国嵌

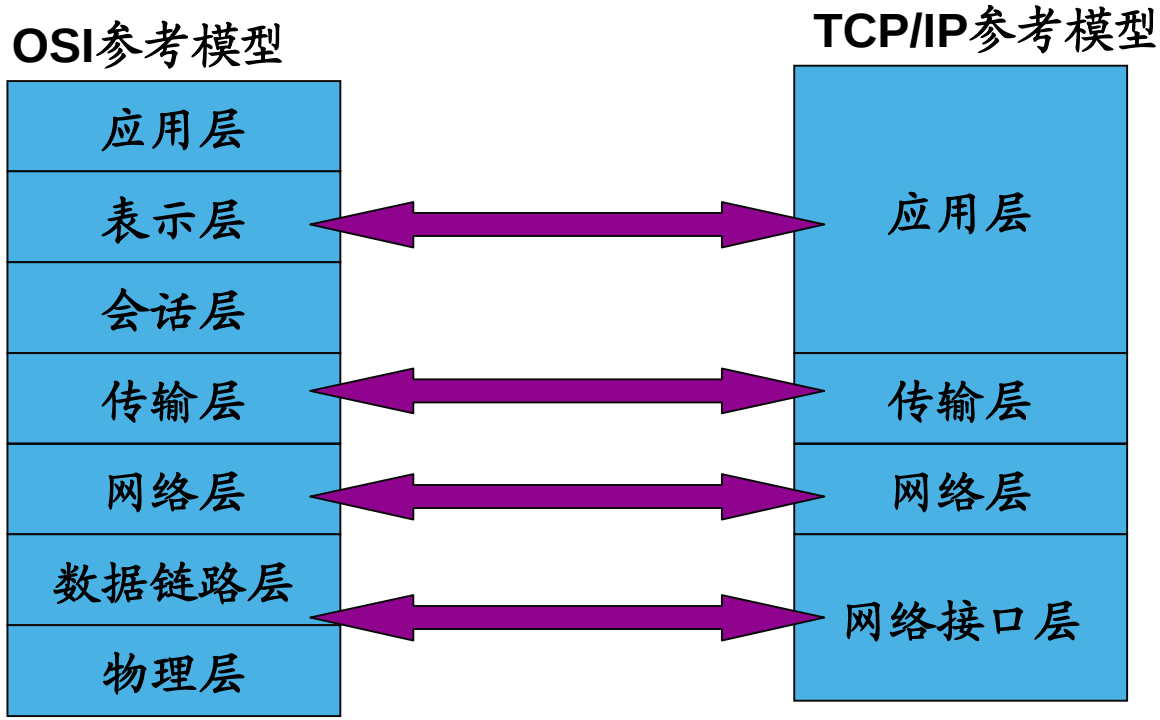
www.enjoylinux.cn

LINUX
网络应用程序设计

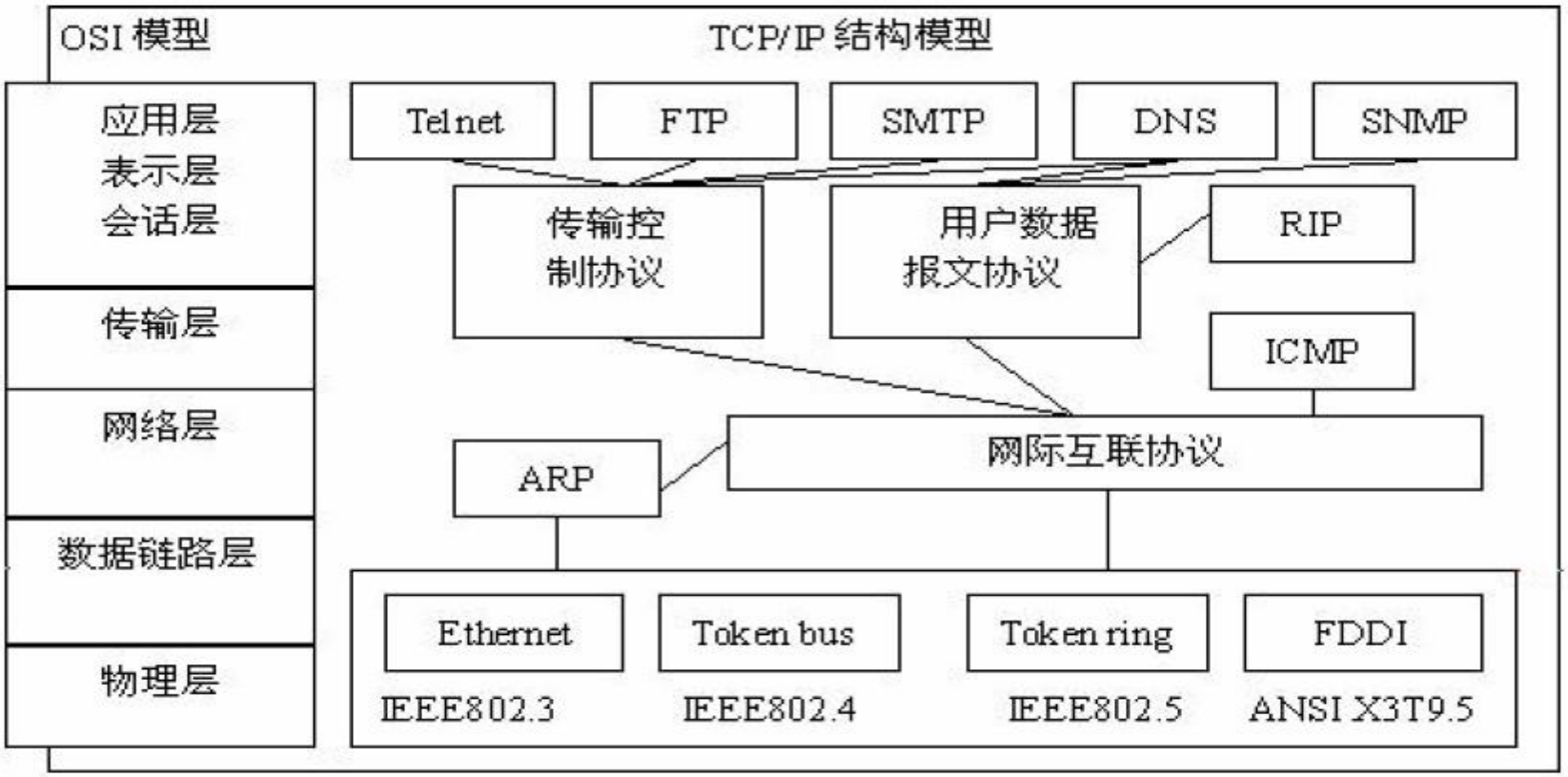
Contents

Contents

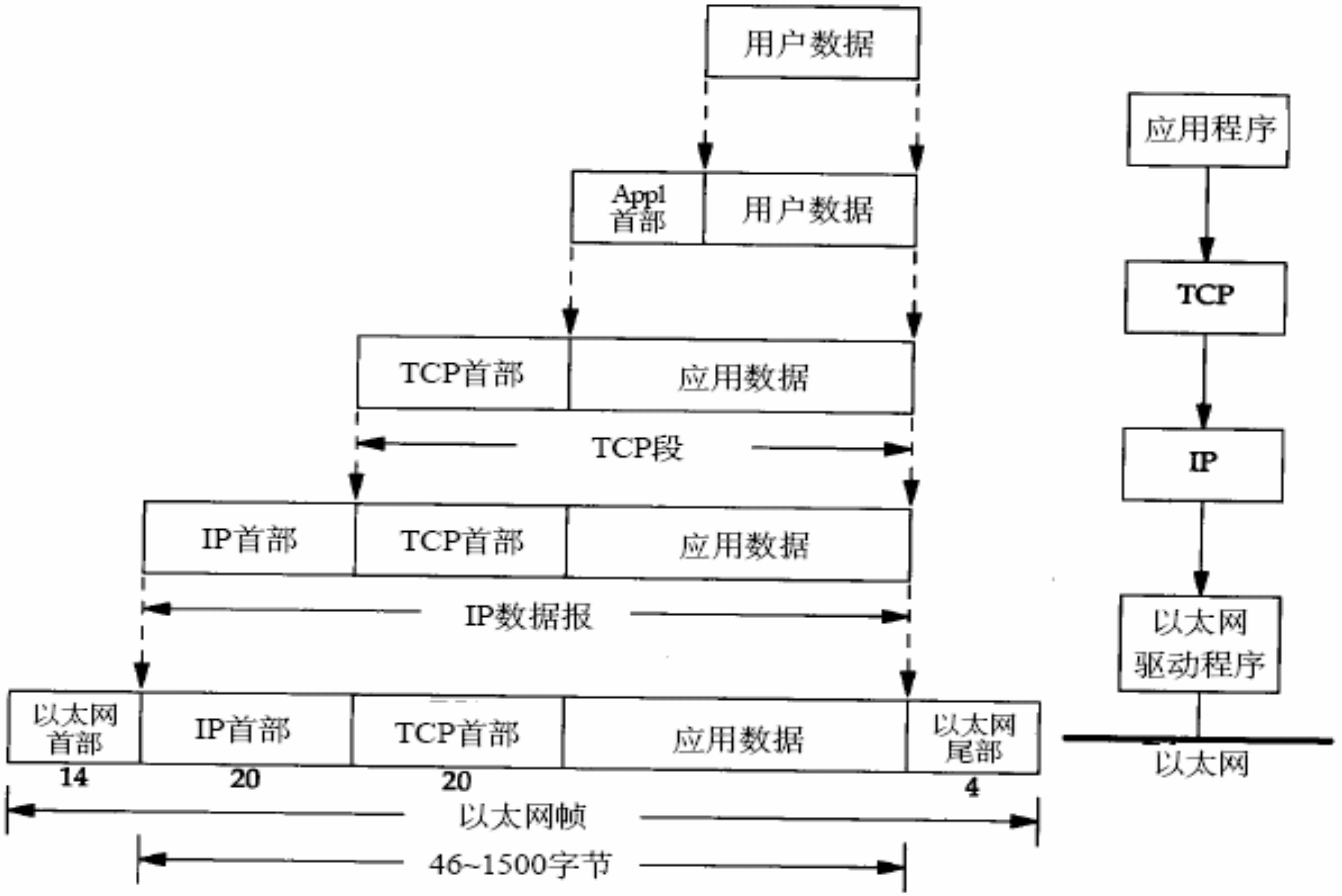
网络模型



网络模型



数据封装（结合wireshark）



TCP/IP协议族

TCP/IP 实际上一个协同工作的通信家族，为网络数据通信提供通路。为讨论方便可TCP/IP 协议组大体上分为三部分:

- Internet 协议（IP）
- 传输控制协议（TCP）和用户数据报协议（UDP）
- 处于 TCP 和 UDP 之上的一组应用协议。它们包括: TELNET，文件传送协议（FTP），域名服务（DNS）和简单的邮件传送程序（SMTP）等。

网络层

第一部分称为网络层。主要包括Internet 协议（IP）、网际控制报文协议（ICMP）和地址解析协议（ARP）：

- Internet 协议（IP）

该协议被设计成互联分组交换通信网，以形成一个网际通信环境。它负责在源主机和目的地主机之间传输来自其较高层软件的称为数据报文的数据块，它在源和目的地之间提供非连接型传递服务。

网络层

- 网际控制报文协议（ICMP）

  它实际上不是IP层部分，但直接同IP层一起工作，报告网络上的某些出错情况。允许网际路由器传输差错信息或测试报文。

- 地址解析协议（ARP）

  ARP 实际上不是网络层部分，它处于IP和数据链路层之间，它是在32位IP地址和48位物理地址之间执行翻译的协议。

传输层协议

第二部分是传输层协议，包括传输控制协议和用户数据报文协议。

- 传输控制协议（TCP）：

该协议对建立网络上用户进程之间的对话负责，它确保进程之间的可靠通信，所提供的功能如下：

1. 监听输入对话建立请求
2. 请求另一网络站点对话
3. 可靠的发送和接收数据
4. 适度的关闭对话

传输层协议

- 用户数据报文协议（UDP）：

  UDP 提供不可靠的非连接型传输层服务，它允许在源和目的地之间传送数据，而不必在传送数据之前建立对话。它主要用于那些非连接型的应用程序，如：视频点播。

应用协议

这部分主要包括Telnet，文件传送协议（FTP 和TFTP），简单文件传送协议（SMTP）和域名服务（DNS）等协议。

IP协议

IP主要有以下四个主要功能:

- 数据传送
- 寻址
- 路由选择
- 数据报文的分段

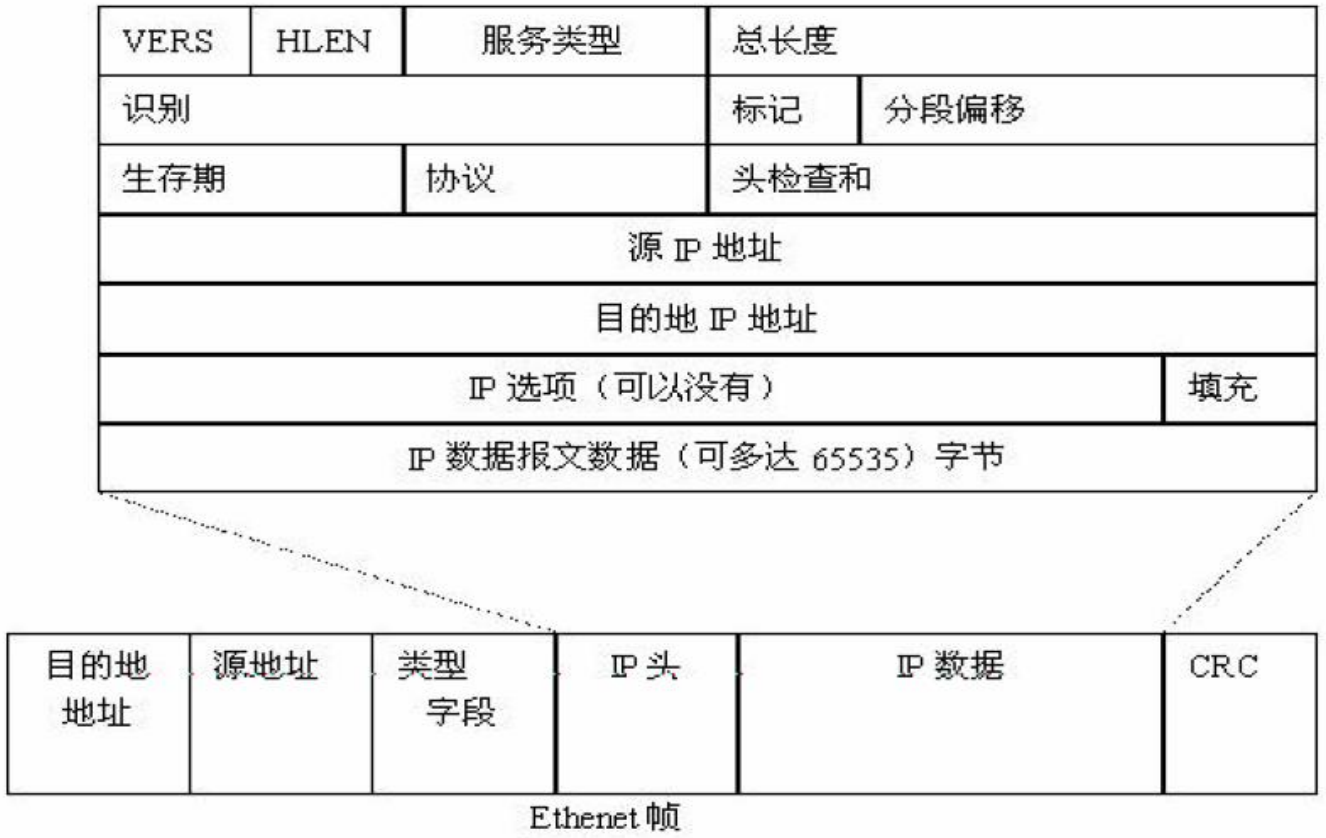
IP协议

IP的主要目的是为数据输入/输出网络提供基本算法，为高层协议提供无连接的传送服务。这意味着在IP将数据递交给接收站点以前不在传输站点和接收站点之间建立对话。它只是封装和传递数据，但不向发送者或接收者报告包的状态，不处理所遇到的故障。

IP协议

IP包由IP协议头与协议数据两部分构成。

IP协议头



IP协议头

www.enjoylinux.cn

结合wireshark分析

TCP协议

TCP是重要的传输层协议，目的是允许数据同网络上的其他节点进行可靠的交换。它能提供端口编号的译码，以识别主机的应用程序，而且完成数据的可靠传输。

- TCP 协议具有严格的内装差错检验算法确保数据的完整性。
- TCP 是面向字节的顺序协议，这意味着包内的每个字节被分配一个顺序编号，并分配给每包一个顺序编号。

TCP协议头

TCP协议头

www.enjoylinux.cn

结合wireshark分析

UDP协议

UDP也是传输层协议，它是无连接的，不可靠的传输服务。当接收数据时它不向发送方提供确认信息，它不提供输入包的顺序，如果出现丢失包或重份包的情况，也不会向发送方发出差错报文。由于它执行功能时具有较低的开销，因而执行速度比TCP快。

UDP协议头

www.enjoylinux.cn

源端口	目的地端口
报文长度	校验和
数据	
数据	
数据	

目的地地址	源地址	类型字段	IP 头	IP 数据	CRC

Ethenet 帧

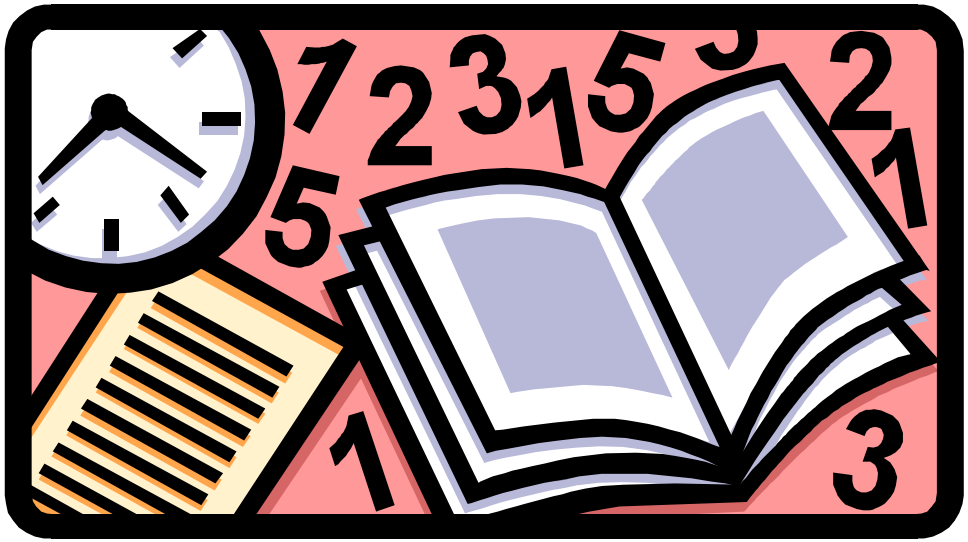
UDP协议头

结合wireshark分析

Contents

Socket

Linux中的网络编程通过Socket(套接字)接口实现，Socket是一种文件描述符。



类型

套接字socket有三种类型：

- 流式套接字（SOCK_STREAM）

  流式的套接字可以提供可靠的、面向连接的通讯流。它使用了TCP协议。TCP保证了数据传输的正确性和顺序性。

类型

- 数据报套接字（SOCK_DGRAM）

  数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证可靠，无差错,它使用数据报协议UDP。

- 原始套接字

  原始套接字允许对低层协议如IP或ICMP直接访问，主要用于新的网络协议的测试等。

地址结构


```
struct sockaddr
{
    u_short sa_family;
    char sa_data[14];
}
```

- **Sa_family:**

  地址族，采用“AF_xxx”的形式，如： AF_INET。

- **Sa_data:**

  14字节的特定协议地址。

地址结构


```
struct sockaddr_in
{
  short int sin_family;  /* Internet地址族 */
  unsigned short int sin_port;  /* 端口号 */
  struct in_addr sin_addr;   /* IP地址 */
  unsigned char sin_zero[8];  /* 填0 */
}
```

编程中一般并不直接针对sockaddr数据结构操作，而是使用与sockaddr等价的sockaddr_in数据结构

地址结构


```
struct  in_addr
{
   unsigned  long  s_addr;
}
```

S_addr: 32位的地址。

地址转换

IP地址通常由数字加点(192.168.0.1)的形式表示，而在struct in_addr中使用的是IP地址是由32位的整数表示的，为了转换我们可以使用下面两个函数:

- int inet_aton(const char *cp,struct in_addr *inp)
- char *inet_ntoa(struct in_addr in)

函数里面 a 代表 ascii n 代表network.第一个函数表示将a.b.c.d形式的IP转换为32位的IP,存储在 inp指针里面。第二个是将32位IP转换为a.b.c.d的格式。

字节序转换

不同类型的 CPU 对变量的字节存储顺序可能不同：有的系统是高位在前，低位在后，而有的系统是低位在前，高位在后，而网络传输的数据顺序是一定要统一的。所以当内部字节存储顺序和网络字节顺序不同时，就一定要进行转换。

字节序转换

32bit的整数(0x01234567)从地址0x100开始:

- 小端字节序:

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

- 大端字节序:

	0x100	0x101	0x102	0x103	
...	01	23	45	67	...

字节序转换

网络字节顺序是TCP/IP中规定好的一种数据表示格式，它与具体的CPU类型、操作系统等无关，从而可以保证数据在不同主机之间传输时能够被正确解释。网络字节顺序采用big endian排序方式。

字节序转换

www.enjoylinux.cn

为什么要进行字节序转换?

例:

INTEL的CPU使用的小端字节序MOTOROLA 68k系列CPU使用的是大端字节序 MOTOROLA发一个16位数据0X1234给INTEL, 传到INTEL时 ,就被INTEL解释为0X3412 。

字节序转换

- htons

  把unsigned short类型从主机序转换到网络序

- htonl

  把unsigned long类型从主机序转换到网络序

- ntohs

  把unsigned short类型从网络序转换到主机序

- ntohl

  把unsigned long类型从网络序转换到主机序

IP与主机名

在网络上标识一台机器可以用IP，也可以使用主机名。

```
struct hostent *gethostbyname(const char *hostname)
```

```
struct hostent
{
        char *h_name;          /* 主机的正式名称 */
        char *h_aliases;       /* 主机的别名 */
        int   h_addrtype;      /* 主机的地址类型  AF_INET*/
        int   h_length;        /* 主机的地址长度 */
        char **h_addr_list;    /* 主机的IP地址列表 */
}
```

```
#define h_addr h_addr_list[0]  /* 主机的第一个IP地址*/
```


地址转换

IP地址通常由数字加点(192.168.0.1)的形式表示，而在struct in_addr中使用的是IP地址是由32位的整数表示的，为了转换我们可以使用下面两个函数:

- int inet_aton(const char *cp,struct in_addr *inp)
- char *inet_ntoa(struct in_addr in)

函数里面 a 代表 ascii n 代表network.第一个函数表示将a.b.c.d形式的IP转换为32位的IP,存储在 inp指针里面。第二个是将32位IP转换为a.b.c.d的格式。

函数

进行Socket编程的常用函数有：

- **socket**

  创建一个socket。

- **bind**

  用于绑定IP地址和端口号到socket。

- **connect**

  该函数用于绑定之后的client端，与服务器建立连接。

操作函数

- **listen**

  设置能处理的最大连接要求，Listen()并未开始接收连线，只是设置socket为listen模式。

- **accept**

  用来接受socket连接。

- **send**

  发送数据

- **recv**

  接收数据

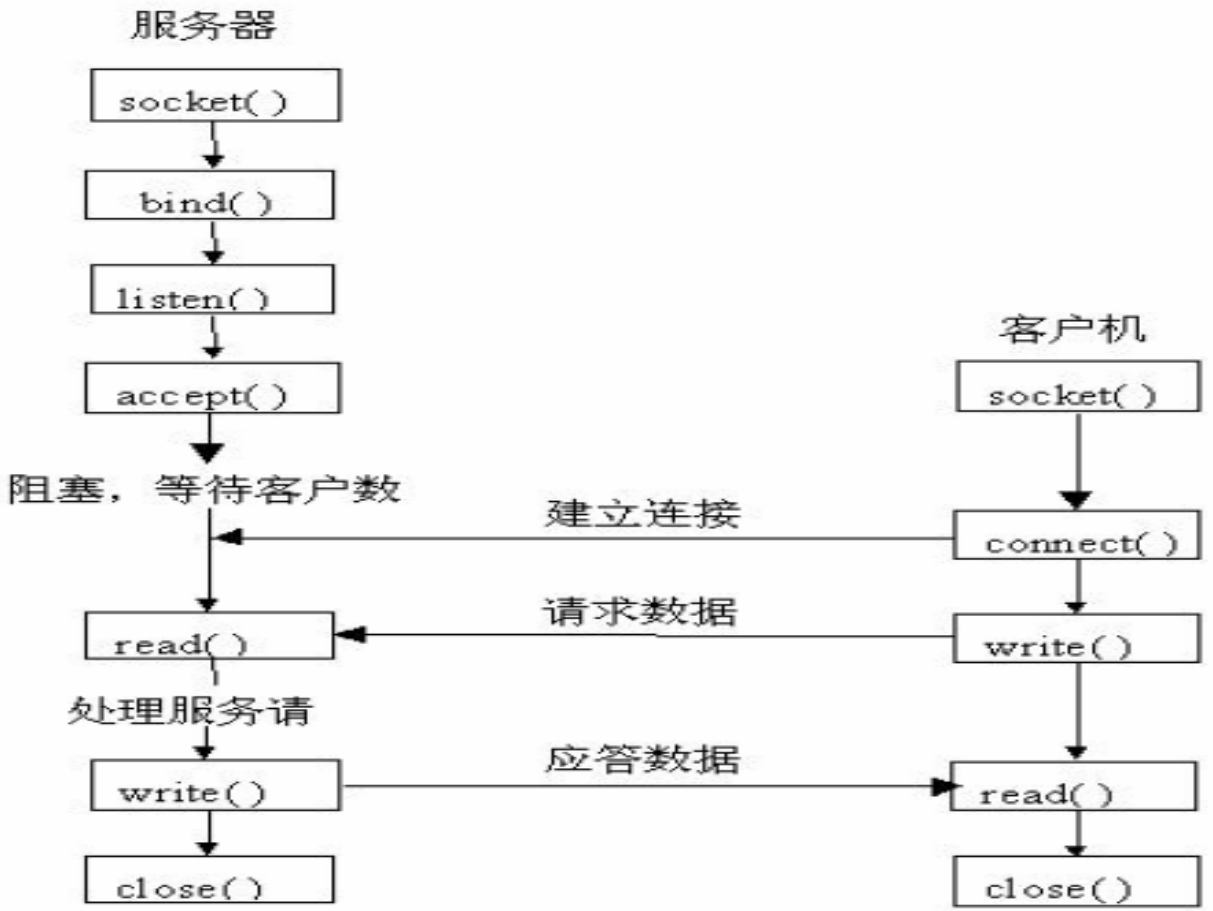
基于TCP-服务器

1. 创建一个socket，用函数socket()
2. 绑定IP地址、端口等信息到socket上，用函数bind()
3. 设置允许的最大连接数，用函数listen()
4. 接收客户端上来的连接，用函数accept()
5. 收发数据，用函数send()和recv()，或者read()和write()
6. 关闭网络连接

基于TCP-客户端

1. 创建一个socket，用函数socket()
2. 设置要连接的对方的IP地址和端口等属性
3. 连接服务器，用函数connect()
4. 收发数据，用函数send()和recv()，或者read()和write()
5. 关闭网络连接

基于TCP



实例

tcp_server.c

tcp_client.c

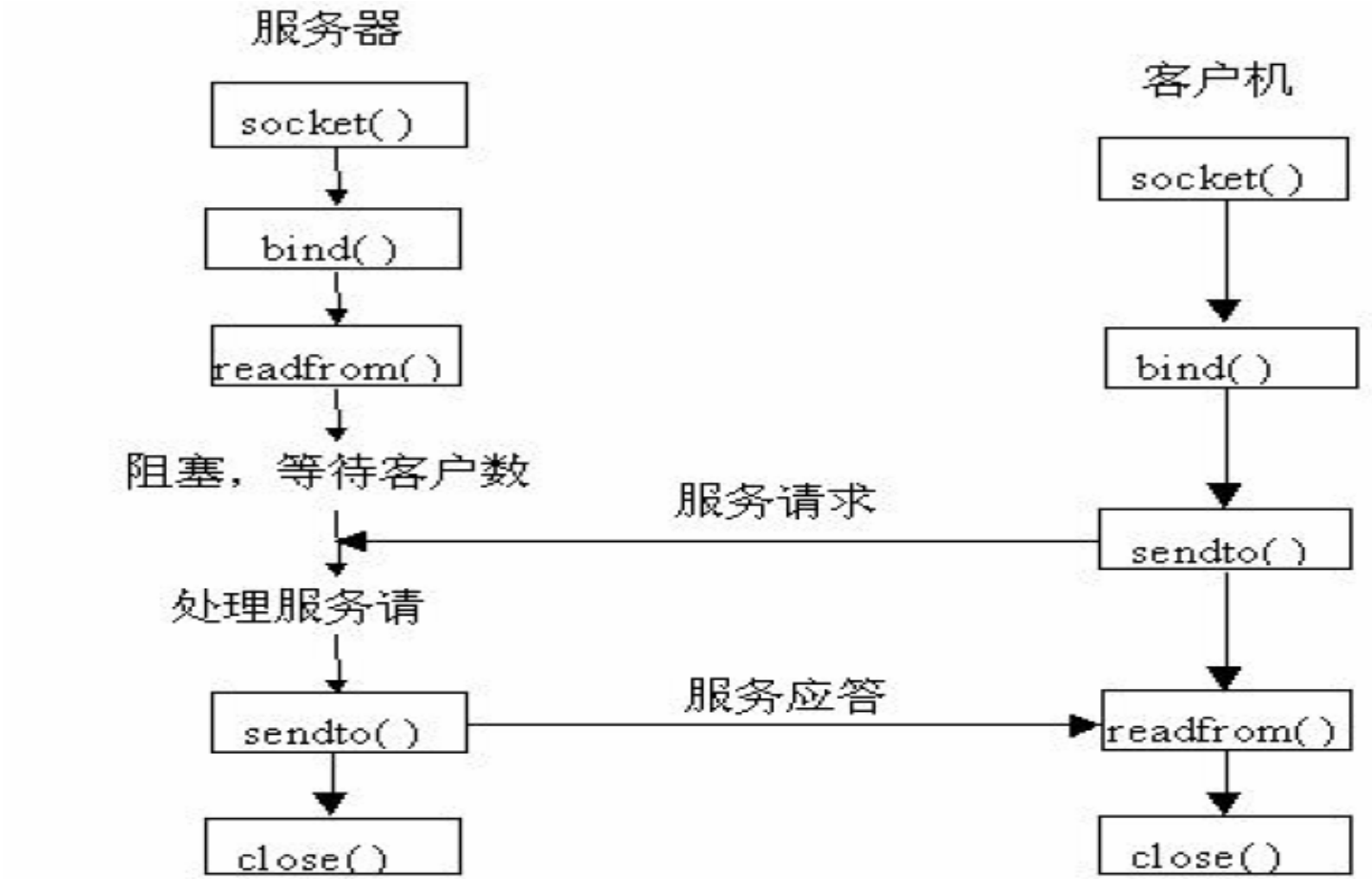
基于UDP-服务器

1. 创建一个socket，用函数socket()
2. 绑定IP地址、端口等信息到socket上，用函数bind()
3. 循环接收数据，用函数recvfrom()
4. 关闭网络连接

基于UDP-客户端

1. 创建一个socket，用函数socket()
2. 绑定IP地址、端口等信息到socket上，用函数bind()
3. 设置对方的IP地址和端口等属性
4. 发送数据，用函数sendto()
5. 关闭网络连接

基于UDP



实例

udp_server.c

udp_client.c

服务器模型

在网络程序里面,一般来说都是许多客户对应一个服务器，为了处理客户的请求,对服务端的程序就提出了特殊的要求。目前最常用的服务器模型有:

- 循环服务器:服务器在同一个时刻只可以响应一个客户端的请求
- 并发服务器:服务器在同一个时刻可以响应多个客户端的请求

UDP循环服务器

UDP循环服务器的实现方法:UDP服务器每次从套接字上读取一个客户端的请求->处理->然后将结果返回给客户机。

```
socket(...);
bind(...);
while(1)
 {
     recvfrom(...);
     process(...);
     sendto(...);
}
```

因为UDP是非面向连接的,没有一个客户端可以老是占住服务端， 服务器对于每一个客户机的请求总是能够满足。

TCP循环服务器

TCP服务器接受一个客户端的连接,然后处理,完成了这个客户的所有请求后,断开连接。算法如下:

```
socket(...);
bind(...);
listen(...);
while(1)
{
    accept(...);
    process(...);
    close(...);
}
```


TCP循环服务器

TCP循环服务器一次只能处理一个客户端的请求。只有在这个客户的所有请求都满足后,服务器才可以继续后面的请求。这样如果有一个客户端占住服务器不放时,其它的客户机都不能工作了，因此,TCP服务器一般很少用循环服务器模型的。

TCP并发服务器

并发服务器的思想是每一个客户机的请求并不由服务器直接处理,而是由服务器创建一个 子进程来处理。算法如下:

```
socket(...);
bind(...);
listen(...);
while(1) {
        accept(...);
        if(fork(..)==0) {
            process(...);
            close(...);
            exit(...);
        }
        close(...);
}
```


TCP并发服务器

TCP并发服务器可以解决TCP循环服务器客户机独占服务器的情况。但同时也带来了问题：为了响应客户的请求,服务器要创建子进程来处理，而创建子进程是一种非常消耗资源的操作。

多路复用I/O

阻塞函数在完成其指定的任务以前不允许程序继续向下执行。例如：当服务器运行到accept语句时，而没有客户请求连接，服务器就会停止在accept语句上等待连接请求的到来。这种情况称为阻塞（blocking），而非阻塞操作则可以立即完成。例如，如果你希望服务器仅仅检查是否有客户在等待连接，有就接受连接，否则就继续做其他事情，则可以通过使用select系统调用来实现。除此之外，select还可以同时监视多个套接字 。

多路复用I/O


```
int select(int maxfd, fd_set *readfds, fd_set *writefds, fe_set
    *exceptfds, const struct timeval *timeout)
```

- **Maxfd:** 文件描述符的范围，比待检的最大文件描述符大1
- **Readfds:**被读监控的文件描述符集
- **Writefds:**被写监控的文件描述符集
- **Exceptfds:**被异常监控的文件描述符集
- **Timeout:**定时器

多路复用I/O

Timeout取不同的值，该调用有不同的表现:

- Timeout值为0，不管是否有文件满足要求，都立刻返回，无文件满足要求返回0，有文件满足要求返回一个正值。
- Timeout为NULL，select将阻塞进程，直到某个文件满足要求
- Timeout值为正整数，就是等待的最长时间，即select在timeout时间内阻塞进程。

多路复用I/O

Select调用返回时，返回值有如下情况:

1. 正常情况下返回满足要求的文件描述符个数;
2. 经过了timeout等待后仍无文件满足要求，返回值为0;
3. 如果select被某个信号中断，它将返回-1并设置errno为EINTR。
4. 如果出错，返回-1并设置相应的errno。

多路复用I/O

1. 设置要监控的文件
2. 调用Select开始监控
3. 判断文件是否发生变化

多路复用I/O

系统提供了4个宏对描述符集进行操作:

```
#include <sys/select.h>
void FD_SET(int fd, fd_set *fdset)
void FD_CLR(int fd, fd_set *fdset)
void FD_ZERO(fd_set *fdset)
void FD_ISSET(int fd, fd_set *fdset)
```

宏FD_SET将文件描述符fd添加到文件描述符集fdset中;

宏FD_CLR从文件描述符集fdset中清除文件描述符fd;

宏FD_ZERO清空文件描述符集fdset;

在调用select后使用FD_ISSET来检测文件描述符集fdset中的文件fd发生了变化。

多路复用I/O


```
FD_ZERO(&fds); //清空集合
sock1 = socket(……);
sock2 = socket(……);
bind(sock1,…);
bind(sock2,…);
listen(sock1,…);
listen(sock1,…);
FD_SET(sock1,&fds); //设置描述符
FD_SET(sock2,&fds); //设置描述符
maxfdp=(sock1>sock2?sock1:sock2) + 1;
switch(select(maxfdp,&fds,NULL,NULL,&timeout))
  case -1: exit(-1);break; //select错误，退出程序
  case 0:break;
  default:
  if(FD_ISSET(sock1,&fds)) //测试sock1是否可读
   accpet(sock1,…)
     ……
```