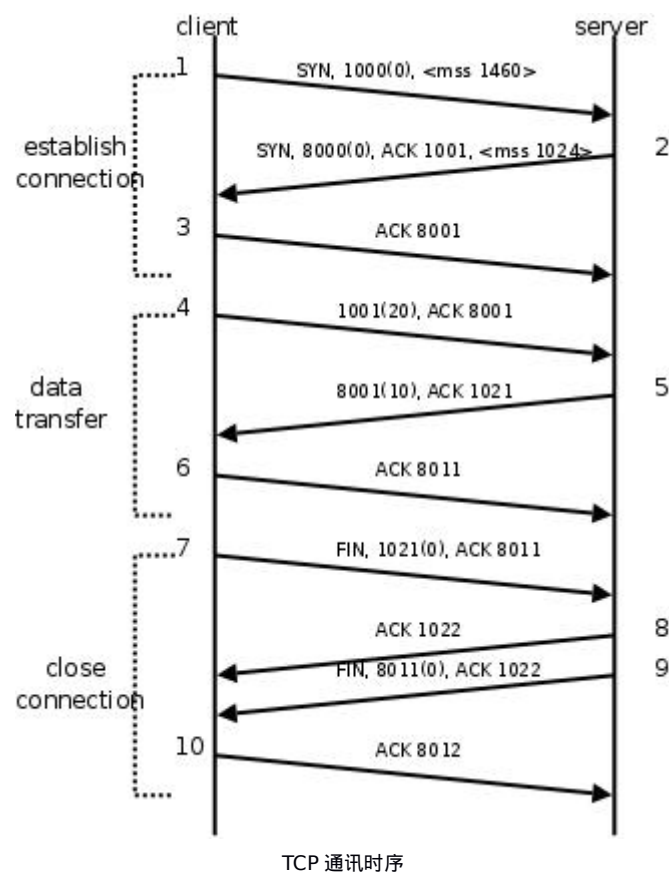
与 UDP 协议一样也有源端口号和目的端口号，通讯的双方由 IP 地址和端口号标识。32 位序号、32 位确认序号、窗口大小稍后详细解释。4 位首部长度和 IP 协议头类似，表示 TCP 协议头的长度，以 4 字节为单位，

因此 TCP 协议头最长可以是 $4 \times 15 = 60$ 字节，如果没有选项字段，TCP 协议头最短 20 字节。URG、ACK、PSH、RST、SYN、FIN 是六个控制位，本节稍后将解释 SYN、ACK、FIN、RST 四个位，其它位的解释从略。16 位检验和将 TCP 协议头和数据都计算在内。紧急指针和各种选项的解释从略。

TCP 协议

TCP 通信时序

下图是一次 TCP 通讯的时序图。TCP 连接建立断开。包含大家熟知的三次握手和四次握手。



在这个例子中，首先客户端主动发起连接、发送请求，然后服务器端响应请求，然后客户端主动关闭连接。两条竖线表示通讯的两端，从上到下表示时间的先后顺序，注意，数据从一端传到网络的另一端也需要时间，所以图中的箭头都是斜的。双方发送的段按时间顺序编号为 1-10，各段中的主要信息在箭头上标出，例如段 2 的箭头上标着 SYN, 8000(0), ACK1001,，表示该段中的 SYN 位置 1，32 位序号是 8000，该段不携带有效载荷（数据字节数为 0），ACK 位置 1，32 位确认序号是 1001，带有一个 mss（Maximum Segment Size，最大报文长度）选项值为 1024。

建立连接（三次握手）的过程：

1. 客户端发送一个带 SYN 标志的 TCP 报文到服务器。这是三次握手过程中的段 1。

客户端发出段 1，SYN 位表示连接请求。序号是 1000，这个序号在网络通讯中用作临时的地址，每发一个数据字节，这个序号要加 1，这样在接收端可以根据序号排出数据包的正确顺序，也可以发现丢包的情况，另外，规定 SYN 位和 FIN 位也要占一个序号，这次虽然没发数据，但是由于发了 SYN 位，因此下次再发送应该用序号 1001。mss 表示最大段尺寸，如果一个段太大，封装成帧后超过了链路层的最大帧长度，就必须在 IP 层分片，为了避免这种情况，客户端声明自己的最大段尺寸，建议服务器端发来的段不要超过这个长度。

2. 服务器端回应客户端，是三次握手中的第 2 个报文段，同时带 ACK 标志和 SYN 标志。它表示对刚才客户端 SYN 的回应；同时又发送 SYN 给客户端，询问客户端是否准备好进行数据通讯。

服务器发出段 2，也带有 SYN 位，同时置 ACK 位表示确认，确认序号是 1001，表示“我接收到序号 1000 及其以前所有的段，请你下次发送序号为 1001 的段”，也就是应答了客户端的连接请求，同时也给客户端发出一个连接请求，同时声明最大尺寸为 1024。

3. 客户必须再次回应服务器端一个 ACK 报文，这是报文段 3。

客户端发出段 3，对服务器的连接请求进行应答，确认序号是 8001。在这个过程中，客户端和服务端分别给对方发了连接请求，也应答了对方的连接请求，其中服务器的请求和应答在一个段中发出，因此一共有三个段用于建立连接，称为“三方握手（three-way-handshake）”。在建立连接的同时，双方协商了一些信息，例如双方发送序号的初始值、最大段尺寸等。

在 TCP 通讯中，如果一方收到另一方发来的段，读出其中的目的端口号，发现本机没有任何进程使用这个端口，就会应答一个包含 RST 位的段给另一方。例如，服务器没有任何进程使用 8080 端口，我们却用 telnet 客户端去连接它，服务器收到客户端发来的 SYN 段就会应答一个 RST 段，客户端的 telnet 程序收到 RST 段后报告错误 Connection refused：

```
$ telnet 192.168.0.200 8080
```

Trying 192.168.0.200...

telnet: Unable to connect to remote host: Connection refused

数据传输的过程：

1. 客户端发出段 4，包含从序号 1001 开始的 20 个字节数据。
2. 服务器发出段 5，确认序号为 1021，对序号为 1001-1020 的数据表示确认收到，同时请求发送序号 1021 开始的数据，服务器在应答的同时也向客户端发送从序号 8001 开始的 10 个字节数据，这称为 piggyback。
3. 客户端发出段 6，对服务器发来的序号为 8001-8010 的数据表示确认收到，请求发送序号 8011 开始的数据。

在数据传输过程中，ACK 和确认序号是非常重要的，应用程序交给 TCP 协议发送的数据会暂存在 TCP 层的发送缓冲区中，发出数据包给对方之后，只有收到对方应答的 ACK 段才知道该数据包确实发到了对方，可以从发送缓冲区中释放掉了，如果因为网络故障丢失了数据包或者丢失了对方发回的 ACK 段，经过等待超时后 TCP 协议自动将发送缓冲区中的数据包重发。

关闭连接（四次握手）的过程：

由于 TCP 连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个 FIN 来终止这个方向的连接。收到一个 FIN 只意味着这一方向上没有数据流动，一个 TCP 连接在收到一个 FIN 后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

1. 客户端发出段 7，FIN 位表示关闭连接的请求。
2. 服务器发出段 8，应答客户端的关闭连接请求。
3. 服务器发出段 9，其中也包含 FIN 位，向客户端发送关闭连接请求。
4. 客户端发出段 10，应答服务器的关闭连接请求。

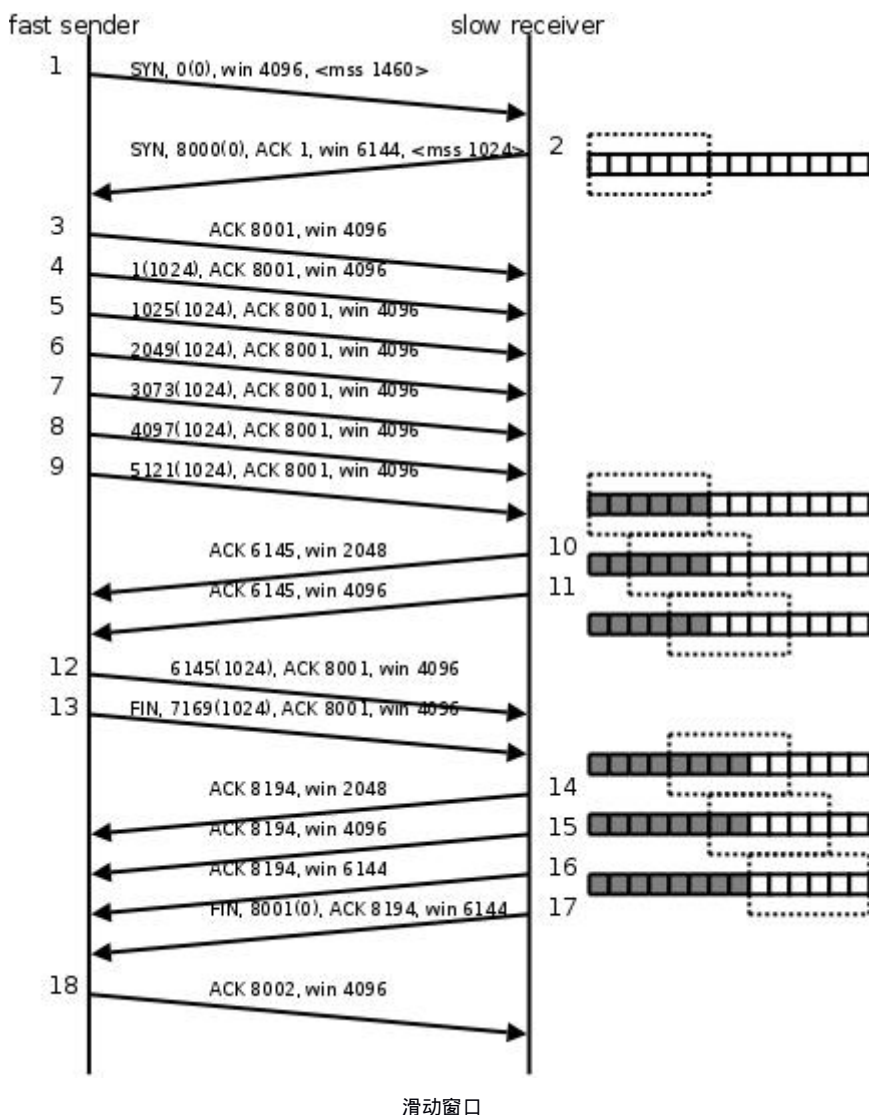
建立连接的过程是三方握手，而关闭连接通常需要 4 个段，服务器的应答和关闭连接请求通常不合并在一个段中，因为有连接半关闭的情况，这种情况下客户端关闭连接之后就不能再发送数据给服务器了，但是服务器还可以

发送数据给客户端，直到服务器也关闭连接为止。

滑动窗口 (TCP 流量控制)

介绍 UDP 时我们描述了这样的问题：如果发送端发送的速度较快，接收端接收到数据后处理的速度较慢，而接收缓冲区的大小是固定的，就会丢失数据。TCP 协议通过“滑动窗口 (Sliding Window)”机制解决这一问题。

看下图的通讯过程：



1. 发送端发起连接，声明最大段尺寸是 1460，初始序号是 0，窗口大小是 4K，表示“我的接收缓冲区还有 4K 字节空闲，你发的数据不要超过 4K”。接收端应答连接请求，声明最大段尺寸是 1024，初始序号是 8000，窗口大小是 6K。发送端应答，三方握手结束。
2. 发送端发出段 4-9，每个段带 1K 的数据，发送端根据窗口大小知道接收端的缓冲区满了，因此停止发送数据。

3. 接收端的应用程序提走 2K 数据，接收缓冲区又有了 2K 空闲，接收端发出段 10，在应答已收到 6K 数据的同时声明窗口大小为 2K。
4. 接收端的应用程序又提走 2K 数据，接收缓冲区有 4K 空闲，接收端发出段 11，重新声明窗口大小为 4K。
5. 发送端发出段 12-13，每个段带 2K 数据，段 13 同时还包含 FIN 位。
6. 接收端应答接收到的 2K 数据（6145-8192），再加上 FIN 位占一个序号 8193，因此应答序号是 8194，连接处于半关闭状态，接收端同时声明窗口大小为 2K。
7. 接收端的应用程序提走 2K 数据，接收端重新声明窗口大小为 4K。
8. 接收端的应用程序提走剩下的 2K 数据，接收缓冲区全空，接收端重新声明窗口大小为 6K。
9. 接收端的应用程序在提走全部数据后，决定关闭连接，发出段 17 包含 FIN 位，发送端应答，连接完全关闭。

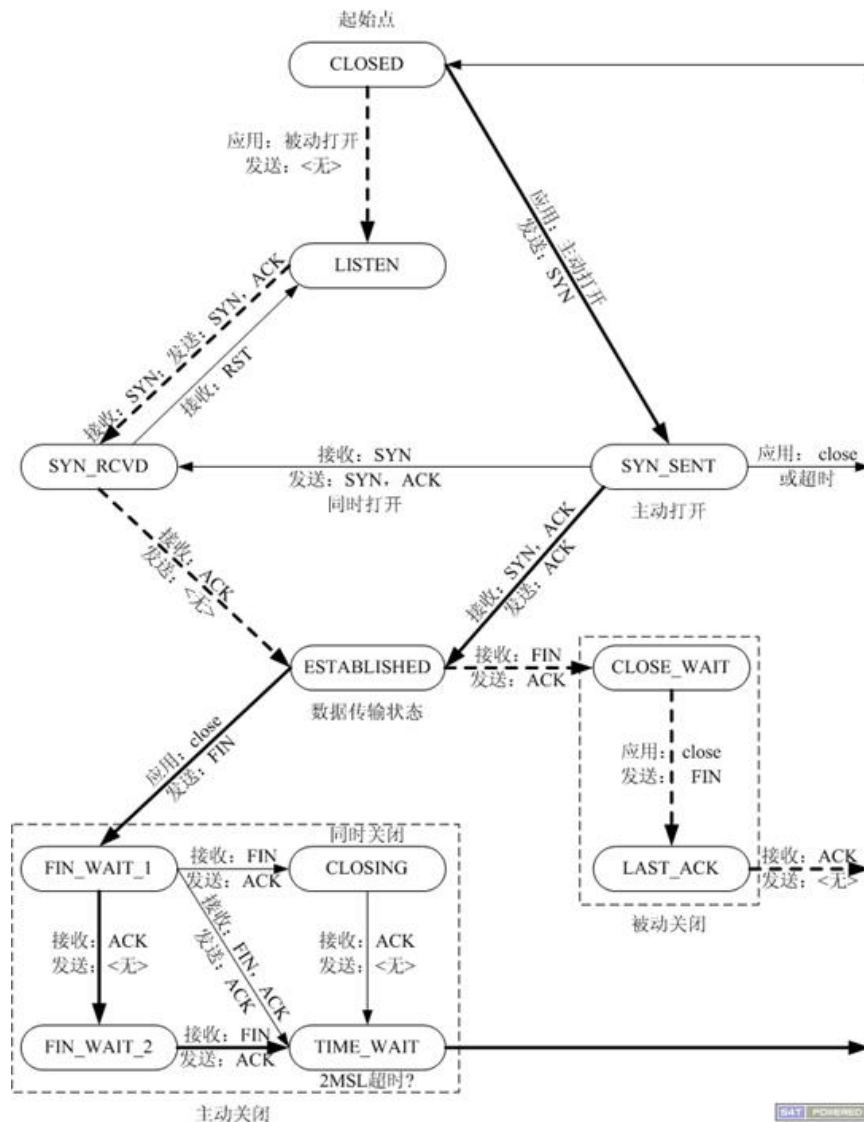
上图在接收端用小方块表示 1K 数据，实心的小方块表示已接收到的数据，虚线框表示接收缓冲区，因此套在虚线框中的空心小方块表示窗口大小，从图中可以看出，随着应用程序提走数据，虚线框是向右滑动的，因此称为滑动窗口。

从这个例子还可以看出，发送端是一 K 一 K 地发送数据，而接收端的应用程序可以两 K 两 K 地提走数据，当然也有可能一次提走 3K 或 6K 数据，或者一次只提走几个字节的数据。也就是说，应用程序所看到的数据是一个整体，或说是一个流（stream），在底层通讯中这些数据可能被拆成很多数据包来发送，但是一个数据包有多少字节对应用程序是不可见的，因此 TCP 协议是面向流的协议。而 UDP 是面向消息的协议，每个 UDP 段都是一条消息，应用程序必须以消息为单位提取数据，不能一次提取任意字节的数据，这一点和 TCP 是很不同的。

TCP 状态转换

这个图 N 多人都知道，它排除和定位网络或系统故障时大有帮助，但是怎样牢牢地将这张图刻在脑中呢？那么你就一定要对这张图的每一个状态，及转换的过程有深刻的认识，不能只停留在一知半解之中。下面对这张图的 11 种状态详细解析一下，以便加强记忆！不过在这之前，先回顾一下 TCP 建立连接的三次握手过程，以及 关闭连接的四次握手过程。

TCP 状态转换图



TCP 状态转换图

CLOSED : 表示初始状态。

LISTEN : 该状态表示服务器端的某个 SOCKET 处于监听状态，可以接受连接。

SYN_SENT : 这个状态与 SYN_RCVD 遥相呼应，当客户端 SOCKET 执行 CONNECT 连接时，它首先发送 SYN 报文，随即进入到了 SYN_SENT 状态，并等待服务端的发送三次握手中的第 2 个报文。SYN_SENT 状态表示客户端已发送 SYN 报文。

SYN_RCVD: 该状态表示接收到 SYN 报文，在正常情况下，这个状态是服务器端的 SOCKET 在建立 TCP 连接时的三次握手会话过程中的一个中间状态，很短暂。此种状态时，当收到客户端的 ACK 报文后，会进入到 ESTABLISHED 状态。

ESTABLISHED：表示连接已经建立。

FIN_WAIT_1：FIN_WAIT_1 和 FIN_WAIT_2 状态的真正含义都是表示等待对方的 FIN 报文。区别是：

FIN_WAIT_1 状态是当 socket 在 ESTABLISHED 状态时，想主动关闭连接，向对方发送了 FIN 报文，此时该 socket 进入到 FIN_WAIT_1 状态。

FIN_WAIT_2 状态是当对方回应 ACK 后，该 socket 进入到 FIN_WAIT_2 状态，正常情况下，对方应马上回应 ACK 报文，所以 FIN_WAIT_1 状态一般较难见到，而 FIN_WAIT_2 状态可用 netstat 看到。

FIN_WAIT_2：主动关闭链接的一方，发出 FIN 收到 ACK 以后进入该状态。称之为半连接或半关闭状态。该状态下的 socket 只能接收数据，不能发。

TIME_WAIT：表示收到了对方的 FIN 报文，并发送出了 ACK 报文，等 2MSL 后即可回到 CLOSED 可用状态。如果 FIN_WAIT_1 状态下，收到对方同时带 FIN 标志和 ACK 标志的报文时，可以直接进入到 TIME_WAIT 状态，而无须经过 FIN_WAIT_2 状态。

CLOSING：这种状态较特殊，属于一种较罕见的状态。正常情况下，当你发送 FIN 报文后，按理来说是应该先收到（或同时收到）对方的 ACK 报文，再收到对方的 FIN 报文。但是 CLOSING 状态表示你发送 FIN 报文后，并没有收到对方的 ACK 报文，反而却也收到了对方的 FIN 报文。什么情况下会出现此种情况呢？如果双方几乎在同时 close 一个 SOCKET 的话，那么就出现了双方同时发送 FIN 报文的情况，也即会出现 CLOSING 状态，表示双方都正在关闭 SOCKET 连接。

CLOSE_WAIT：此种状态表示在等待关闭。当对方关闭一个 SOCKET 后发送 FIN 报文给自己，系统会回应一个 ACK 报文给对方，此时则进入到 CLOSE_WAIT 状态。接下来呢，察看是否还有数据发送给对方，如果没有可以 close 这个 SOCKET，发送 FIN 报文给对方，即关闭连接。所以在 CLOSE_WAIT 状态下，需要关闭连接。

LAST_ACK：该状态是被动关闭一方在发送 FIN 报文后，最后等待对方的 ACK 报文。当收到 ACK 报文后，即可以进入到 CLOSED 可用状态。

半关闭

当 TCP 链接中 A 发送 FIN 请求关闭，B 端回应 ACK 后（A 端进入 FIN_WAIT_2 状态），B 没有立即发送 FIN 给 A 时，A 方处在半链接状态，此时 A 可以接收 B 发送的数据，但是 A 已不能再向 B 发送数据。

从程序的角度，可以使用 API 来控制实现半连接状态。

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

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

sockfd: 需要关闭的 socket 的描述符

how: 允许为 shutdown 操作选择以下几种方式:

SHUT_RD(0): 关闭 sockfd 上的读功能，此选项将不允许 sockfd 进行读操作。

该套接字**不再接受数据**，任何当前在套接字接受缓冲区的数据将被无声的丢弃掉。

SHUT_WR(1): 关闭 sockfd 的写功能，此选项将不允许 sockfd 进行写操作。进程不能在此套接字发出写操作。

SHUT_RDWR(2): 关闭 sockfd 的读写功能。相当于调用 shutdown 两次：首先是以 SHUT_RD, 然后以 SHUT_WR。

使用 close 中止一个连接，但它只是减少描述符的引用计数，并不直接关闭连接，只有当描述符的引用计数为 0 时才关闭连接。

shutdown 不考虑描述符的引用计数，直接关闭描述符。也可选择中止一个方向的连接，只中止读或只中止写。

注意:

1. 如果有多个进程共享一个套接字，close 每被调用一次，计数减 1，直到计数为 0 时，也就是所用进程都调用了 close，套接字将被释放。
2. 在多进程中如果一个进程调用了 shutdown(sfd, SHUT_RDWR)后，其它的进程将无法进行通信。但，如果一个进程 close(sfd)将不会影响到其它进程。

2MSL

2MSL (Maximum Segment Lifetime) TIME_WAIT 状态的存在有两个理由：

(1) 让 **4 次握手关闭流程更加可靠**；4 次握手的最后一个 ACK 是由主动关闭方发送出去的，若这个 ACK 丢失，被动关闭方会再次发一个 FIN 过来。若主动关闭方能够保持一个 2MSL 的 TIME_WAIT 状态，则有更大的机

会让丢失的 ACK 被再次发送出去。

(2) 防止 lost duplicate 对后续新建正常链接的传输造成破坏。lost uplicate 在实际的网络中非常常见，经常是由于路由器产生故障，路径无法收敛，导致一个 packet 在路由器 A，B，C 之间做类似死循环的跳转。IP 头部有个 TTL，限制了一个包在网络中的最大跳数，因此这个包有两种命运，要么最后 TTL 变为 0，在网络中消失；要么 TTL 在变为 0 之前路由器路径收敛，它凭借剩余的 TTL 跳数终于到达目的地。但非常可惜的是 TCP 通过超时重传机制在早些时候发送了一个跟它一模一样的包，并先于它达到了目的地，因此它的命运也就注定被 TCP 协议栈抛弃。

另外一个概念叫做 incarnation connection，指跟上次的 socket pair 一模一样的新连接，叫做 incarnation of previous connection。lost uplicate 加上 incarnation connection，则会对我们的传输造成致命的错误。

TCP 是流式的，所有包到达的顺序是不一致的，依靠序列号由 TCP 协议栈做顺序的拼接，假设一个 incarnation connection 这时收到的 seq=1000，来了一个 lost duplicate 为 seq=1000，len=1000，则 TCP 认为这个 lost duplicate 合法，并存放入了 receive buffer，导致传输出现错误。通过一个 2MSL TIME_WAIT 状态，确保所有的 lost duplicate 都会消失掉，避免对新连接造成错误。

该状态为什么设计在**主动关闭这一方**：

(1) 发最后 ACK 的是主动关闭一方。

(2) 只要有一方保持 TIME_WAIT 状态，就能起到避免 incarnation connection 在 2MSL 内的重新建立，不需要两方都有。

如何正确对待 2MSL TIME_WAIT？

RFC 要求 socket pair 在处于 TIME_WAIT 时，不能再起一个 incarnation connection。但绝大部分 TCP 实现，强加了更为严格的限制。在 2MSL 等待期间，socket 中使用的本地端口在默认情况下不能再被使用。

若 A 10.234.5.5 : 1234 和 B 10.55.55.60 : 6666 建立了连接 A 主动关闭，那么在 A 端只要 port 为 1234，无论对方的 port 和 ip 是什么，都不允许再起服务。这甚至比 RFC 限制更为严格，RFC 仅仅是要求 socket pair

不一致，而实现当中只要这个 port 处于 TIME_WAIT，就不允许起连接。这个限制对主动打开方来说是无所谓的，因为一般用的是临时端口；但对于被动打开方，一般是 server，就悲剧了，因为 server 一般是熟知端口。比如 http，一般端口是 80，不可能允许这个服务在 2MSL 内不能起来。

解决方案是给服务器的 socket 设置 SO_REUSEADDR 选项，这样的话就算熟知端口处于 TIME_WAIT 状态，在这个端口上依旧可以将服务启动。当然，虽然有了 SO_REUSEADDR 选项，但 sockt pair 这个限制依旧存在。比如上面的例子，A 通过 SO_REUSEADDR 选项依旧在 1234 端口上起了监听，但这时我们若是从 B 通过 6666 端口去连它，TCP 协议会告诉我们连接失败，原因为 Address already in use.

RFC 793 中规定 MSL 为 2 分钟，实际应用中常用的是 30 秒，1 分钟和 2 分钟等。

RFC (Request For Comments)，是一系列以编号排定的文件。收集了有关因特网相关资讯，以及 UNIX 和因特网社群的软件文件。

程序设计中的问题

做一个测试，首先启动 server，然后启动 client，用 Ctrl-C 终止 server，马上再运行 server，运行结果：

```
itcast$ ./server
bind error: Address already in use
```

这是因为，虽然 server 的应用程序终止了，但 TCP 协议层的连接并没有完全断开，因此不能再次监听同样的 server 端口。我们用 netstat 命令查看一下：

```
itcast$ netstat -apn |grep 6666
tcp        1      0 192.168.1.11:38103      192.168.1.11:6666      CLOSE_WAIT 3525/client
tcp        0      0 192.168.1.11:6666      192.168.1.11:38103      FIN_WAIT2  -
```

server 终止时，socket 描述符会自动关闭并发 FIN 段给 client，client 收到 FIN 后处于 CLOSE_WAIT 状态，但是 client 并没有终止，也没有关闭 socket 描述符，因此不会发 FIN 给 server，因此 server 的 TCP 连接处于 FIN_WAIT2 状态。

现在用 Ctrl-C 把 client 也终止掉，再观察现象：

```
itcast$ netstat -apn |grep 6666
tcp        0      0 192.168.1.11:6666      192.168.1.11:38104      TIME_WAIT  -
```

```
itcast$ ./server
bind error: Address already in use
```

client 终止时自动关闭 socket 描述符 ,server 的 TCP 连接收到 client 发的 FIN 段后处于 TIME_WAIT 状态。TCP 协议规定 ,主动关闭连接的一方要处于 **TIME_WAIT** 状态 ,等待两个 MSL (maximum segment lifetime) 的时间后才能回到 CLOSED 状态 ,因为我们先 Ctrl-C 终止了 server ,所以 server 是主动关闭连接的一方 ,在 TIME_WAIT 期间仍然不能再次监听同样的 server 端口。

MSL 在 RFC 1122 中规定为两分钟 ,但是各操作系统的实现不同 ,在 Linux 上一般经过半分钟后就可以再次启动 server 了。至于为什么要规定 TIME_WAIT 的时间 ,可参考 UNP 2.7 节。

端口复用

在 server 的 TCP 连接没有完全断开之前不允许重新监听是不合理的。因为 ,TCP 连接没有完全断开指的是 connfd (127.0.0.1:6666) 没有完全断开 ,而我们重新监听的是 lis-tenfd (0.0.0.0:6666) ,虽然是占用同一个端口 ,但 IP 地址不同 ,connfd 对应的是与某个客户端通讯的一个具体的 IP 地址 ,而 listenfd 对应的是 wildcard address。解决问题的方法是使用 setsockopt()设置 socket 描述符的选项 SO_REUSEADDR 为 1 ,表示允许创建端口号相同但 IP 地址不同的多个 socket 描述符。

在 server 代码的 socket()和 bind()调用之间插入如下代码 :

```
int opt = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

有关 setsockopt 可以设置的其它选项请参考 UNP 第 7 章。

TCP 异常断开

心跳检测机制

在 TCP 网络通信中 ,经常会出现客户端和服务端之间的非正常断开 ,需要实时检测查询链接状态。常用的解决方法就是在程序中加入心跳机制。

Heart-Beat 线程

这个是最常用的简单方法。在接收和发送数据时个人设计一个守护进程(线程)，定时发送 Heart-Beat 包，客户端/服务器收到该小包后，立刻返回相应的包即可检测对方是否实时在线。

该方法的好处是通用，但缺点就是会改变现有的通讯协议！大家一般都是使用业务层心跳来处理，主要是灵活可控。

UNIX 网络编程不推荐使用 SO_KEEPALIVE 来做心跳检测，还是在业务层以心跳包做检测比较好，也方便控制。

设置 TCP 属性

SO_KEEPALIVE 保持连接检测对方主机是否崩溃，避免（服务器）永远阻塞于 TCP 连接的输入。设置该选项后，如果 2 小时内在此套接口的任一方向都没有数据交换，TCP 就自动给对方发一个保持存活探测分节(keepalive probe)。这是一个对方必须响应的 TCP 分节.它会导致以下三种情况：对方接收一切正常：以期望的 ACK 响应。2 小时后，TCP 将发出另一个探测分节。对方已崩溃且已重新启动：以 RST 响应。套接口的待处理错误被置为 ECONNRESET，套接口本身则被关闭。对方无任何响应：源自 berkeley 的 TCP 发送另外 8 个探测分节，相隔 75 秒一个，试图得到一个响应。在发出第一个探测分节 11 分钟 15 秒后若仍无响应就放弃。套接口的待处理错误被置为 ETIMEOUT，套接口本身则被关闭。如 ICMP 错误是“host unreachable(主机不可达)”，说明对方主机并没有崩溃，但是不可达，这种情况下待处理错误被置为 EHOSTUNREACH。

根据上面的介绍我们可以知道对端以一种非优雅的方式断开连接的时候，我们可以设置 SO_KEEPALIVE 属性使得我们在 2 小时以后发现对方的 TCP 连接是否依然存在。

```
keepAlive = 1;
setsockopt(listenfd, SOL_SOCKET, SO_KEEPALIVE, (void*)&keepAlive, sizeof(keepAlive));
```

如果我们不能接受如此之长的等待时间，从 TCP-Keepalive-HOWTO 上可以知道一共有两种方式可以设置，一种是修改内核关于网络方面的配置参数，另外一种就是 SOL_TCP 字段的 TCP_KEEPIDL，TCP_KEEPINTVL，TCP_KEEPCNT 三个选项。

1. The tcp_keepidle parameter specifies the interval of inactivity that causes TCP to generate a KEEPALIVE transmission for an application that requests them. tcp_keepidle defaults to

14400 (two hours).

/*开始首次 KeepAlive 探测前的 TCP 空闲时间 */

2. The `tcp_keepintvl` parameter specifies the interval between the nine retries that are attempted if a `KEEPAIVE` transmission is not acknowledged. `tcp_keep intvl` defaults to 150 (75 seconds).

/* 两次 KeepAlive 探测间的时间间隔 */

3. The `tcp_keepcnt` option specifies the maximum number of keepalive probes to be sent. The value of `TCP_KEEPCNT` is an integer value between 1 and `n`, where `n` is the value of the systemwide `tcp_keepcnt` parameter.

/* 判定断开前的 KeepAlive 探测次数*/

```
int keepIdle = 1000;
int keepInterval = 10;
int keepCount = 10;

Setsockopt(listenfd, SOL_TCP, TCP_KEEPIIDLE, (void *)&keepIdle, sizeof(keepIdle));
Setsockopt(listenfd, SOL_TCP, TCP_KEEPIIDLE, (void *)&keepInterval, sizeof(keepInterval));
Setsockopt(listenfd, SOL_TCP, TCP_KEEPCNT, (void *)&keepCount, sizeof(keepCount));
```

`SO_KEEPAIVE` 设置空闲 2 小时才发送一个“保持存活探测分节”，不能保证实时检测。对于判断网络断开时间太长，对于需要及时响应的程序不太适应。

当然也可以修改时间间隔参数，但是会影响到所有打开此选项的套接口！关联了完成端口的 `socket` 可能会忽略掉该套接字选项。

网络名词术语解析

路由(route)

路由（名词）

数据包从源地址到目的地址所经过的路径，由一系列路由节点组成。

路由（动词）

某个路由节点为数据包选择投递方向的选路过程。

路由器工作原理

路由器（Router）是连接因特网中各局域网、广域网的设备，它会根据信道的情况自动选择和设定路由，以最佳路径，按前后顺序发送信号的设备。

传统地，路由器工作于 OSI 七层协议中的第三层，其主要任务是接收来自一个网络接口的数据包，根据其中所含的目的地址，决定转发到下一个目的地址。因此，路由器首先得在转发路由表中查找它的目的地址，若找到了目的地址，就在数据包的帧格前添加下一个 MAC 地址，同时 IP 数据包头的 TTL（Time To Live）域也开始减数，并重新计算校验和。当数据包被送到输出端口时，它需要按顺序等待，以便被传送到输出链路上。

路由器在工作时能够按照某种路由通信协议查找设备中的路由表。如果到某一特定节点有一条以上的路径，则基本预先确定的路由准则是选择最优（或最经济）的传输路径。由于各种网络段和其相互连接情况可能会因环境变化而变化，因此路由情况的信息一般也按所使用的路由信息协议的规定而定时更新。

网络中，每个路由器的基本功能都是按照一定的规则来动态地更新它所保持的路由表，以便保持路由信息的有效性。为了便于在网络间传送报文，路由器总是先按照预定的规则把较大的数据分解成适当大小的数据包，再将这些数据包分别通过相同或不同路径发送出去。当这些数据包按先后秩序到达目的地后，再把分解的数据包按照一定顺序包装成原有的报文形式。路由器的分层寻址功能是路由器的重要功能之一，该功能可以帮助具有很多节点站的网络来存储寻址信息，同时还能在网络间截获发送到远地网段的报文，起转发作用；选择最合理的路由，引导通信也是路由器基本功能；多协议路由器还可以连接使用不同通信协议的网络段，成为不同通信协议网络段之间的通信平台。

路由和交换之间的主要区别就是交换发生在 OSI 参考模型第二层（数据链路层），而路由发生在第三层，即网络层。这一区别决定了路由和交换在移动信息的过程中需使用不同的控制信息，所以两者实现各自功能的方式是不同的。

路由表(Routing Table)

在计算机网络中，路由表或称路由择域信息库（RIB）是一个存储在路由器或者联网计算机中的电子表格（文

件)或类数据库。路由表存储着指向特定网络地址的路径。

路由条目

路由表中的一行，每个条目主要由目的网络地址、子网掩码、下一跳地址、发送接口四部分组成，如果要发送的数据包的目的网络地址匹配路由表中的某一行，就按规定的接口发送到下一跳地址。

缺省路由条目

路由表中的最后一行，主要由下一跳地址和发送接口两部分组成，当目的地址与路由表中其它行都不匹配时，就按缺省路由条目规定的接口发送到下一跳地址。

路由节点

一个具有路由能力的主机或路由器，它维护一张路由表，通过查询路由表来决定向哪个接口发送数据包。

以太网交换机工作原理

以太网交换机是基于以太网传输数据的交换机，以太网采用共享总线型传输媒体方式的局域网。以太网交换机的结构是每个端口都直接与主机相连，并且一般都工作在全双工方式。交换机能同时连通许多对端口，使每一对相互通信的主机都能像独占通信媒体那样，进行无冲突地传输数据。

以太网交换机工作于 OSI 网络参考模型的第二层 (即数据链路层) 是一种基于 MAC (Media Access Control , 介质访问控制) 地址识别、完成以太网数据帧转发的网络设备。

hub 工作原理

集线器实际上就是中继器的一种，其区别仅在于集线器能够提供更多的端口服务，所以集线器又叫多口中继器。

集线器功能是随机选出某一端口的设备，并让它独占全部带宽，与集线器的上联设备 (交换机、路由器或服务器等) 进行通信。从 Hub 的工作方式可以看出，它在网络中只起到信号放大和重发作用，其目的是扩大网络的传输范围，而不具备信号的定向传送能力，是一个标准的共享式设备。其次是 Hub 只与它的上联设备 (如上层 Hub、

交换机或服务器)进行通信，同层的各端口之间不会直接进行通信，而是通过上联设备再将信息广播到所有端口上。

由此可见，即使是在同一 Hub 的不同两个端口之间进行通信，都必须要经过两步操作：

第一步是将信息上传到上联设备；

第二步是上联设备再将该信息广播到所有端口上。

半双工/全双工

Full-duplex（全双工）全双工是在通道中同时双向数据传输的能力。

Half-duplex（半双工）在通道中同时只能沿着一个方向传输数据。

DNS 服务器

DNS 是域名系统（Domain Name System）的缩写，是因特网的一项核心服务，它作为可以将域名和 IP 地址相互映射的一个分布式数据库，能够使人更方便的访问互联网，而不用去记住能够被机器直接读取的 IP 地址串。

它是由解析器以及域名服务器组成的。域名服务器是指保存有该网络中所有主机的域名和对应 IP 地址，并具有将域名转换为 IP 地址功能的服务器。

局域网(LAN)

local area network，一种覆盖一座或几座大楼、一个校园或者一个厂区等地理区域的小范围的计算机网。

1. 覆盖的地理范围较小，只在一个相对独立的局部范围内联，如一座或集中的建筑群内。
2. 使用专门铺设的传输介质进行联网，数据传输速率高（10Mb/s ~ 10Gb/s）
3. 通信延迟时间短，可靠性较高
4. 局域网可以支持多种传输介质

广域网(WAN)

wide area network，一种用来实现不同地区的局域网或城域网的互连，可提供不同地区、城市和国家之间的计算机通信的远程计算机网。

覆盖的范围比局域网（LAN）和城域网（MAN）都广。广域网的通信子网主要使用分组交换技术。

广域网的通信子网可以利用公用分组交换网、卫星通信网和无线分组交换网，它将分布在不同地区的局域网或计算机系统互连起来，达到资源共享的目的。如互联网是世界范围内最大的广域网。

1. 适应大容量与突发性通信的要求；
2. 适应综合业务服务的要求；
3. 开放的设备接口与规范化的协议；
4. 完善的通信服务与网络管理。

端口

逻辑意义上的端口，一般是指 TCP/IP 协议中的端口，端口号的范围从 0 到 65535，比如用于浏览网页服务的 80 端口，用于 FTP 服务的 21 端口等等。

1. 端口号小于 256 的定义为常用端口，服务器一般都是通过常用端口号来识别的。
2. 客户端只需保证该端口号在本机上是惟一的就可以了。客户端端口号因存在时间很短暂又称临时端口号；
3. 大多数 TCP/IP 实现给临时端口号分配 1024—5000 之间的端口号。大于 5000 的端口号是为其他服务器预留的。

我们应该在自定义端口时，避免使用 well-known 的端口。如：80、21 等等。

MTU

MTU:通信术语 最大传输单元（Maximum Transmission Unit，MTU）

是指一种通信协议的某一层上面所能通过的最大数据包大小（以字节为单位）。最大传输单元这个参数通常与通信接口有关（网络接口卡、串口等）。

以下是一些协议的 MTU：

FDDI 协议：4352 字节

以太网（**Ethernet**）协议：**1500** 字节

PPPoE（ADSL）协议：1492 字节

X.25 协议（Dial Up/Modem）：576 字节

Point-to-Point：4470 字节

常见网络知识面试题

1. TCP 如何建立链接
2. TCP 如何通信
3. TCP 如何关闭链接
4. 什么是滑动窗口
5. 什么是半关闭
6. 局域网内两台机器如何利用 TCP/IP 通信
7. internet 上两台主机如何进行通信
8. 如何在 internet 上识别唯一一个进程

答：通过“IP 地址+端口号”来区分不同的服务

9. 为什么说 TCP 是可靠的链接，UDP 不可靠
10. 路由器和交换机的区别

Socket 编程

套接字概念

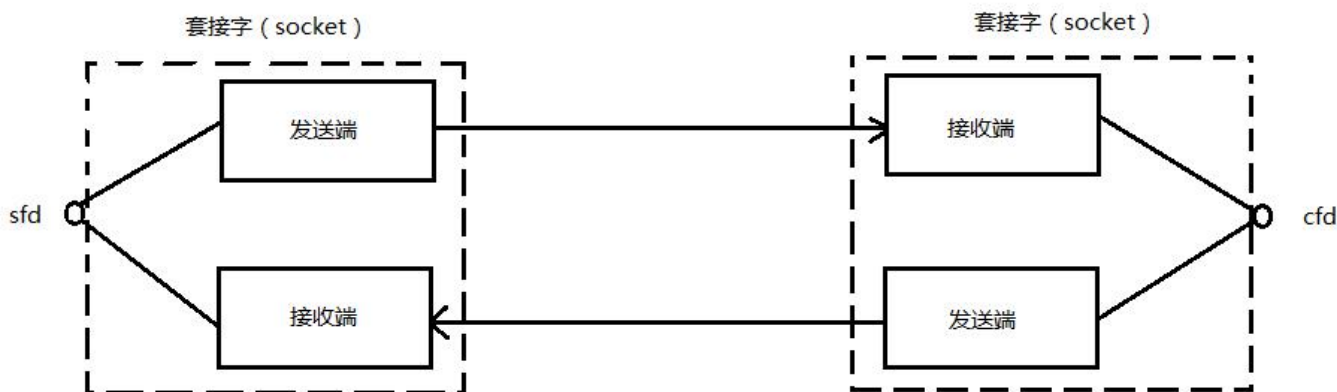
Socket 本身有“插座”的意思，在 Linux 环境下，用于表示进程间网络通信的特殊文件类型。本质为内核借助缓冲区形成的伪文件。

既然是文件，那么理所当然的，我们可以使用文件描述符引用套接字。与管道类似的，Linux 系统将其封装成文件的目的是为了统一接口，使得读写套接字和读写文件的操作一致。区别是管道主要应用于本地进程间通信，而套接字多应用于网络进程间数据的传递。

套接字的内核实现较为复杂，不宜在学习初期深入学习。

在 TCP/IP 协议中，“IP 地址+TCP 或 UDP 端口号”唯一标识网络通讯中的一个进程。“IP 地址+端口号”就对应一个 socket。欲建立连接的两个进程各自有一个 socket 来标识，那么这两个 socket 组成的 socket pair 就唯一标识一个连接。因此可以用 Socket 来描述网络连接的一对一关系。

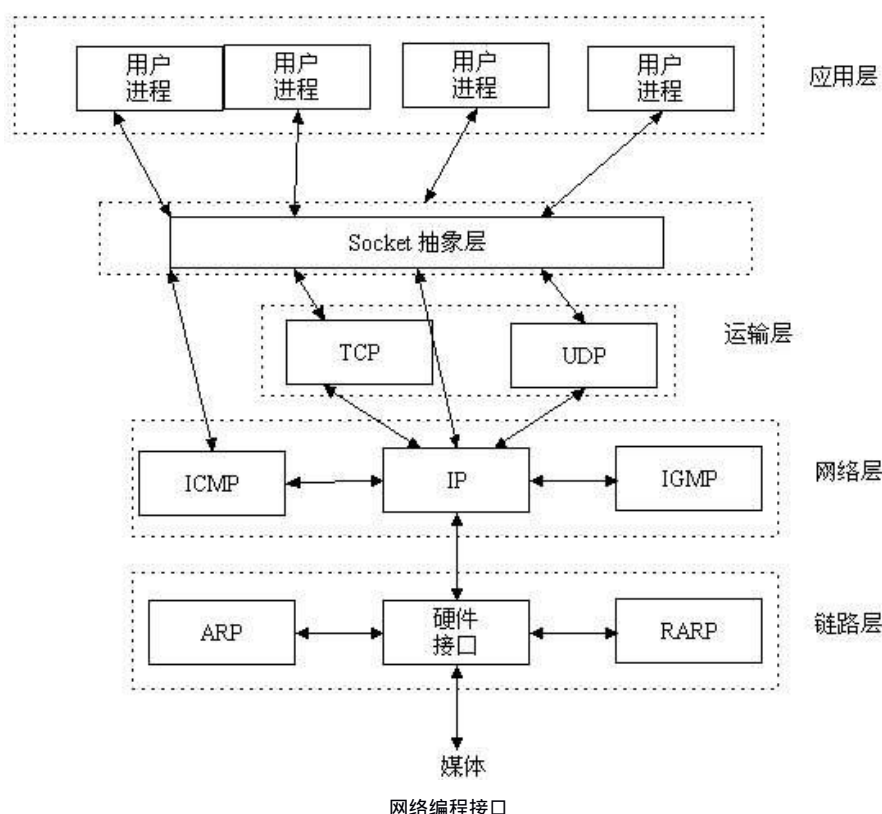
套接字通信原理如下图所示：



套接字通讯原理示意

在网络通信中，套接字一定是成对出现的。一端的发送缓冲区对应对端的接收缓冲区。我们使用同一个文件描述符来描述发送缓冲区和接收缓冲区。

TCP/IP 协议最早在 BSD UNIX 上实现，为 TCP/IP 协议设计的应用层编程接口称为 socket API。本章的主要内容是 socket API，主要介绍 TCP 协议的函数接口，最后介绍 UDP 协议和 UNIX Domain Socket 的函数接口。



预备知识

网络字节序

我们已经知道，内存中的多字节数据相对于内存地址有大端和小端之分，磁盘文件中的多字节数据相对于文件中的偏移地址也有大端小端之分。网络数据流同样有大端小端之分，那么如何定义网络数据流的地址呢？发送主机通常将发送缓冲区中的数据按内存地址从低到高的顺序发出，接收主机把从网络上接到的字节依次保存在接收缓冲区中，也是按内存地址从低到高的顺序保存，因此，网络数据流的地址应这样规定：先发出的数据是低地址，后发出的数据是高地址。

TCP/IP 协议规定，**网络数据流应采用大端字节序**，即低地址高字节。例如上一节的 UDP 段格式，地址 0-1 是 16 位的源端口号，如果这个端口号是 1000 (0x3e8)，则地址 0 是 0x03，地址 1 是 0xe8，也就是先发 0x03，再发 0xe8，这 16 位在发送主机的缓冲区中也应该是低地址存 0x03，高地址存 0xe8。但是，如果发送主机是小端字节序的，这 16 位被解释成 0xe803，而不是 1000。因此，发送主机把 1000 填到发送缓冲区之前需要做字

字节序的转换。同样地，接收主机如果是小端字节序的，接到 16 位的源端口号也要做字节序的转换。如果主机是大端字节序的，发送和接收都不需要做转换。同理，32 位的 IP 地址也要考虑网络字节序和主机字节序的问题。

为使网络程序具有可移植性，使同样的 C 代码在大端和小端计算机上编译后都能正常运行，可以调用以下库函数做**网络字节序和主机字节序的转换**。

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

h 表示 host，n 表示 network，l 表示 32 位长整数，s 表示 16 位短整数。

如果主机是小端字节序，这些函数将参数做相应的大小端转换然后返回，如果主机是大端字节序，这些函数不做转换，将参数原封不动地返回。

IP 地址转换函数

早期：

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);
```

只能处理 IPv4 的 ip 地址

不可重入函数

注意参数是 struct in_addr

现在：

```
#include <arpa/inet.h>
int inet_pton(int af, const char *src, void *dst);
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

支持 IPv4 和 IPv6

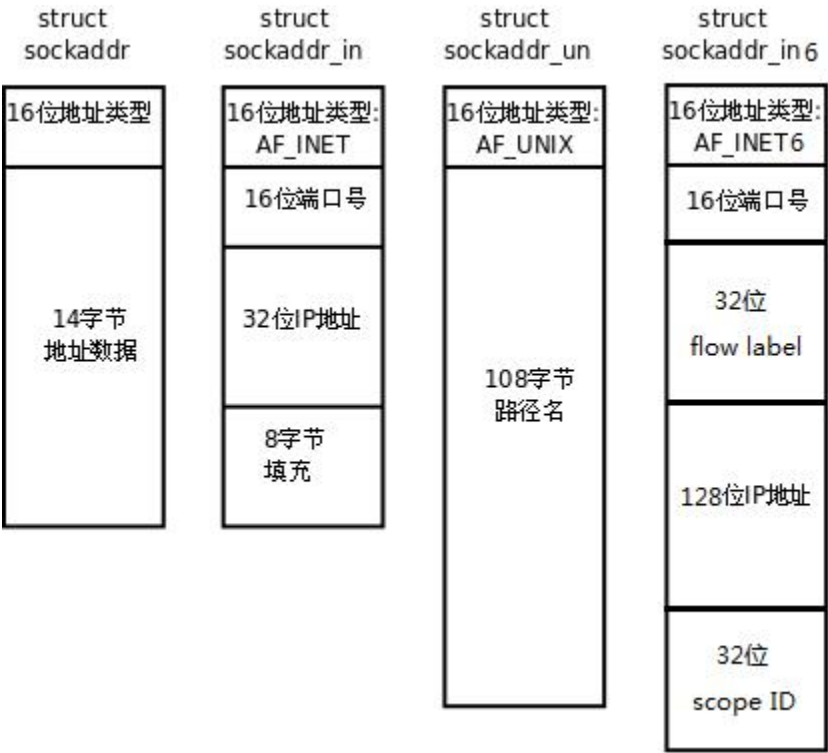
可重入函数

其中 `inet_pton` 和 `inet_ntop` 不仅可以转换 IPv4 的 `in_addr`，还可以转换 IPv6 的 `in6_addr`。

因此函数接口是 `void *addrptr`。

sockaddr 数据结构

`struct sockaddr` 很多网络编程函数诞生早于 IPv4 协议，那时候都使用的是 `sockaddr` 结构体，为了向前兼容，现在 `sockaddr` 退化成了（`void *`）的作用，传递一个地址给函数，至于这个函数是 `sockaddr_in` 还是 `sockaddr_in6`，由地址族确定，然后函数内部再强制类型转化为所需的地址类型。



sockaddr 数据结构

```
struct sockaddr {
    sa_family_t sa_family;    /* address family, AF_xxx */
    char sa_data[14];        /* 14 bytes of protocol address */
};
```

使用 `sudo grep -r "struct sockaddr_in {" /usr` 命令可查看到 `struct sockaddr_in` 结构体的定义。一般其默认的存储位置：`/usr/include/linux/in.h` 文件中。

```
struct sockaddr_in {
    __kernel_sa_family_t sin_family;    /* Address family */    地址结构类型
```

```

__be16 sin_port;                /* Port number */      端口号
struct in_addr sin_addr;        /* Internet address */  IP 地址
/* Pad to size of `struct sockaddr'. */
unsigned char __pad[__SOCK_SIZE__ - sizeof(short int) -
sizeof(unsigned short int) - sizeof(struct in_addr)];
};

```

```

struct in_addr {                /* Internet address. */
    __be32 s_addr;
};

```

```

struct sockaddr_in6 {
    unsigned short int sin6_family; /* AF_INET6 */
    __be16 sin6_port;              /* Transport layer port # */
    __be32 sin6_flowinfo;          /* IPv6 flow information */
    struct in6_addr sin6_addr;      /* IPv6 address */
    __u32 sin6_scope_id;           /* scope id (new in RFC2553) */
};

```

```

struct in6_addr {
    union {
        __u8 u6_addr8[16];
        __be16 u6_addr16[8];
        __be32 u6_addr32[4];
    } in6_u;
#define s6_addr      in6_u.u6_addr8
#define s6_addr16    in6_u.u6_addr16
#define s6_addr32     in6_u.u6_addr32
};

```

```

#define UNIX_PATH_MAX 108
struct sockaddr_un {
    __kernel_sa_family_t sun_family; /* AF_UNIX */
    char sun_path[UNIX_PATH_MAX];    /* pathname */
};

```

Pv4 和 IPv6 的地址格式定义在 `netinet/in.h` 中，IPv4 地址用 `sockaddr_in` 结构体表示，包括 16 位端口号和 32 位 IP 地址，IPv6 地址用 `sockaddr_in6` 结构体表示，包括 16 位端口号、128 位 IP 地址和一些控制字段。

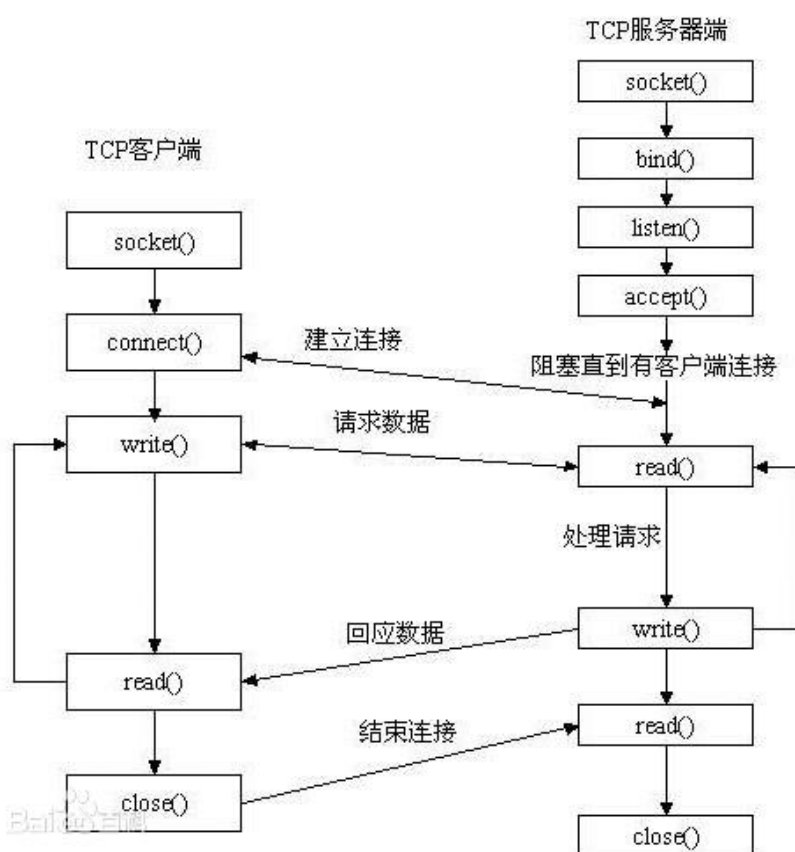
UNIX Domain Socket 的地址格式定义在 `sys/un.h` 中，用 `sock-addr_un` 结构体表示。各种 socket 地址结构体的开头都是相同的，前 16 位表示整个结构体的长度（并不是所有 UNIX 的实现都有长度字段，如 Linux 就没有），后 16 位表示地址类型。IPv4、IPv6 和 Unix Domain Socket 的地址类型分别定义为常数 `AF_INET`、`AF_INET6`、`AF_UNIX`。这样，只要取得某种 `sockaddr` 结构体的首地址，不需要知道具体是哪种类型的 `sockaddr` 结构体，就

可以根据地址类型字段确定结构体中的内容。因此，socket API 可以接受各种类型的 sockaddr 结构体指针做参数，例如 bind、accept、connect 等函数，这些函数的参数应该设计成 void *类型以便接受各种类型的指针，但是 sock API 的实现早于 ANSI C 标准化，那时还没有 void *类型，因此这些函数的参数都用 struct sockaddr *类型表示，在传递参数之前要强制类型转换一下，例如：

```
struct sockaddr_in servaddr;  
bind(listen_fd, (struct sockaddr *)&servaddr, sizeof(servaddr));    /* initialize servaddr */
```

网络套接字函数

socket 模型创建流程图



socket API

socket 函数

```
#include <sys/types.h> /* See NOTES */  
#include <sys/socket.h>  
int socket(int domain, int type, int protocol);  
domain:
```

AF_INET 这是大多数用来产生 socket 的协议，使用 TCP 或 UDP 来传输，用 IPv4 的地址

AF_INET6 与上面类似，不过是用 IPv6 的地址

AF_UNIX 本地协议，使用在 Unix 和 Linux 系统上，一般都是当客户端和服务端在同一台及其上的时候使用

type:

SOCK_STREAM 这个协议是按照顺序的、可靠的、数据完整的基于字节流的连接。这是一个使用最多的 socket 类型，这个 socket 是使用 TCP 来进行传输。

SOCK_DGRAM 这个协议是无连接的、固定长度的传输调用。该协议是不可靠的，使用 UDP 来进行它的连接。

SOCK_SEQPACKET 该协议是双线路的、可靠的连接，发送固定长度的数据包进行传输。必须把这个包完整的接受才能进行读取。

SOCK_RAW socket 类型提供单一的网络访问，这个 socket 类型使用 ICMP 公共协议。（ping、traceroute 使用该协议）

SOCK_RDM 这个类型是很少使用的，在大部分的操作系统上没有实现，它是提供给数据链路层使用，不保证数据包的顺序

protocol:

传 0 表示使用默认协议。

返回值：

成功：返回指向新创建的 socket 的文件描述符，失败：返回-1，设置 errno

socket()打开一个网络通讯端口，如果成功的话，就像 open()一样返回一个文件描述符，应用程序可以像读写文件一样用 read/write 在网络上收发数据，如果 socket()调用出错则返回-1。对于 IPv4，domain 参数指定为 AF_INET。对于 TCP 协议，type 参数指定为 SOCK_STREAM，表示面向流的传输协议。如果是 UDP 协议，则 type 参数指定为 SOCK_DGRAM，表示面向数据报的传输协议。protocol 参数的介绍从略，指定为 0 即可。

bind 函数

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

sockfd：

socket 文件描述符

addr:

构造出 IP 地址加端口号

addrlen:

sizeof(addr)长度

返回值：

成功返回 0，失败返回-1，设置 errno

服务器程序所监听的网络地址和端口号通常是固定不变的，客户端程序得知服务器程序的地址和端口号后就可以向服务器发起连接，因此服务器需要调用 bind 绑定一个固定的网络地址和端口号。

bind()的作用是将参数 sockfd 和 addr 绑定在一起，使 sockfd 这个用于网络通讯的文件描述符监听 addr 所描述的地址和端口号。前面讲过，struct sockaddr *是一个通用指针类型，addr 参数实际上可以接受多种协议的 sockaddr 结构体，而它们的长度各不相同，所以需要第三个参数 addrlen 指定结构体的长度。如：

```
struct sockaddr_in servaddr;  
bzero(&servaddr, sizeof(servaddr));  
servaddr.sin_family = AF_INET;  
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);  
servaddr.sin_port = htons(6666);
```

首先将整个结构体清零，然后设置地址类型为 AF_INET，网络地址为 **INADDR_ANY**，这个宏表示本地的任意 IP 地址，因为服务器可能有多个网卡，每个网卡也可能绑定多个 IP 地址，这样设置可以在所有的 IP 地址上监听，直到与某个客户端建立了连接时才确定下来到底用哪个 IP 地址，端口号为 6666。

listen 函数

```
#include <sys/types.h> /* See NOTES */  
#include <sys/socket.h>  
int listen(int sockfd, int backlog);  
sockfd:  
    socket 文件描述符  
backlog:  
    排队建立 3 次握手队列和刚刚建立 3 次握手队列的连接数和
```

查看系统默认 backlog

```
cat /proc/sys/net/ipv4/tcp_max_syn_backlog
```

典型的服务器程序可以同时服务于多个客户端，当有客户端发起连接时，服务器调用的 accept() 返回并接受这个连接，如果有大量的客户端发起连接而服务器来不及处理，尚未 accept 的客户端就处于连接等待状态，listen() 声明 sockfd 处于监听状态，并且最多允许有 backlog 个客户端处于连接等待状态，如果接收到更多的连接请求就忽略。listen() 成功返回 0，失败返回 -1。

accept 函数

```
#include <sys/types.h> /* See NOTES */  
#include <sys/socket.h>  
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);  
sockfd:  
    socket 文件描述符  
addr:  
    传出参数，返回链接客户端地址信息，含 IP 地址和端口号  
addrlen:  
    传入传出参数（值-结果），传入 sizeof(addr) 大小，函数返回时返回真正接收到地址结构体的大小  
返回值：  
    成功返回一个新的 socket 文件描述符，用于和客户端通信，失败返回 -1，设置 errno
```

三方握手完成后，服务器调用 `accept()` 接受连接，如果服务器调用 `accept()` 时还没有客户端的连接请求，就阻塞等待直到有客户端连接上来。`addr` 是一个传出参数，`accept()` 返回时传出客户端的地址和端口号。`addrlen` 参数是一个传入传出参数（value-result argument），传入的是调用者提供的缓冲区 `addr` 的长度以避免缓冲区溢出问题，传出的是客户端地址结构体的实际长度（有可能没有占满调用者提供的缓冲区）。如果给 `addr` 参数传 `NULL`，表示不关心客户端的地址。

我们的服务器程序结构是这样的：

```
while (1) {
    cliaddr_len = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
    n = read(connfd, buf, MAXLINE);
    .....
    close(connfd);
}
```

整个是一个 `while` 死循环，每次循环处理一个客户端连接。由于 `cliaddr_len` 是传入传出参数，每次调用 `accept()` 之前应该重新赋初值。`accept()` 的参数 `listenfd` 是先前的监听文件描述符，而 `accept()` 的返回值是另外一个文件描述符 `connfd`，之后与客户端之间就通过这个 `connfd` 通讯，最后关闭 `connfd` 断开连接，而不关闭 `listenfd`，再次回到循环开头 `listenfd` 仍然用作 `accept` 的参数。`accept()` 成功返回一个文件描述符，出错返回 -1。

connect 函数

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

`sockfd`:
socket 文件描述符

`addr`:
传入参数，指定服务器端地址信息，含 IP 地址和端口号

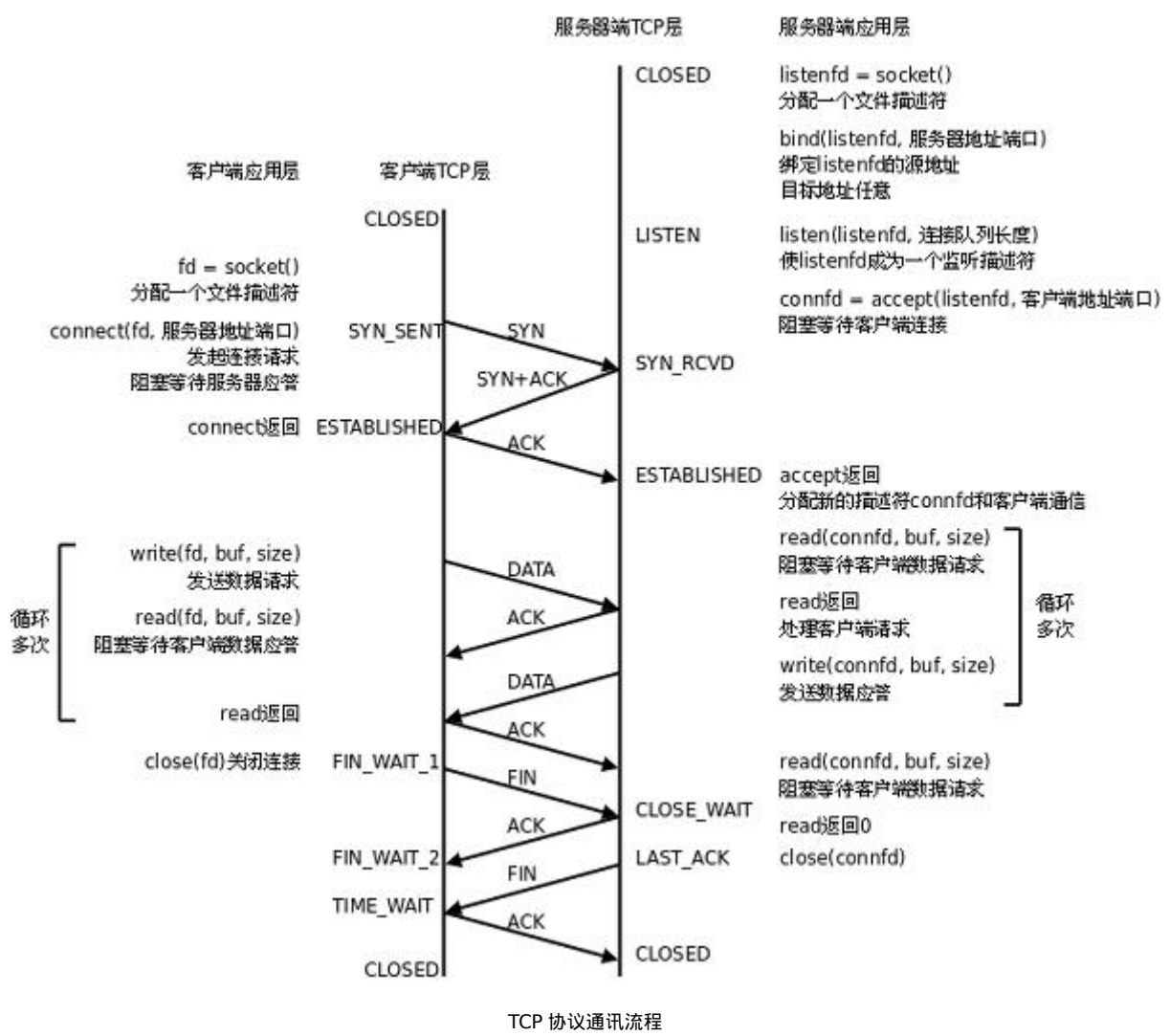
`addrlen`:
传入参数，传入 `sizeof(addr)` 大小

返回值：
成功返回 0，失败返回 -1，设置 `errno`

客户端需要调用 `connect()` 连接服务器，`connect` 和 `bind` 的参数形式一致，区别在于 `bind` 的参数是自己的地址，而 `connect` 的参数是对方的地址。`connect()` 成功返回 0，出错返回 -1。

C/S 模型-TCP

下图是基于 TCP 协议的客户端/服务器程序的一般流程：



服务器调用 `socket()`、`bind()`、`listen()`完成初始化后，调用 `accept()`阻塞等待，处于监听端口的状态，客户端调用 `socket()`初始化后，调用 `connect()`发出 `SYN` 段并阻塞等待服务器应答，服务器应答一个 `SYN-ACK` 段，客户端收到后从 `connect()`返回，同时应答一个 `ACK` 段，服务器收到后从 `accept()`返回。

数据传输的过程：

建立连接后，TCP 协议提供全双工的通信服务，但是一般的客户端/服务器程序的流程是由客户端主动发起请求，服务器被动处理请求，一问一答的方式。因此，服务器从 `accept()`返回后立刻调用 `read()`，读 `socket` 就像读管道一样，如果没有数据到达就阻塞等待，这时客户端调用 `write()`发送请求给服务器，服务器收到后从 `read()`返回，对客户端的请求进行处理，在此期间客户端调用 `read()`阻塞等待服务器的应答，服务器调用 `write()`将处理

结果发回给客户端，再次调用 read()阻塞等待下一条请求，客户端收到后从 read()返回，发送下一条请求，如此循环下去。

如果客户端没有更多的请求了，就调用 close()关闭连接，就像写端关闭的管道一样，服务器的 read()返回 0，这样服务器就知道客户端关闭了连接，也调用 close()关闭连接。注意，任何一方调用 close()后，连接的两个传输方向都关闭，不能再发送数据了。如果一方调用 shutdown()则连接处于半关闭状态，仍可接收对方发来的数据。

在学习 socket API 时要注意应用程序和 TCP 协议层是如何交互的：应用程序调用某个 socket 函数时 TCP 协议层完成什么动作，比如调用 connect()会发出 SYN 段 应用程序如何知道 TCP 协议层的状态变化，比如从某个阻塞的 socket 函数返回就表明 TCP 协议收到了某些段，再比如 read()返回 0 就表明收到了 FIN 段

server

下面通过最简单的客户端/服务器程序的实例来学习 socket API。

server.c 的作用是从客户端读字符，然后将每个字符转换为大写并回送给客户端。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <strings.h>
#include <string.h>
#include <ctype.h>
#include <arpa/inet.h>

#define SERV_PORT 9527

int main(void)
{
    int sfd, cfd;
    int len, i;
    char buf[BUFSIZ], clie_IP[BUFSIZ];

    struct sockaddr_in serv_addr, clie_addr;
    socklen_t clie_addr_len;

    /*创建一个 socket 指定 IPv4 协议族 TCP 协议*/
    sfd = socket(AF_INET, SOCK_STREAM, 0);
```

```

/*初始化一个地址结构 man 7 ip 查看对应信息*/
bzero(&serv_addr, sizeof(serv_addr));           //将整个结构体清零
serv_addr.sin_family = AF_INET;                 //选择协议族为 IPv4
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY); //监听本地所有 IP 地址
serv_addr.sin_port = htons(SERV_PORT);          //绑定端口号

/*绑定服务器地址结构*/
bind(sfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

/*设定链接上限,注意此处不阻塞*/
listen(sfd, 64);                                //同一时刻允许向服务器发起链接请求的数量

printf("wait for client connect ...\n");

/*获取客户端地址结构大小*/
clie_addr_len = sizeof(clie_addr_len);
/*参数 1 是 sfd; 参 2 传出参数, 参 3 传入参数, 全部是 client 端的参数*/
cfd = accept(sfd, (struct sockaddr *)&clie_addr, &clie_addr_len);           /*监听客户端链接, 会阻塞*/

printf("client IP:%s\tpport:%d\n",
       inet_ntop(AF_INET, &clie_addr.sin_addr.s_addr, clie_IP, sizeof(clie_IP)),
       ntohs(clie_addr.sin_port));

while (1) {
    /*读取客户端发送数据*/
    len = read(cfd, buf, sizeof(buf));
    write(STDOUT_FILENO, buf, len);

    /*处理客户端数据*/
    for (i = 0; i < len; i++)
        buf[i] = toupper(buf[i]);

    /*处理完数据回写给客户端*/
    write(cfd, buf, len);
}

/*关闭链接*/
close(sfd);
close(cfd);

return 0;
}
client

```

client.c 的作用是从命令行参数中获得一个字符串发给服务器，然后接收服务器返回的字符串并打印。

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>

```

```

#include <sys/socket.h>
#include <arpa/inet.h>

#define SERV_IP "127.0.0.1"
#define SERV_PORT 9527

int main(void)
{
    int sfd, len;
    struct sockaddr_in serv_addr;
    char buf[BUFSIZ];

    /*创建一个 socket 指定 IPv4 TCP*/
    sfd = socket(AF_INET, SOCK_STREAM, 0);

    /*初始化一个地址结构:*/
    bzero(&serv_addr, sizeof(serv_addr));           //清零
    serv_addr.sin_family = AF_INET;                 //IPv4 协议族
    inet_pton(AF_INET, SERV_IP, &serv_addr.sin_addr.s_addr); //指定 IP 字符串类型转换为网络字节序 参 3:传出参数
    serv_addr.sin_port = htons(SERV_PORT);          //指定端口 本地转网络字节序

    /*根据地址结构链接指定服务器进程*/
    connect(sfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    while (1) {
        /*从标准输入获取数据*/
        fgets(buf, sizeof(buf), stdin);
        /*将数据写给服务器*/
        write(sfd, buf, strlen(buf));                //写个服务器
        /*从服务器读回转换后数据*/
        len = read(sfd, buf, sizeof(buf));
        /*写至标准输出*/
        write(STDOUT_FILENO, buf, len);
    }

    /*关闭链接*/
    close(sfd);

    return 0;
}

```

编译的 Makefile 如下:

```

src = $(wildcard *.c)
targets = $(patsubst %.c, %, $(src))

CC = gcc
CFLAGS = -Wall -g

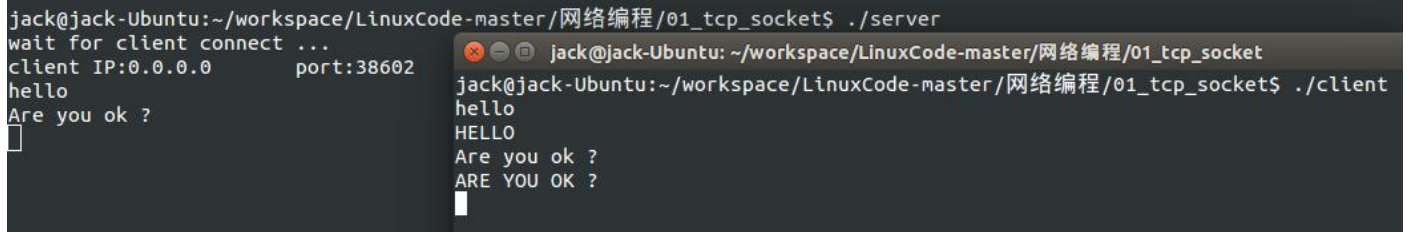
```

```
all:$(targets)

$(targets):%:%.c
    $(CC) $< -o $@ $(CFLAGS)

.PHONY:clean all
clean:
    -rm -rf $(targets)
```

编译运行结果如下:



The screenshot shows two terminal windows. The left window is running the server program, and the right window is running the client program. The server output is:
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/01_tcp_socket\$./server
wait for client connect ...
client IP:0.0.0.0 port:38602
hello
Are you ok ?
[blank line]
The right window output is:
jack@jack-Ubuntu: ~/workspace/LinuxCode-master/网络编程/01_tcp_socket
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/01_tcp_socket\$./client
hello
HELLO
Are you ok ?
ARE YOU OK ?
[blank line]

由于客户端不需要固定的端口号，因此不必调用 `bind()`，客户端的端口号由内核自动分配。注意，客户端不是不允许调用 `bind()`，只是没有必要调用 `bind()` 固定一个端口号，服务器也不是必须调用 `bind()`，但如果服务器不调用 `bind()`，内核会自动给服务器分配监听端口，每次启动服务器时端口号都不一样，客户端要连接服务器就会遇到麻烦。

客户端和服务端启动后可以使用 `netstat` 命令查看链接情况：

```
netstat -apn|grep 6666
```

出错处理封装函数

上面的例子不仅功能简单，而且简单到几乎没有什么错误处理，我们知道，系统调用不能保证每次都成功，必须进行出错处理，这样一方面可以保证程序逻辑正常，另一方面可以迅速得到故障信息。

为使错误处理的代码不影响主程序的可读性，我们把与 `socket` 相关的一些系统函数加上错误处理代码包装成新的函数，做成一个模块 `wrap.c`：

wrap.c

```
#include "wrap.h"

void perr_exit(const char *s)
{
```

```

    perror(s);
    exit(-1);
}

int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr)
{
    int n;

again:
    if ((n = accept(fd, sa, salenptr)) < 0) {
        if ((errno == ECONNABORTED) || (errno == EINTR))
            goto again;
        else
            perr_exit("accept error");
    }
    return n;
}

int Bind(int fd, const struct sockaddr *sa, socklen_t salen)
{
    int n;

    if ((n = bind(fd, sa, salen)) < 0)
        perr_exit("bind error");

    return n;
}

int Connect(int fd, const struct sockaddr *sa, socklen_t salen)
{
    int n;
    n = connect(fd, sa, salen);
    if (n < 0) {
        perr_exit("connect error");
    }

    return n;
}

int Listen(int fd, int backlog)
{
    int n;

    if ((n = listen(fd, backlog)) < 0)
        perr_exit("listen error");

    return n;
}

```

```

int Socket(int family, int type, int protocol)
{
    int n;

    if ((n = socket(family, type, protocol)) < 0)
        perr_exit("socket error");

    return n;
}

ssize_t Read(int fd, void *ptr, size_t nbytes)
{
    ssize_t n;

again:
    if ( (n = read(fd, ptr, nbytes)) == -1) {
        if (errno == EINTR)
            goto again;
        else
            return -1;
    }

    return n;
}

ssize_t Write(int fd, const void *ptr, size_t nbytes)
{
    ssize_t n;

again:
    if ((n = write(fd, ptr, nbytes)) == -1) {
        if (errno == EINTR)
            goto again;
        else
            return -1;
    }

    return n;
}

int Close(int fd)
{
    int n;

    if ((n = close(fd)) == -1)
        perr_exit("close error");

    return n;
}

```

/*参三: 应该读取的字节数*/

//socket 4096 readn(cfd, buf, 4096) nleft = 4096-1500

```

ssize_t Readn(int fd, void *vptr, size_t n)
{
    size_t  nleft;           //unsigned int  剩余未读取的字节数
    ssize_t nread;           //int  实际读到的字节数
    char    *ptr;

    ptr = vptr;
    nleft = n;               //n  未读取字节数

    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0;
            else
                return -1;
        } else if (nread == 0)
            break;

        nleft -= nread;      //nleft = nleft - nread
        ptr += nread;
    }
    return n - nleft;
}

```

```

ssize_t Writen(int fd, const void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nwritten;
    const char *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
            if (nwritten < 0 && errno == EINTR)
                nwritten = 0;
            else
                return -1;
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return n;
}

```

```

ssize_t my_read(int fd, char *ptr)
{
    static int read_cnt;
    static char *read_ptr;

```



```

static char read_buf[100];

if (read_cnt <= 0) {
again:
    if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {    //"hello\n"
        if (errno == EINTR)
            goto again;
        return -1;
    } else if (read_cnt == 0)
        return 0;

    read_ptr = read_buf;
}
read_cnt--;
*ptr = *read_ptr++;

return 1;
}

/*readline --- fgets*/
//传出参数 vptr
ssize_t Readline(int fd, void *vptr, size_t maxlen)
{
    ssize_t n, rc;
    char    c, *ptr;
    ptr = vptr;

    for (n = 1; n < maxlen; n++) {
        if ((rc = my_read(fd, &c)) == 1) {    //ptr[] = hello\n
            *ptr++ = c;
            if (c == '\n')
                break;
        } else if (rc == 0) {
            *ptr = 0;
            return n-1;
        } else
            return -1;
    }
    *ptr = 0;

    return n;
}

```

wrap.h

```

#ifndef __WRAP_H_
#define __WRAP_H_

#include <stdlib.h>
#include <stdio.h>

```

```

#include <unistd.h>
#include <errno.h>
#include <sys/socket.h>

void perr_exit(const char *s);
int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr);
int Bind(int fd, const struct sockaddr *sa, socklen_t salen);
int Connect(int fd, const struct sockaddr *sa, socklen_t salen);
int Listen(int fd, int backlog);
int Socket(int family, int type, int protocol);
ssize_t Read(int fd, void *ptr, size_t nbytes);
ssize_t Write(int fd, const void *ptr, size_t nbytes);
int Close(int fd);
ssize_t Readn(int fd, void *vptr, size_t n);
ssize_t Writen(int fd, const void *vptr, size_t n);
ssize_t my_read(int fd, char *ptr);
ssize_t Readline(int fd, void *vptr, size_t maxlen);

#endif

```

使用案例：

Server.c:

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <strings.h>
#include <string.h>
#include <ctype.h>
#include <arpa/inet.h>

#include "wrap.h"

#define SERV_PORT 6666

int main(void)
{
    int sfd, cfd;
    int len, i;
    char buf[BUFSIZ], clie_IP[BUFSIZ];

    struct sockaddr_in serv_addr, clie_addr;
    socklen_t clie_addr_len;

    sfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;

```

```

serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(SERV_PORT);

Bind(sfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

Listen(sfd, 2);

printf("wait for client connect ...\n");

clie_addr_len = sizeof(clie_addr_len);
cfd = Accept(sfd, (struct sockaddr *)&clie_addr, &clie_addr_len);
printf("cfd = ----%d\n", cfd);

printf("client IP: %s  port:%d\n",
       inet_ntop(AF_INET, &clie_addr.sin_addr.s_addr, clie_IP, sizeof(clie_IP)),
       ntohs(clie_addr.sin_port));

while (1) {
    len = Read(cfd, buf, sizeof(buf));
    Write(STDOUT_FILENO, buf, len);

    for (i = 0; i < len; i++)
        buf[i] = toupper(buf[i]);
    Write(cfd, buf, len);
}

Close(sfd);
Close(cfd);

return 0;
}

```

Client.c

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#include "wrap.h"

#define SERV_IP "127.0.0.1"
#define SERV_PORT 6666

int main(void)
{
    int sfd, len;
    struct sockaddr_in serv_addr;
    char buf[BUFSIZ];

```

```

sfd = Socket(AF_INET, SOCK_STREAM, 0);

bzero(&serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
inet_pton(AF_INET, SERV_IP, &serv_addr.sin_addr.s_addr);
serv_addr.sin_port = htons(SERV_PORT);

Connect(sfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

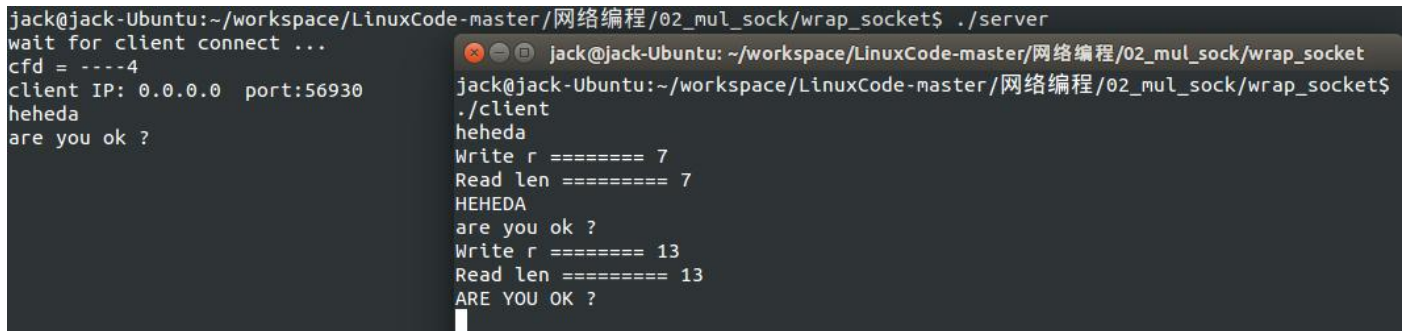
while (1) {
    fgets(buf, sizeof(buf), stdin);
    int r = Write(sfd, buf, strlen(buf));
    printf("Write r ===== %d\n", r);
    len = Read(sfd, buf, sizeof(buf));
    printf("Read len ===== %d\n", len);
    Write(STDOUT_FILENO, buf, len);
}

Close(sfd);

return 0;
}

```

编译运行结果如下:



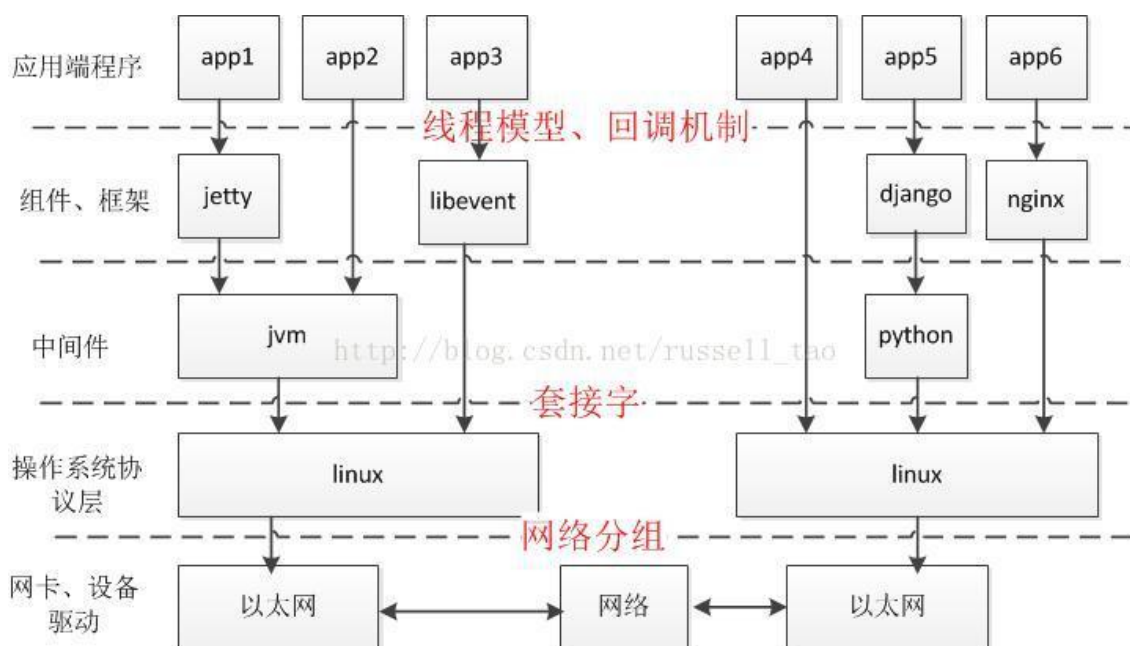
```

jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/02_mul_sock/wrap_socket$ ./server
wait for client connect ...
cfd = ----4
client IP: 0.0.0.0 port:56930
heheda
are you ok ?

jack@jack-Ubuntu: ~/workspace/LinuxCode-master/网络编程/02_mul_sock/wrap_socket
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/02_mul_sock/wrap_socket$ ./client
heheda
Write r ===== 7
Read len ===== 7
HEHEDA
are you ok ?
Write r ===== 13
Read len ===== 13
ARE YOU OK ?

```

高并发服务器



多进程并发服务器

使用多进程并发服务器时要考虑以下几点：

1. 父进程最大文件描述个数(父进程中需要 close 关闭 accept 返回的新文件描述符)
2. 系统内创建进程个数(与内存大小相关)
3. 进程创建过多是否降低整体服务性能(进程调度)

server

```
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <sys/wait.h>
#include <ctype.h>
#include <unistd.h>

#include "wrap.h"

#define MAXLINE 8192
```

```

#define SERV_PORT 8000

void do_sigchild(int num)
{
    while (waitpid(0, NULL, WNOHANG) > 0)
        ;
}

int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    int i, n;
    pid_t pid;
    struct sigaction newact;

    newact.sa_handler = do_sigchild;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGCHLD, &newact, NULL);

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    int opt = 1;
    setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    Listen(listenfd, 20);

    printf("Accepting connections ...\n");
    while (1) {
        cliaddr_len = sizeof(cliaddr);
        connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
        pid = fork();
        if (pid == 0) {
            Close(listenfd);
            while (1) {
                n = Read(connfd, buf, MAXLINE);
                if (n == 0) {
                    printf("the other side has been closed.\n");
                }
            }
        }
    }
}

```

```

        break;
    }
    printf("received from %s at PORT %d\n",
        inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
        ntohs(cliaddr.sin_port));

    for (i = 0; i < n; i++)
        buf[i] = toupper(buf[i]);

    Write(STDOUT_FILENO, buf, n);
    Write(connfd, buf, n);
}
Close(connfd);
return 0;
} else if (pid > 0) {
    Close(connfd);
} else
    perr_exit("fork");
}
return 0;
}

```

client

```

/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "wrap.h"

#define MAXLINE 8192
#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

```



```

Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

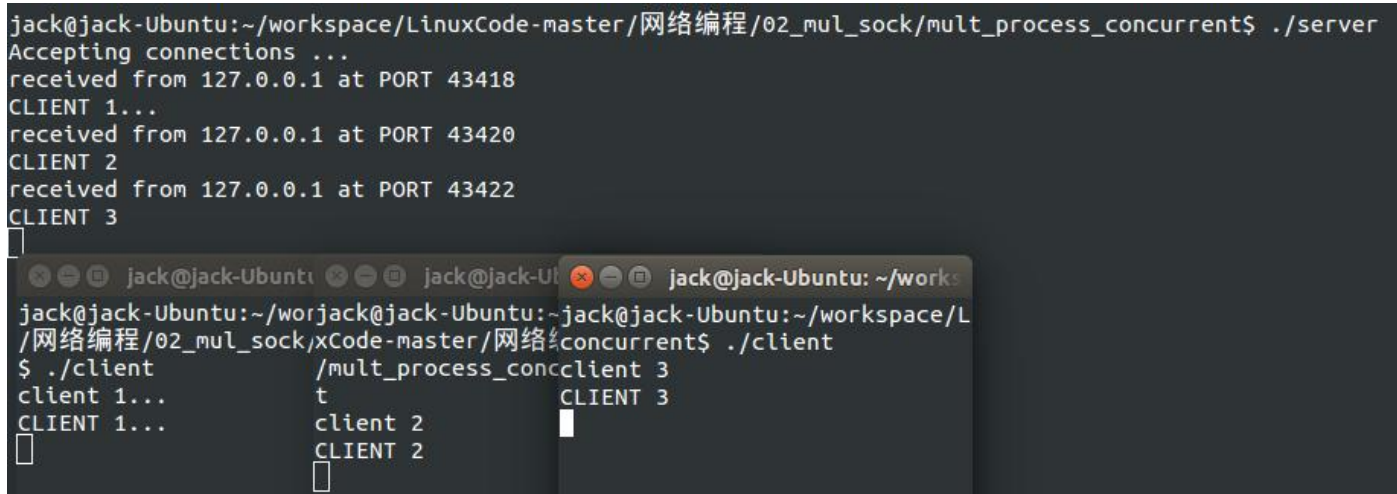
while (fgets(buf, MAXLINE, stdin) != NULL) {
    Write(sockfd, buf, strlen(buf));
    n = Read(sockfd, buf, MAXLINE);
    if (n == 0) {
        printf("the other side has been closed.\n");
        break;
    }
    else
        Write(STDOUT_FILENO, buf, n);
}

Close(sockfd);

return 0;
}

```

编译运行结果如下：



The image shows a terminal window with three panes. The top pane shows the server program output: 'Accepting connections ...', 'received from 127.0.0.1 at PORT 43418', 'CLIENT 1...', 'received from 127.0.0.1 at PORT 43420', 'CLIENT 2', 'received from 127.0.0.1 at PORT 43422', and 'CLIENT 3'. The bottom-left pane shows the client program output: 'client 1...', 'CLIENT 1...', and 'CLIENT 2'. The bottom-right pane shows the client program output: 'client 3' and 'CLIENT 3'.

```

jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/02_mul_sock/mult_process_concurrent$ ./server
Accepting connections ...
received from 127.0.0.1 at PORT 43418
CLIENT 1...
received from 127.0.0.1 at PORT 43420
CLIENT 2
received from 127.0.0.1 at PORT 43422
CLIENT 3
[ ]

jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/02_mul_sock$ ./client
client 1...
CLIENT 1...
[ ]

jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/02_mul_sock$ ./client
client 3
CLIENT 3
[ ]

```

多线程并发服务器

在使用线程模型开发服务器时需考虑以下问题：

1. 调整进程内最大文件描述符上限
2. 线程如有共享数据，考虑线程同步
3. 服务于客户端线程退出时，退出处理。（退出值，分离态）
4. 系统负载，随着链接客户端增加，导致其它线程不能及时得到 CPU

server

```
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>

#include "wrap.h"

#define MAXLINE 8192
#define SERV_PORT 8000

struct s_info {                                //定义一个结构体，将地址结构跟 cfd 捆绑
    struct sockaddr_in cliaddr;
    int connfd;
};

void *do_work(void *arg)
{
    int n,i;
    struct s_info *ts = (struct s_info*)arg;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];    //define INET_ADDRSTRLEN 16 可用"+d"查看

    while (1) {
        n = Read(ts->connfd, buf, MAXLINE);        //读客户端
        if (n == 0) {
            printf("the client %d closed...\n", ts->connfd);
            break;                                //跳出循环,关闭 cfd
        }
        printf("received from %s at PORT %d\n",
            inet_ntop(AF_INET, &(ts->cliaddr.sin_addr), str, sizeof(str)),
            ntohs((ts->cliaddr.sin_port)));        //打印客户端信息(IP/PORT)

        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);            //小写-->大写

        Write(STDOUT_FILENO, buf, n);            //写出至屏幕
        Write(ts->connfd, buf, n);                //回写给客户端
    }
    Close(ts->connfd);

    return (void *)0;
}
```

```

int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    pthread_t tid;
    struct s_info ts[256];    //根据最大线程数创建结构体数组.
    int i = 0;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);    //创建一个 socket, 得到 lfd

    bzero(&servaddr, sizeof(servaddr));    //地址结构清零
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);    //指定本地任意 IP
    servaddr.sin_port = htons(SERV_PORT);    //指定端口号 8000

    Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)); //绑定

    Listen(listenfd, 128);    //设置同一时刻链接服务器上限数

    printf("Accepting client connect ...\n");

    while (1) {
        cliaddr_len = sizeof(cliaddr);
        connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);    //阻塞监听客户端链接请求
        ts[i].cliaddr = cliaddr;
        ts[i].connfd = connfd;

        /* 达到线程最大数时 , pthread_create 出错处理, 增加服务器稳定性 */
        pthread_create(&tid, NULL, do_work, (void*)&ts[i]);
        pthread_detach(tid);    //子线程分离,防止僵线程产生.
        i++;
    }

    return 0;
}

```

client

```

/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "wrap.h"

#define MAXLINE 80

```

```

#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;
    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr.s_addr);
    servaddr.sin_port = htons(SERV_PORT);

    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        Write(sockfd, buf, strlen(buf));
        n = Read(sockfd, buf, MAXLINE);
        if (n == 0)
            printf("the other side has been closed.\n");
        else
            Write(STDOUT_FILENO, buf, n);
    }

    Close(sockfd);

    return 0;
}

```

Makefile

```

src = $(wildcard *.c)
obj = $(patsubst %.c, %.o, $(src))

all: server client

server: server.o wrap.o
    gcc server.o wrap.o -o server -Wall -lpthread
client: client.o wrap.o
    gcc client.o wrap.o -o client -Wall -lpthread

%.o:%.c
    gcc -c $< -Wall

.PHONY: clean all
clean:
    -rm -rf server client $(obj)

```

编译运行结果如下:

```
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/02_mul_sock/mult_thread_concurrent$ ./server
Accepting client connect ...
received from 127.0.0.1 at PORT 43444
CLIEN1 HEHEDA
received from 127.0.0.1 at PORT 43446
CLIENT2 ARE YOU OK ?
received from 127.0.0.1 at PORT 43448
CLIENT3 I AM FAN .
[]

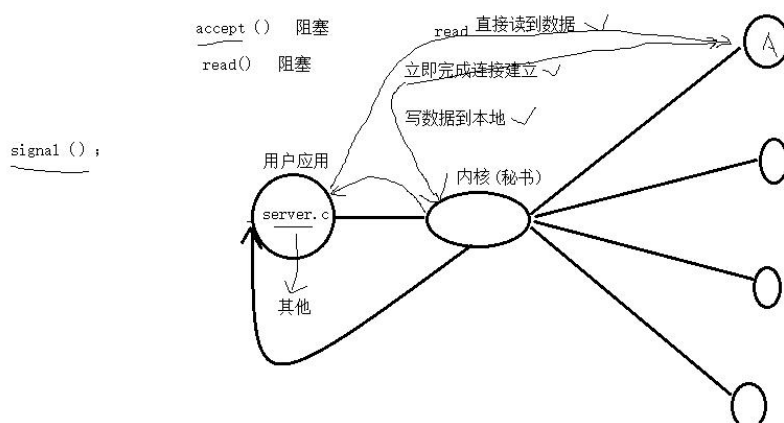
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/02_mul_sock/mult_thread_concurrent$ ./client
cli1 heheda
CLIEN1 HEHEDA
[]

jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/02_mul_sock/mult_thread_concurrent$ ./client
client2 are you ok ?
CLIENT2 ARE YOU OK ?
[]

jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/02_mul_sock/mult_thread_concurrent$ ./client
client3 i am fan .
CLIENT3 I AM FAN .
[]
```

多路 I/O 转接服务器

多路 IO 转接服务器也叫做多任务 IO 服务器。该类服务器实现的主旨思想是，不再由应用程序自己监视客户端连接，取而代之由内核替应用程序监视文件。



主要使用的方法有三种

select

1. select 能监听的文件描述符个数受限于 FD_SETSIZE, 一般为 1024, 单纯改变进程打开的文件描述符个数并不能改变 select 监听文件个数
2. 解决 1024 以下客户端时使用 select 是很合适的, 但如果链接客户端过多, select 采用的是轮询模型, 会

大大降低服务器响应效率，不应在 select 上投入更多精力

参1: 所监听的所有文件描述符中, 最大的文件描述符+1 fd4+1

参2/3/4: 所监听的文件描述符 “可读” 事件 readfds();

所监听的文件描述符 “可读” 事件 writefds();

所监听的文件描述符 “异常” 事件

返回: 成功: 所监听的所有的 监听集合中, 满足条件的总数。

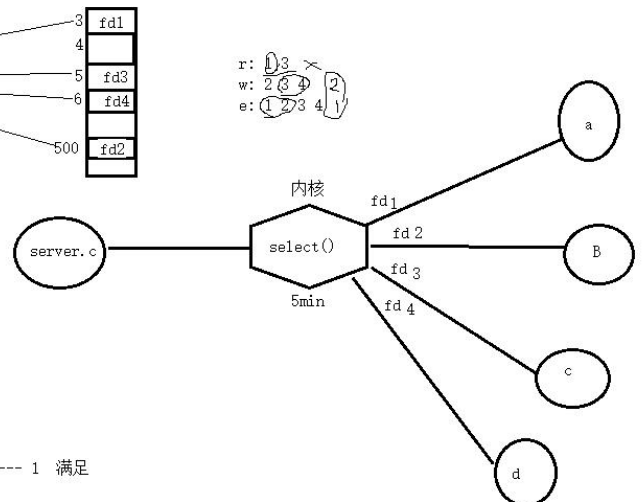
失败:

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

将set清空 0
将fd 从 set中清除出去。
判断 fd 是否在集合中。
将fd 设置到 set集合中去。

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET (fd1, &readfds);
FD_SET (fd2, &readfds);
FD_SET (fd3, &readfds);
select (); -----> 总数
```

```
for ();
FD_ISSET(fd1,&readfds) --- 1 满足
```



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

```
/* According to earlier standards */
```

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

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

```
#include <unistd.h>
```

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

nfds: 监控的文件描述符集里最大文件描述符加 1, 因为此参数会告诉内核检测前多少个文件描述符的状态

readfds: 监控有读数据到达文件描述符集合, 传入传出参数

writefds: 监控写数据到达文件描述符集合, 传入传出参数

exceptfds: 监控异常发生达文件描述符集合,如带外数据到达异常, 传入传出参数

timeout: 定时阻塞监控时间, 3 种情况

1.NULL, 永远等下去

2.设置 timeval, 等待固定时间

3.设置 timeval 里时间均为 0, 检查描述字后立即返回, 轮询

```
struct timeval {
```

```
    long tv_sec; /* seconds */
```

```
    long tv_usec; /* microseconds */
```

```
};
```

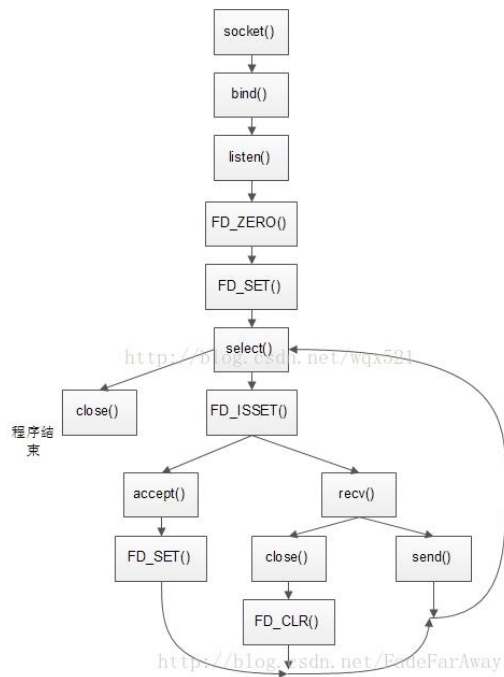
```
void FD_CLR(int fd, fd_set *set); //把文件描述符集合里 fd 清 0
```

```
int FD_ISSET(int fd, fd_set *set); //测试文件描述符集合里 fd 是否置 1
```

```
void FD_SET(int fd, fd_set *set); //把文件描述符集合里 fd 位置 1
```

```
void FD_ZERO(fd_set *set); //把文件描述符集合里所有位清 0
```

select 函数原理如下:



server

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <ctype.h>

#include "wrap.h"

#define SERV_PORT 6666

int main(int argc, char *argv[])
{
    int i, j, n, maxi;

    int nready, client[FD_SETSIZE];          /* 自定义数组 client, 防止遍历 1024 个文件描述符 FD_SETSIZE 默认为 1024 */
    int maxfd, listenfd, connfd, sockfd;
    char buf[BUFSIZ], str[INET_ADDRSTRLEN]; /* #define INET_ADDRSTRLEN 16 */

    struct sockaddr_in clie_addr, serv_addr;
    socklen_t clie_addr_len;
    fd_set rset, allset;                    /* rset 读事件文件描述符集合 allset 用来暂存 */

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    int opt = 1;
    setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

```



```

bzero(&serv_addr, sizeof(serv_addr));
serv_addr.sin_family= AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port= htons(SERV_PORT);

Bind(listenfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
Listen(listenfd, 128);

maxfd = listenfd;                                /* 起初 listenfd 即为最大文件描述符 */

maxi = -1;                                        /* 将来用作 client[]的下标, 初始值指向 0 个元素之前下标
位置 */
for (i = 0; i < FD_SETSIZE; i++)
    client[i] = -1;                                /* 用-1 初始化 client[] */

FD_ZERO(&allset);
FD_SET(listenfd, &allset);                        /* 构造 select 监控文件描述符集 */

while (1) {
    rset = allset;                                /* 每次循环时都从新设置 select 监控信号集 */
    nready = select(maxfd+1, &rset, NULL, NULL, NULL);//当有客户端连接时将返回连接个数,否则阻塞
    if (nready < 0)
        perr_exit("select error");

    if (FD_ISSET(listenfd, &rset)) {                /* 说明有新的客户端链接请求 */

        clie_addr_len = sizeof(clie_addr);
        connfd = Accept(listenfd, (struct sockaddr *)&clie_addr, &clie_addr_len);    /* Accept 不会阻塞 */
        printf("received from %s at PORT %d\n",
            inet_ntop(AF_INET, &clie_addr.sin_addr, str, sizeof(str)),
            ntohs(clie_addr.sin_port));

        for (i = 0; i < FD_SETSIZE; i++)
            if (client[i] < 0) {                    /* 找 client[]中没有使用的位置 */
                client[i] = connfd;                /* 保存 accept 返回的文件描述符到 client[]里 */
                break;
            }

        if (i == FD_SETSIZE) {                    /* 达到 select 能监控的文件个数上限 1024 */
            fputs("too many clients\n", stderr);
            exit(1);
        }

        FD_SET(connfd, &allset);                    /* 向监控文件描述符集合 allset 添加新的文件描述符
connfd */
        if (connfd > maxfd)
            maxfd = connfd;                        /* select 第一个参数需要 */
    }
}

```

```

        if (i > maxi)
            maxi = i;                                /* 保证 maxi 存的总是 client[]最后一个元素下标 */

        if (--nready == 0)
            continue;
    }

    for (i = 0; i <= maxi; i++) {                    /* 检测哪个 clients 有数据就绪 */

        if ((sockfd = client[i]) < 0)
            continue;
        if (FD_ISSET(sockfd, &rset)) {

            if ((n = Read(sockfd, buf, sizeof(buf))) == 0) { /* 当 client 关闭链接时,服务器端也关闭对应链接 */
                Close(sockfd);
                FD_CLR(sockfd, &allset);                /* 解除 select 对此文件描述符的监控 */
                client[i] = -1;
            } else if (n > 0) {
                for (j = 0; j < n; j++)
                    buf[j] = toupper(buf[j]);
                Write(sockfd, buf, n);
                Write(STDOUT_FILENO, buf, n);
            }
            if (--nready == 0)
                break;                                /* 跳出 for, 但还在 while 中 */
        }
    }
}
Close(listenfd);
return 0;
}

```

client

```

/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 6666

int main(int argc, char *argv[])
{

```

```

struct sockaddr_in servaddr;
char buf[MAXLINE];
int sockfd, n;

if (argc != 2)
    printf("./client IP\n");

sockfd = Socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
servaddr.sin_port = htons(SERV_PORT);

Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
printf("-----connect ok-----\n");

while (fgets(buf, MAXLINE, stdin) != NULL) {
    Write(sockfd, buf, strlen(buf));
    n = Read(sockfd, buf, MAXLINE);
    if (n == 0) {
        printf("the other side has been closed.\n");
        break;
    }
    else
        Write(STDOUT_FILENO, buf, n);
}
Close(sockfd);

return 0;
}

```

Makefile

```

src = $(wildcard *.c)
obj = $(patsubst %.c, %.o, $(src))

all: server client

server: server.o wrap.o
    gcc server.o wrap.o -o server -Wall
client: client.o wrap.o
    gcc client.o wrap.o -o client -Wall

%.o:%.c
    gcc -c $< -Wall

.PHONY: clean all

```

clean:

```
-rm -rf server client ${obj}
```

pselect

pselect 原型如下。此模型应用较少，有需要的同学可参考 select 模型自行编写 C/S

```
#include <sys/select.h>
int pselect(int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, const struct timespec *timeout,
            const sigset_t *sigmask);
struct timespec {
    long tv_sec; /* seconds */
    long tv_nsec; /* nanoseconds */
};
用 sigmask 替代当前进程的阻塞信号集，调用返回后还原原有阻塞信号集
```

Poll

poll 的机制与 select 类似，与 select 在本质上没有多大差别，管理多个描述符也是进行轮询，根据描述符的状态进行处理，但是 poll 没有最大文件描述符数量的限制。poll 和 select 同样存在一个缺点就是，包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间，而不论这些文件描述符是否就绪，它的开销随着文件描述符数量的增加而线性增大。

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
struct pollfd {
    int fd; /* 文件描述符 */
    short events; /* 监控的事件 */
    short revents; /* 监控事件中满足条件返回的事件 */
};
```

POLLIN	普通或带外优先数据可读,即 POLLRDNORM POLLRDBAND
POLLRDNORM	数据可读
POLLRDBAND	优先级带数据可读
POLLPRI	高优先级可读数据
POLLOUT	普通或带外数据可写
POLLWRNORM	数据可写
POLLWRBAND	优先级带数据可写
POLLERR	发生错误
POLLHUP	发生挂起
POLLNVAL	描述字不是一个打开的文件

nfds	监控数组中有多少文件描述符需要被监控	
timeout	毫秒级等待	
-1	阻塞等，#define INFTIM -1	Linux 中没有定义此宏
0	立即返回，不阻塞进程	
>0	等待指定毫秒数，如当前系统时间精度不够毫秒，向上取值	

如果不再监控某个文件描述符时，可以把 pollfd 中，fd 设置为-1，poll 不再监控此 pollfd，下次返回时，把 revents 设置为 0。

使用 poll()和 select()不一样，你不需要显式地请求异常情况报告。

POLLIN | POLLPRI 等价于 select()的读事件，POLLOUT | POLLWRBAND 等价于 select()的写事件。POLLIN 等价于 POLLRDNORM | POLLRDBAND，而 POLLOUT 则等价于 POLLWRNORM。例如，要同时监视一个文件描述符是否可读和可写，我们可以设置 events 为 POLLIN | POLLOUT。在 poll 返回时，我们可以检查 revents 中的标志，对应于文件描述符请求的 events 结构体。如果 POLLIN 事件被设置，则文件描述符可以被读取而不阻塞。如果 POLLOUT 被设置，则文件描述符可以写入而不导致阻塞。这些标志并不是互斥的：它们可能被同时设置，表示这个文件描述符的读取和写入操作都会正常返回而不阻塞。

timeout 参数指定等待的毫秒数，无论 I/O 是否准备好，poll 都会返回。timeout 指定为负数值表示无限超时，使 poll()一直挂起直到一个指定事件发生；timeout 为 0 指示 poll 调用立即返回并列出准备好 I/O 的文件描述符，但并不等待其它的事件。这种情况下，poll()就像它的名字那样，一旦选举出来，立即返回。

返回值和错误代码

成功时，poll()返回结构体中 revents 域不为 0 的文件描述符个数；如果在超时前没有任何事件发生，poll()返回 0；失败时，poll()返回-1，并设置 errno 为下列值之一：

EBADF	一个或多个结构体中指定的文件描述符无效。
EFAULTfds	指针指向的地址超出进程的地址空间。
EINTR	请求的事件之前产生一个信号，调用可以重新发起。
EINVALnfd	参数超出 PLIMIT_NOFILE 值。
ENOMEM	可用内存不足，无法完成请求。

server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```

#include <poll.h>
#include <errno.h>
#include <ctype.h>

#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 8000
#define OPEN_MAX 1024

int main(int argc, char *argv[])
{
    int i, j, maxi, listenfd, connfd, sockfd;
    int nready; /*接收 poll 返回值, 记录满足监听事件的 fd 个数*/
    ssize_t n;

    char buf[MAXLINE], str[INET_ADDRSTRLEN];
    socklen_t clilen;
    struct pollfd client[OPEN_MAX];
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    int opt = 1;
    setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
    Listen(listenfd, 128);

    client[0].fd = listenfd; /* 要监听的第一个文件描述符 存入 client[0]*/
    client[0].events = POLLIN; /* listenfd 监听普通读事件 */

    for (i = 1; i < OPEN_MAX; i++)
        client[i].fd = -1; /* 用-1 初始化 client[]里剩下元素 0 也是文件描述符,不能用 */

    maxi = 0; /* client[]数组有效元素中最大元素下标 */

    for ( ; ; ) {
        nready = poll(client, maxi+1, -1); /* 阻塞监听是否有客户端链接请求 */

        if (client[0].revents & POLLIN) { /* listenfd 有读事件就绪 */

            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);/* 接收客户端请求 Accept 不会阻塞 */

```

```

printf("received from %s at PORT %d\n",
      inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
      ntohs(cliaddr.sin_port));

for (i = 1; i < OPEN_MAX; i++)
    if (client[i].fd < 0) {
        client[i].fd = connfd;      /* 找到 client[] 中空闲的位置,存放 accept 返回的 connfd */
        break;
    }

if (i == OPEN_MAX)                /* 达到了最大客户端数 */
    perr_exit("too many clients");

client[i].events = POLLIN;        /* 设置刚刚返回的 connfd,监控读事件 */
if (i > maxi)
    maxi = i;                     /* 更新 client[] 中最大元素下标 */
if (--nready <= 0)
    continue;                     /* 没有更多就绪事件时,继续回到 poll 阻塞 */
}

for (i = 1; i <= maxi; i++) {     /* 前面的 if 没满足,说明没有 listenfd 满足. 检测 client[] 看是那个 connfd
就绪 */
    if ((sockfd = client[i].fd) < 0)
        continue;

    if (client[i].revents & POLLIN) {

        if ((n = Read(sockfd, buf, MAXLINE)) < 0) {
            /* connection reset by client */
            if (errno == ECONNRESET) { /* 收到 RST 标志 */
                printf("client[%d] aborted connection\n", i);
                Close(sockfd);
                client[i].fd = -1;    /* poll 中不监控该文件描述符,直接置为-1 即可,不用像 select 中那样移除 */
            } else
                perr_exit("read error");

        } else if (n == 0) {        /* 说明客户端先关闭链接 */
            printf("client[%d] closed connection\n", i);
            Close(sockfd);
            client[i].fd = -1;
        } else {
            for (j = 0; j < n; j++)
                buf[j] = toupper(buf[j]);
            Writen(sockfd, buf, n);
        }
        if (--nready <= 0)
            break;
    }
}
}

```

```
}  
    return 0;  
}
```

client

```
#include <stdio.h>  
#include <string.h>  
#include <unistd.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
  
#include "wrap.h"  
  
#define MAXLINE 80  
#define SERV_PORT 8000  
  
int main(int argc, char *argv[])  
{  
    struct sockaddr_in servaddr;  
    char buf[MAXLINE];  
    int sockfd, n;  
  
    sockfd = Socket(AF_INET, SOCK_STREAM, 0);  
  
    bzero(&servaddr, sizeof(servaddr));  
    servaddr.sin_family = AF_INET;  
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);  
    servaddr.sin_port = htons(SERV_PORT);  
  
    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));  
  
    while (fgets(buf, MAXLINE, stdin) != NULL) {  
        Write(sockfd, buf, strlen(buf));  
        n = Read(sockfd, buf, MAXLINE);  
  
        if (n == 0) {  
            printf("the other side has been closed.\n");  
            break;  
        } else  
            Write(STDOUT_FILENO, buf, n);  
    }  
  
    Close(sockfd);  
  
    return 0;  
}
```


ppoll

GNU 定义了 ppoll（非 POSIX 标准），可以支持设置信号屏蔽字，大家可参考 poll 模型自行实现 C/S。

```
#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <poll.h>
int ppoll(struct pollfd *fds, nfds_t nfds,
          const struct timespec *timeout_ts, const sigset_t *sigmask);
```

epoll

epoll 是 Linux 下多路复用 IO 接口 select/poll 的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统 CPU 利用率，因为它会复用文件描述符集合来传递结果而不用迫使开发者每次等待事件之前都必须重新准备要被侦听的文件描述符集合，另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核 IO 事件异步唤醒而加入 Ready 队列的描述符集合就行了。

目前 epell 是 linux 大规模并发网络程序中的热门首选模型。

epoll 除了提供 select/poll 那种 IO 事件的电平触发（Level Triggered）外，还提供了边沿触发（Edge Triggered），这就使得用户空间程序有可能缓存 IO 状态，减少 epoll_wait/epoll_pwait 的调用，提高应用程序效率。

可以使用 cat 命令查看一个进程可以打开的 socket 描述符上限。

```
cat /proc/sys/fs/file-max
```

如有需要，可以通过修改配置文件的方式修改该上限值。

```
sudo vi /etc/security/limits.conf
在文件尾部写入以下配置,soft 软限制，hard 硬限制。如下图所示。
* soft nfile 65536
* hard nfile 100000
```

```

39 #           - nice - max nice priority allowed to raise to values: [-20, 19]
40 #           - rtprio - max realtime priority
41 #           - chroot - change root to directory (Debian-specific)
42 #
43 #<domain>      <type>  <item>          <value>
44 #
45
46 #*             soft    core             0
47 #root          hard    core            100000
48 #*             hard    rss              10000
49 #@student      hard    nproc           20
50 #@faculty      soft    nproc           20
51 #@faculty      hard    nproc           50
52 #ftp           hard    nproc           0
53 #ftp           -       chroot            /ftp
54 #@student      -       maxlogins       4
55 *             soft    nofile           65536
56 *             hard    nofile           100000
57
58 # End of file
/etc/security/limits.conf
"/etc/security/limits.conf" 58L, 2243C 已写入

```

基础 API

1. 创建一个 epoll 句柄，参数 size 用来告诉内核监听的文件描述符的个数，跟内存大小有关。

```

#include <sys/epoll.h>
int epoll_create(int size)      size : 监听数目

```

2. 控制某个 epoll 监控的文件描述符上的事件：注册、修改、删除。

```

#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
    epfd :    为 epoll_creat 的句柄
    op :      表示动作，用 3 个宏来表示：
                EPOLL_CTL_ADD (注册新的 fd 到 epfd)，
                EPOLL_CTL_MOD (修改已经注册的 fd 的监听事件)，
                EPOLL_CTL_DEL (从 epfd 删除一个 fd)；
    event :   告诉内核需要监听的事件

struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};
typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;

```

EPOLLIN : 表示对应的文件描述符可以读（包括对端 SOCKET 正常关闭）

EPOLLOUT : 表示对应的文件描述符可以写

EPOLLPRI : 表示对应的文件描述符有紧急的数据可读 (这里应该表示有带外数据到来)

EPOLLERR : 表示对应的文件描述符发生错误

EPOLLHUP : 表示对应的文件描述符被挂断;

EPOLLET : 将 EPOLL 设为边缘触发(Edge Triggered)模式, 这是相对于水平触发(Level Triggered)而言的

EPOLLONESHOT : 只监听一次事件, 当监听完这次事件之后, 如果还需要继续监听这个 socket 的话, 需要再次把这个 socket 加入到 EPOLL 队列里

3. 等待所监控文件描述符上有事件的产生, 类似于 select()调用。

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

int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)

    events :      用来存内核得到事件的集合,
    maxevents :   告之内核这个 events 有多大, 这个 maxevents 的值不能大于创建 epoll_create()时的 size,
    timeout :     是超时时间
        -1 : 阻塞
        0 : 立即返回, 非阻塞
        >0 : 指定毫秒
    返回值 : 成功返回有多少文件描述符就绪, 时间到时返回 0, 出错返回-1
```

server

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/epoll.h>
#include <errno.h>
#include <ctype.h>

#include "wrap.h"

#define MAXLINE 8192
#define SERV_PORT 8000
#define OPEN_MAX 5000 //创建连接上限

int main(int argc, char *argv[])
{
    int i, listenfd, connfd, sockfd;
    int n, num = 0;
    ssize_t nready, efd, res;
    char buf[MAXLINE], str[INET_ADDRSTRLEN];
    socklen_t clien;

    struct sockaddr_in cliaddr, servaddr;
    struct epoll_event tep, ep[OPEN_MAX]; //tep: epoll_ctl 参数 ep[] : epoll_wait 参数
```

```

listenfd = Socket(AF_INET, SOCK_STREAM, 0);

int opt = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));    //端口复用

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

Bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

Listen(listenfd, 20);

efd = epoll_create(OPEN_MAX);          //创建 epoll 模型, efd 指向红黑树根节点
if (efd == -1)
    perr_exit("epoll_create error");

tep.events = EPOLLIN; tep.data.fd = listenfd;          //指定 lfd 的监听时间为"读"
res = epoll_ctl(efd, EPOLL_CTL_ADD, listenfd, &tep);    //将 lfd 及对应的结构体设置到树上,efd 可找到该树
if (res == -1)
    perr_exit("epoll_ctl error");

for ( ; ; ) {
    /*epoll 为 server 阻塞监听事件, ep 为 struct epoll_event 类型数组, OPEN_MAX 为数组容量, -1 表永久阻塞*/
    nready = epoll_wait(efd, ep, OPEN_MAX, -1);
    if (nready == -1)
        perr_exit("epoll_wait error");

    for (i = 0; i < nready; i++) {
        if (!(ep[i].events & EPOLLIN))          //如果不是"读"事件, 继续循环
            continue;

        if (ep[i].data.fd == listenfd) {        //判断满足事件的 fd 是不是 lfd
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);    //接受链接

            printf("received from %s at PORT %d\n",
                inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
                ntohs(cliaddr.sin_port));
            printf("cfd %d---client %d\n", connfd, ++num);

            tep.events = EPOLLIN; tep.data.fd = connfd;
            res = epoll_ctl(efd, EPOLL_CTL_ADD, connfd, &tep);
            if (res == -1)
                perr_exit("epoll_ctl error");

        } else {                                //不是 lfd,

```

```

    sockfd = ep[i].data.fd;
    n = Read(sockfd, buf, MAXLINE);

    if (n == 0) { //读到 0,说明客户端关闭链接
        res = epoll_ctl(efd, EPOLL_CTL_DEL, sockfd, NULL); //将该文件描述符从红黑树摘除
        if (res == -1)
            perr_exit("epoll_ctl error");
        Close(sockfd); //关闭与该客户端的链接
        printf("client[%d] closed connection\n", sockfd);

    } else if (n < 0) { //出错
        perror("read n < 0 error: ");
        res = epoll_ctl(efd, EPOLL_CTL_DEL, sockfd, NULL);
        Close(sockfd);

    } else { //实际读到了字节数
        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]); //转大写,写回给客户端

        Write(STDOUT_FILENO, buf, n);
        Writen(sockfd, buf, n);
    }
}
}
}
Close(listenfd);
Close(efd);

return 0;
}

```

client

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "wrap.h"

#define MAXLINE 8192
#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];

```

```

int sockfd, n;

sockfd = Socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
servaddr.sin_port = htons(SERV_PORT);

Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

while (fgets(buf, MAXLINE, stdin) != NULL) {
    Write(sockfd, buf, strlen(buf));
    n = Read(sockfd, buf, MAXLINE);
    if (n == 0) {
        printf("the other side has been closed.\n");
        break;
    }
    else
        Write(STDOUT_FILENO, buf, n);
}
Close(sockfd);

return 0;
}

```

Makefile

```

src = $(wildcard *.c)
obj = $(patsubst %.c, %.o, $(src))

all: server client

server: server.o wrap.o
    gcc server.o wrap.o -o server -Wall
client: client.o wrap.o
    gcc client.o wrap.o -o client -Wall

%.o:%.c
    gcc -c $< -Wall

.PHONY: clean all
clean:
    -rm -rf server client $(obj)

```

epoll 进阶

事件模型

EPOLL 事件有两种模型：

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

Level Triggered (LT) 水平触发只要有数据都会触发。

思考如下步骤：

1. 假定我们已经把一个用来从管道中读取数据的文件描述符(RFD)添加到 epoll 描述符。
2. 管道的另一端写入了 2KB 的数据
3. 调用 `epoll_wait`，并且它会返回 RFD，说明它已经准备好读取操作
4. 读取 1KB 的数据
5. 调用 `epoll_wait`.....

在这个过程中，有两种工作模式：

ET 模式

ET 模式即 Edge Triggered 工作模式。

如果我们在第 1 步将 RFD 添加到 epoll 描述符的时候使用了 `EPOLLET` 标志，那么在第 5 步调用 `epoll_wait` 之后将有可能挂起，因为剩余的数据还存在于文件的输入缓冲区内，而且数据发出端还在等待一个针对已经发出数据的反馈信息。只有在监视的文件句柄上发生了某个事件的时候 ET 工作模式才会汇报事件。因此在第 5 步的时候，调用者可能会放弃等待仍在存在于文件输入缓冲区内的剩余数据。epoll 工作在 ET 模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。最好以下面的方式调用 ET 模式的 epoll 接口，在后面会介绍避免可能的缺陷。

1) 基于非阻塞文件句柄

2) 只有当 read 或者 write 返回 EAGAIN(非阻塞读, 暂时无数据)时才需要挂起、等待。但这并不是说每次

read 时都需要循环读, 直到读到产生一个 EAGAIN 才认为此次事件处理完成, 当 read 返回的读到的数据

长度小于请求的数据长度时, 就可以确定此时缓冲中已没有数据了, 也就可以认为此事读事件已处理完成。

LT 模式

LT 模式即 Level Triggered 工作模式。

与 ET 模式不同的是, 以 LT 方式调用 epoll 接口的时候, 它就相当于一个速度比较快的 poll, 无论后面的数据是否被使用。

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

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

实例一：

基于管道 epoll LT 触发模式

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <errno.h>
#include <unistd.h>

#define MAXLINE 10

int main(int argc, char *argv[])
{
```



```

int efd, i;
int pfd[2];
pid_t pid;
char buf[MAXLINE], ch = 'a';

pipe(pfd);
pid = fork();

if (pid == 0) {           //子 写
    close(pfd[0]);
    while (1) {
        //aaaa\n
        for (i = 0; i < MAXLINE/2; i++)
            buf[i] = ch;
        buf[i-1] = '\n';
        ch++;
        //bbbb\n
        for (; i < MAXLINE; i++)
            buf[i] = ch;
        buf[i-1] = '\n';
        ch++;
        //aaaa\nbbbb\n
        write(pfd[1], buf, sizeof(buf));
        sleep(5);
    }
    close(pfd[1]);
}

} else if (pid > 0) {     //父 读
    struct epoll_event event;
    struct epoll_event revent[10];           //epoll_wait 就绪返回 event
    int res, len;

    close(pfd[1]);
    efd = epoll_create(10);

    //    event.events = EPOLLIN | EPOLLET;    // ET 边沿触发
    event.events = EPOLLIN;                  // LT 水平触发 (默认)
    event.data.fd = pfd[0];
    epoll_ctl(efd, EPOLL_CTL_ADD, pfd[0], &event);

    while (1) {
        res = epoll_wait(efd, revent, 10, -1);
        printf("res %d\n", res);
        if (revent[0].data.fd == pfd[0]) {
            len = read(pfd[0], buf, MAXLINE/2);
            write(STDOUT_FILENO, buf, len);
        }
    }
}

```

```

        close(pfd[0]);
        close(efd);

    } else {
        perror("fork");
        exit(-1);
    }

    return 0;
}

```

编译运行结果如下：

```

res 1
aaaa
res 1
bbbb
res 1
cccc
res 1
dddd
res 1
eeee
res 1
ffff
res 1
gggg
res 1
hhhh

```

现象是先打印十个字符，对应于上图就是先打印 aaaa 和 bbbb 然后过了 5s 继续打印 cccc 和 dddd 然后以此类推，这就是 LT 模式，只要缓冲区有数据就触发。然后将代码修改

```
event.events = EPOLLIN | EPOLLET;    // ET 边沿触发
```

编译运行结果如下：

```

res 1
aaaa
res 1
bbbb
res 1
cccc

```

每过 5s 打印一串字符（如 aaaa），程序设定每隔 5s 子进程向管道发送一串字符，这就是 ET 模式，只有在接收到数据时才触发。

实例二：

基于网络 C/S 模型的 epoll ET 触发模式

server

```
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/epoll.h>
#include <unistd.h>

#define MAXLINE 10
#define SERV_PORT 9000

int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    int efd;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    listen(listenfd, 20);

    struct epoll_event event;
    struct epoll_event revent[10];
    int res, len;

    efd = epoll_create(10);
    event.events = EPOLLIN | EPOLLET;    /* ET 边沿触发 */
    //event.events = EPOLLIN;           /* 默认 LT 水平触发 */

    printf("Accepting connections ...\n");

    cliaddr_len = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
    printf("received from %s at PORT %d\n",
```

```

        inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
        ntohs(cliaddr.sin_port));

event.data.fd = connfd;
epoll_ctl(efd, EPOLL_CTL_ADD, connfd, &event);

while (1) {
    res = epoll_wait(efd, reseat, 10, -1);

    printf("res %d\n", res);
    if (reseat[0].data.fd == connfd) {
        len = read(connfd, buf, MAXLINE/2);          //readn(500)
        write(STDOUT_FILENO, buf, len);
    }
}

return 0;
}

```

client

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define MAXLINE 10
#define SERV_PORT 9000
int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, i;
    char ch = 'a';

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (1) {
        //aaaa\n
        for (i = 0; i < MAXLINE/2; i++)

```

```

        buf[i] = ch;
    buf[i-1] = '\n';
    ch++;
    //bbbb\n
    for (; i < MAXLINE; i++)
        buf[i] = ch;
    buf[i-1] = '\n';
    ch++;
    //aaaa\nbbbb\n
    write(sockfd, buf, sizeof(buf));
    sleep(5);
}
close(sockfd);

return 0;
}

```

编译运行结果如下：

```

Accepting connections ...
received from 127.0.0.1 at PORT 55526
res 1
aaaa
res 1
bbbb
res 1
cccc

```

客户端向服务端连续发送 aaaa 和 bbbb 字符，然后过 5s 发送 cccc 和 dddd，再然后以此类推发送，一轮发送十个字符，服务端设置 epoll 为 ET 模式，每次接受 5 个字符，所以读端阻塞了，只有等到 5s 后客户端在此发送才会触发接受字符，所以现象是每隔 5s 打印一串字符。

实例三：

基于网络 C/S 非阻塞模型的 epoll ET 触发模式

server

```

#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/epoll.h>
#include <unistd.h>

```

```

#include <fcntl.h>

#define MAXLINE 10
#define SERV_PORT 8000

int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    int efd, flag;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    listen(listenfd, 20);

    //////////////////////////////////////
    struct epoll_event event;
    struct epoll_event revent[10];
    int res, len;

    efd = epoll_create(10);

    event.events = EPOLLIN | EPOLLET;    /* ET 边沿触发，默认是水平触发 */

    //event.events = EPOLLIN;
    printf("Accepting connections ...\n");
    cliaddr_len = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
    printf("received from %s at PORT %d\n",
           inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
           ntohs(cliaddr.sin_port));

    flag = fcntl(connfd, F_GETFL);      /* 修改 connfd 为非阻塞读 */
    flag |= O_NONBLOCK;
    fcntl(connfd, F_SETFL, flag);

    event.data.fd = connfd;
    epoll_ctl(efd, EPOLL_CTL_ADD, connfd, &event);    //将 connfd 加入监听红黑树
    while (1) {

```

```

printf("epoll_wait begin\n");
res = epoll_wait(efd, revent, 10, -1);      //最多 10 个, 阻塞监听
printf("epoll_wait end res %d\n", res);

if (revent[0].data.fd == connfd) {
    while ((len = read(connfd, buf, MAXLINE/2)) > 0)    //非阻塞读, 轮询
        write(STDOUT_FILENO, buf, len);
}
}

return 0;
}

```

client

```

/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define MAXLINE 10
#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, i;
    char ch = 'a';

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (1) {
        //aaaa\n
        for (i = 0; i < MAXLINE/2; i++)
            buf[i] = ch;
        buf[i-1] = '\n';
        ch++;
        //bbbb\n
    }
}

```

```

    for (; i < MAXLINE; i++)
        buf[i] = ch;
    buf[i-1] = '\n';
    ch++;
    //aaaa\nbbbb\n
    write(sockfd, buf, sizeof(buf));
    sleep(10);
}

close(sockfd);

return 0;
}

```

编译运行结果如下：

```

Accepting connections ...
received from 127.0.0.1 at PORT 52646
epoll_wait begin
epoll_wait end res 1
aaaa
bbbb
epoll_wait begin
epoll_wait end res 1
cccc
dddd
epoll_wait begin

```

服务器端一次性将客户端发送的十个字符全部接收并打印了。服务器端主要做了一件事：

```

flag = fcntl(connfd, F_GETFL);          /* 修改 connfd 为非阻塞读 */
flag |= O_NONBLOCK;
fcntl(connfd, F_SETFL, flag);

```

将 connfd 设置为非阻塞，由于：

```
while ((len = read(connfd, buf, MAXLINE/2)) > 0)    //非阻塞读，轮询
```

一次只能读五个字节，然而 epoll 设置为 ET 模式，只有等待客户端 5s 后的在此发送才能继续读出，这样就形成了阻塞，效率不高。如果设置 connfd 为非阻塞，那么一次性就可以读出数据，避免了等待和反复调用 epoll_wait() 函数，节省了时间和开销。

epoll 反应堆模型

下面代码实现的思想：epoll 反应堆模型：（ libevent 网络编程开源库 核心思想 ）

1.普通多路 IO 转接服务器： 红黑树 —— 添加待监听的结点 —— epoll_ctl —— EPOLLIN —— fd —— 监

听 —— epoll_wait ——返回满足监听事件的 fd 的总个数 —— 传出参数 events 数组 —— 内部元素 —— 满足对应监听事件的 fd —— 判断对应事件 —— Accept、Read。 ——循环 epoll_wait 监听

2.epoll 反应堆模型： 创建红黑树 —— 添加监听结点 —— epoll_ctl —— EPOLLIN —— fd —— 监听 —— epoll_wait —— 将结点从树上摘下 —— 大写转小写 —— 修改 fd 的监听事件 —— EPOLLOUT —— 重新添加到红黑树 —— 监听 —— epoll_wait —— 写数据到客户端 —— 再将结点从树上摘下 —— 修改监听时间 —— EPOLLIN —— 挂上红黑树监听。

添加监听写事件的目的：“滑动窗口”已满，绕过写。epoll_wait 满足后再进行写。

```
/*
 *epoll 基于非阻塞 I/O 事件驱动
 */
#include <stdio.h>
#include <sys/socket.h>
#include <sys/epoll.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#define MAX_EVENTS 1024 //监听上限数
#define BUFLLEN 4096
#define SERV_PORT 8080

void recvdata(int fd, int events, void *arg);
void senddata(int fd, int events, void *arg);

/* 描述就绪文件描述符相关信息 */

struct myevent_s {
    int fd; //要监听的文件描述符
    int events; //对应的监听事件
    void *arg; //泛型参数
    void (*call_back)(int fd, int events, void *arg); //回调函数
    int status; //是否在监听:1->在红黑树上(监听), 0->不在(不监听)
    char buf[BUFLLEN];
    int len;
    long last_active; //记录每次加入红黑树 g_efd 的时间值
};
```

```

int g_efd; //全局变量, 保存 epoll_create 返回的文件描述符
struct myevent_s g_events[MAX_EVENTS+1]; //自定义结构体类型数组. +1-->listen fd

/*将结构体 myevent_s 成员变量 初始化*/

void eventset(struct myevent_s *ev, int fd, void (*call_back)(int, int, void *), void *arg)
{
    ev->fd = fd;
    ev->call_back = call_back;
    ev->events = 0;
    ev->arg = arg;
    ev->status = 0;
    //memset(ev->buf, 0, sizeof(ev->buf));
    //ev->len = 0;
    ev->last_active = time(NULL); //调用 eventset 函数的时间

    return;
}

/* 向 epoll 监听的红黑树 添加一个 文件描述符 */

void eventadd(int efd, int events, struct myevent_s *ev)
{
    struct epoll_event epv = {0, {0}};
    int op;
    epv.data.ptr = ev;
    epv.events = ev->events = events; //EPOLLIN 或 EPOLLOUT

    if (ev->status == 1) { //已经在红黑树 g_efd 里
        op = EPOLL_CTL_MOD; //修改其属性
    } else { //不在红黑树里
        op = EPOLL_CTL_ADD; //将其加入红黑树 g_efd, 并将 status 置 1
        ev->status = 1;
    }

    if (epoll_ctl(efd, op, ev->fd, &epv) < 0) //实际添加/修改
        printf("event add failed [fd=%d], events[%d]\n", ev->fd, events);
    else
        printf("event add OK [fd=%d], op=%d, events[%0X]\n", ev->fd, op, events);

    return ;
}

/* 从 epoll 监听的 红黑树中删除一个 文件描述符*/

void eventdel(int efd, struct myevent_s *ev)
{
    struct epoll_event epv = {0, {0}};

```

```

if (ev->status != 1)                                     //不在红黑树上
    return ;

epv.data.ptr = ev;
ev->status = 0;                                           //修改状态
epoll_ctl(efd, EPOLL_CTL_DEL, ev->fd, &epv);           //从红黑树 efd 上将 ev->fd 摘除

return ;
}

/* 当有文件描述符就绪, epoll 返回, 调用该函数 与客户端建立链接 */

void acceptconn(int lfd, int events, void *arg)
{
    struct sockaddr_in cin;
    socklen_t len = sizeof(cin);
    int cfd, i;

    if ((cfd = accept(lfd, (struct sockaddr *)&cin, &len)) == -1) {
        if (errno != EAGAIN && errno != EINTR) {
            /* 暂时不做出错处理 */
        }
        printf("%s: accept, %s\n", __func__, strerror(errno));
        return ;
    }

    do {
        for (i = 0; i < MAX_EVENTS; i++)
            if (g_events[i].status == 0)                //从全局数组 g_events 中找一个空闲元素
                                                        //类似于 select 中找值为-1 的元素
                break;                                    //跳出 for

        if (i == MAX_EVENTS) {
            printf("%s: max connect limit[%d]\n", __func__, MAX_EVENTS);
            break;                                        //跳出 do while(0) 不执行后续代码
        }

        int flag = 0;
        if ((flag = fcntl(cfd, F_SETFL, O_NONBLOCK)) < 0) {
            //将 cfd 也设置为非阻塞
            printf("%s: fcntl nonblocking failed, %s\n", __func__, strerror(errno));
            break;
        }

        /* 给 cfd 设置一个 myevent_s 结构体, 回调函数 设置为 recvdata */

        eventset(&g_events[i], cfd, recvdata, &g_events[i]);
        eventadd(g_efd, EPOLLIN, &g_events[i]);         //将 cfd 添加到红黑树 g_efd 中, 监听读事件

    } while(0);
}

```

```

printf("new connect [%s:%d][time:%ld], pos[%d]\n",
      inet_ntoa(cin.sin_addr), ntohs(cin.sin_port), g_events[i].last_active, i);
return ;
}

void recvdata(int fd, int events, void *arg)
{
    struct myevent_s *ev = (struct myevent_s *)arg;
    int len;

    len = recv(fd, ev->buf, sizeof(ev->buf), 0);          //读文件描述符，数据存入 myevent_s 成员 buf 中

    eventdel(g_efd, ev);          //将该节点从红黑树上摘除

    if (len > 0) {

        ev->len = len;
        ev->buf[len] = '\0';          //手动添加字符串结束标记
        printf("C[%d]:%s\n", fd, ev->buf);

        eventset(ev, fd, senddata, ev);          //设置该 fd 对应的回调函数为 senddata
        eventadd(g_efd, EPOLLOUT, ev);          //将 fd 加入红黑树 g_efd 中,监听其写事件

    } else if (len == 0) {
        close(ev->fd);
        /* ev-g_events 地址相减得到偏移元素位置 */
        printf("[fd=%d] pos[%ld], closed\n", fd, ev-g_events);
    } else {
        close(ev->fd);
        printf("recv[fd=%d] error[%d]:%s\n", fd, errno, strerror(errno));
    }

    return;
}

void senddata(int fd, int events, void *arg)
{
    struct myevent_s *ev = (struct myevent_s *)arg;
    int len;

    len = send(fd, ev->buf, ev->len, 0);          //直接将数据 回写给客户端。未作处理
    /*
    printf("fd=%d\tev->buf=%s\tev->len=%d\n", fd, ev->buf, ev->len);
    printf("send len = %d\n", len);
    */

    if (len > 0) {

```

```

    printf("send[fd=%d], [%d]%s\n", fd, len, ev->buf);
    eventdel(g_efd, ev); //从红黑树 g_efd 中移除
    eventset(ev, fd, recvdata, ev); //将该 fd 的 回调函数改为 recvdata
    eventadd(g_efd, EPOLLIN, ev); //从新添加到红黑树上， 设为监听读事件

} else {
    close(ev->fd); //关闭链接
    eventdel(g_efd, ev); //从红黑树 g_efd 中移除
    printf("send[fd=%d] error %s\n", fd, strerror(errno));
}

return ;
}

/*创建 socket, 初始化 lfd */

void initlistensocket(int efd, short port)
{
    int lfd = socket(AF_INET, SOCK_STREAM, 0);
    fcntl(lfd, F_SETFL, O_NONBLOCK); //将 socket 设为非阻塞

    /* void eventset(struct myevent_s *ev, int fd, void (*call_back)(int, int, void *), void *arg); */
    eventset(&g_events[MAX_EVENTS], lfd, acceptconn, &g_events[MAX_EVENTS]);

    /* void eventadd(int efd, int events, struct myevent_s *ev) */
    eventadd(efd, EPOLLIN, &g_events[MAX_EVENTS]);

    struct sockaddr_in sin;
    memset(&sin, 0, sizeof(sin)); //bzero(&sin, sizeof(sin))
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(port);

    bind(lfd, (struct sockaddr *)&sin, sizeof(sin));

    listen(lfd, 20);

    return ;
}

int main(int argc, char *argv[])
{
    unsigned short port = SERV_PORT;

    if (argc == 2)
        port = atoi(argv[1]); //使用用户指定端口.如未指定,用默认端口

    g_efd = epoll_create(MAX_EVENTS+1); //创建红黑树,返回给全局 g_efd
    if (g_efd <= 0)

```

```

printf("create efd in %s err %s\n", __func__, strerror(errno));

initlistensocket(g_efd, port);                                //初始化监听 socket

struct epoll_event events[MAX_EVENTS+1];                    //保存已经满足就绪事件的文件描述符数组
printf("server running:port[%d]\n", port);

int checkpos = 0, i;
while (1) {
    /* 超时验证，每次测试 100 个链接，不测试 listenfd 当客户端 60 秒内没有和服务器通信，则关闭此客户端链接 */

    long now = time(NULL);                                    //当前时间
    for (i = 0; i < 100; i++, checkpos++) {                  //一次循环检测 100 个。使用 checkpos 控制检测对象
        if (checkpos == MAX_EVENTS)
            checkpos = 0;
        if (g_events[checkpos].status != 1)                  //不在红黑树 g_efd 上
            continue;

        long duration = now - g_events[checkpos].last_active; //客户端不活跃的时间

        if (duration >= 60) {
            close(g_events[checkpos].fd);                    //关闭与该客户端链接
            printf("[fd=%d] timeout\n", g_events[checkpos].fd);
            eventdel(g_efd, &g_events[checkpos]);            //将该客户端 从红黑树 g_efd 移除
        }
    }
}

/*监听红黑树 g_efd，将满足的事件的文件描述符加至 events 数组中，1 秒没有事件满足，返回 0*/
int nfd = epoll_wait(g_efd, events, MAX_EVENTS+1, 1000);
if (nfd < 0) {
    printf("epoll_wait error, exit\n");
    break;
}

for (i = 0; i < nfd; i++) {
    /*使用自定义结构体 myevent_s 类型指针，接收 联合体 data 的 void *ptr 成员*/
    struct myevent_s *ev = (struct myevent_s *)events[i].data.ptr;

    if ((events[i].events & EPOLLIN) && (ev->events & EPOLLIN)) { //读就绪事件
        ev->call_back(ev->fd, events[i].events, ev->arg);
    }
    if ((events[i].events & EPOLLOUT) && (ev->events & EPOLLOUT)) { //写就绪事件
        ev->call_back(ev->fd, events[i].events, ev->arg);
    }
}
}
/* 退出前释放所有资源 */
return 0;
}

```

Libevent 核心原理

Libevent 是一个事件驱动框架，不能仅说他是一个网络库。nodejs 就是采用与 libevent 类似的 libev 来做核心驱动的。

Libevent 支持三种事件：io 事件、信号事件、时间事件，并且事件的设置和使用方式是一样的。libevent 的核心原理是采用 io 多路复用的方式来单线程处理事件。至于为什么这么说，下面会分别对三种事件进行解释。

io 事件：io 事件包含 socket 可读、可写、断开、设备可读、可写等和 IO 相关的事件，libevent 主要采用了 epoll 模型来进行 i/o 事件的多路复用（我说的是 linux 上，libevent 也封装了 select, poll 模型，下面仅说采用 epoll 的情况）。一句话解释 epoll 模型：就是在内核管理的设备或者资源上设置等待队列，当资源出现的时候，系统会通知 epoll_wait 唤醒，进行事件的处理。总的来说，i/o 事件的事件驱动依赖于操作系统。

时间事件：时间事件，可以简单的解释一下，如果我们想在 500ms 后执行一段代码，那么，就可以在 libevent 上面设置一个时间事件，代码封装到回调函数里面去。如果看过 epoll 或者 select 的使用方式，你就会知道 epoll_wait 可以设置一个 timeout，等待 timeout 这个时长，如果没有事件发生，也会返回。时间事件的处理就 将所有时间事件要等待时间最少的设置为 timeout 时间，这样，即使什么 i/o 事件也没有发生，也能在 timeout 后，处理该处理的时间事件。

信号事件：简单的解释一下，即使和 i/o 无关的信号，也可以作为一个事件进行处理，信号在 linux 中是进程间通信方式之一，A 可以发出一个信号，B 可以接受信号，B 接受之后可以进行一些操作，问题是，libevent 想把信号事件也统一一起处理，其原理是，将信号事件这种和 I/O 无关的事件转换为和 I/O 有关的，充分利用现有模型统一处理。事实上，信号事件是采用一个 client socket 和一个 server socket，client socket 只能写，server socket 只能读，server socket 将 recv 到的 client socket 的文件描述符放到了 epoll 的事件集合中，永远不删。当收到系统信号的时候，通过 client socket 进行发送，server socket 收到数据，就会触发 epoll_wait 唤醒，如果发现这是 server socket 的事件，就会对信号事件进行遍历找到那个等待的。

线程池并发服务器

1. 预先创建阻塞于 accept 多线程，使用互斥锁上锁保护 accept
2. 预先创建多线程，由主线程调用 accept

threadpool.h

```
#ifndef __THREADPOOL_H_
#define __THREADPOOL_H_

typedef struct threadpool_t threadpool_t;

/**
 * @function threadpool_create
 * @desc Creates a threadpool_t object.
 * @param thr_num  thread num
 * @param max_thr_num  max thread size
 * @param queue_max_size  size of the queue.
 * @return a newly created thread pool or NULL
 */
threadpool_t *threadpool_create(int min_thr_num, int max_thr_num, int queue_max_size);

/**
 * @function threadpool_add
 * @desc add a new task in the queue of a thread pool
 * @param pool  Thread pool to which add the task.
 * @param function Pointer to the function that will perform the task.
 * @param argument Argument to be passed to the function.
 * @return 0 if all goes well, else -1
 */
int threadpool_add(threadpool_t *pool, void*(*function)(void *arg), void *arg);

/**
 * @function threadpool_destroy
 * @desc Stops and destroys a thread pool.
 * @param pool  Thread pool to destroy.
 * @return 0 if destroy success else -1
 */
int threadpool_destroy(threadpool_t *pool);

/**
 * @desc get the thread num
 * @param pool threadpool
 * @return # of the thread
 */
```



```

int threadpool_all_threadnum(threadpool_t *pool);

/**
 * desc get the busy thread num
 * @param pool threadpool
 * return # of the busy thread
 */
int threadpool_busy_threadnum(threadpool_t *pool);

#endif

```

Threadpool.c

```

#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include "threadpool.h"

#define DEFAULT_TIME 10 /*10s 检测一次*/
#define MIN_WAIT_TASK_NUM 10 /*如果 queue_size > MIN_WAIT_TASK_NUM 添加新的线程到线程池*/
#define DEFAULT_THREAD_VARY 10 /*每次创建和销毁线程的个数*/
#define true 1
#define false 0

typedef struct {
    void *(*function)(void *); /* 函数指针，回调函数 */
    void *arg; /* 上面函数的参数 */
} threadpool_task_t; /* 各子线程任务结构体 */

/* 描述线程池相关信息 */
struct threadpool_t {
    pthread_mutex_t lock; /* 用于锁住本结构体 */
    pthread_mutex_t thread_counter; /* 记录忙状态线程个数 de 琐 -- busy_thr_num */
    pthread_cond_t queue_not_full; /* 当任务队列满时，添加任务的线程阻塞，等待此条件变量 */
    pthread_cond_t queue_not_empty; /* 任务队列里不为空时，通知等待任务的线程 */

    pthread_t *threads; /* 存放线程池中每个线程的 tid。数组 */
    pthread_t adjust_tid; /* 存管理线程 tid */
    threadpool_task_t *task_queue; /* 任务队列 */

    int min_thr_num; /* 线程池最小线程数 */
    int max_thr_num; /* 线程池最大线程数 */
    int live_thr_num; /* 当前存活线程个数 */

```

```

int busy_thr_num;                /* 忙状态线程个数 */
int wait_exit_thr_num;          /* 要销毁的线程个数 */

int queue_front;                /* task_queue 队头下标 */
int queue_rear;                 /* task_queue 队尾下标 */
int queue_size;                 /* task_queue 队中实际任务数 */
int queue_max_size;             /* task_queue 队列可容纳任务数上限 */

int shutdown;                   /* 标志位，线程池使用状态，true 或 false */
};

/**
 * @function void *threadpool_thread(void *threadpool)
 * @desc the worker thread
 * @param threadpool the pool which own the thread
 */
void *threadpool_thread(void *threadpool);

/**
 * @function void *adjust_thread(void *threadpool);
 * @desc manager thread
 * @param threadpool the threadpool
 */
void *adjust_thread(void *threadpool);

/**
 * check a thread is alive
 */
int is_thread_alive(pthread_t tid);
int threadpool_free(threadpool_t *pool);

threadpool_t *threadpool_create(int min_thr_num, int max_thr_num, int queue_max_size)
{
    int i;
    threadpool_t *pool = NULL;
    do {
        if((pool = (threadpool_t *)malloc(sizeof(threadpool_t))) == NULL) {
            printf("malloc threadpool fail");
            break; /*跳出 do while*/
        }

        pool->min_thr_num = min_thr_num;
        pool->max_thr_num = max_thr_num;
        pool->busy_thr_num = 0;
        pool->live_thr_num = min_thr_num;          /* 活着的线程数 初值=最小线程数 */
        pool->queue_size = 0;                       /* 有 0 个产品 */
        pool->queue_max_size = queue_max_size;
        pool->queue_front = 0;
        pool->queue_rear = 0;
    } while(0);
}

```

```

pool->shutdown = false;                                /* 不关闭线程池 */

/* 根据最大线程上限数，给工作线程数组开辟空间，并清零 */
pool->threads = (pthread_t *)malloc(sizeof(pthread_t)*max_thr_num);
if (pool->threads == NULL) {
    printf("malloc threads fail");
    break;
}
memset(pool->threads, 0, sizeof(pthread_t)*max_thr_num);

/* 队列开辟空间 */
pool->task_queue = (threadpool_task_t *)malloc(sizeof(threadpool_task_t)*queue_max_size);
if (pool->task_queue == NULL) {
    printf("malloc task_queue fail");
    break;
}

/* 初始化互斥锁、条件变量 */
if (pthread_mutex_init(&(pool->lock), NULL) != 0
    || pthread_mutex_init(&(pool->thread_counter), NULL) != 0
    || pthread_cond_init(&(pool->queue_not_empty), NULL) != 0
    || pthread_cond_init(&(pool->queue_not_full), NULL) != 0)
{
    printf("init the lock or cond fail");
    break;
}

/* 启动 min_thr_num 个 work thread */
for (i = 0; i < min_thr_num; i++) {
    pthread_create(&(pool->threads[i]), NULL, threadpool_thread, (void *)pool);/*pool 指向当前线程池*/
    printf("start thread 0x%x...\n", (unsigned int)pool->threads[i]);
}
pthread_create(&(pool->adjust_tid), NULL, adjust_thread, (void *)pool);/* 启动管理者线程 */

return pool;

} while (0);

threadpool_free(pool);    /* 前面代码调用失败时，释放 pool 存储空间 */

return NULL;
}

/* 向线程池中 添加一个任务 */
int threadpool_add(threadpool_t *pool, void*(*function)(void *arg), void *arg)
{
    pthread_mutex_lock(&(pool->lock));

    /* ==为真，队列已经满，调 wait 阻塞 */

```

```

while ((pool->queue_size == pool->queue_max_size) && (!pool->shutdown)) {
    pthread_cond_wait(&(pool->queue_not_full), &(pool->lock));
}
if (pool->shutdown) {
    pthread_mutex_unlock(&(pool->lock));
}

/* 清空 工作线程 调用的回调函数 的参数 arg */
if (pool->task_queue[pool->queue_rear].arg != NULL) {
    free(pool->task_queue[pool->queue_rear].arg);
    pool->task_queue[pool->queue_rear].arg = NULL;
}

/*添加任务到任务队列里*/
pool->task_queue[pool->queue_rear].function = function;
pool->task_queue[pool->queue_rear].arg = arg;
pool->queue_rear = (pool->queue_rear + 1) % pool->queue_max_size;      /* 队尾指针移动, 模拟环形 */
pool->queue_size++;

/*添加完任务后, 队列不为空, 唤醒线程池中 等待处理任务的线程*/
pthread_cond_signal(&(pool->queue_not_empty));
pthread_mutex_unlock(&(pool->lock));

return 0;
}

/* 线程池中各个工作线程 */
void *threadpool_thread(void *threadpool)
{
    threadpool_t *pool = (threadpool_t *)threadpool;
    threadpool_task_t task;

    while (true) {
        /* Lock must be taken to wait on conditional variable */
        /*刚创建出线程, 等待任务队列里有任务, 否则阻塞等待任务队列里有任务后再唤醒接收任务*/
        pthread_mutex_lock(&(pool->lock));

        /*queue_size == 0 说明没有任务, 调 wait 阻塞在条件变量上, 若有任务, 跳过该 while*/
        while ((pool->queue_size == 0) && (!pool->shutdown)) {
            printf("thread 0x%x is waiting\n", (unsigned int)pthread_self());
            pthread_cond_wait(&(pool->queue_not_empty), &(pool->lock));
        }

        /*清除指定数目的空闲线程, 如果要结束的线程个数大于 0, 结束线程*/
        if (pool->wait_exit_thr_num > 0) {
            pool->wait_exit_thr_num--;
        }

        /*如果线程池里线程个数大于最小值时可以结束当前线程*/
        if (pool->live_thr_num > pool->min_thr_num) {
            printf("thread 0x%x is exiting\n", (unsigned int)pthread_self());
            pool->live_thr_num--;
        }
    }
}

```

```

        pthread_mutex_unlock(&(pool->lock));
        pthread_exit(NULL);
    }
}

/*如果指定了 true，要关闭线程池里的每个线程，自行退出处理*/
if (pool->shutdown) {
    pthread_mutex_unlock(&(pool->lock));
    printf("thread 0x%x is exiting\n", (unsigned int)pthread_self());
    pthread_exit(NULL);    /* 线程自行结束 */
}

/*从任务队列里获取任务，是一个出队操作*/
task.function = pool->task_queue[pool->queue_front].function;
task.arg = pool->task_queue[pool->queue_front].arg;

pool->queue_front = (pool->queue_front + 1) % pool->queue_max_size;    /* 出队，模拟环形队列 */
pool->queue_size--;

/*通知可以有新的任务添加进来*/
pthread_cond_broadcast(&(pool->queue_not_full));

/*任务取出后，立即将 线程池锁 释放*/
pthread_mutex_unlock(&(pool->lock));

/*执行任务*/
printf("thread 0x%x start working\n", (unsigned int)pthread_self());
pthread_mutex_lock(&(pool->thread_counter));    /*忙状态线程数变量锁*/
pool->busy_thr_num++;    /*忙状态线程数+1*/
pthread_mutex_unlock(&(pool->thread_counter));
(*(task.function))(task.arg);    /*执行回调函数任务*/
//task.function(task.arg);    /*执行回调函数任务*/

/*任务结束处理*/
printf("thread 0x%x end working\n", (unsigned int)pthread_self());
pthread_mutex_lock(&(pool->thread_counter));
pool->busy_thr_num--;    /*处理掉一个任务，忙状态数线程数-1*/
pthread_mutex_unlock(&(pool->thread_counter));
}

pthread_exit(NULL);
}

/* 管理线程 */
void *adjust_thread(void *threadpool)
{
    int i;
    threadpool_t *pool = (threadpool_t *)threadpool;

```

```

while (!pool->shutdown) {

    sleep(DEFAULT_TIME);                                /*定时 对线程池管理*/

    pthread_mutex_lock(&(pool->lock));
    int queue_size = pool->queue_size;                    /* 关注 任务数 */
    int live_thr_num = pool->live_thr_num;                 /* 存活 线程数 */
    pthread_mutex_unlock(&(pool->lock));

    pthread_mutex_lock(&(pool->thread_counter));
    int busy_thr_num = pool->busy_thr_num;                 /* 忙着的线程数 */
    pthread_mutex_unlock(&(pool->thread_counter));

    /* 创建新线程 算法： 任务数大于最小线程池个数，且存活的线程数少于最大线程个数时 如：30>=10 && 40<100*/
    if (queue_size >= MIN_WAIT_TASK_NUM && live_thr_num < pool->max_thr_num) {
        pthread_mutex_lock(&(pool->lock));
        int add = 0;

        /*一次增加 DEFAULT_THREAD 个线程*/
        for (i = 0; i < pool->max_thr_num && add < DEFAULT_THREAD_VARY
            && pool->live_thr_num < pool->max_thr_num; i++) {
            if (pool->threads[i] == 0 || !is_thread_alive(pool->threads[i])) {
                pthread_create(&(pool->threads[i]), NULL, threadpool_thread, (void *)pool);
                add++;
                pool->live_thr_num++;
            }
        }

        pthread_mutex_unlock(&(pool->lock));
    }

    /* 销毁多余的空闲线程 算法：忙线程 X2 小于 存活的线程数 且 存活的线程数 大于 最小线程数时*/
    if ((busy_thr_num * 2) < live_thr_num && live_thr_num > pool->min_thr_num) {

        /* 一次销毁 DEFAULT_THREAD 个线程，随机 10 个即可 */
        pthread_mutex_lock(&(pool->lock));
        pool->wait_exit_thr_num = DEFAULT_THREAD_VARY;    /* 要销毁的线程数 设置为 10 */
        pthread_mutex_unlock(&(pool->lock));

        for (i = 0; i < DEFAULT_THREAD_VARY; i++) {
            /* 通知处在空闲状态的线程，他们会自行终止*/
            pthread_cond_signal(&(pool->queue_not_empty));
        }
    }
}

return NULL;
}

```

```

int threadpool_destroy(threadpool_t *pool)
{
    int i;
    if (pool == NULL) {
        return -1;
    }
    pool->shutdown = true;

    /*先销毁管理线程*/
    pthread_join(pool->adjust_tid, NULL);

    for (i = 0; i < pool->live_thr_num; i++) {
        /*通知所有的空闲线程*/
        pthread_cond_broadcast(&(pool->queue_not_empty));
    }
    for (i = 0; i < pool->live_thr_num; i++) {
        pthread_join(pool->threads[i], NULL);
    }
    threadpool_free(pool);

    return 0;
}

```

```

int threadpool_free(threadpool_t *pool)
{
    if (pool == NULL) {
        return -1;
    }

    if (pool->task_queue) {
        free(pool->task_queue);
    }
    if (pool->threads) {
        free(pool->threads);
        pthread_mutex_lock(&(pool->lock));
        pthread_mutex_destroy(&(pool->lock));
        pthread_mutex_lock(&(pool->thread_counter));
        pthread_mutex_destroy(&(pool->thread_counter));
        pthread_cond_destroy(&(pool->queue_not_empty));
        pthread_cond_destroy(&(pool->queue_not_full));
    }
    free(pool);
    pool = NULL;

    return 0;
}

```

```

int threadpool_all_threadnum(threadpool_t *pool)
{

```

```

    int all_threadnum = -1;
    pthread_mutex_lock(&(pool->lock));
    all_threadnum = pool->live_thr_num;
    pthread_mutex_unlock(&(pool->lock));
    return all_threadnum;
}

int threadpool_busy_threadnum(threadpool_t *pool)
{
    int busy_threadnum = -1;
    pthread_mutex_lock(&(pool->thread_counter));
    busy_threadnum = pool->busy_thr_num;
    pthread_mutex_unlock(&(pool->thread_counter));
    return busy_threadnum;
}

int is_thread_alive(pthread_t tid)
{
    int kill_rc = pthread_kill(tid, 0);    //发 0 号信号，测试线程是否存活
    if (kill_rc == ESRCH) {
        return false;
    }

    return true;
}

/*测试*/

#ifdef 1
/* 线程池中的线程，模拟处理业务 */
void *process(void *arg)
{
    printf("thread 0x%x working on task %d\n ",(unsigned int)pthread_self(),*(int *)arg);
    sleep(1);
    printf("task %d is end\n",*(int *)arg);

    return NULL;
}

int main(void)
{
    /*threadpool_t *threadpool_create(int min_thr_num, int max_thr_num, int queue_max_size);*/

    threadpool_t *thp = threadpool_create(3,100,100);/*创建线程池，池里最小 3 个线程，最大 100，队列最大 100*/
    printf("pool initied");

    //int *num = (int *)malloc(sizeof(int)*20);
    int num[20], i;
    for (i = 0; i < 20; i++) {
        num[i]=i;
    }
}

```



```

    printf("add task %d\n",i);
    threadpool_add(thp, process, (void*)&num[i]);    /* 向线程池中添加任务 */
}
sleep(10);                                          /* 等子线程完成任务 */
threadpool_destroy(thp);

return 0;
}
#endif

```

UDP 服务器

传输层主要应用的协议模型有两种，一种是 TCP 协议，另外一种则是 UDP 协议。TCP 协议在网络通信中占主导地位，绝大多数的网络通信借助 TCP 协议完成数据传输。但 UDP 也是网络通信中不可或缺的重要通信手段。

相较于 TCP 而言，UDP 通信的形式更像是发短信。不需要在数据传输之前建立、维护连接。只专心获取数据就好。省去了三次握手的过程，通信速度可以大大提高，但与之伴随的通信的稳定性和正确率便得不到保证。因此，我们称 UDP 为“无连接的不可靠报文传递”。

那么与我们熟知的 TCP 相比，UDP 有哪些优点和不足呢？由于无需创建连接，所以 UDP 开销较小，数据传输速度快，实时性较强。多用于对实时性要求较高的通信场合，如视频会议、电话会议等。但随之也伴随着数据传输不可靠，传输数据的正确率、传输顺序和流量都得不到控制和保证。所以，通常情况下，使用 UDP 协议进行数据传输，为保证数据的正确性，我们需要在应用层添加辅助校验协议来弥补 UDP 的不足，以达到数据可靠传输的目的。

与 TCP 类似的，UDP 也有可能出现缓冲区被填满后，再接收数据时丢包的现象。由于它没有 TCP 滑动窗口的机制，通常采用如下两种方法解决：

- 1) 服务器应用层设计流量控制，控制发送数据速度。
- 2) 借助 `setsockopt` 函数改变接收缓冲区大小。如：

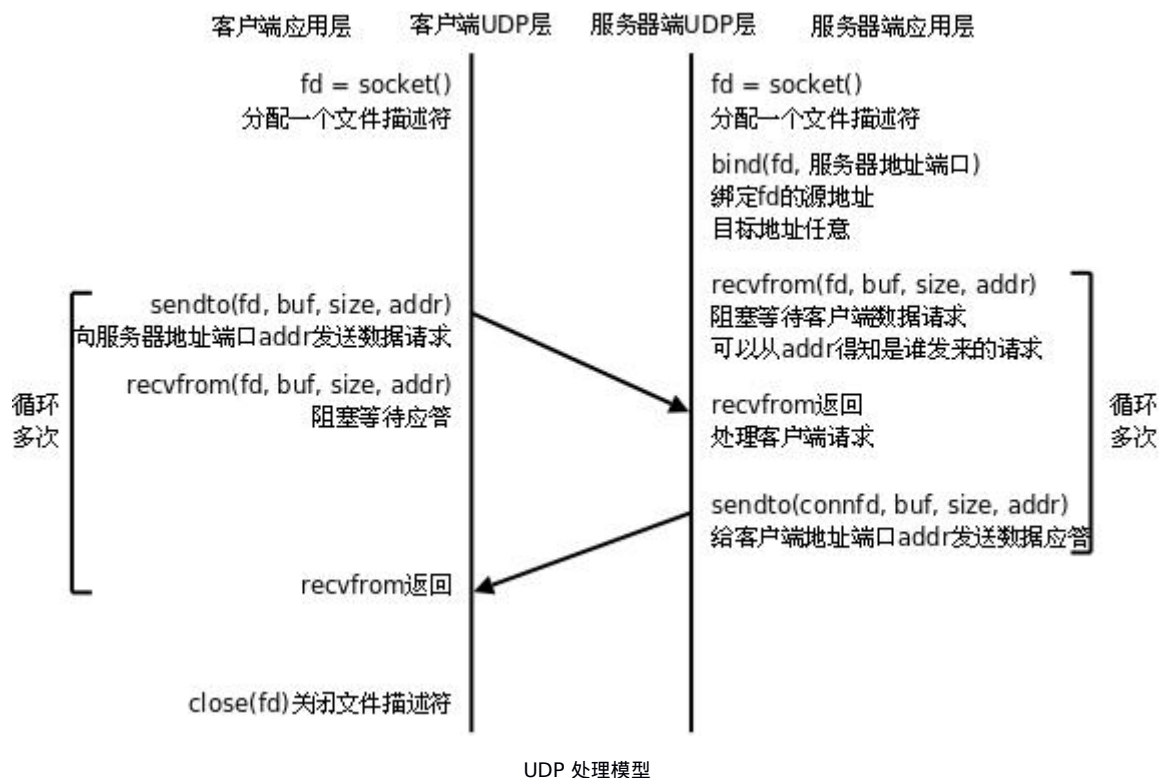
```

#include <sys/socket.h>
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);

int n = 220x1024
setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

```

C/S 模型-UDP



由于 UDP 不需要维护连接，程序逻辑简单了很多，但是 UDP 协议是不可靠的，保证通讯可靠性的机制需要在应用层实现。

编译运行 server，在两个终端里各开一个 client 与 server 交互，看看 server 是否具有并发服务的能力。用 Ctrl+C 关闭 server，然后再运行 server，看此时 client 还能否和 server 联系上。和前面 TCP 程序的运行结果相比较，体会无连接的含义。

sendto 函数

```
#include <sys/types.h>
#include <sys/socket.h>

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

第一个参数 sockfd:正在监听端口的套接口文件描述符，通过 socket 获得

第二个参数 buf：发送缓冲区，往往是使用者定义的数组，该数组装有要发送的数据

第三个参数 len:发送缓冲区的大小，单位是字节

第四个参数 flags:填 0 即可

第五个参数 dest_addr:指向接收数据的主机地址信息的结构体，也就是该参数指定数据要发送到哪个主机哪个进程

第六个参数 addrlen:表示第五个参数所指向内容的长度

返回值：成功：返回发送成功的数据长度

失败：-1

recvfrom 函数

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

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

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);
```

第一个参数 sockfd:正在监听端口的套接口文件描述符，通过 socket 获得

第二个参数 buf：接收缓冲区，往往是使用者定义的数组，该数组装有接收到的数据

第三个参数 len:接收缓冲区的大小，单位是字节

第四个参数 flags:填 0 即可

第五个参数 src_addr:指向发送数据的主机地址信息的结构体，也就是我们可以从该参数获取到数据是谁发出的

第六个参数 addrlen:表示第五个参数所指向内容的长度

返回值：成功：返回接收成功的数据长度

失败：-1

server

```
#include <string.h>
```

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
#include <arpa/inet.h>
```

```
#include <ctype.h>
```

```
#define SERV_PORT 8000
```

```
int main(void)
```

```
{
```

```
    struct sockaddr_in serv_addr, clie_addr;
```

```
    socklen_t clie_addr_len;
```

```
    int sockfd;
```

```
    char buf[BUFSIZ];
```

```
    char str[INET_ADDRSTRLEN];
```

```
    int i, n;
```

```
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
```

```
    bzero(&serv_addr, sizeof(serv_addr));
```

```
    serv_addr.sin_family = AF_INET;
```

```
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
    serv_addr.sin_port = htons(SERV_PORT);
```

```
    bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
```

```
    printf("Accepting connections ...\n");
```

```
    while (1) {
```

```
        clie_addr_len = sizeof(clie_addr);
```

```
        n = recvfrom(sockfd, buf, BUFSIZ, 0, (struct sockaddr *)&clie_addr, &clie_addr_len);
```

```
        if (n == -1)
```

```
            perror("recvfrom error");
```

```

    printf("received from %s at PORT %d\n",
           inet_ntop(AF_INET, &clie_addr.sin_addr, str, sizeof(str)),
           ntohs(clie_addr.sin_port));

    for (i = 0; i < n; i++)
        buf[i] = toupper(buf[i]);

    n = sendto(sockfd, buf, n, 0, (struct sockaddr *)&clie_addr, sizeof(clie_addr));
    if (n == -1)
        perror("sendto error");
}
close(sockfd);

return 0;
}

```

client

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <ctype.h>

#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    int sockfd, n;
    char buf[BUFSIZ];

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    while (fgets(buf, BUFSIZ, stdin) != NULL) {
        n = sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr, sizeof(servaddr));
        if (n == -1)
            perror("sendto error");

        n = recvfrom(sockfd, buf, BUFSIZ, 0, NULL, 0); //NULL:不关心对端信息
        if (n == -1)
            perror("recvfrom error");
    }
}

```

```
    write(STDOUT_FILENO, buf, n);
}

close(sockfd);

return 0;
}
```

编译运行结果如下：

A terminal window showing the execution of a UDP server and client. The server process is running in the background, displaying 'Accepting connections ...' and three 'received from 127.0.0.1 at PORT 48294' messages. In the foreground, a new terminal window shows the client process running './client', which sends the messages 'hahha', 'HAHHA', 'hello', 'HELLO', 'are you ok ?', and 'ARE YOU OK ?' to the server.

```
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/04.udp/UDP/udp_socket$ ./server
Accepting connections ...
received from 127.0.0.1 at PORT 48294
received from 127.0.0.1 at PORT 48294
received from 127.0.0.1 at PORT 48294

jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/04.udp/UDP/udp_socket$ ./client
hahha
HAHHA
hello
HELLO
are you ok ?
ARE YOU OK ?
```

广播

广播域：广播域是网络中能接收任一台主机发出的广播帧的所有主机集合。也就是说，如果广播域内的其中一台主机发出一个广播帧，同一广播域内所有的其它主机都可以收到该广播帧。

广播域的计算：如何知道一台主机是属于哪一个广播域呢？其实计算很简单，只要用主机的 IP 地址与子网掩码进行与运算即可知道该主机属于哪一个广播域。例如：一台主机的 IP 地址为 192.168.23.150，子网掩码为 255.255.255.0，那么它所属的广播域就是 $192.168.23.150 \& 255.255.255.0 = 192.168.23.0$ 。那么其它的在广播域 192.168.23.0 内的所有主机就可以到该设备发送的广播包。如果把子网掩码改为 255.255.0.0，那么它所属的广播域就是 $192.168.23.150 \& 255.255.0.0 = 192.168.0.0$ 。那么其它的在广播域 192.168.0.0 内的所有主机都可以收到该设备发送的广播包。

广播地址的计算：要想相同广播域内的其它主机能收到的广播帧，还需要在发送广播包的时候指定当前所属广播域内的广播地址。广播地址的计算方法为子网掩码取反再与广播域进行或运算。

例如：如果主机当前所属广播域为 192.168.0.0，子网掩码为 255.255.0.0，那么广播地址则为

192.168.255.255。

使用 UDP 进行跨网段广播：要使主机 A 发送的广播包能够被另一网段的主机 B 收到，那么只需要更改主机 A 的子网掩码使得与主机 B 在同一个广播域内，再使用新的广播域的广播地址发送广播包即可。

例如：要使用 192.168.23.150 发送广播包让 192.168.27.135 收到，只需要设置 192.168.23.150 的子网掩码为 255.255.0.0，然后再使用广播地址 192.168.255.255 即可。

server

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <string.h>
#include <arpa/inet.h>
#include <net/if.h>

#define SERVER_PORT 8000 /* 无关紧要 */
#define MAXLINE 1500

#define BROADCAST_IP "192.168.1.255"
#define CLIENT_PORT 9000 /* 重要 */

int main(void)
{
    int sockfd;
    struct sockaddr_in serveraddr, clientaddr;
    char buf[MAXLINE];

    /* 构造用于 UDP 通信的套接字 */
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET; /* IPv4 */
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY); /* 本地任意 IP INADDR_ANY = 0 */
    serveraddr.sin_port = htons(SERVER_PORT);

    bind(sockfd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));

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

    /*构造 client 地址 IP+端口 192.168.7.255+9000 */
    bzero(&clientaddr, sizeof(clientaddr));
```

```

clientaddr.sin_family = AF_INET;
inet_pton(AF_INET, BROADCAST_IP, &clientaddr.sin_addr.s_addr);
clientaddr.sin_port = htons(CLIENT_PORT);

int i = 0;
while (1) {
    sprintf(buf, "Drink %d glasses of water\n", i++);
    //fgets(buf, sizeof(buf), stdin);
    sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr *)&clientaddr, sizeof(clientaddr));
    sleep(1);
}
close(sockfd);
return 0;
}

```

Client

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define SERVER_PORT 8000
#define MAXLINE 4096
#define CLIENT_PORT 9000

int main(int argc, char *argv[])
{
    struct sockaddr_in localaddr;
    int confd;
    ssize_t len;
    char buf[MAXLINE];

    //1.创建一个 socket
    confd = socket(AF_INET, SOCK_DGRAM, 0);

    //2.初始化本地端地址
    bzero(&localaddr, sizeof(localaddr));
    localaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "0.0.0.0", &localaddr.sin_addr.s_addr);
    localaddr.sin_port = htons(CLIENT_PORT);

    int ret = bind(confd, (struct sockaddr *)&localaddr, sizeof(localaddr)); //显示绑定不能省略
    if (ret == 0)
        printf("...bind ok...\n");
}

```



```

while (1) {
    len = recvfrom(confd, buf, sizeof(buf), 0, NULL, 0);
    write(STDOUT_FILENO, buf, len);
}
close(confd);

return 0;
}

```

编译运行结果如下：

```

/LinuxCode-master/网络编程/04.udp/UDP/broadcast$ ./server
jack@jack-Ubuntu: ~/workspace/LinuxCode-master/网络编程/04.udp/UDP
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/04.udp/UDP
...bind ok...
Drink 40 glasses of water
Drink 41 glasses of water
Drink 42 glasses of water
Drink 43 glasses of water

```

服务器使用：

```

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

```

设置以广播方式发送，使用：

```

/*构造 client 地址 IP+端口 192.168.7.255+9000 */
bzero(&clientaddr, sizeof(clientaddr));
clientaddr.sin_family = AF_INET;
inet_pton(AF_INET, BROADCAST_IP, &clientaddr.sin_addr.s_addr);
clientaddr.sin_port = htons(CLIENT_PORT);

```

构建广播的地址和接受的端口号，这里主机的广播地址和子网掩码为：

```

wlan0    Link encap:以太网  硬件地址 d0:53:49:15:09:ef
         inet 地址:192.168.1.105  广播:192.168.1.255  掩码:255.255.255.0
         inet6 地址: fe80::d253:49ff:fe15:9ef/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
         接收数据包:22520  错误:0  丢弃:0  过载:0  帧数:0
         发送数据包:18185  错误:0  丢弃:0  过载:0  载波:0
         碰撞:0  发送队列长度:1000
         接收字节:25244613 (25.2 MB)  发送字节:2681303 (2.6 MB)

```

所以局域网内 IP 地址为 192.168.1.xxx 的客户端可以收到这个广播，但需要监听端口 9000：

```

bzero(&localaddr, sizeof(localaddr));
localaddr.sin_family = AF_INET;
inet_pton(AF_INET, "0.0.0.0", &localaddr.sin_addr.s_addr);

```



```
localaddr.sin_port = htons(CLIENT_PORT);
```

```
int ret = bind(confd, (struct sockaddr*)&localaddr, sizeof(localaddr)); //显示绑定不能省略
```

多播(组播)

组播组可以是永久的也可以是临时的。组播组地址中，有一部分由官方分配的，称为永久组播组。永久组播组保持不变的是它的 ip 地址，组中的成员构成可以发生变化。永久组播组中成员的数量都可以是任意的，甚至可以为零。那些没有保留下来供永久组播组使用的 ip 组播地址，可以被临时组播组利用。

224.0.0.0 ~ 224.0.0.255	为预留的组播地址（永久组地址），地址 224.0.0.0 保留不做分配，其它地址供路由协议使用；
224.0.1.0 ~ 224.0.1.255	是公用组播地址，可以用于 Internet；欲使用需申请。
224.0.2.0 ~ 238.255.255.255	为用户可用的组播地址（临时组地址），全网范围内有效；
239.0.0.0 ~ 239.255.255.255	为本地管理组播地址，仅在特定的本地范围内有效。

可使用 ip ad 命令查看网卡编号，如：

```
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/04.udp/UDP/multicast$ ip ad
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN group default qlen 1000
    link/ether 08:62:66:56:10:b9 brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether d0:53:49:15:09:ef brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.105/24 brd 192.168.1.255 scope global wlan0
        valid_lft forever preferred_lft forever
    inet6 fe80::d253:49ff:fe15:9ef/64 scope link
        valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:d0:f6:0e:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
        valid_lft forever preferred_lft forever
```

结构 ip_mreqn

```
struct ip_mreqn
{
    struct in_addr imr_multiaddr; /*多播组的 IP 地址*/
    struct in_addr imr_address; /*本地址网络接口的 IP 地址*/
    int imr_ifindex; /*网络接口序号*/
}
```

该结构体的两个成员分别用于指定所加入的多播组的组 IP 地址，和所要加入组的那个本地接口的 IP 地址。该命令字没有源过滤的功能，它相当于实现 IGMPv1 的多播加入服务接口。

函数 if_nametoindex()

```
unsigned if_nametoindex(const char *ifname);
```

if_nametoindex 可以根据网卡名，获取网卡序号。指定网络接口名称字符串作为参数；若该接口存在，则返回相应的索引，否则返回 0

server

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <net/if.h>

#define SERVER_PORT 8000
#define CLIENT_PORT 9000
#define MAXLINE 1500

#define GROUP "239.0.0.2"

int main(void)
{
    int sockfd;
    struct sockaddr_in serveraddr, clientaddr;
    char buf[MAXLINE] = "scut_whut\n";
    struct ip_mreqn group;

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);          /* 构造用于 UDP 通信的套接字 */

    bzero(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;                  /* IPv4 */
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);   /* 本地任意 IP INADDR_ANY = 0 */
    serveraddr.sin_port = htons(SERVER_PORT);

    bind(sockfd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));

    inet_pton(AF_INET, GROUP, &group.imr_multiaddr); /* 设置组地址 */
    inet_pton(AF_INET, "0.0.0.0", &group.imr_address); /* 本地任意 IP */
    group.imr_ifindex = if_nametoindex("eth0");        /* 给出网卡名,转换为对应编号: eth0 --> 编号 命令:ip ad */

    setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_IF, &group, sizeof(group)); /* 组播权限 */

    bzero(&clientaddr, sizeof(clientaddr));           /* 构造 client 地址 IP+端口 */
    clientaddr.sin_family = AF_INET;
    inet_pton(AF_INET, GROUP, &clientaddr.sin_addr.s_addr); /* IPv4 239.0.0.2+9000 */
    clientaddr.sin_port = htons(CLIENT_PORT);

    int i = 0;
```

```

while (1) {
    sprintf(buf, "scut_whut%d\n", i++);
    //fgets(buf, sizeof(buf), stdin);
    sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr *)&clientaddr, sizeof(clientaddr));
    sleep(1);
}

close(sockfd);

return 0;
}

```

client

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <net/if.h>

#define SERVER_PORT 8000
#define CLIENT_PORT 9000

#define GROUP "239.0.0.2"

int main(int argc, char *argv[])
{
    struct sockaddr_in localaddr;
    int confd;
    ssize_t len;
    char buf[BUFSIZ];

    struct ip_mreqn group;                                /* 组播结构体 */

    confd = socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&localaddr, sizeof(localaddr));                /* 初始化 */
    localaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "0.0.0.0", &localaddr.sin_addr.s_addr);
    localaddr.sin_port = htons(CLIENT_PORT);

    bind(confd, (struct sockaddr *)&localaddr, sizeof(localaddr));

    inet_pton(AF_INET, GROUP, &group.imr_multiaddr);      /* 设置组地址 */
    inet_pton(AF_INET, "0.0.0.0", &group.imr_address);    /* 使用本地任意 IP 添加到组播组 */
    group.imr_ifindex = if_nametoindex("eth0");           /* 通过网卡名-->编号 ip ad */


    setsockopt(confd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &group, sizeof(group)); /* 设置 client 加入多播组 */
}

```

```
while (1) {
    len = recvfrom(confd, buf, sizeof(buf), 0, NULL, 0);
    write(STDOUT_FILENO, buf, len);
}
close(confd);

return 0;
}
```

编译运行结果如下：

A terminal window with a dark background. The prompt is 'jack@jack-Ubuntu: ~/workspace/LinuxCode-master/网络编程/04.udp/UDP/multicast\$'. The user has run './server' and then './client'. The client outputs a list of scut numbers: scut_whut9, scut_whut10, scut_whut11, scut_whut12, scut_whut13, scut_whut14, and scut_whut15.

```
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/04.udp/UDP/multicast$ ./server
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/04.udp/UDP/multicast$ ./client
scut_whut9
scut_whut10
scut_whut11
scut_whut12
scut_whut13
scut_whut14
scut_whut15
```

socket IPC (本地套接字 domain)

socket API 原本是为网络通讯设计的，但后来在 socket 的框架上发展出一种 IPC 机制，就是 UNIX Domain Socket。虽然网络 socket 也可用于同一台主机的进程间通讯（通过 loopback 地址 127.0.0.1），但是 UNIX Domain Socket 用于 IPC 更有效率：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。这是因为，IPC 机制本质上是可靠的通讯，而网络协议是为不可靠的通讯设计的。UNIX Domain Socket 也提供面向流和面向数据包两种 API 接口，类似于 TCP 和 UDP，但是面向消息的 UNIX Domain Socket 也是可靠的，消息既不会丢失也不会顺序错乱。

UNIX Domain Socket 是全双工的，API 接口语义丰富，相比其它 IPC 机制有明显的优越性，目前已成为使用最广泛的 IPC 机制，比如 X Window 服务器和 GUI 程序之间就是通过 UNIXDomain Socket 通讯的。

使用 UNIX Domain Socket 的过程和网络 socket 十分相似，也要先调用 socket() 创建一个 socket 文件描述符，address family 指定为 AF_UNIX，type 可以选择 SOCK_DGRAM 或 SOCK_STREAM，protocol 参数仍然指定为 0 即可。

UNIX Domain Socket 与网络 socket 编程最明显的不同在于地址格式不同，用结构体 `sockaddr_un` 表示，网络编程的 socket 地址是 IP 地址加端口号，而 UNIX Domain Socket 的地址是一个 socket 类型的文件在文件系统中的路径，这个 socket 文件由 `bind()`调用创建，如果调用 `bind()`时该文件已存在，则 `bind()`错误返回。

对比网络套接字地址结构和本地套接字地址结构：

```
struct sockaddr_in {
    __kernel_sa_family_t sin_family;          /* Address family */    地址结构类型
    __be16 sin_port;                            /* Port number */      端口号
    struct in_addr sin_addr;                    /* Internet address */ IP 地址
};
struct sockaddr_un {
    __kernel_sa_family_t sun_family;          /* AF_UNIX */          地址结构类型
    char sun_path[UNIX_PATH_MAX];             /* pathname */         socket 文件名(含路径)
};
```

以下程序将 UNIX Domain socket 绑定到一个地址。

```
size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
#define offsetof(type, member) ((int)&((type *)0)->MEMBER)
```

server

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <strings.h>
#include <string.h>
#include <ctype.h>
#include <arpa/inet.h>
#include <sys/un.h>
#include <stddef.h>

#include "wrap.h"

#define SERV_ADDR "serv.socket"

int main(void)
{
    int lfd, cfd, len, size, i;
```

```

struct sockaddr_un servaddr, cliaddr;
char buf[4096];

lfd = Socket(AF_UNIX, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_UNIX;
strcpy(servaddr.sun_path, SERV_ADDR);

len = offsetof(struct sockaddr_un, sun_path) + strlen(servaddr.sun_path);    /* servaddr total len */

unlink(SERV_ADDR);                                /* 确保 bind 之前 serv.sock 文件不存在,bind 会创建该文件 */
Bind(lfd, (struct sockaddr *)&servaddr, len);      /* 参 3 不能是 sizeof(servaddr) */

Listen(lfd, 20);

printf("Accept ...\n");
while (1) {
    len = sizeof(cliaddr);
    cfd = Accept(lfd, (struct sockaddr *)&cliaddr, (socklen_t *)&len);

    len -= offsetof(struct sockaddr_un, sun_path);    /* 得到文件名的长度 */
    cliaddr.sun_path[len] = '\0';                    /* 确保打印时,没有乱码出现 */

    printf("client bind filename %s\n", cliaddr.sun_path);

    while ((size = read(cfd, buf, sizeof(buf))) > 0) {
        for (i = 0; i < size; i++)
            buf[i] = toupper(buf[i]);
        write(cfd, buf, size);
    }
    close(cfd);
}
close(lfd);

return 0;
}

```

client

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <strings.h>
#include <string.h>
#include <ctype.h>

```

```

#include <arpa/inet.h>
#include <sys/un.h>
#include <stddef.h>

#include "wrap.h"

#define SERV_ADDR "serv.socket"
#define CLIE_ADDR "clie.socket"

int main(void)
{
    int  cfd, len;
    struct sockaddr_un servaddr, cliaddr;
    char buf[4096];

    cfd = Socket(AF_UNIX, SOCK_STREAM, 0);

    bzero(&cliaddr, sizeof(cliaddr));
    cliaddr.sun_family = AF_UNIX;
    strcpy(cliaddr.sun_path, CLIE_ADDR);

    len = offsetof(struct sockaddr_un, sun_path) + strlen(cliaddr.sun_path);    /* 计算客户端地址结构有效长度 */

    unlink(CLIE_ADDR);
    Bind(cfd, (struct sockaddr *)&cliaddr, len);                                /* 客户端也需要 bind, 不能依赖自动绑定 */
*/

    bzero(&servaddr, sizeof(servaddr));                                         /* 构造 server 地址 */
    servaddr.sun_family = AF_UNIX;
    strcpy(servaddr.sun_path, SERV_ADDR);

    len = offsetof(struct sockaddr_un, sun_path) + strlen(servaddr.sun_path);  /* 计算服务器端地址结构有效长度 */

    Connect(cfd, (struct sockaddr *)&servaddr, len);

    while (fgets(buf, sizeof(buf), stdin) != NULL) {
        write(cfd, buf, strlen(buf));
        len = read(cfd, buf, sizeof(buf));
        write(STDOUT_FILENO, buf, len);
    }

    close(cfd);

    return 0;
}

```

Makefile

```
src = $(wildcard *.c)
obj = $(patsubst %.c, %.o, $(src))
soktfile = $(wildcard *.socket)

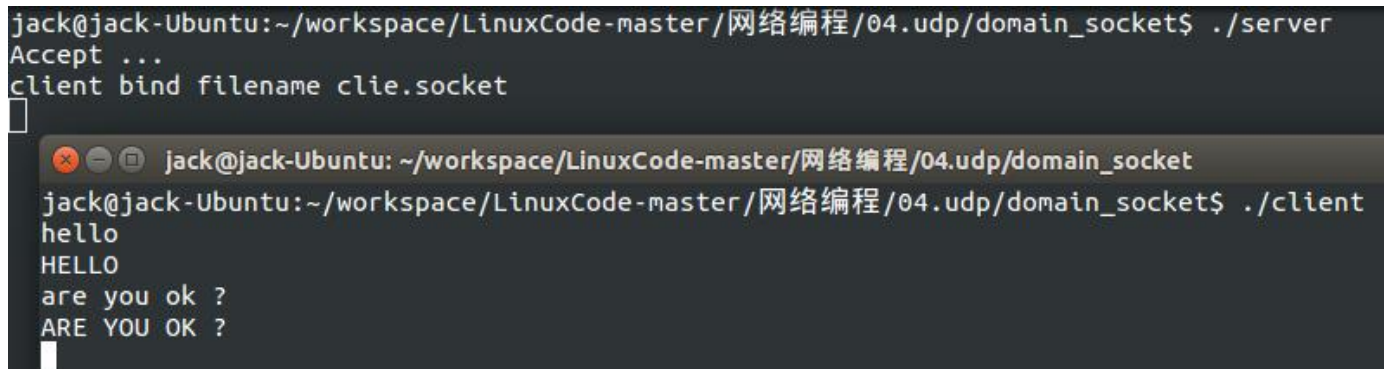
all: server client

server: server.o wrap.o
    gcc server.o wrap.o -o server -Wall
client: client.o wrap.o
    gcc client.o wrap.o -o client -Wall

%.o:%.c
    gcc -c $< -Wall

.PHONY: clean all
clean:
    -rm -rf server client $(obj) $(soktfile)
```

编译运行结果如下：



```
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/04.udp/domain_socket$ ./server
Accept ...
client bind filename clie.socket
jack@jack-Ubuntu: ~/workspace/LinuxCode-master/网络编程/04.udp/domain_socket
jack@jack-Ubuntu:~/workspace/LinuxCode-master/网络编程/04.udp/domain_socket$ ./client
hello
HELLO
are you ok ?
ARE YOU OK ?
```

其它常用函数

名字与地址转换

gethostbyname 根据给定的主机名，获取主机信息。

过时，仅用于 IPv4，且线程不安全。

```
#include <stdio.h>
#include <netdb.h>
#include <arpa/inet.h>

extern int h_errno;
```



```

int main(int argc, char *argv[])
{
    struct hostent *host;
    char str[128];
    host = gethostbyname(argv[1]);
    printf("%s\n", host->h_name);

    while (*(host->h_aliases) != NULL)
        printf("%s\n", *host->h_aliases++);

    switch (host->h_addrtype) {
        case AF_INET:
            while (*(host->h_addr_list) != NULL)
                printf("%s\n", inet_ntop(AF_INET, (*host->h_addr_list++), str, sizeof(str)));
            break;
        default:
            printf("unknown address type\n");
            break;
    }
    return 0;
}

```

gethostbyaddr 函数。

此函数只能获取域名解析服务器的 url 和/etc/hosts 里登记的 IP 对应的域名。

```

#include <stdio.h>
#include <netdb.h>
#include <arpa/inet.h>

extern int h_errno;

int main(int argc, char *argv[])
{
    struct hostent *host;
    char str[128];
    struct in_addr addr;

    inet_pton(AF_INET, argv[1], &addr);
    host = gethostbyaddr((char *)&addr, 4, AF_INET);
    printf("%s\n", host->h_name);

    while (*(host->h_aliases) != NULL)
        printf("%s\n", *host->h_aliases++);
    switch (host->h_addrtype) {
        case AF_INET:
            while (*(host->h_addr_list) != NULL)

```

```
        printf("%s\n", inet_ntop(AF_INET, (*host->h_addr_list++), str, sizeof(str)));
        break;
    default:
        printf("unknown address type\n");
        break;
    }
    return 0;
}
```

getservbyname

getservbyport

根据服务程序名字或端口号获取信息。使用频率不高。

getaddrinfo

getnameinfo

freeaddrinfo

可同时处理 IPv4 和 IPv6，线程安全的。

套接口和地址关联

getsockname

根据 accpet 返回的 sockfd，得到临时端口号

getpeername

根据 accpet 返回的 sockfd，得到远端链接的端口号，在 exec 后可以获取客户端信息。