Linux 网络编程

计算机网络中实现通信必须有一些约定，这些约定即被称为通信协议，如对速率、传输、代码、代码结构、传输控制步骤、出错控制等的约定。也就是说，为了在两个节点之间成功地进行通信，两个节点之间必须约定使用共同的“语言”。这些被通信各方共同遵循的约定、语言、规矩，有时又被称为协议(protocol)。

1. 网络基础

1.1 网络分层模型

最为通用的网络协议是TCP/IP协议，TCP/IP是“transmission Control Protocol/Internet Protocol”的简写，中文译名为传输控制协议/互联网络协议， 它规范了网络上的所有通信设备，尤其是一个主机与另一个主机之间的数据往来格式以及传送方式。

- TCP/IP模型

  TCP/IP 分不同层次进行开发，每一层分别负责不同的通信功能。一个协议族，比如TCP/IP，是一组不同层次上的多个协议的组合。TCP/IP通常被认为是一个四层协议系统，每一层负责不同的功能：

（1）链路层：有时也称作数据链路层或 网络接口层，通常包括操作系统中的设备驱动程序和计算机中对应的网络接口卡。它们一起处理与电缆的物理接口细节。

（2）网络层：有时也称作互联网层，例如分组的选路。在TCP/IP协议族中，网络层协议包括IP协议（网际协议），ICMP协议（Internet Control Message Protocol： Internet互联网控制报文协议），以及IGMP协议(Internet Group Management Protocol： Internet组管理协议)，以及ARP（Address Resolution Protocol 地址解析协议）与RARP（Reverse Address Resolution Protocol） 反向地址解析协议;

（3）传输层：主要为两台主机上的应用程序提供端到端的通信。在TCP/IP协议族中，有两个互不相同的传输协议：TCP (Transmission Control Protocol传输控制协议)和UDP(User Datagram Protocol用户数据报协议)。

（4）应用层：负责处理特定的应用程序细节。

- OSI模型

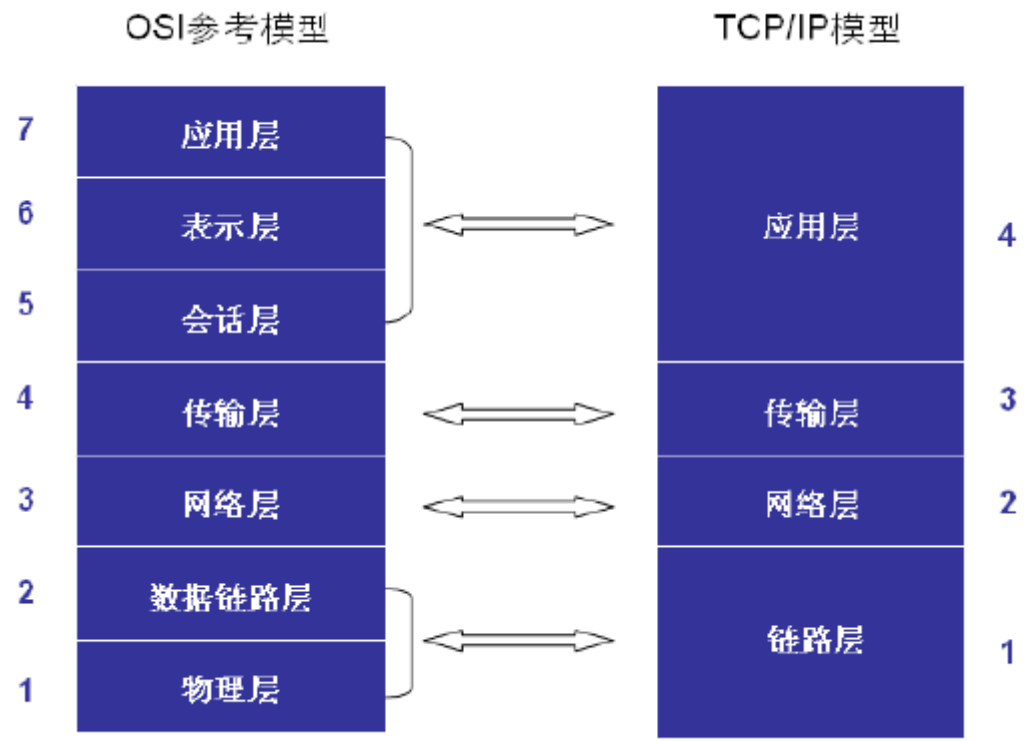
  1978年，国际标准化组织(ISO：Open System Interconnection)开发了开放式系统互联参考模型，以促进计算机系统的开放互联。开放式互联就是可在多个厂家的环境中支持互联。

该模型为计算机间开放式通信所需要定义的功能层次建立了全球的标准。

OSI模型将通信会话需要的各种进程划分成7个相对独立的功能层次，这些层次的组织是：

具体7层	功能与连接方式
应用层	网络服务与使用者应用程序间的一个接口。
表示层	数据表示、数据安全、数据压缩。
会话层	是建立在传输层之上，利用传输层提供的服务，使应用建立和维持会话，并能使会话获得同步。
传输层	在系统之间提供可靠的、透明的数据传送，提供端到端的错误恢复和流控制。（端口号）
网络层	基于网络层地址（IP地址）进行不同网络系统间的路径选择。
数据链路层	在物理层上建立、维持、释放数据链接链接以及差错校验等功能，通过使用接收系统的硬件地址或物理地址来寻址。
物理层	网络硬件设备之间的接口。

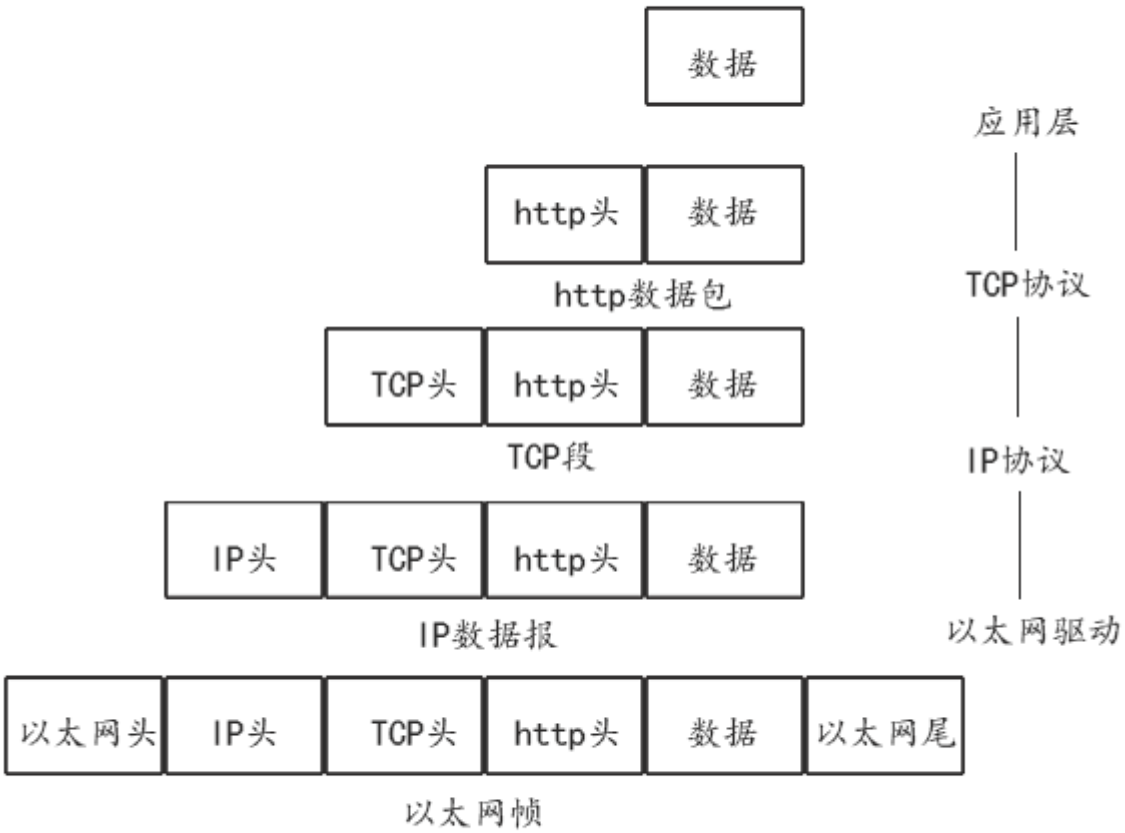
- TCP/IP参考模型比OSI模型更灵活



1.2 数据的封装和拆装

位于TCP/IP四层模型各个层的数据通常用一个公共的机制来封装：定义描述元信息和数据包的部分真实信息的报头的协议，这些元信息可以是数据源、目的地和其他的附加属性。当信息在不同的层之间传递时，都会在每一个层被封装上协议的特有头部。而当我们收到数据时会一层层的剥离头部，直至取出数据。

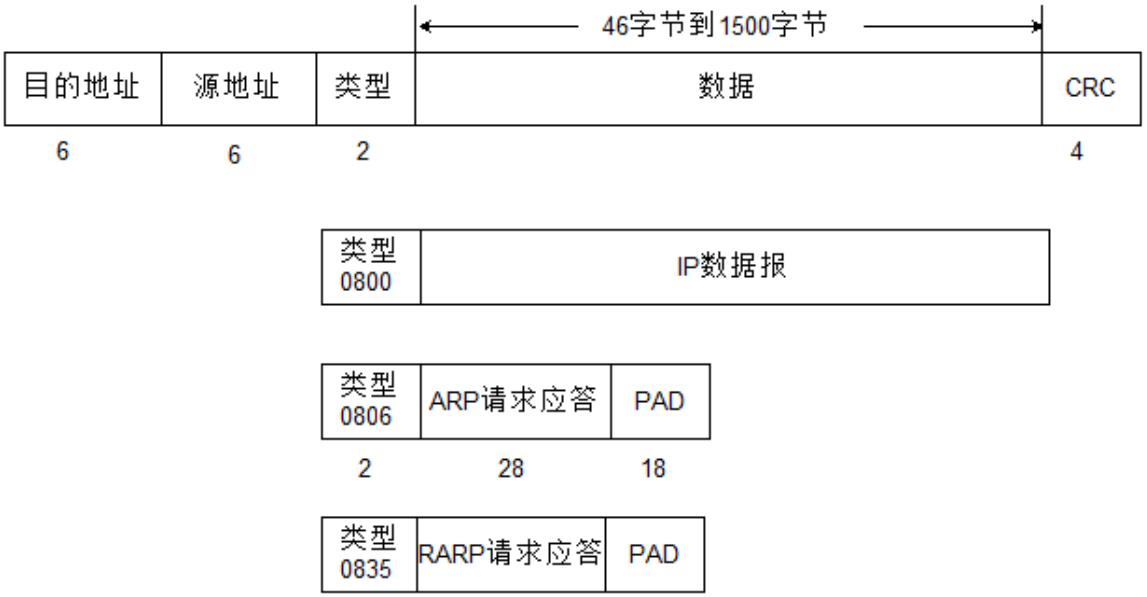
比如以HTTP协议发送的数据包为例：



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

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。用ifconfig命令看一下，“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。帧末尾是CRC校验码。

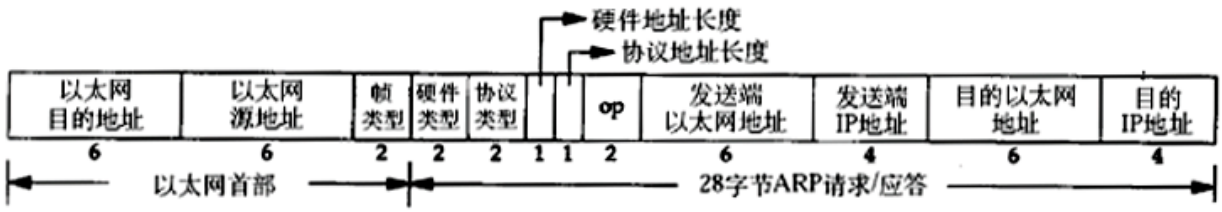


图为以太网帧

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了，则需要对数据包进行分片（fragmentation）ifconfig命令的输出中也有“MTU:1500”。注意，MTU这个概念指数据帧中有效载荷的最大长度，不包括帧首部的长度。

- ARP（Address Resolution Protocol）地址解析协议

> 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。



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。

`arp -a`用于查看缓存中的所有项目。

1.3 IP协议

IP协议是最为通用的网络层协议。所有的TCP、UDP数据都以IP数据报文的格式传输。

IP协议是一个无连接、不可靠的协议。不可靠的意思是它不能保证IP数据报能成功地到达目的地。IP仅提供最好的传输服务，如果发生某种错误时，如中途某个路由器故障，IP有一个简单的错误处理算法：丢弃该数据报，然后发送ICMP消息报给信息源端。

任何可靠性要求必须由传输层来提供如(TCP)。无连接这个术语的意思是IP并不维护任何关于后续数据报的状态信息。每个数据报的处理是相互独立的。这也说明，IP数据报可以不按发送顺序接收。

如果一“信源”向同一个“信宿”发送两个连续的数据报文（先发A，然后发B），每个数据报都是独立进行路由选择，可能选择不同的路线，因此B可能在A到达之前先到达。

- IP段

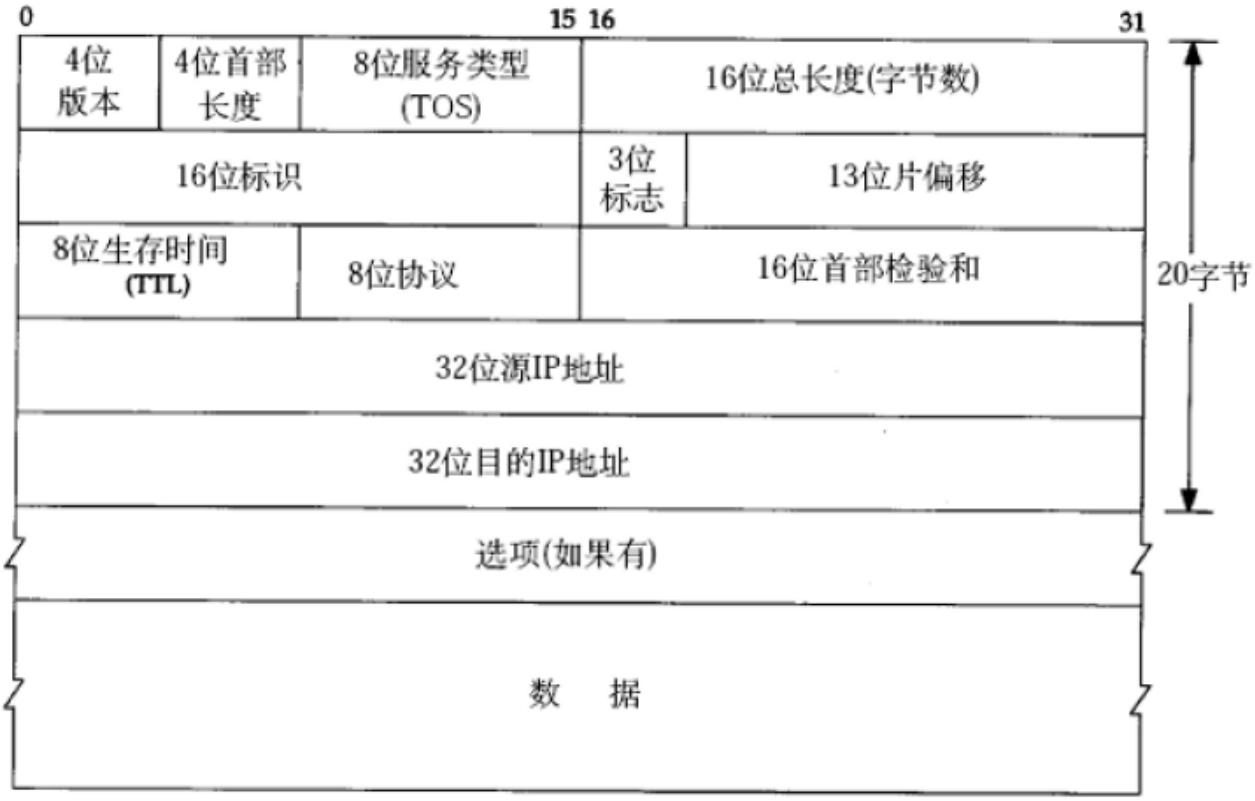


图 10.9: IP数据报格式

1.4 TCP/UDP协议

- TCP协议

TCP提供一种面向连接的、可靠的字节流服务，它位于TCP/IP模型的传输层。面向连接意味着两个使用TCP的应用（通常是一个客户端和一个服务器）在彼此交换数据之前必须先建立一个TCP连接。

- TCP通过下列方式来提供可靠性：

1、应用数据被分割成TCP认为最合适发送的数据块。由TCP传递给IP层的信息单位称为报文段(segment)。当TCP发出一个段后，它启动一个定时器，等待接收端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

2、当TCP收到发自TCP连接另一端的数据的时候，它将发送一个确认（这个确认不被立即发送，通常将推迟几百毫秒，尽可能的和数据一起发送）。

3、TCP通过校验和的形式，提供对TCP首部和TCP数据的基本校验功能。如果接收端计算出来的校验和，与数据包中的校验和不相等，TCP将丢弃这个报文段，发送端将超时并重发。

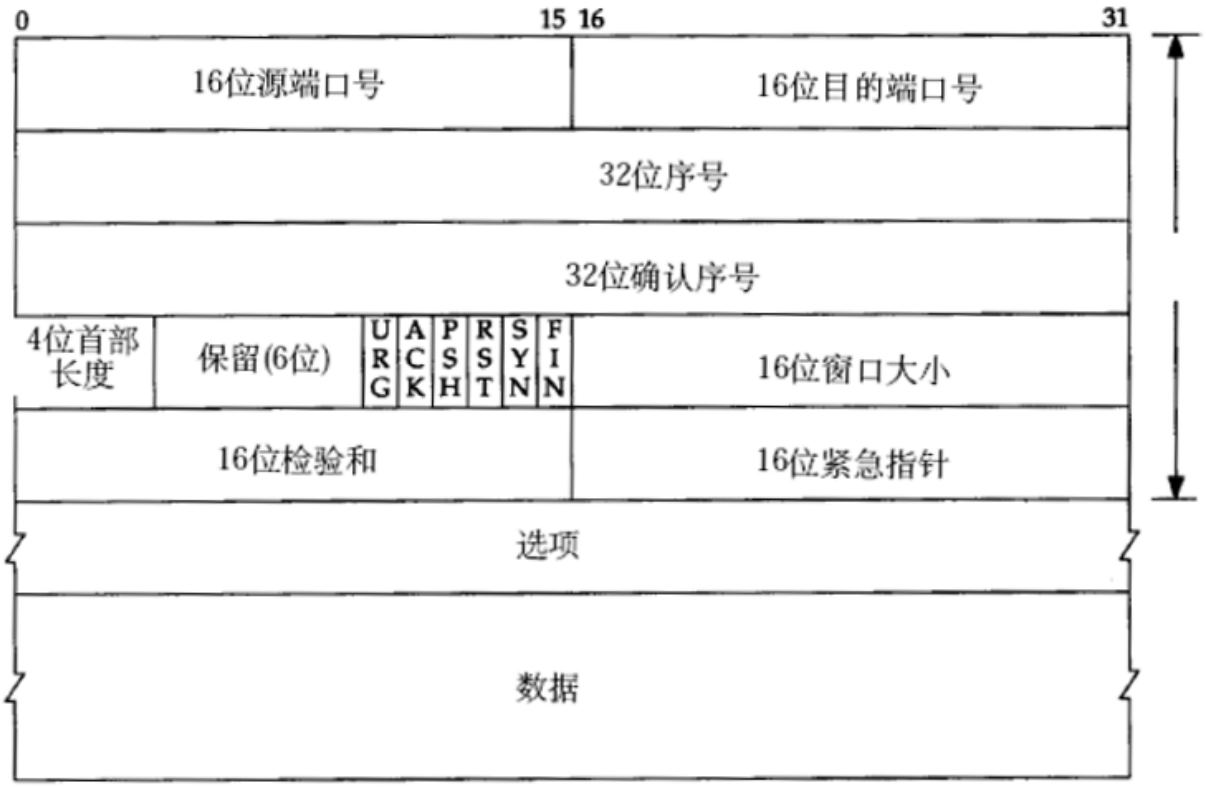
4、既然TCP报文段作为IP数据报来传输，而IP数据报的到达可能会失序，因此TCP报文也会失序。

5、既然IP数据报会发生重复，TCP的接收端必须丢弃重复的数据。

6、TCP还能提供流量控制，TCP连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许另一端发送接收端缓冲区所能接纳的数据。

综上所述，TCP是一个较为可靠的数据传输协议，但是TCP确认的数据不能保证被应用层收到，比如，当TCP确认后的数据已放入套接字缓冲区中，而此时恰巧应用进程非正常退出，所以编写一个好的网络程序，我们需要注意许多细节。

- TCP 数据包



- UDP协议

  UDP与TCP之间存在本质差异，UDP是无连接的、不可靠的数据报协议。

- UDP数据包格式

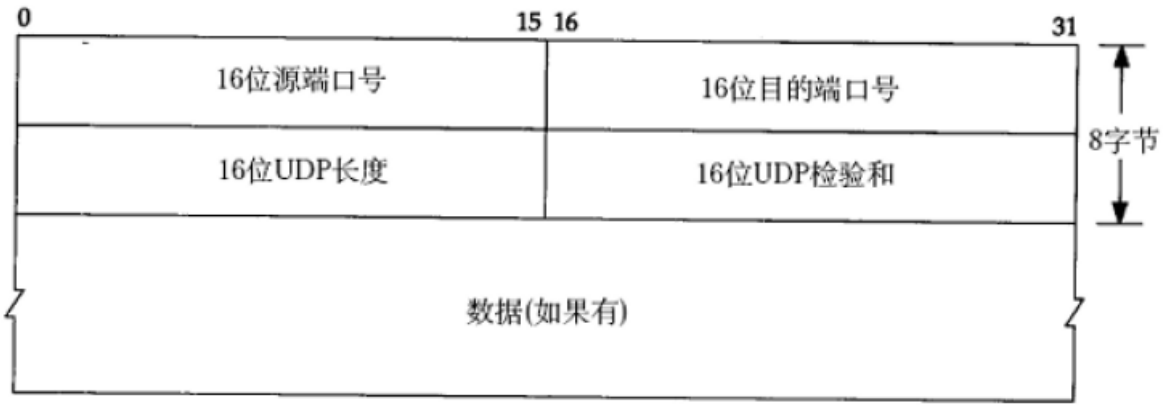
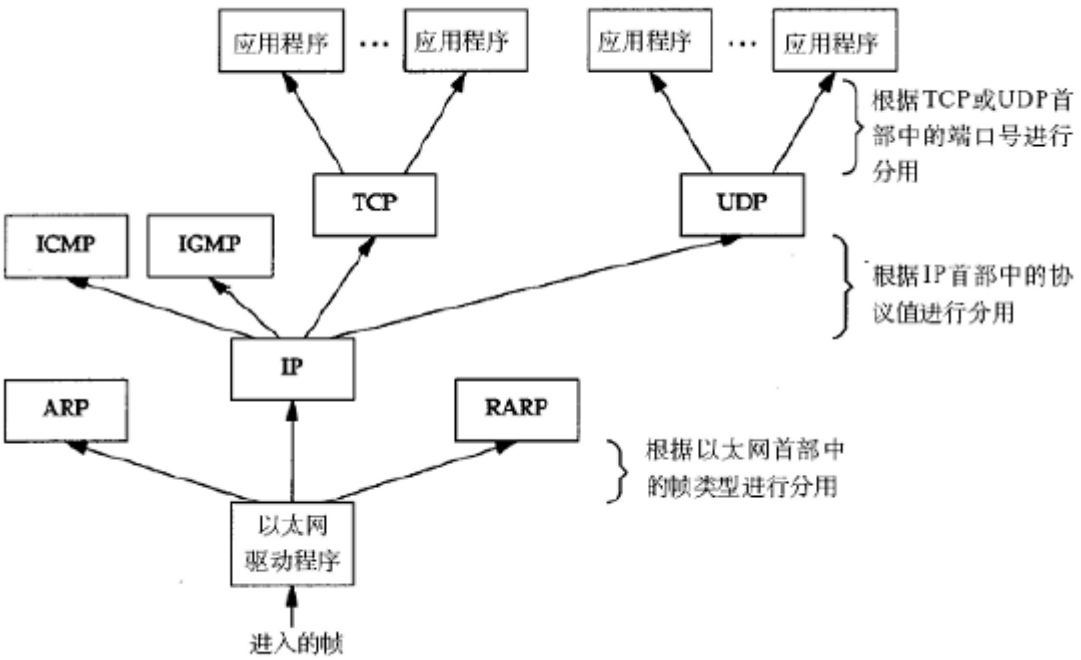


图 10.10: UDP数据段

- 所有协议的分层



1.5 IP地址

我们知道，在Internet中，有成千上万的计算机进行通信，那么，计算机和计算机之间进行通信的时候如何找到对方呢。

在网络中，计算机的唯一性是通过IP地址来标识的。也就是说，为了实现Internet上不同计算机之间的通信，每台计算机都必须有一个不与其他计算机重复的地址，即IP地址。

连接到Internet上的每一台主机都有一个或多个IP地址，它是在Internet的所有计算机中唯一标识该计算机的一个32位的无符号整数，这个整数可表示4294967296个地址。这样巨大的数字很难记忆，因此为了反映这些整数作为地址的特征以便于记忆，人们使用一种点分十进制的表示法来书写IP地址。

标准点分十进制表示法将32为的地址按8位一组分成4组，每一个8位的数均用十进制的数表示，所产生的4组数字之间用"."连接在一起。如果`192.168.2.1`它对应的无符号整数是`0xc0a80201`

11000000 (0xc0)	**10101000 (0xa8)**	**00000010 (0x02)**	**00000001 (0x01)**
192	168	2	1

Internet委员会定义了5种IP地址类型以适合不同容量的网络，即A类~E类。

其中A、B、C类（如下表格）由InternetNIC在全球范围内统一分配，D、E类为特殊地址。

类别	最大网络数	**IP**地址范围	最大主机数	私有**IP**地址范围
A	126（2^7-2)	0.0.0.0 - 127.255.255.255	16777214	10.0.0.0 - 10.255.255.255
B	16384(2^14)	128.0.0.0-191.255.255.255	65534	172.16.0.0 - 172.31.255.255
C	2097152(2^21)	192.0.0.0-223.255.255.255	254	192.168.0.0 - 192.168.255.255

D类地址和E类地址。这两类地址用途比较特殊，在这里只是简单介绍一下：D类地址称为广播地址，供特殊协议向选定的节点发送信息时用。E类地址保留给将来使用。

1.6 服务和端口号

Internet通信域中套接字地址由主机的IP地址加上端口号组成，IP地址用来标识Internet中唯一的主机，端口号则用来区别同一台主机中不同的服务程序。同一台计算机中每一个网络服务器程序使用不同的端口号，因为可以有很多个服务程序，因此我们需要告诉计算机数据应传递给哪个服务程序。

端口号是一个16位无符号整数，每一台计算机可以有65535个端口号（0端口号被保留）。Internet中 大部分计算机中相同的服务程序使用相同的端口号，这些端口号是“知名的”，ftp文件传输服务程序使用端口号21，telnet使用端口号23。

Linux中，小于1024的端口号保留用于标准的服务程序，它们也称之为特权端口号，因为只有root用户执行的服务程序才能使用它们。特权端口号的这种特征可以防止普通用户用任意的服务程序从知名端口号接收数据获取他人重要的信息。

Linux系统中有一个记录“知名”服务的配置文件，这个配置文件通常命名位/etc/services，它里面记录了IANA(Internet Assigned Numbers Authority)因特网号码指派管理局“知名端口号”与服务的对应关系。

1.7 域名

在Internet中，对于主机的IP地址，除了用点分十进制表示之外，也可以使用域名表示。域名的优点是它更便于记忆。例如，Internet地址: 59.151.21.100也可以记为www.google.cn。

在Internet中使用应用范围或地域来划分域名的后缀，例如edu代表教育，com代表商业机构，org代表非营利性组织，gov代表政府机构。世界上的其他国家都使用两个字母的国家代码作为最右端的域，例如cn代表中国，jp代表日本。

Linux系统内部都用一个数据库来记住主机名与主机IP地址之间的映射，这一数据库要么是/etc/hosts文件，要么由DNS服务系统提供，在互联网中，域名系统(DNS)是由很多层分布在互联网中的域名数据库组成，由它来提供IP地址和主机名之间的映射信息，进行域名到IP地址之间的转换。

1.8 子网掩码

子网掩码只有一个作用，就是将某个IP地址划分成网络地址和主机地址两部分。

子网掩码——屏蔽一个IP地址的网络部分的“全1”比特模式。对于A类地址来说，默认的子网掩码是255.0.0.0；对于B类地址来说默认的子网掩码是255.255.0.0；对于C类地址来说默认的子网掩码是255.255.255.0。

例如：192.168.1.22的ip地址，它的子网掩码是255.255.255.0，那么它的网络地址就是192.168.1，它的主机地址是22。

1.9 网关

大家都知道，从一个房间走到另一个房间，必然要经过一扇门。同样，从一个网络向另一个网络发送信网关息，也必须经过一道“关口”，这道关口就是网关。顾名思义，网关（Gateway）就是一个网络连接到另一个网络的“关口”。也就是网络关卡。

1.20 路由器与交换机

交换机工作在数据链路层，交换机建立起来的局域网，通过MAC识别每一台主机，不会给每台主机分配IP地址。

路由器工作在网络层，路由器也有交换机同样的功能，路由器还多了路由的功能，并且能够给每一台主机分配IP地址。

2 socket编程

2.1 套接字(Socket)和Socket API简介

套接字是一种通讯机制，是一种使用标准Unix/Linux文件描述字与其他主机进程通讯的手段。从程序员的角度来看，套接字很像文件描述字（一个整数），因为它同文件和管道一样可以用read() / write()来读写数据。但是，套接字与普通文件描述字不同。

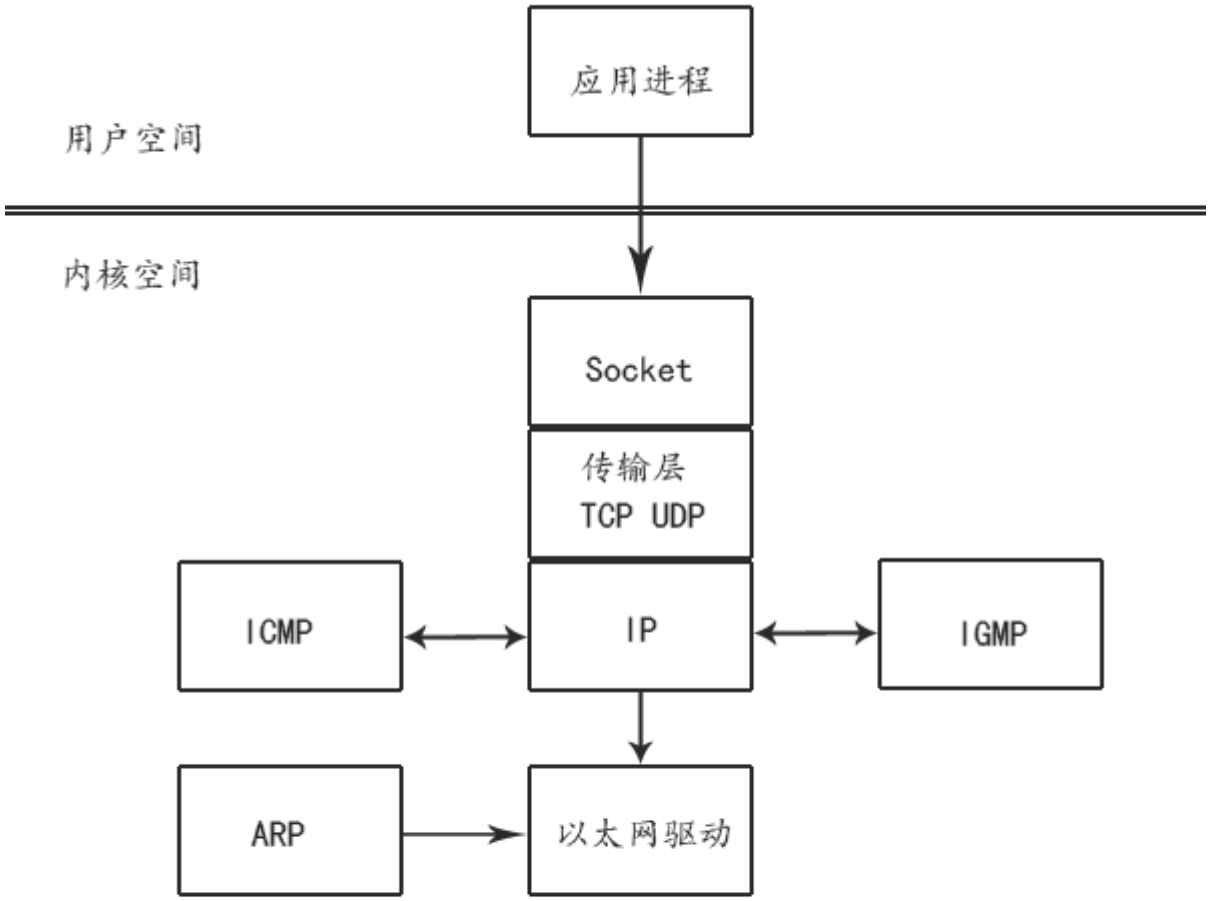
首先，套接字不像文件描述符那么简单，它除了需要地址端口等信息之外，还明确包含有关于通信的其他属性（如地址族，协议族）。

其次，套接字的使用可以是非对称的，它通常明确的区分通信的两个进程为客户进程和服务进程，并且允许不同系统或主机上的多个客户与单个服务进程相连。

最后，套接字的创建以及控制套接字的各种操作与文件描述字也有所不同，这些操作的不同是由于建立套接字网络比磁盘访问要复杂而带来的。

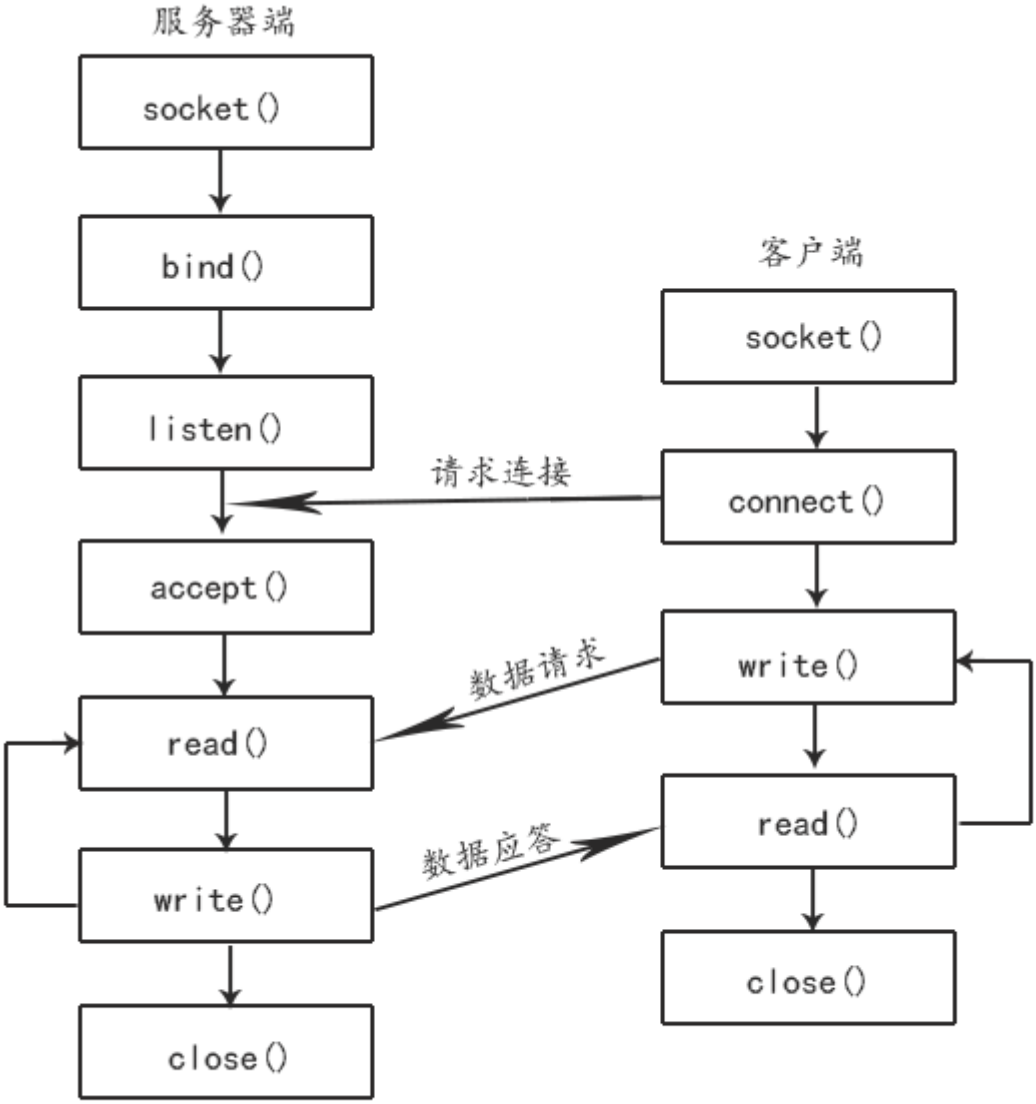
进程可以通过Socket API来实现对套接字进行操作，从而实现访问网络，Socket API是最初作为TCP/IP协议代码的。Socket API在其他操作系统中得到广泛的支持，包括众多的类Unix操作系统（AIX，MAC，Linux等）和Windows系统。

Socket层在内核中实现，如果应用程序想要调用它就要通过系统调用方式。



2.2 网络编程基础知识

- 服务器与客户端模型



- 服务器

  （1）服务进程通过系统调用`socket()`创建一个套接字。

  （2）服务进程要给该套接字绑定一个端口，通过`bind()`来实现。

  （3）服务进程调用`listen()`等待客户进程与该套接字连接。因为服务器必须允许多个客户同时与该套接字连接，所以`listen()`函数用于为客户端连接创建一个连接队列，调用`accept()`接受这些连接。

  （4）服务进程每调用一次`accept()`便创建一个新的套接字，不同与`socket()`创建的套接字，这个新的套接字完全只用于特定的客户通信，而`socket()`创建的套接字则保留用于等待其他客户链接的到来。

  （5）使用`accept()`创建的套接字与客户端进行读写操作。

  （6）使用`close()`或`shutdown()`来断开连接，释放资源。

- 客户端

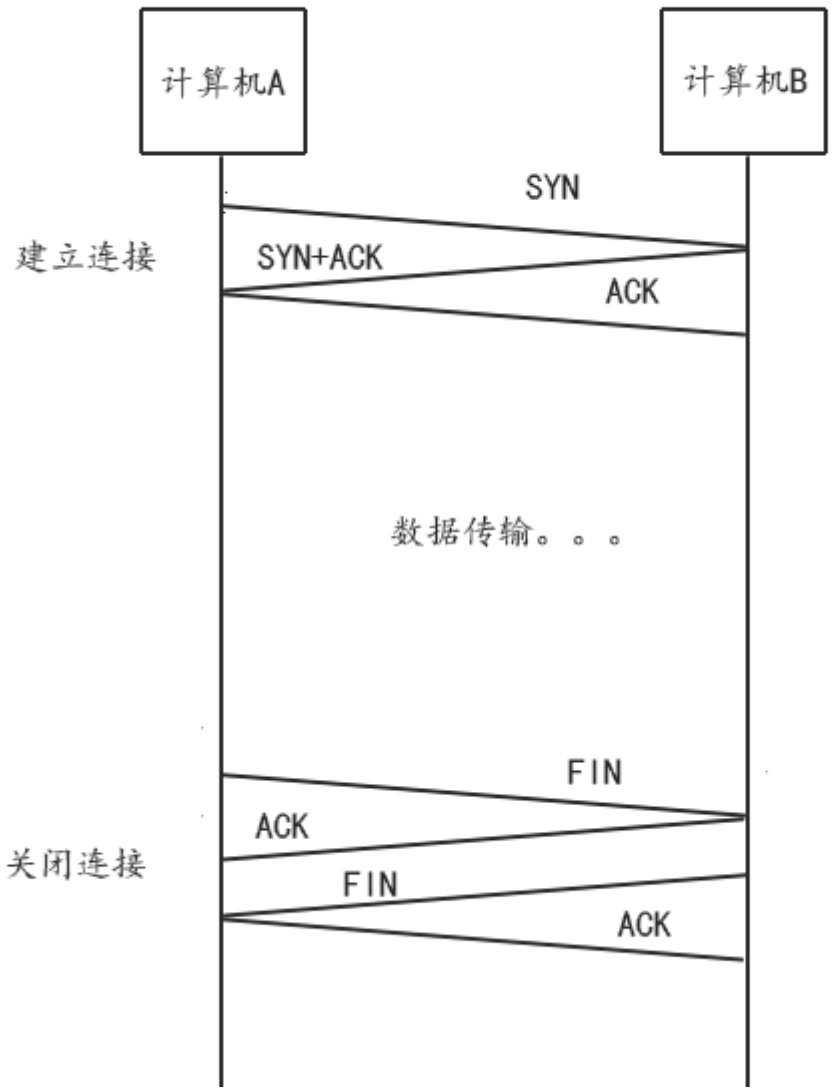
  （1）客户进程通过系统调用`socket()`创建一个套接字。

  （2）然后通过ip地址以及端口作为参数，调用`connect()`与服务器建立连接。

  （3）使用`socket()`创建的套接字与服务器进行读写操作。

（4）使用`close()`或`shutdown()`来关闭套接字，释放资源。

2.3 TCP连接与断开



TCP不同的控制报文的作用如下：

SYN报文：建立连接的请求。

FIN报文：用来关闭连接的请求。

ACK报文：用来确认数据。

- TCP建立一次连接需要经历三次握手过程：

  （1）服务器端做好监听准备，通常是调用`socket()`、`bind()`、和`listen()`函数监听服务器的某个已知端口。

（2）客户端通过调用`socket()`和`connect()`函数，导致客户端TCP层发送一个SYN报文请求连接，以及初始序号（例如`I`）。

（3）服务器接收到SYN报文后，发送包含服务器初始序号（例如`J`）的SYN报文段作为应答，同时回复ACK报文确认客户端发出的SYN，确认序号设置为`I+1`。

（4）客户端接收到SYN报文后，回复ACK报文确认服务器发出的SYN，确认序号设置为`J+1`。

- TCP连接的关闭过程：

  （1）A主机发送一个FIN终止这个方向连接，序号为`M`，通常是调用`close()`导致。

  （2）B主机收到FIN，响应一个ACK，ACK确认序号设置为`M+1`，同时必须通知应用程序对方已经终止了连接，通常是对I/O操作函数返回文件结束标志(通过read()、write()察觉到这个结束标志)。

  （3）B主机也调用`close()`, 发送一个FIN给A主机，序号为`N`。

  （4）A主机收到FIN后将响应一个ACK，序号为`N+1`.

2.4 字节序

不同种类的计算机在存储“字”数据是，使用不同的字节序协定。

大端字节序： 是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中。

小端字节序： 是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中。

x86框架采用小端字节序，ARM架构启动时可以 设置大小端字节序。

互联网是个复杂的结合体，由各式各样的主机构成，为了规范字节序，网络字节序采用大端字节序。

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。

2.5 地址族

一般而言，在使用套接字通信之前，我们首先要为套接字选定一个地址族，简单的说，为套接字指定地址族的目的是告诉套接字使用哪一种网络层进行通讯(IPv4 或 IPv6)。

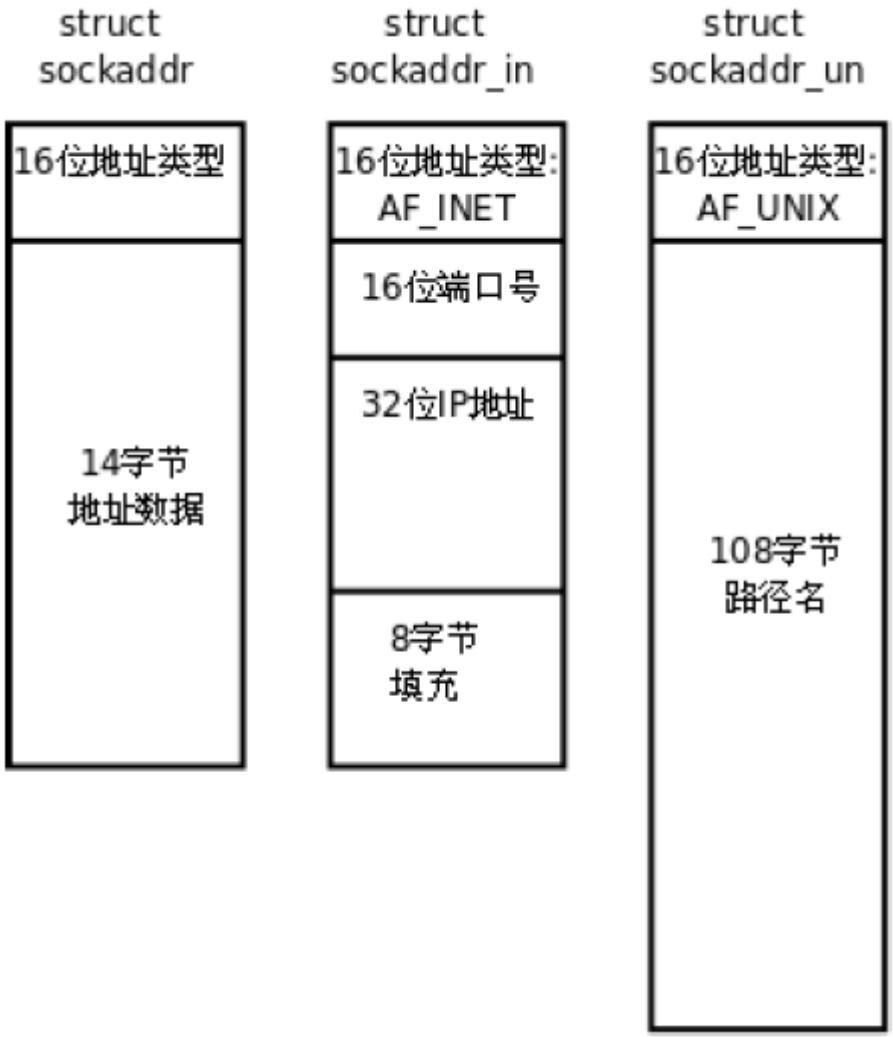
在Linux系统中，sys/socket.h头文件定义了繁多的地址族类型，它们都是“AF_”或"IPv6".

AF_INET 使用TCP或UDP来传输，用IPv4的地址
AF_INET6 使用TCP或UDP来传输，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器在同一台及其上的时候使用。

2.6 地址结构

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。



```
#include <sys/socket.h>
struct sockaddr {
    sa_family_t sa_family; /* 地址族 */
    char sa_data[14]; /*地址值，实际可能更长*/
};

/*IPV4*/
struct sockaddr_in {
    __kernel_sa_family_t sin_family; /* AF_INET4 */
    __be16 sin_port; /*端头号*/
    struct in_addr sin_addr; /* IPv4 地址 */
    /* 8字节填充 */
    unsigned char __pad[__SOCK_SIZE__ - sizeof(short int) -
    sizeof(unsigned short int) - sizeof(struct in_addr)];
};

struct in_addr {
    __be32 s_addr;
};


/*IPV6*/
struct sockaddr_in6 {
    unsigned short int sin6_family; /*AF_INET6*/
    __be16 sin6_port; /* 端口 # */
    __be32 sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr; /* IPv6 地址 128bit*/
    __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

/*本地socket通信*/
struct sockaddr_un {
    __kernel_sa_family_t sun_family; /* AF_UNIX */
    char sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

2.7 IP转换

ipv4以及的ipv6点分十进制表达法与二进制整数之间的互转。

```
#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);
```

`inet_pton` 把字符串 `src` 转换成ip地址保存在 `dst` 中。该函数调用成功返回大于0的整数。

`inet_ntop()` 把网络字节序的ip地址 `src` 转换成字符串保存在 `dst` 中作为返回值返回，参数 `size` 为 `dst` 所包含的字节数。

`af` 用来指定要转换的ip属于什么协议族。

```
#include <arpa/inet.h>
#include <sys/socket.h>
#include <string.h>
int main(int argc, char** argv)
{
    struct sockaddr_in sin;
    char buf[16];
    memset(&sin, 0, sizeof(sin));
    sin.sin_family=AF_INET;
    sin.sin_port=htons(3001);
    inet_pton(AF_INET, "192.168.1.111", &sin.sin_addr.s_addr);
    printf("%s\n", inet_ntop(AF_INET, &sin.sin_addr, buf, sizeof(buf)));
    return 0;
}
```

3 socket API

3.1 socket

```
#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，不过是来用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器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。

- protocol：0 默认协议, 一般填默认的。

- 返回值：

  成功返回一个新的套接字，失败返回-1，设置errno

3.2 给本地套接字赋予地址和端口(bind)

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

sockfd：
　　socket套接字
addr:
　　填写IP地址、端口号、协议族等信息。
addrlen:
　　addr结构体实例的大小
返回值：
　　成功返回0，失败返回-1, 设置errno

　　bind()的作用是将参数sockfd和addr绑定在一起。

```
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(8000);
```

　　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8000。

3.3 给连接排队(listen)

当我们给服务器端套接调用bind()指定了本地IP地址后，这就如同电话总机已经有了具体的号码，而端口号就如同我们的分机号码，此时需要把分机号码的振铃打开，同时还要对同时拨入的多个电话进行排队。

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

sockfd：

参数为成功调用socket函数返回的套接字，并已经成功调用了bind()。

backlog：

参数告诉套接字在忙于处理上一个请求时，还可以接受多少个进入的请求，换句话说，这决定了挂起连接的队列的大小。

返回：

调用成功返回0，失败返回-1，设置errno。

- 查看系统默认backlog

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

3.4 接受网络连接(accept)

`accept()`调用类似于听见电话铃声后接起电话，此时，你已经建立起一个与你的客户的连接，这个连接保存直到你或你的客户挂线，网络服务器端也有类似的过程，它使用`accept()`函数接受客户端的连接。

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

sockdf:

参数为成功调用socket函数返回的套接字，并已经成功调用了`bind()`和`listen()`

addr:

传出参数，返回连接客户端地址信息，含IP地址和端口号。（如果使用IPv4地址族，那么它需要我们填个指向sockaddr_in结构的指针）

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

3.5 连接远程主机(connect)

请求连接是客户端的动作，TCP/IP客户端需要调用`connect()`去连接一个服务器，原型如下：

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

sockdf:

参数是成功调用socket()返回的套接字。

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,如IPV4传入sizeof(struct sockaddr_in)大小

如果调用成功，表示连接已经建立（三次握手完成），返回0。否则返回 -1并将错误码代存放于全局变量errno之中。

3.6 Socket I/O

对于套接字的I/O(读写)有很多API可以用，如send()、recv()、readv()、writev()、sendmsg()、recvmsg()、read()、write()等等。这里我们只介绍常用的I/O接口read()和write()。

```
#include <unistd.h>
int read(int sockfs, void *buf, size_t nbytes);
int write(int sockfd, void *buf, size_t nbytes);
```

3.7 关闭套接字

关闭套接字可以简单的调用close()函数:

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

调用close()会触发四次握手关闭连接。

3.8 搭建一个TCP服务器

源码：*tcp_server.c*

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MAXBYTES 1024
#define SERVER_PORT 8888
int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, clifd;
    char str[INET_ADDRSTRLEN];
    char buf[MAXBYTES];
    int i, n, len;
    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(SERVER_PORT);
    bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
    listen(listenfd, 128);
    printf("Accepting connections ...\n");
    while (1)
    {
        bzero(&buf, sizeof(buf));
        cliaddr_len = sizeof(cliaddr);
        clifd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
        if(clifd == -1)
        {
            perror("clifd");
            continue;
        }
        len = read(clifd, buf, sizeof(buf) - 1);
        if(len != -1)
        {
            printf("accept from %s at PORT %d :[%d] %s\n"
                , inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str))
                , ntohs(cliaddr.sin_port), len, buf);
        }
        else
        {
            perror("read");
        }
        close(clifd);
    }
    return 0;
```

测试服务器是否好用可以使用一下命令：

```
nc 127.0.0.1 8888
```

3.9 搭建一个TCP客户端

源码：*tcp_client.c*

```
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_PORT 8888
int main(int argc, char *argv[])
{
    struct sockaddr_in serveraddr;
    int sockfd, n, ret;
    char buf[] = "helloworld";
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &serveraddr.sin_addr);
    serveraddr.sin_port = htons(SERVER_PORT);
    ret = connect(sockfd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
    if(ret == -1)
      perror("connect");

    ret = write(sockfd, buf, strlen(buf));
    if(ret == -1)
      perror("connect");
    close(sockfd);
    return 0;
}
```

需要注意的事：

1.当socket写端关闭的时候，读端read将一直读取到0字节，可以使用这一特性来关闭读端套接字。

2.当socket读端关闭的时候，写端write继续写入数据，将会报错，并且收到一个SIGPIPE信号。

3.当网络异常的时候（如拔掉网线），这个时候read不会返回错误，将阻塞等待数据；write也不会返回错误，仍然可以写入数据。

4.socket在有个buffer，并且这个buffer的大小是一定的，当一直往socket write的时候，达到这个buffer的最大值后，write将阻塞。

服务器改写如下：

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

#define MAXBYTES 1024
#define SERVER_PORT 8888
int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, clifd;
    char str[INET_ADDRSTRLEN];
    char buf[MAXBYTES];
    int i, n, len;

    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(SERVER_PORT);
    bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
    listen(listenfd, 128);
    printf("Accepting connections ...\n");
    while (1)
    {
        bzero(&buf, sizeof(buf));
        cliaddr_len = sizeof(cliaddr);
        clifd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len); //可能被信号中断需要
注意
        if(clifd == -1)
        {
            perror("read");
            return -1;
        }
        while(1)
        {
            len = read(clifd, buf, sizeof(buf) - 1);
            if(len > 0)
            {
                printf("accept from %s at PORT %d :[%d] %s\n"
                        , inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str))
                        , ntohs(cliaddr.sin_port), len, buf);
            }
           else if(len == 0) //写端关闭的时候将永远读到0字节，如果读到0字节就可以关闭套接字了
           {
               close(clifd);
                break;
           }

            else
```

```
            {
                perror("read");
            }
            sleep(1);
        }

    }
    close(listenfd);
    return 0;

}
```

客户端改写如下：

```c
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#define SERVER_PORT 8888
void sig_handler(int sig)
{
    printf("SIGPIPE\n");
}

int main(int argc, char *argv[])
{
    signal(SIGPIPE, sig_handler); //当读端关闭的时候，如果继续写，则会触发SIGPIPE信号
    struct sockaddr_in serveraddr;
    int sockfd, n, ret;
    char buf[] = "helloworld";
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    inet_pton(AF_INET, "192.168.1.103", &serveraddr.sin_addr);
    serveraddr.sin_port = htons(SERVER_PORT);

    int conntimes = 0;
    while(connect(sockfd, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) == -1) //有可能链接失败，通过这种方式定时链接
    {
        if(conntimes++ > 10)
          return -1;
        sleep(1);
        perror("connect");

    }

    int num = 0;
    while(1)
    {
        ret = write(sockfd, buf, strlen(buf));
        if(ret == -1)
        {
            perror("write");
        }
        else
            num += ret;
        printf("send bytes %d\n", num);
        sleep(1);
    }
    close(sockfd);
    return 0;
}
```

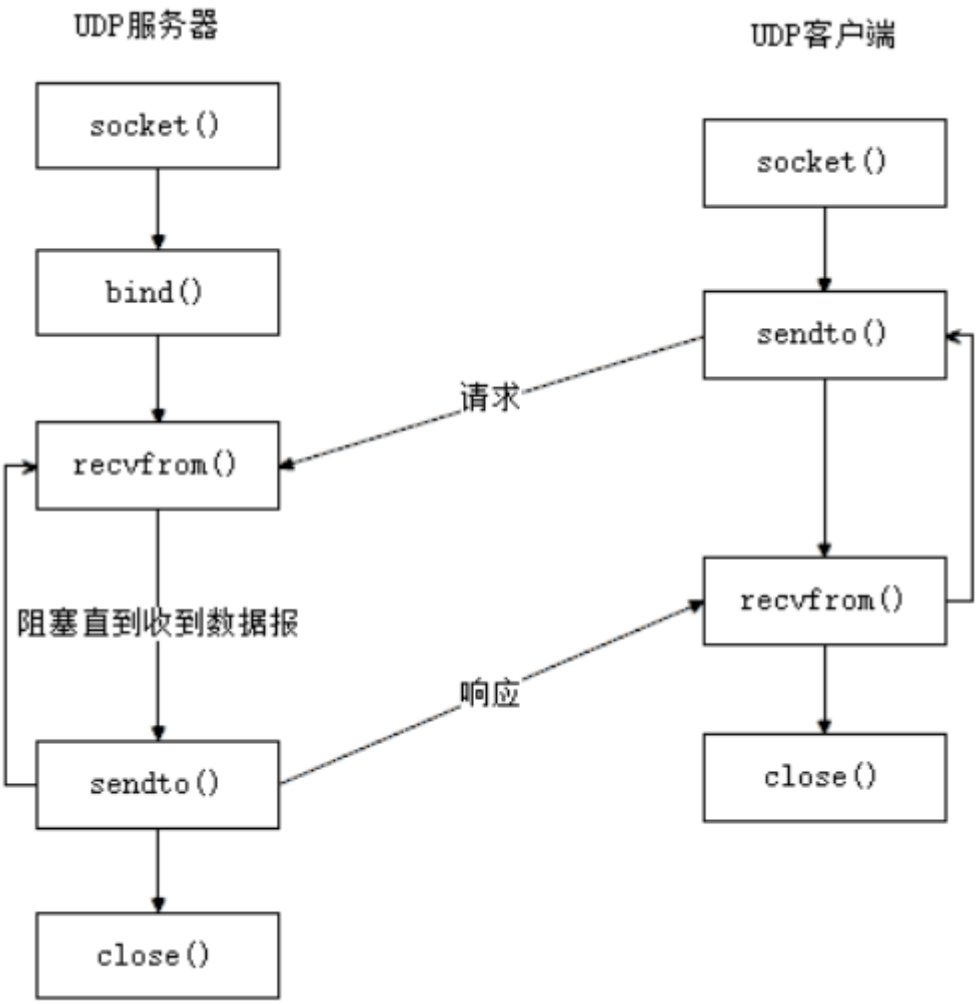
4 UDP编程

4.1 UDP概述

TCP与UDP应用程序之间存在本质差异，UDP是无连接的、不可靠的数据报协议，而TCP是面向连接的，提供可靠的字节流。然而，有些情况下更适合用UDP而不是TCP。

在互联网中个，有很多流行的应用层协议是用UDP实现的，如DNS(域名系统)、NFS（网络文件系统）。另外UDP还提供了广播和多播的特性，这是TCP无法做到的。

下面是典型的UDP服务器客户端。客户端不与服务器建立连接，它调用函数`sendto()`给服务器发送数据报，此函数要求目的地址作为其参数。类似的，服务器不用调用`accept()`接受连接,它只管调用函数`recvfrom()`，等待来自某客户端数据到达。`recvfrom()`在读入数据的同时返回客户的协议地址，所以服务器可以根据`recvfrom()`返回的客户端地址发送应答。



- 创建一个UDP套接字

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

采用SOCK_DGRAM数据报模式中的默认协议，也就是UDP协议来传输。

- 使用`recvfrom()`和`sendto()`来收发UDP数据

函数原型为：

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                        struct sockaddr *src_addr, socklen_t *addrlen);

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

sockfd:

`socket()`函数返回的套接字。

buf:

指定保存数据的缓冲区的指针。

len:

指出要读写字节数。

flags：

参数很少使用，一般我们会传递一个0。

返回值:

如果正确接收返回接收到的字节数，失败返回-1，并且设置errno。

4.2 UDP服务端

源码：*udp_server.c*

```c
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#define MAXBYTES 80
int main(int argc, char** argv)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int sockfd;
    char buf[MAXBYTES];
    char str[INET_ADDRSTRLEN];
    int i, len;
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(8888);
    bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
    printf("Accepting connections ...\n");

    while (1)
    {
        cliaddr_len = sizeof(cliaddr);
        len = recvfrom(sockfd, buf, MAXBYTES, 0, (struct sockaddr *)&cliaddr,   &cliaddr_len);
       if(len != -1)
       {
            printf("received from %s at PORT %d : [%d]%s\n",
            inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
            ntohs(cliaddr.sin_port), len, buf);
       }
    }
    close(sockfd);
}
```

4.3 UDP客户端

源码：*UDP_client.c*

```c
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define MAXBYTES 20

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    int sockfd, len;
    char buf[MAXBYTES];
    char str[INET_ADDRSTRLEN];
    socklen_t servaddr_len;
    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(8888);

    while (fgets(buf, MAXBYTES, stdin) != NULL)
    {
        len = sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)&servaddr,
sizeof(servaddr));
        if (len == -1)
        {
            perror("sendto");
            exit(-1);
        }
    }
    close(sockfd);
    return 0;
}
```

4.4 错误值处理

源码：*mysocket.h*

```
#ifndef __MYSOCKET_H__
#define __MYSOCKET_H__
void perr_exit(const char *s);
int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr);
void Bind(int fd, const struct sockaddr *sa, socklen_t salen);
void Connect(int fd, const struct sockaddr *sa, socklen_t salen);
void 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);
void Close(int fd);
ssize_t Readn(int fd, void *vptr, size_t n);
ssize_t Writen(int fd, const void *vptr, size_t n);
static ssize_t my_read(int fd, char *ptr);
ssize_t Readline(int fd, void *vptr, size_t maxlen);
#endif
```

源码：*mysocket.c*

```c
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/socket.h>
#include <unistd.h>
#include "mysocket.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;
}
void Bind(int fd, const struct sockaddr *sa, socklen_t salen)
{
    if (bind(fd, sa, salen) < 0) { perr_exit("bind error"); }
}
void Connect(int fd, const struct sockaddr *sa, socklen_t salen)
{
    if (connect(fd, sa, salen) < 0) { perr_exit("connect error"); }
}
void Listen(int fd, int backlog)
{
    if (listen(fd, backlog) < 0) { perr_exit("listen error"); }
}
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;
}

void Close(int fd)
{
    if (close(fd) == -1) { perr_exit("close error"); }
}

ssize_t Readn(int fd, void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nread;
    char *ptr;
    ptr = vptr;
    nleft = n;
    while(nleft > 0)
    {
        if((nread = read(fd, ptr, nleft)) == -1)
        {
            if (errno == EINTR) { continue; } else { return -1;}
        }
        else if(nread == 0) { break; }
        nleft -= nread;
        ptr += nread;
    }
    return n - nleft;
}
ssize_t Writen(int fd, const void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nwritten;
    char *ptr;
    ptr = (char*)vptr;
    nleft = n;
    while (nleft > 0)
    {
        if ( (nwritten = write(fd, ptr, nleft)) == -1)
        {
            if (errno == EINTR) { continue; } else  { return -1; }
        }
        else if(nwritten == 0) { break; }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return n - nleft;
}

static 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)
        {
            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;
}
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++ = c;
            if (c == '\n') { break; }
        }
        else if (rc == 0) { *ptr = 0; return n - 1; }
        else { return -1; }
    }
    *ptr = 0;
    return n;
}
```

5 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(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)

源码：*process_server.c*

```c
#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 <ctype.h>
#define MAXLINE 128
#define SERV_PORT 8888

void do_sigchild(int num)
{
    while(waitpid(-1, 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;
    signal(SIGCHLD, do_sigchild);
    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);
    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(connfd, buf, n);
            }
            Close(connfd);
            return 0;
        }

        else if (pid > 0) { Close(connfd); }
        else { perr_exit("fork"); }

    }
}
```

源码：*client.c*

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "mysocket.h"
#define MAXLINE 128
#define SERV_PORT 8888

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");
        }
        else
        {
            Write(STDOUT_FILENO, buf, n);
        }
    }
    Close(sockfd);
    return 0;
}
```

6 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.服务与客户端线程退出时，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

源码：*thread_server.c*

```c
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <ctype.h>
#include "mysocket.h"
#define MAXLINE 128
#define SERV_PORT 8888
struct s_info {
    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];
    pthread_detach(pthread_self());
    while (1)
    {
        n = Read(ts->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, &(*ts).cliaddr.sin_addr, str, sizeof(str)),
                ntohs((*ts).cliaddr.sin_port));
        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);
        Write(ts->connfd, buf, n);
    }
    Close(ts->connfd);
}
int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    int i = 0;
    pthread_t tid;
    struct s_info ts[383];
    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);

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

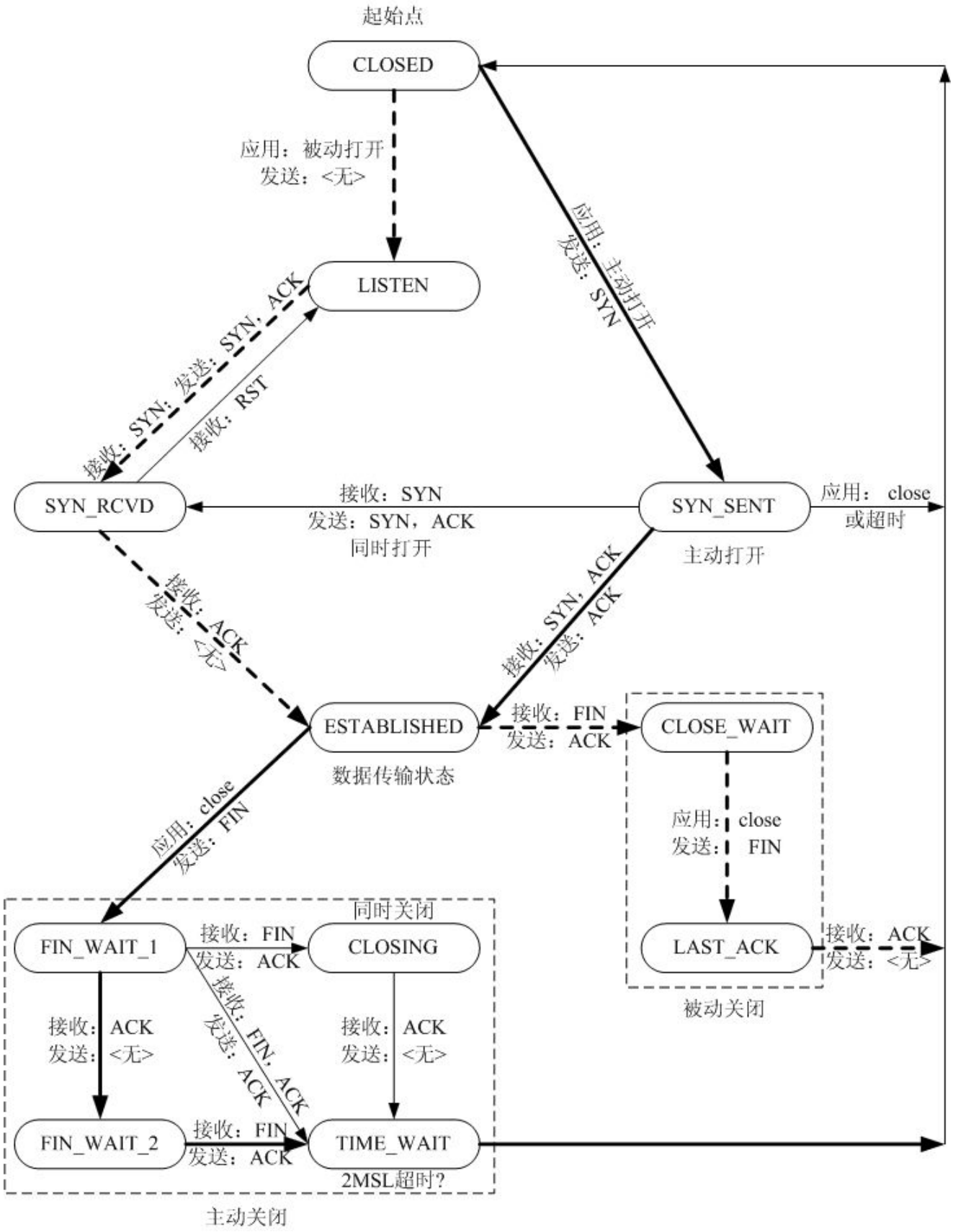
```
    while (1)
    {
        cliaddr_len = sizeof(cliaddr);
        connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
        ts[i].cliaddr = cliaddr;
        ts[i].connfd = connfd;
        /*一般要对pthread_create做出错处理，提高服务器稳定性，例如达到线程最大数时*/
        pthread_create(&tid, NULL, do_work, (void*)&ts[i]);
        i++;
    }
    return 0;
}
```

源码：*client.c*

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "mysocket.h"
#define MAXLINE 128
#define SERV_PORT 8888
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");
        else
            Write(STDOUT_FILENO, buf, n);
    }
    Close(sockfd);
    return 0;
}
```

7 TCP状态

TCP 状态转换图


起始点
CLOSED
应用：被动打开
发送：<无>
LISTEN
应用：主动打开
发送：SYN
接收：SYN，发送：SYN，ACK
接收：RST
SYN_RCVD
接收：SYN
发送：SYN，ACK
同时打开
SYN_SENT
应用： close
或超时
主动打开
接收：
发送：ACK
<无>
接收：SYN，ACK
发送：ACK
ESTABLISHED
数据传输状态
接收：FIN
发送：ACK
CLOSE_WAIT
应用：close
发送：FIN
应用：close
发送：FIN
同时关闭
FIN_WAIT_1
接收：FIN
发送：ACK
CLOSING
接收：ACK
发送：<无>
接收：FIN，ACK
发送：ACK
接收：ACK
发送：<无>
LAST_ACK
接收：ACK
发送：<无>
被动关闭
FIN_WAIT_2
接收：FIN
发送：ACK
TIME_WAIT
2MSL超时?
主动关闭

1、建立连接协议（三次握手）

（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报文1。

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

（3） 客户必须再次回应服务器一个ACK报文，这是报文段3。

2、连接终止协议（四次握手）

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） TCP客户端发送一个FIN，用来关闭客户到服务器的数据传送（报文段4）。

（2） 服务器收到这个FIN，它发回一个ACK，确认序号为收到的序号加1（报文段5）。和SYN一样，一个FIN将占用一个序号。

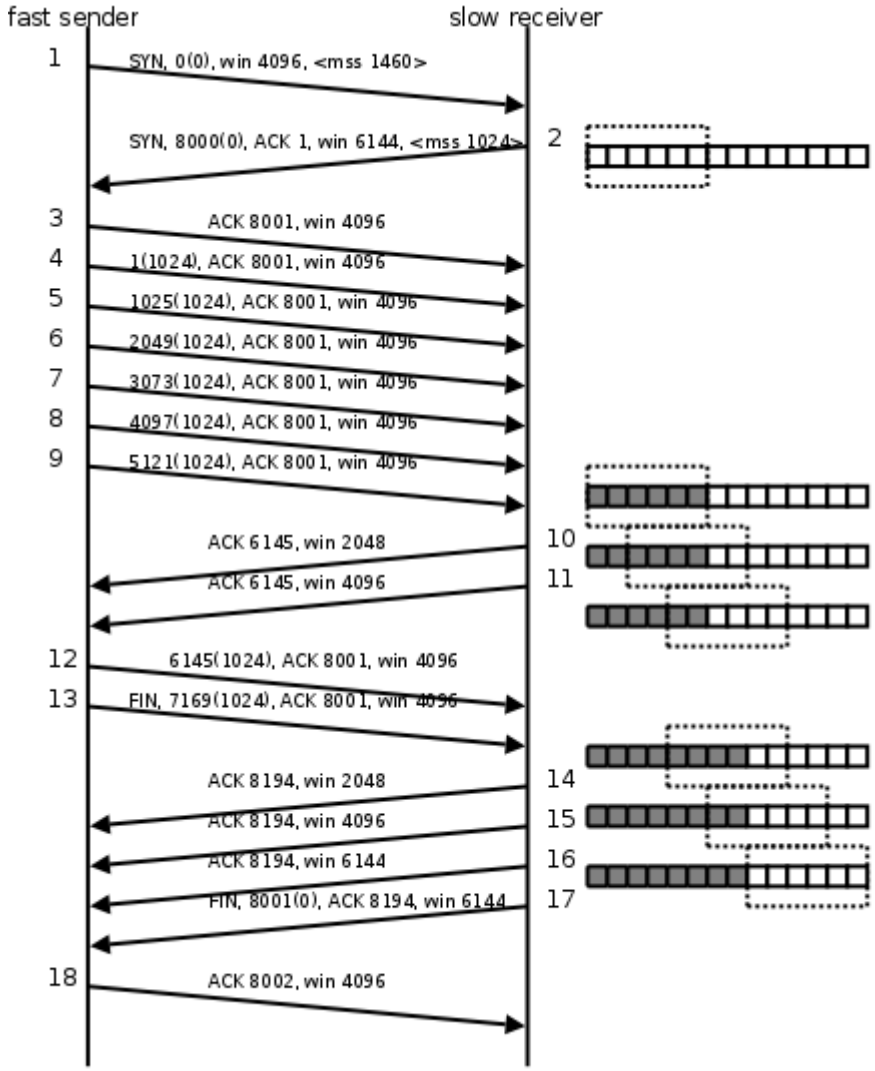
（3） 服务器关闭客户端的连接，发送一个FIN给客户端（报文段6）。

（4） 客户段发回ACK报文确认，并将确认序号设置为收到序号加1（报文段7）。

```
CLOSED: #这个没什么好说的了，表示初始状态。
LISTEN: #这个也是非常容易理解的一个状态，表示服务器端的某个SOCKET处于监听状态，可以接受连接了。
SYN_RCVD: #这个状态表示接受到了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，基本上用netstat你是很难看到这种状态的，除非你特意写了一个客户端测试程序，故意将三次TCP握手过程中最后一个ACK报文不予发送。因此这种状态时，当收到客户端的ACK报文后，它会进入到ESTABLISHED状态。
SYN_SENT: #这个状态与SYN_RCVD遥想呼应，当客户端SOCKET执行CONNECT连接时，它首先发送SYN报文，因此也随即它会进入到了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。
ESTABLISHED: #这个容易理解了，表示连接已经建立了。
FIN_WAIT_1: #这个状态要好好解释一下，其实FIN_WAIT_1和FIN_WAIT_2状态的真正含义都是表示等待对方的FIN报文。而这两种状态的区别是：FIN_WAIT_1状态实际上是当SOCKET在ESTABLISHED状态时，它想主动关闭连接，向对方发送了FIN报文，此时该SOCKET即进入到FIN_WAIT_1状态。而当对方回应ACK报文后，则进入到FIN_WAIT_2状态，当然在实际的正常情况下，无论对方何种情况下，都应该马上回应ACK报文，所以FIN_WAIT_1状态一般是比较难见到的，而FIN_WAIT_2状态还有时常常可以用netstat看到，如果长时间没有收到ACK，过一段时间后也会重置为CLOSED状态。
FIN_WAIT_2: #上面已经详细解释了这种状态，实际上FIN_WAIT_2状态下的SOCKET，表示半连接，也即有一方要求close连接，但另外还告诉对方，我暂时还有点数据需要传送给你，稍后再关闭连接。
TIME_WAIT: #表示收到了对方的FIN报文，并发送出了ACK报文，就等2MSL后即可回到CLOSED可用状态了。如果FIN_WAIT_1状态下，收到了对方同时带FIN标志和ACK标志的报文时，可以直接进入到TIME_WAIT状态，而无须经过FIN_WAIT_2状态。
CLOSING: #这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。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？其实细想一下，也不难得出结论：那就是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: #这种状态的含义其实是表示在等待关闭。怎么理解呢？当对方close一个SOCKET后发送FIN报文给自己，你系统毫无疑问地会回应一个ACK报文给对方，此时则进入到CLOSE_WAIT状态。接下来呢，实际上你真正需要考虑的事情是察看你是否还有数据发送给对方，如果没有的话，那么你也就可以close这个SOCKET，发送FIN报文给对方，也即关闭连接。所以你在CLOSE_WAIT状态下，需要完成的事情是等待你去关闭连接。
LAST_ACK: #这个状态还是比较容易好理解的，它是被动关闭一方在发送FIN报文后，最后等待对方的ACK报文。当收到ACK报文后，也即可以进入到CLOSED可用状态了。
```

8 TCP流量控制(滑动窗口)

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。

9 TCP与UDP的不同接包处理方式

1.UDP发包的问题

问：udp 发送两次数据，第一次 100字节 ，第二次200字节， 接包方一次recvfrom(1000), 收到是 100，还是200，还是300？

答：UDP是数据报文协议，是以数据包方式，所以每次可以接收100，200，在理想情况下，第一次是无论recvfrom多少都是接收到100。当然，可能由于网络原因，第二个包先到的话，有可能是200了。对可能会由于网络原因乱序，所以可能先收到200，所以自定义的udp协议包头里都要加上一个序列号，标识发送与收包对应

2.TCP的发包问题

问：同样如果换成tcp, 第一次发送 100字节 ，第二次发送200字节，recv(1000)会接收到多少？

答：tcp是流协议，所以recv(1000)，会收到300 tcp自己处理好了重传，保证数据包的完整性

3.有分片的情况下如下处理

问：如果MTU是1500，使用UDP发送 2000，那么recvfrom(2000)是收到1500，还是2000?

答： 还是接收2000，数据分片由IP层处理了，放到UDP还是一个完整的包。接收到的包是由路由路径上最少的MTU来分片，注意转到UDP已经在是组装好的(组装出错的包会经crc校验出错而丢弃)，是一个完整的数据包

4.分片后的处理

问：如果500那个片丢了怎么办？udp又没有重传

答：UDP里有个CRC检验，如果包不完整就会丢弃，也不会通知是否接收成功，所以UDP是不可靠的传输协议，而且TCP不存在这个问题，有自己的重传机制。在内网来说，UDP基本不会有丢包，可靠性还是有保障。当然如果是要求有时序性和高可靠性，还是走TCP，不然就要自己提供重传和乱序处理(UDP内网发包处理量可以达7w~10w/s)

5.不同连接到同一个端口的包处理

问：TCP

A -> C 发100

B -> C 发200

AB同时同一端口

C recv(1000) ,会收到多少？

答：A与C是一个tcp连接，B与C又是另一个tcp连接， 所以不同socket，所以分开处理。每个socket有自己的接收缓冲和发送缓冲

6.什么是**TCP**粘包

由于TCP是流协议，对于一个socket的包，如发送 10AAAAABBBBB两次，由于网络原因第一次又分成两次发送， 10AAAAAB和BBBB，如果接包的时候先读取10(包长度)再读入后续数据，当接收得快，发送的慢时，就会出现先接收了 10AAAAAB,会解释错误 ,再接到到BBBB10AAAAABBBBB，也解释错误的情况。这就是TCP的粘包。

解决的办法TLV方式，先接收包头，在包头里指定包体长度来接收。设置包头包尾的检查位（如群空间0x2开头，0x3结束来检查一个包是否完整）。对于TCP来说：1）不存在丢包，错包，所以不会出现数据出错 2）如果包头检测错误，即为非法或者请求，直接重置即可

7.TCE怎么处理**TCP**包

TCE的TCP包处理：

1.先按可用buf接收TCP包

2.按照协议头检查，提出多个请求包

3.如果检查失败，当前接收缓存就会重置(位置从0开始) ->

1）由于TCP保证了包的完整性，不会出现错包

2）如果是非法请求（对应的一个socket接收的数据包），会丢弃

10 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。

```
#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
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "mysocket.h"
#include <ctype.h>
#define MAXLINE 128
#define SERV_PORT 8888
int main(int argc, char *argv[])
{
    int i, maxi, maxfd, listenfd, connfd, sockfd;
    int nready, client[FD_SETSIZE]; /* FD_SETSIZE 默认为1024 */
    ssize_t n;
    fd_set rset, allset;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN]; /* #define INET_ADDRSTRLEN 16 */
    socklen_t cliaddr_len;
    struct sockaddr_in cliaddr, servaddr;
    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); /* 默认最大128 */
    maxfd = listenfd; /* 初始化*/
    maxi = -1; /* client[]的下标*/
    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);
        if (nready < 0)
            perr_exit("select error");
        if (FD_ISSET(listenfd, &rset))
        { /* new client connection */
            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));
            for (i = 0; i < FD_SETSIZE; i++)
                if (client[i] < 0)
                {
                    client[i] = connfd; /* 保存accept返回的文件描述符到client[]里*/
                    break;
                }
            /* 达到select能监控的文件个数上限1024 */

            if (i == FD_SETSIZE)
```

```
            {
                fputs("too many clients\n", stderr);
                exit(1);
            }
            FD_SET(connfd, &allset); /* 添加一个新的文件描述符到监控信号集里*/
            if (connfd > maxfd)
                maxfd = connfd; /* select第一个参数需要*/
            if (i > maxi)
                maxi = i; /* 更新client[]最大下标值*/
            if (--nready == 0)
                continue; /* 如果没有更多的就绪文件描述符继续回到上面select阻塞监听,负责处理未
                             处理完的就绪文件描述符*/
        }
        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, MAXLINE)) == 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监控此文件描述符*/
                    client[i] = -1;
                }
                else
                {
                    int j;
                    for (j = 0; j < n; j++)
                        buf[j] = toupper(buf[j]);
                    Write(sockfd, buf, n);
                }
                if (--nready == 0)
                    break;
            }
        }
    }
    Close(listenfd);
    return 0;
}
```

源码：*client.c*

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "mysocket.h"
#define MAXLINE 128
#define SERV_PORT 8888
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");
        else
            Write(STDOUT_FILENO, buf, n);
    }
    Close(sockfd);
    return 0;
}
```

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

11 poll

```
#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。

源码*server.c*

```c
/* server.c */
#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 "mysocket.h"
#define MAXLINE 80
#define SERV_PORT 8888
#define OPEN_MAX 1024
int main(int argc, char *argv[])
{
    int i, j, maxi, listenfd, connfd, sockfd;
    int nready;
    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);
    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);
    client[0].fd = listenfd;
    client[0].events = POLLRDNORM; /* 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*/
    maxi = 0; /* client[]数组有效元素中最大元素下标*/
    while(1)
    {
        nready = poll(client, maxi+1, -1); /* 阻塞*/
        if (client[0].revents & POLLRDNORM)
        { /* 有客户端链接请求*/
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);
            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)
                perr_exit("too many clients");

            client[i].events = POLLRDNORM | POLLERR; /* 设置刚刚返回的connfd，监控读事件,以及错误监
```

```
控*/
            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++)
        { /* 检测client[] */
            if ( (sockfd = client[i].fd) < 0)
                continue;
            if (client[i].revents & (POLLRDNORM | POLLERR))
            {
                if ( (n = Read(sockfd, buf, MAXLINE)) < 0)
                {
                    if (errno == ECONNRESET)
                    { /* 当收到RST标志时*/
                        /* connection reset by client */
                        printf("client[%d] aborted connection\n", i);
                        Close(sockfd);
                        client[i].fd = -1;
                    } else
                        perr_exit("read error");
                }
                else if (n == 0)
                {
                    /* connection closed by client */
                    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; /* no more readable descriptors */
            }
        }
    }
    return 0;
}
```

源码*client.c*

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "mysocket.h"
#define MAXLINE 128
#define SERV_PORT 8888
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");
        else
            Write(STDOUT_FILENO, buf, n);
    }
    Close(sockfd);
    return 0;
}
```

12 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。一个进程打开大数目的socket描述符。

```
sudo vi /etc/security/limits.conf
#写入以下配置,soft软限制，hard硬限制
* soft nofile 65536
* hard nofile 100000
```

epoll 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个数，跟内存大小有关。

```
int epoll_create(int size)
//size：告诉内核监听的数目
```

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 {
    _uint32t 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)来说的
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
```

1. 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：用来从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：指定微秒

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.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/epoll.h>
#include <errno.h>
#include <ctype.h>
#include "mysocket.h"
#define MAXLINE 80
#define SERV_PORT 8888
#define OPEN_MAX 1024
int main(int argc, char *argv[])
{
    int i, j, maxi, listenfd, connfd, sockfd;
    int nready, efd, res;
    ssize_t n;
    char buf[MAXLINE], str[INET_ADDRSTRLEN];
    socklen_t clilen;
    int client[OPEN_MAX];
    struct sockaddr_in cliaddr, servaddr;
    struct epoll_event tep, ep[OPEN_MAX];
    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);
    for (i = 0; i < OPEN_MAX; i++)
        client[i] = -1;
    maxi = -1;
    efd = epoll_create(OPEN_MAX);
    if (efd == -1)
        perr_exit("epoll_create");
    tep.events = EPOLLIN; tep.data.fd = listenfd;
    res = epoll_ctl(efd, EPOLL_CTL_ADD, listenfd, &tep);
    if (res == -1)
        perr_exit("epoll_ctl");
    while(1)
    {
        nready = epoll_wait(efd, ep, OPEN_MAX, -1); /* 阻塞监听*/
        if (nready == -1)
            perr_exit("epoll_wait");
        for (i = 0; i < nready; i++)
        {
            if (!(ep[i].events & EPOLLIN))
                continue;
            if (ep[i].data.fd == listenfd)
            {
                clilen = sizeof(cliaddr);
                connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);

                printf("received from %s at PORT %d\n"
```

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

                for (j = 0; j < OPEN_MAX; j++)
                {
                    if (client[j] < 0)
                    {
                        client[j] = connfd; /* save descriptor */
                        break;
                    }
                }
                if (j == OPEN_MAX)
                    perr_exit("too many clients");
                if (j > maxi)
                    maxi = j; /* max index in client[] array */
                tep.events = EPOLLIN; tep.data.fd = connfd;
                res = epoll_ctl(efd, EPOLL_CTL_ADD, connfd, &tep);
                if (res == -1)
                    perr_exit("epoll_ctl");
            }
            else
            {
                sockfd = ep[i].data.fd;
                n = Read(sockfd, buf, MAXLINE);
                if (n == 0) {
                    for (j = 0; j <= maxi; j++)
                    {
                        if (client[j] == sockfd)
                        {
                            client[j] = -1;
                            break;
                        }
                    }
                    res = epoll_ctl(efd, EPOLL_CTL_DEL, sockfd, NULL);
                    if (res == -1)
                        perr_exit("epoll_ctl");
                    Close(sockfd);
                    printf("client[%d] closed connection\n", j);
                }
                else
                {
                    for (j = 0; j < n; j++)
                        buf[j] = toupper(buf[j]);
                    Writen(sockfd, buf, n);
                }
            }
        }
    }
    close(listenfd);
    close(efd);
    return 0;
}
```

client.c

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "mysocket.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");
        else
            Write(STDOUT_FILENO, buf, n);
    }
    Close(sockfd);
    return 0;
}
```

13 setsockopt

当有一个有相同本地地址和端口的socket1处于TIME_WAIT状态时，而你启动的程序的socket2要占用该地址和端口，使用setsockopt来设置端口复用。

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

sockfd：标识一个套接口的描述字。
level：选项定义的层次；支持SOL_SOCKET、IPPROTO_TCP、IPPROTO_IP和IPPROTO_IPV6。
optname：需设置的选项。SO_BROADCAST允许套接字使用广播、SO_REUSEADDR允许端口复用。

optval：指针，指向存放选项待设置的新值的缓冲区。

optlen：optval缓冲区长度。

端口复用：

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

广播属性:

```
int flag = 1;
int len = sizeof(flag);
setsockopt(listenfd, SOL_SOCKET, SO_BROADCAST, &flag, len);
```

14 UDP广播

Broadcast Address（广播地址）是专门用于同时向网络中所有工作站进行发送的一个地址。在使用TCP/IP协议的网络中，主机标识段host ID为全1的IP地址为广播地址，广播的分组传送给host ID段所涉及的所有计算机。例如，对于10.1.1.0 （255.255.255.0 ）网段，其广播地址为10.1.1.255 （255 即为2 进制的11111111 ），当发出一个目的地址为10.1.1.255 的分组（封包）时，它将被分发给该网段上的所有计算机。

广播地址应用于网络内的所有主机

1)受限广播

它不被路由发送，但会被送到相同物理网络段上的所有主机IP地址的网络字段和主机字段全为1就是地址255.255.255.255

2)直接广播

网络广播会被路由，并会发送到专门网络上的每台主机IP地址的网络字段定义这个网络，主机字段通常全为1，如 192.168.10.255

广播接收端代码:

```
#include<stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#include <arpa/inet.h>
#define MAXBYTES 80
#define BROADCASTIP "192.168.1.255"
int main(int argc, char** argv)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int sockfd;
    int opt = 1;
    int opt_len = sizeof(opt);
    char buf[MAXBYTES] = "helloworld";
    char str[INET_ADDRSTRLEN];
    int i, len;
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(8888);
    bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
    printf("Accepting connections ...\n");

    while (1)
    {
        cliaddr_len = sizeof(cliaddr);
        len = recvfrom(sockfd, buf, MAXBYTES, 0
                , (struct sockaddr *)&cliaddr,  &cliaddr_len);
        if(len != -1)
        {
            buf[len] = '\0';
            printf("Received datagram from %s--%s\n",inet_ntoa(cliaddr.sin_addr),buf);
        }

    }
}
```

广播发送端代码:

```c
#include<stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#include <arpa/inet.h>
#define MAXBYTES 80
#define BROADCASTIP "192.168.1.255"
int main(int argc, char** argv)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int sockfd;
    int opt = 1;
    int opt_len = sizeof(opt);
    char buf[MAXBYTES] = "helloworld";
    char str[INET_ADDRSTRLEN];
    int i, len;
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, BROADCASTIP, &servaddr.sin_addr.s_addr);
    servaddr.sin_port = htons(8888);
    setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &opt, opt_len);
    printf("send now ...\n");
    while (1)
    {
        len = sendto(sockfd, buf, strlen(buf), 0
                , (struct sockaddr *)&servaddr, sizeof(servaddr));
        if(len != -1)
        {
            printf("send to ok\n");
            sleep(5);
        }

    }
}
```

15 域名转换

1. 通过域名获取ip地址等信息

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
/*
    name：指向主机名的指针。
    返回一个hostent指针记录着主机信息。
*/
struct hostent
{
    char *h_name; //表示的是主机的规范名。例如www.google.com的规范名其实是www.l.google.com
    char ** h_aliases;//表示的是主机的别名.www.google.com就是google他自己的别名。有的时候，有的主机可能有好几个别名，这些，其实都是为了易于用户记忆而为自己的网站多取的名字。
    short h_addrtype;//表示的是主机ip地址的类型，到底是ipv4(AF_INET)，还是pv6(AF_INET6)
    short h_length;//表示的是主机ip地址的长度
    char ** h_addr_list;//表示的是主机的ip地址
};
```

示例代码：

```
#include <stdio.h>
#include <netdb.h>
#include <arpa/inet.h>
int main(void)
{
    struct hostent* hent;
    int i = 0;
    char addr[16];

    hent = gethostbyname("www.baidu.com");
    printf("h_name: %s\n", hent->h_name);

    while (hent->h_aliases[i] != NULL)
        printf("aliase:%s\n", hent->h_aliases[i++]);

    i = 0;
    while(hent->h_addr_list[i] != NULL)
        printf("ip addr %s\n", inet_ntop(hent->h_addrtype, hent->h_addr_list[i++], addr, sizeof(addr)));

    return 0;
}
```

1. 通过ip地址获取到规范名别名等信息

```
#include <netdb.h>
struct hostent * gethostbyaddr(const  void * addr, socklen_t len, int family);
/*
返回：若成功则为非空指针，若出错则为NULL且设置h_errno
*/
```