华清远见 FARSIGHT 嵌入式培训专家

嵌入式学院 华清远见旗下品牌

文件I/O

课程内容

- 文件IO编程
- 网络编程

课程目标

- 了解文件概念
- 掌握IO函数
  - open()/creat()/close()/read()/write()/lseek()…
- 目录操作

文件概念：

- 定义：

  文件：一组相关数据的有序集合。

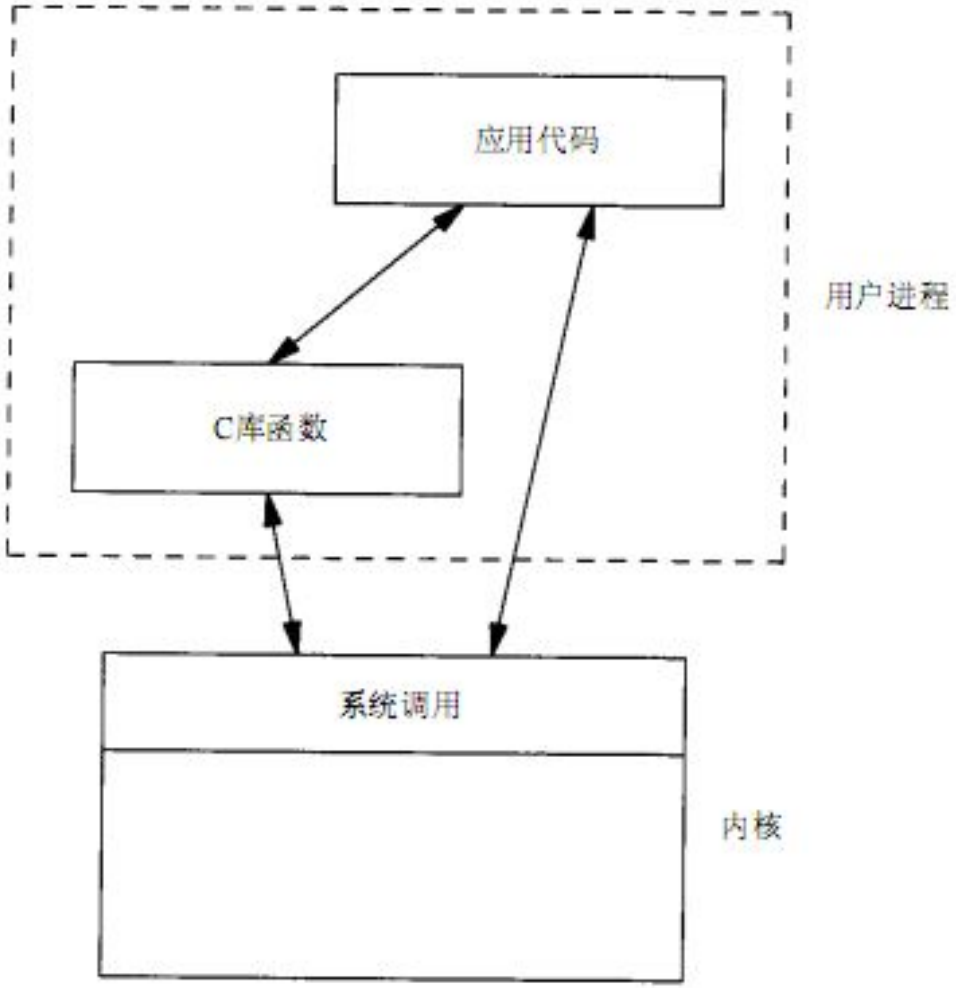
  文件名：这个数据集合的名称。

- 按类型分类：
  - 常规文件
    - ASCII码文件
    - 二进制的文件
  - 目录
  - 字符设备
  - 块设备
  - 有名管道
  - 套接口
  - 符号链接

UNIX - 输入和输出

1. 文件描述符
   ① 顺序分配的非负整数
   ② 内核用以标识一个特定进程正在访问的文件
   ③ 其他资源(socket、pipe等)的访问标识
2. 标准输入、标准输出和标准出错
   ① 由shell默认打开，分别为0/1/2
3. 不用缓存的I/O
   ① 通过文件描述符进行访问
   ② open()/read()/write()/lseek()/close()…
4. 标准I/O
   ① 通过FILE*进行访问
   ② printf()/fprintf()/fopen()/fread()/fwrite()/fseek()/fclose()…

1.8 UNIX基础知识-系统调用和库函数



1.8 UNIX基础知识-系统调用和库函数

1. 系统调用
   - ① 用户空间进程访问内核的接口
   - ② 把用户从底层的硬件编程中解放出来
   - ③ 极大的提高了系统的安全性
   - ④ 使用户程序具有可移植性
2. 库函数
   - ① 库函数为了实现某个功能而封装起来的API集合
   - ② 提供统一的编程接口，更加便于应用程序的移植

1.8 UNIX基础知识-系统调用和库函数

系统调用与库函数的比较

① 所有的操作系统都提供多种服务的入口点，通过这些入口点，程序向内核请求服务

② 从执行者的角度来看，系统调用和库函数之间有重大的区别，但从用户的角度来看，其区别并不非常的重要。

③ 应用程序可以调用系统调用或者库函数，库函数则会调用系统调用来完成其功能。

④ 系统调用通常提供一个访问系统的最小界面，而库函数通常提供比较复杂的功能。

文件I/O函数

文件I/O介绍

- open()
- close()
- read()
- write()
- lseek()

文件I/O – 介绍

文件I/O

① 不带缓冲

- 不带缓冲指的是每个read和write都调用内核中的相应系统调用
- 不带缓冲的I/O函数不是ANSI C的组成部分，但是是POSIX和XPG3的组成部分

② 通过文件描述符来访问文件

文件I/O常用函数

① open()/creat()
② close()
③ read()
④ write()
⑤ lseek()

文件I/O –文件描述符

- 对于内核而言，所有打开文件都由文件描述符引用。
- 文件描述符是一个**非负整数**。当打开一个现存文件或创建一个新文件时，内核向进程返回一个文件描述符。
- 当读、写一个文件时，用open或creat返回的文件描述符标识该文件，将其作为参数传送给read或write。
- 在posix.1应用程序中，幻数0、1、2应被代换成符号常数STDIN_FILENO、STDOUT_FILENO、STDERR_FILENO。这些常数都定义在头文件<unistd.h>中。

文件I/O – open()

原型	int open(const char *pathname, int flags, mode_t mode);		
参数	pathname	被打开的文件名（可包括路径名）。	
	flags	O_RDONLY：只读方式打开文件。	这三个参数互斥
		O_WRONLY：可写方式打开文件。	
		O_RDWR：读写方式打开文件。	
		O_CREAT：如果该文件不存在，就创建一个新的文件，并用第三的参数为其设置权限。	
		O_EXCL：如果使用O_CREAT时文件存在，则可返回错误消息。这一参数可测试文件是否存在。	
		O_NOCTTY：使用本参数时，如文件为终端，那么终端不可以作为调用open()系统调用的那个进程的控制终端。	
		O_TRUNC：如文件已经存在，那么打开文件时先删除文件中原有数据。	
		O_APPEND：以添加方式打开文件，所以对文件的写操作都在文件的末尾进行。	
	mode	被打开文件的存取权限，为8进制表示法。	

文件I/O – read()

调用read()函数可以从一个已打开的可读文件中读取数据。

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

read()调用成功返回读取的字节数，如果返回0，表示到达文件末尾，如果返回-1，表示出错，通过errno设置错误码。

读操作从文件的当前位移量处开始，在成功返回之前，该位移量增加实际读取的字节数。

buf参数需要有调用者来分配内存，并在使用后，由调用者释放分配的内存。

文件I/O – write()

调用write()函数可以向一个已打开的可写文件中写入数据。

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

write()调用成功返回已写的字节数，失败返回-1，并设置errno。

write()的返回值通常与count不同，因此需要循环将全部待写的数据全部写入文件。

write()出错的常见原因：磁盘已满或者超过了一个给定进程的文件长度限制。

对于普通文件，写操作从文件的当前位移量处开始，如果在打开文件时，指定了O_APPEND参数，则每次写操作前，将文件位移量设置在文件的当前结尾处，在一次成功的写操作后，该文件的位移量增加实际写的字节数。

文件I/O – lseek()

调用 lseek()函数可以显示的定位一个已打开的文件。

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

off_t lseek(int fildes, off_t offset, int whence);
```

原型	off_t lseek(int fd,off_t offset,int whence);	
参数	fd：文件描述符。	
	offset：偏移量，每一读写操作所需要移动的距离，单位是字节的数量，可正可负（向前移，向后移）	
	whence (当前位置基点)：	SEEK_SET：当前位置为文件的开头，新位置为偏移量的大小。
		SEEK_CUR：当前位置为文件指针的位置，新位置为当前位置加上偏移量。
		SEEK_END：当前位置为文件的结尾，新位置为文件的大小加上偏移量的大小。
返回值	成功：文件的当前位移	
	-1：出错	

华 清 远 见
FARSIGHT
嵌 入 式 培 训 专 家

嵌入式学院
华清远见旗下品牌

Linux网络编程

主要内容

1. Internet与TCP/IP协议
   - ① Internet历史
   - ② OSI模型与TCP/IP协议体系结构
   - ③ TCP/IP协议

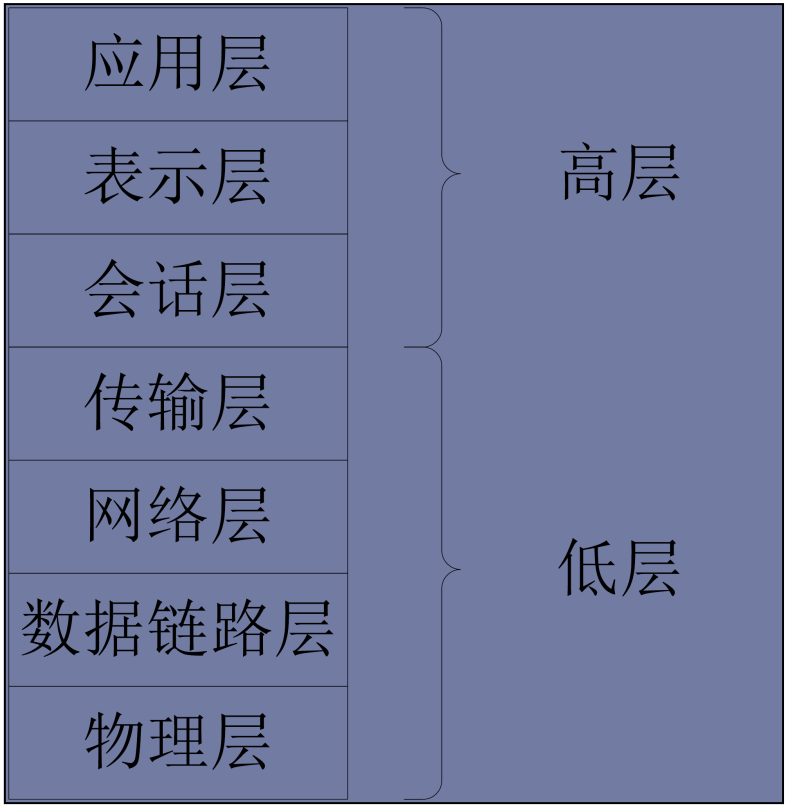
2. TCP/IP网络程序设计
   - ① 预备知识
   - ② TCP服务器/客户
   - ③ 服务器模型

Internet的历史

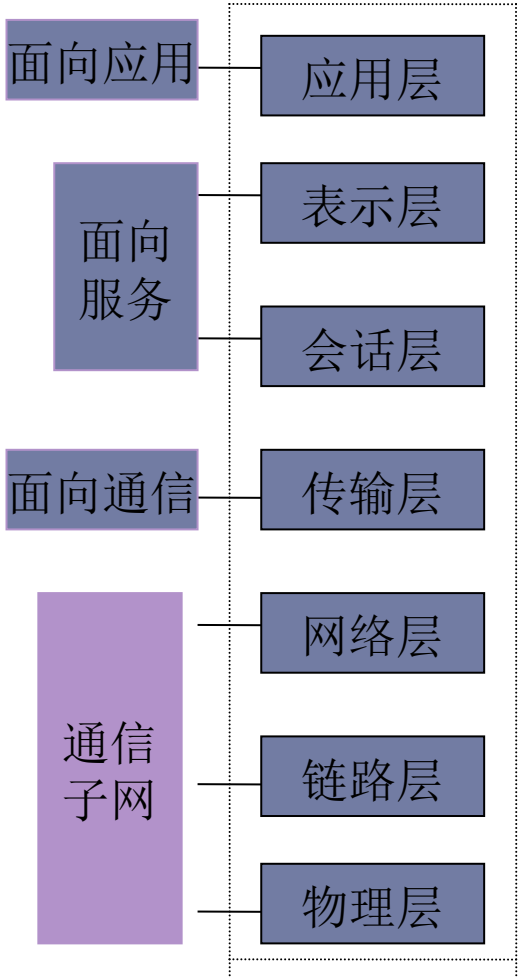
- Internet—“冷战”的产物
  - 1957年10月和11月，前苏联先后有两颗“Sputnik”卫星上天
  - 1958年美国总统艾森豪威尔向美国国会提出建立DARPA (Defense Advanced Research Project Agency)，即国防部高级研究计划署，简称ARPA
  - 1968年6月DARPA提出“资源共享计算机网络” (Resource Sharing Computer Networks)，目的在于让DARPA的所有电脑互连起来，这个网络就叫做ARPAnet，即“阿帕网”，是Interne的最早雏形

OSI开放系统互联模型

- OSI模型相关的协议已经很少使用，但模型本身非常通用
- OSI模型是一个理想化的模型，尚未有完整的实现
- OSI模型共有七层(右图)



OSI模型七层结构



应用程序：FTP、E-mail、Telnet

数据格式定义、数据转换/加密

建立通信进程的逻辑名字与物理名字之间的联系

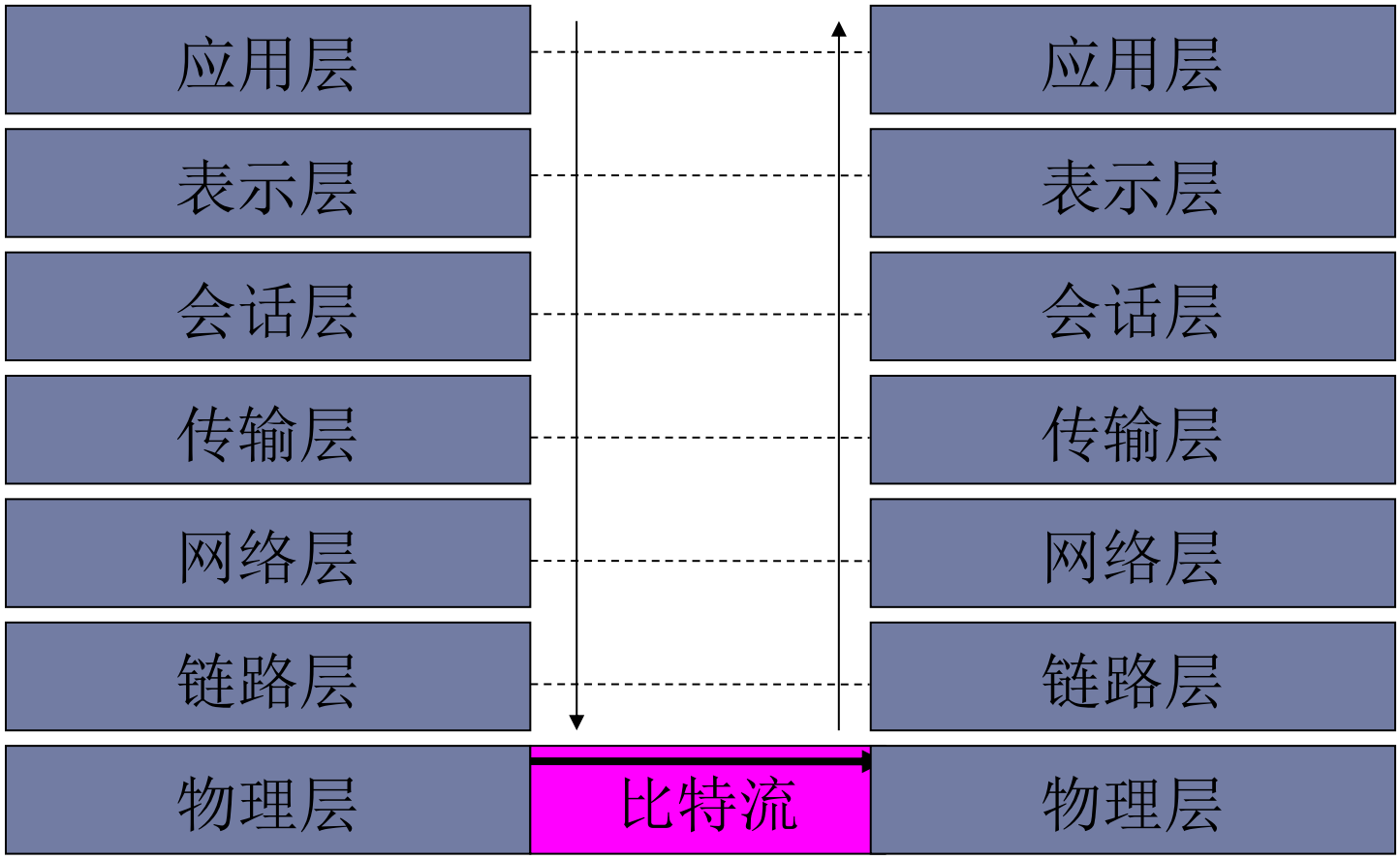
差错处理/恢复，流量控制，提供可靠的数据传输

数据分组、路由选择

数据组成可发送、接收的帧

传输物理信号、接口、信号形式、速率

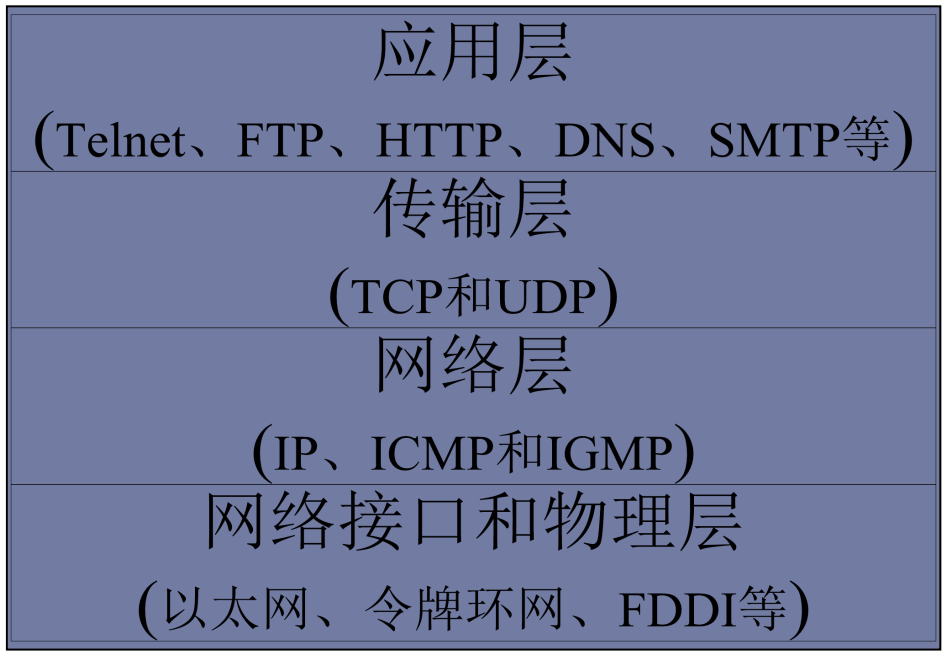
OSI七层结构



TCP/IP协议族的体系结构

- TCP/IP协议是Internet事实上的工业标准。

- 一共有四层



TCP/IP与OSI参考模型的对应关系

OSI模型	TCP/IP协议	
应用层	应用层	Telnet、WWW、FTP等
表示层		
会话层		
传输层	传输层	TCP与UDP
网络层	网络层	IP、ICMP和IGMP
数据链路层	网络接口与物理层	网卡驱动
物理层		物理接口

一个比喻

- 如果把网络数据包的投递过程看成是给远方的一位朋友寄一封信，那么:
  - IP地址就是这位朋友的所在位置，如上海交大XX系，邮局依靠此信息进行信件的投递，网络数据则依靠IP地址信息进行路由
  - 端口号就是这位朋友的名字，传达室依靠这个信息最终把这封信交付给这位收信者，数据包则依靠端口号送达给接收进程

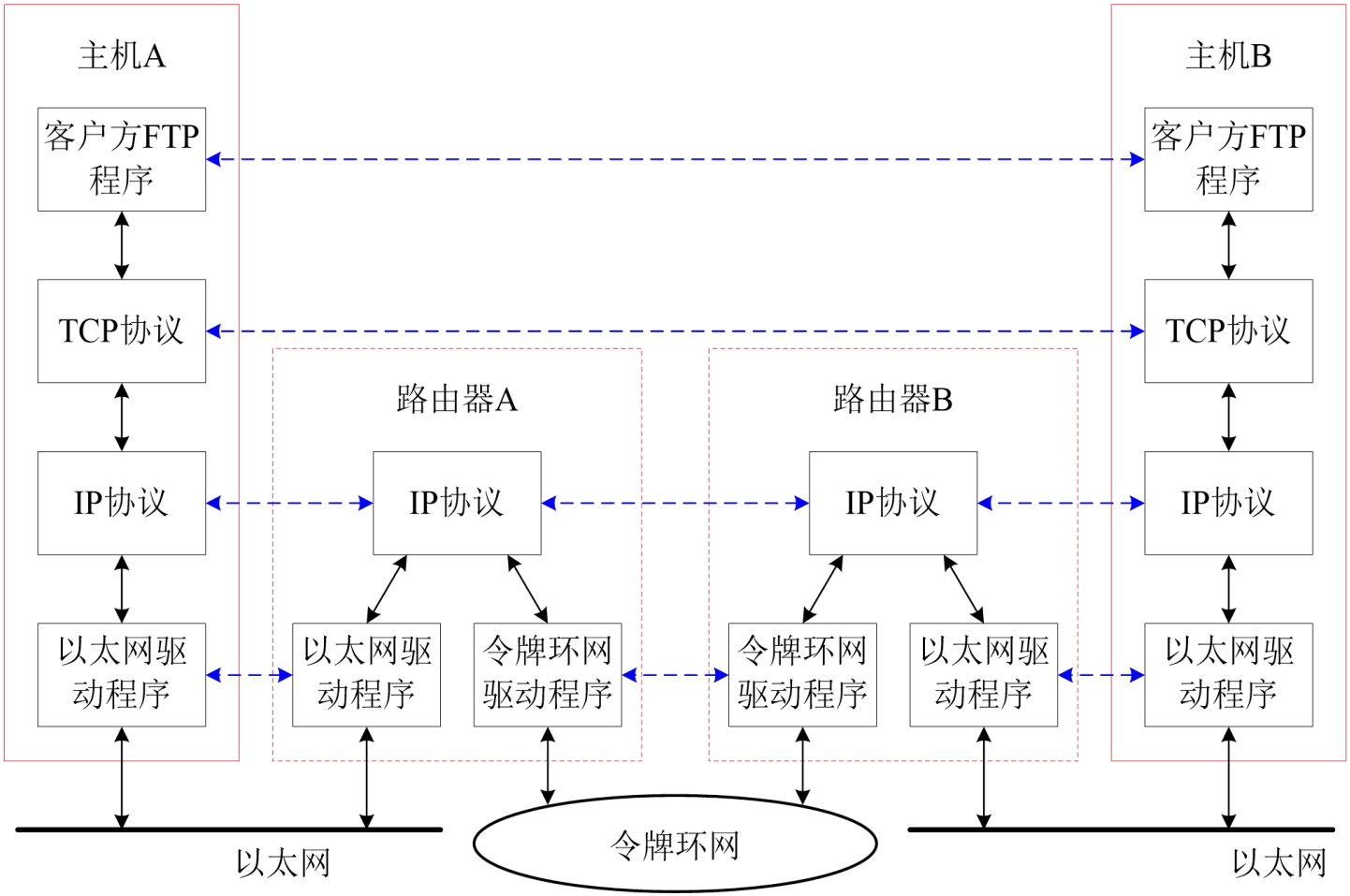
TCP/IP协议

- TCP/IP协议
  - 传输控制/网际协议(Transfer Control Protocol/Internet Protocol) 又称作网络通讯协议
  - Internet国际互联网络的基础，RFC793
  - 一组协议，通常称它为TCP/IP协议族
  - 四个层次：网络接口层、网际层、传输层、应用层

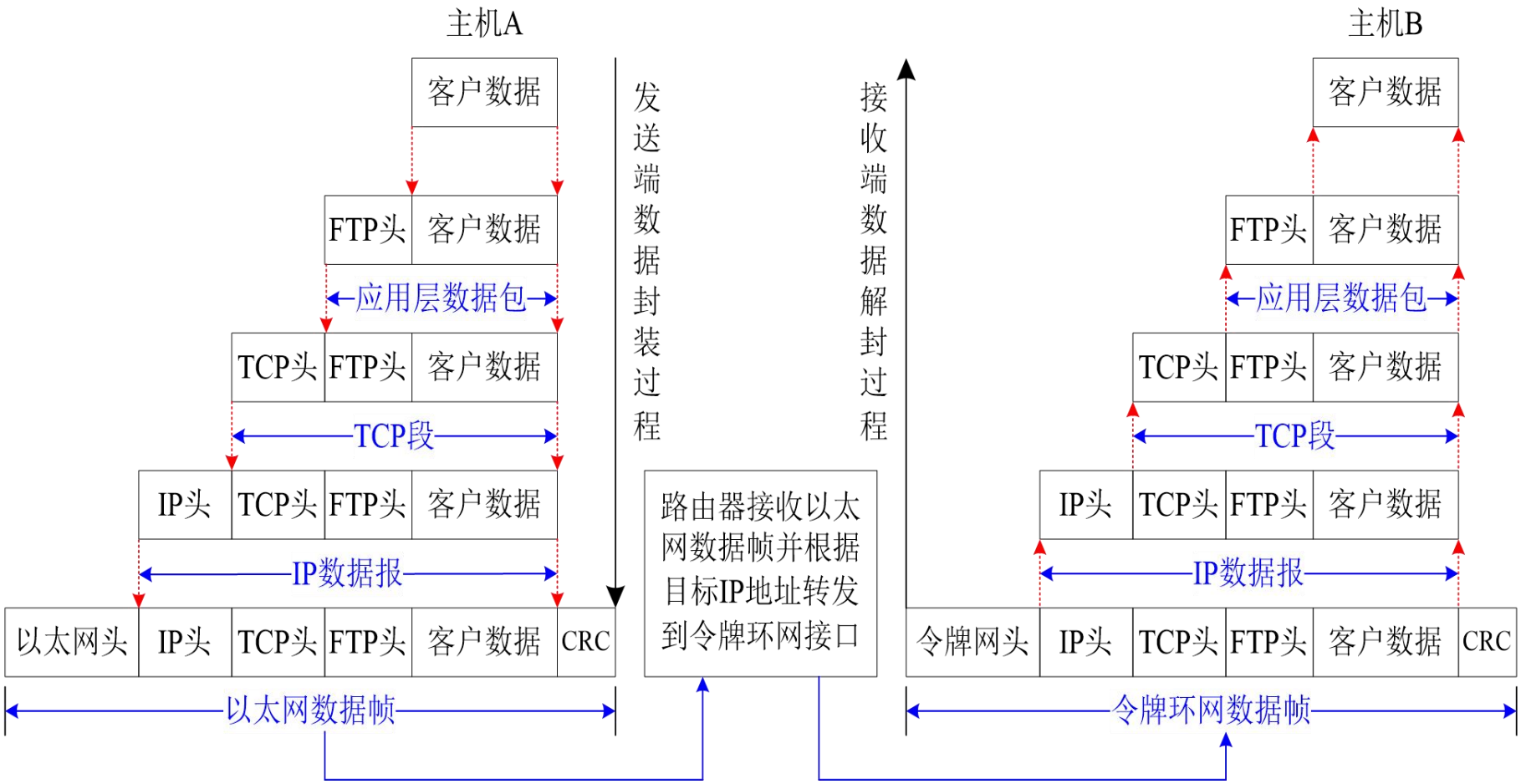
TCP/IP协议族

- 常用协议
  - TCP(Transport Control Protocol)传输控制协议
  - IP(Internetworking Protocol)网间协议
  - UDP(User Datagram Protocol)用户数据报协议
  - ICMP(Internet Control Message Protocol)互联网控制信息协议
  - SMTP(Simple Mail Transfer Protocol)简单邮件传输协议
  - SNMP(Simple Network manage Protocol)简单网络管理协议
  - HTTP(Hypertext Transfer Protocol) 超文本传输协议
  - FTP(File Transfer Protocol)文件传输协议
  - ARP(Address Resolution Protocol)地址解析协议

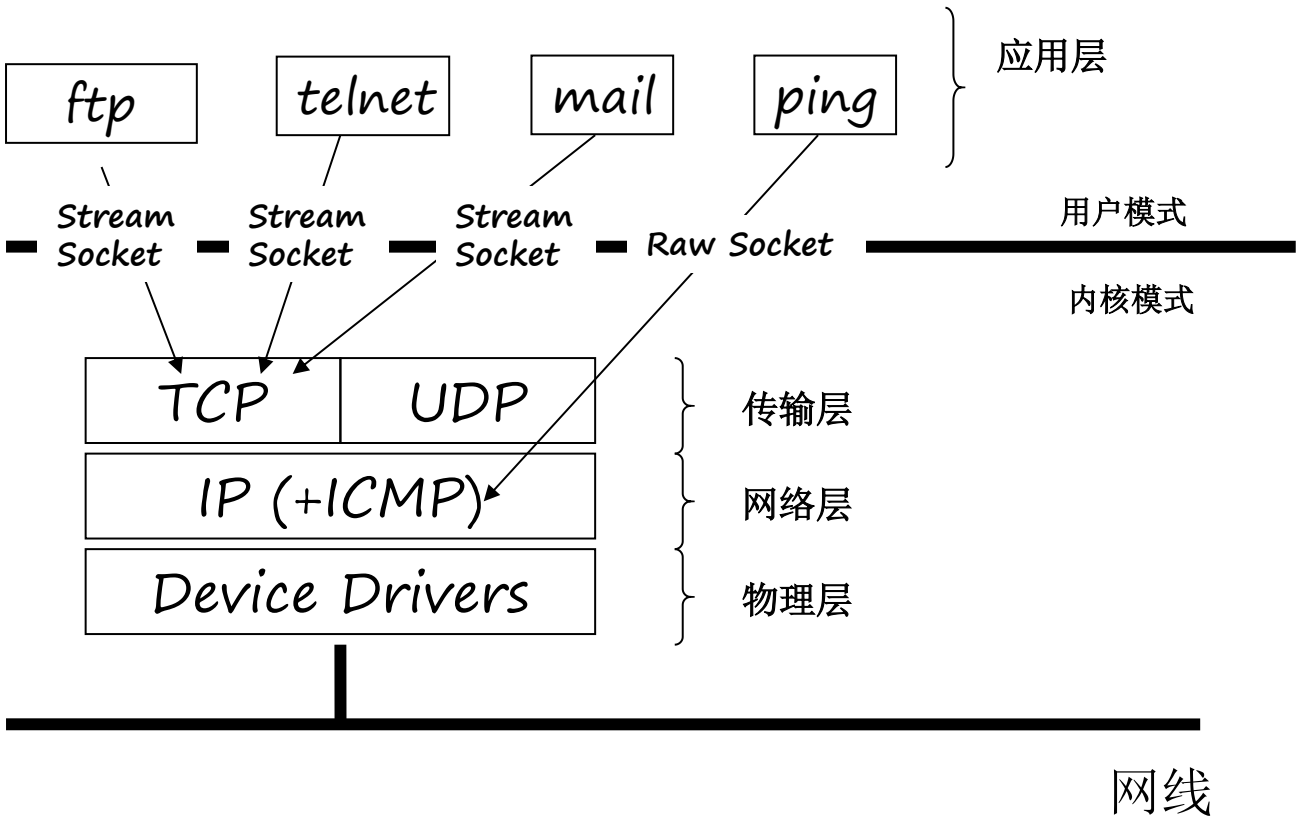
TCP/IP协议通信模型



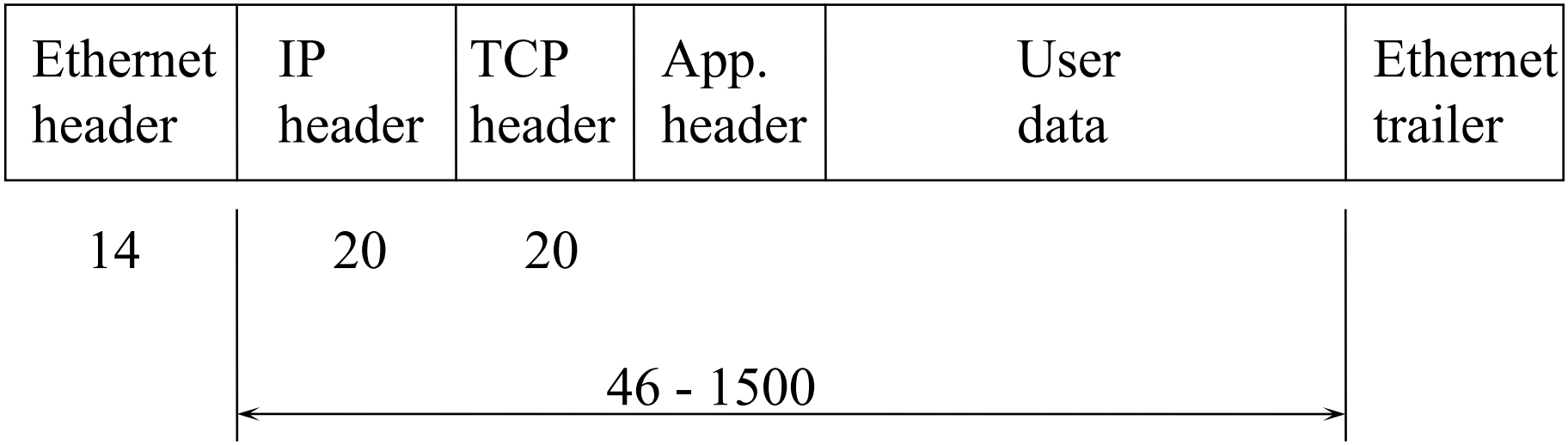
数据的封装与传递过程


主机A
主机B
客户数据
FTP头
客户数据
应用层数据包
TCP头
FTP头
客户数据
TCP段
IP头
TCP头
FTP头
客户数据
IP数据报
以太网头
IP头
TCP头
FTP头
客户数据
CRC
以太网数据帧
发送端数据封装过程
路由器接收以太网数据帧并根据目标IP地址转发到令牌环网接口
接收端数据解封过程
令牌网头
令牌环网数据帧

TCP/IP结构


ftp
telnet
mail
ping
应用层
Stream Socket
Stream Socket
Stream Socket
Raw Socket
用户模式
内核模式
TCP
UDP
传输层
IP (+ICMP)
网络层
Device Drivers
物理层
网线

TCP/IP协议下的数据包

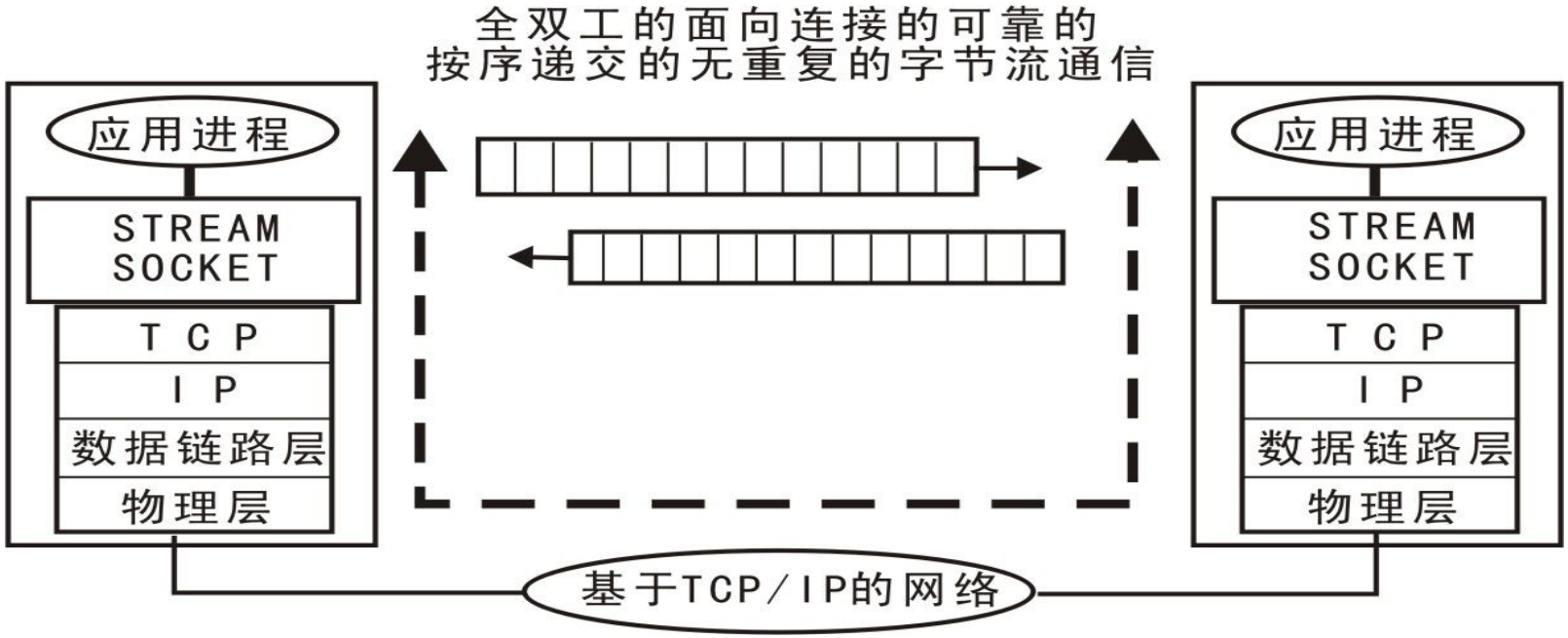


TCP协议特点

- TCP（即传输控制协议）：是一种面向连接的传输层协议，它能提供高可靠性通信(即数据无误、数据无丢失、数据无失序、数据无重复到达的通信)

- 适用情况：
  1. 适合于对传输质量要求较高，以及传输大量数据的通信。
  2. 在需要可靠数据传输的场合，通常使用TCP协议
  3. MSN/QQ等即时通讯软件的用户登录账户管理相关的功能通常采用TCP协议

TCP传输


全双工的面向连接的可靠的
按序递交的无重复的字节流通信
应用进程
STREAM SOCKET
TCP
IP
数据链路层
物理层
应用进程
STREAM SOCKET
TCP
IP
数据链路层
物理层
基于TCP/IP的网络

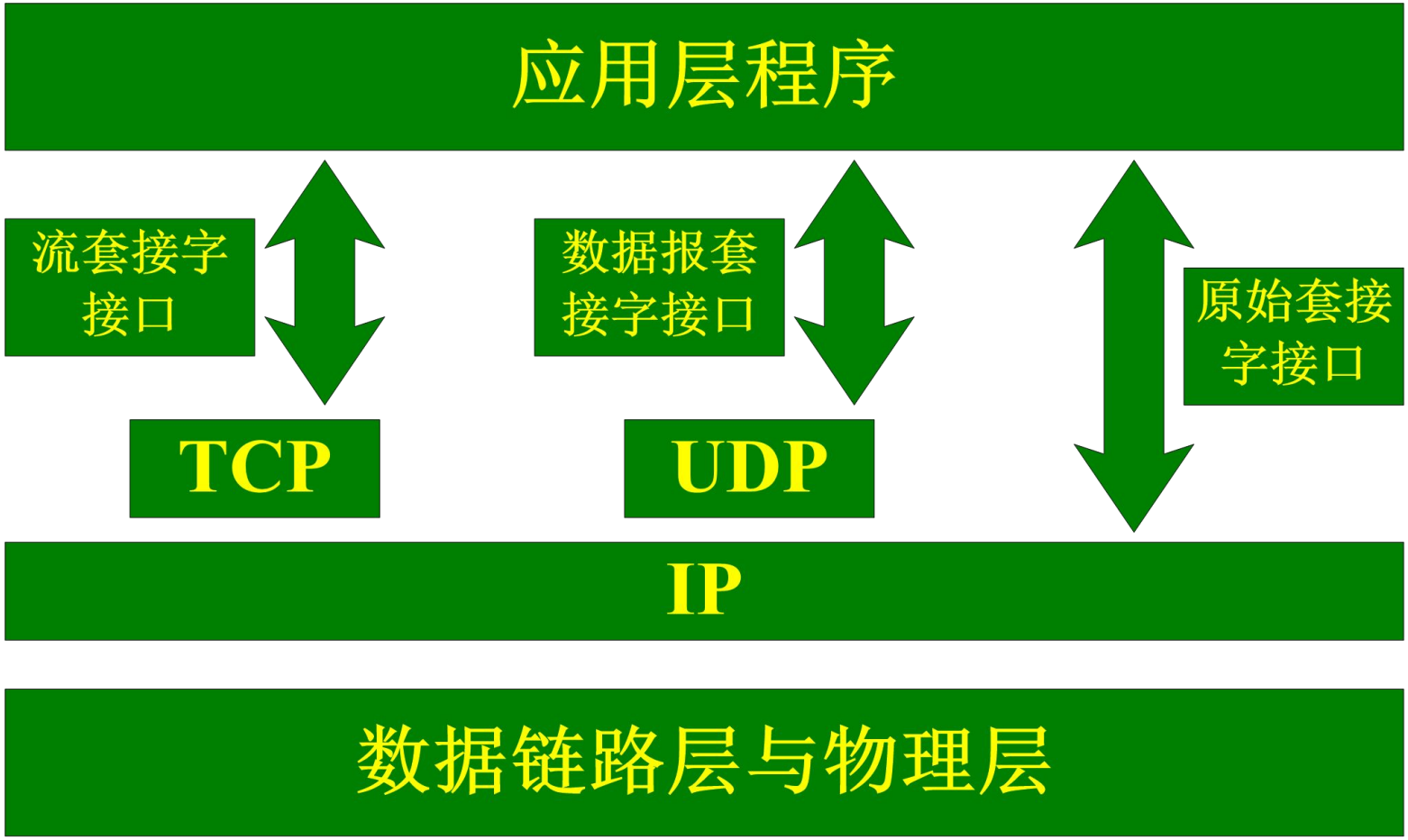
TCP/IP网络编程预备知识

- Socket
- IP地址
- 端口号
- 字节序

Socket类型

- 流式套接字(SOCK_STREAM)
  - 提供了一个面向连接、可靠的数据传输服务，数据无差错、无重复的发送且按发送顺序接收。内设置流量控制，避免数据流淹没慢的接收方。数据被看作是字节流，无长度限制。
- 数据报套接字(SOCK_DGRAM)
  - 提供无连接服务。数据包以独立数据包的形式被发送，不提供无差错保证，数据可能丢失或重复，顺序发送，可能乱序接收。
- 原始套接字(SOCK_RAW)
  - 可以对较低层次协议如IP、ICMP直接访问。

Socket的位置



IP地址

- IP地址是Internet中主机的标识
  - Internet中的主机要与别的机器通信必须具有一个IP地址
  - IP地址为32位（IPv4）或者128位（IPv6）
  - 每个数据包都必须携带目的IP地址和源IP地址，路由器依靠此信息为数据包选择路由
- 表示形式：常用点分形式，如202.38.64.10，最后都会转换为一个32位的无符号整数。
- IP地址分类
- 子网掩码

IP地址的转换

- inet_aton()
  - 将strptr所指的字符串转换成32位的网络字节序二进制值

    #include <arpa/inet.h>

    int inet_aton(const char *strptr,struct in_addr *addrptr);

- inet_addr()
  - 功能同上，返回转换后的地址。

    in_addr_t inet_addr(const char *strptr);

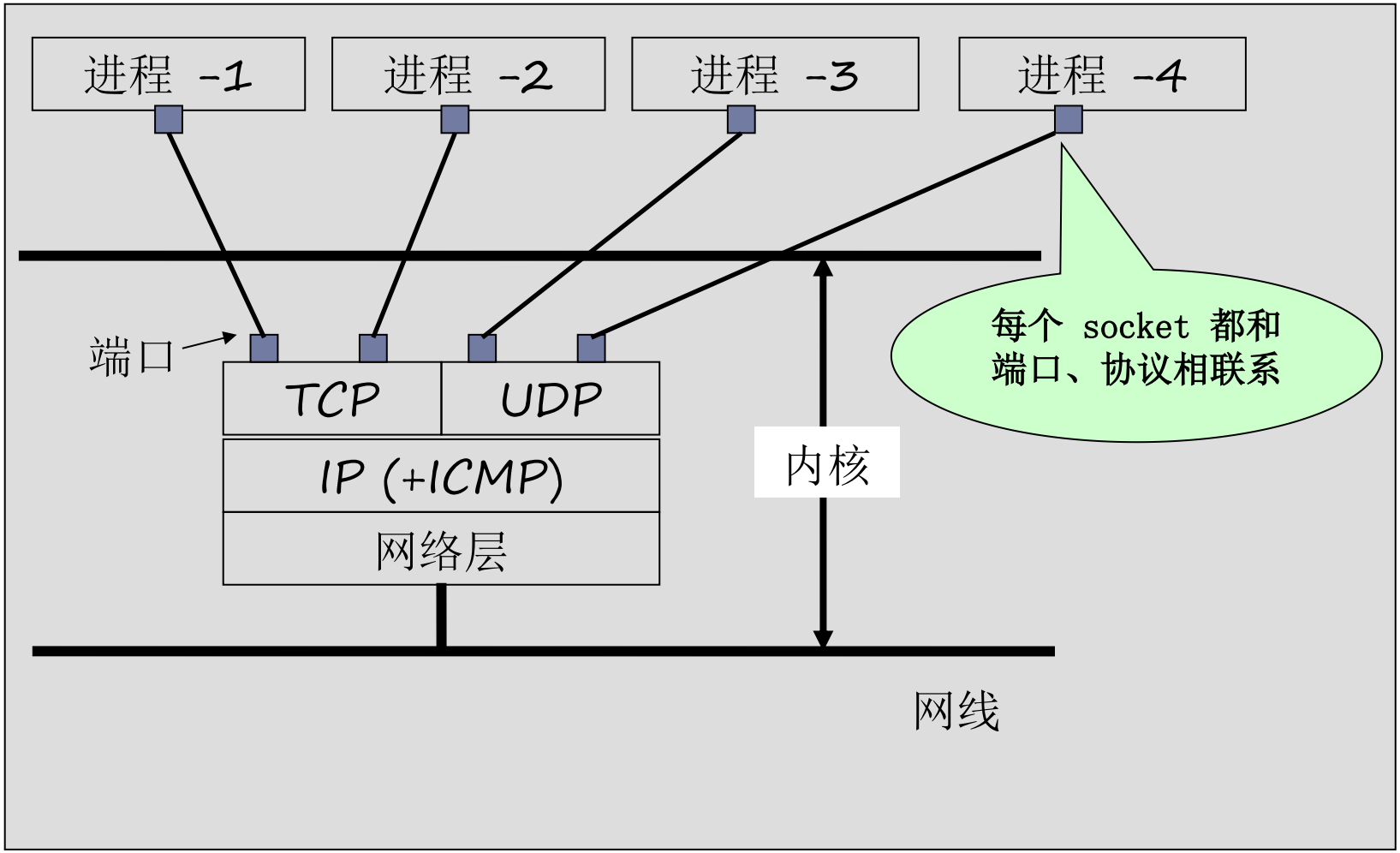
- inet_ntoa()
  - 将32位网络字节序二进制地址转换成点分十进制的字符串。

    char *inet_ntoa(stuct in_addr inaddr);

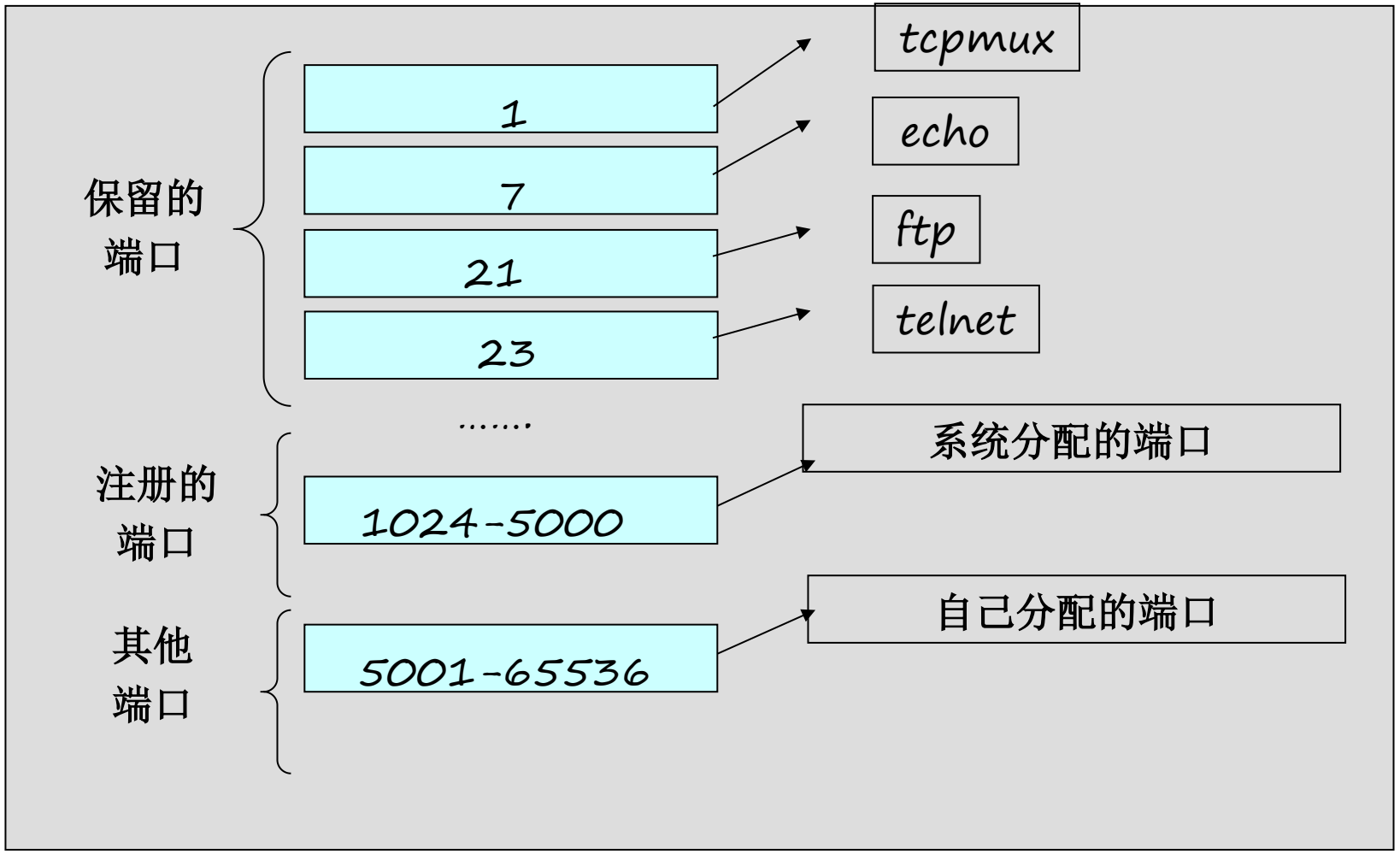
端口号

- 为了区分一台主机接收到的数据包应该转交给哪个进程来进行处理，使用端口号来区别
- TCP端口号与UDP端口号独立
- 端口号一般由IANA (Internet Assigned Numbers Authority) 管理
  - 众所周知端口：1~1023（1~255之间为众所周知端口，256~1023端口通常由UNIX系统占用）
  - 已登记端口：1024~49151
  - 动态或私有端口：49152~65535

套接字和端口


进程 -1
进程 -2
进程 -3
进程 -4
端口
TCP
UDP
IP (+ICMP)
网络层
内核
每个 socket 都和
端口、协议相联系
网线

套接字和端口


保留的
端口
1
7
21
23
tcpmux
echo
ftp
telnet
.....
注册的
端口
1024-5000
系统分配的端口
其他
端口
5001-65536
自己分配的端口

字节序

- 不同类型CPU的主机中，内存存储多字节整数序列有两种方法，称为主机字节序(HBO)：
  - 小端序（little-endian） - 低序字节存储在低地址
    - 将低字节存储在起始地址，称为“Little-Endian”字节序，Intel、AMD等采用的是这种方式；
  - 大端序（big-endian）- 高序字节存储在低地址
    - 将高字节存储在起始地址，称为“Big-Endian”字节序，由ARM、Motorola等所采用
- 网络中传输的数据必须按网络字节序，即大端字节序
- 在大部分PC机上，当应用进程将整数送入socket前，需要转化成网络字节序；当应用进程从socket取出整数后，要转化成小端字节序（原因？）

字节序

- 网络字节序(NBO - Network Byte Order)
  - 使用统一的字节顺序，避免兼容性问题
- 主机字节序(HBO - Host Byte Order)
  - 不同的机器HBO是不一样的，这与CPU的设计有关
  - Motorola 68K系列HBO与NBO是一致的
  - Intel X86系列和ARM系列，HBO与NBO不一致

字节序转换函数

- 把给定系统所采用的字节序称为主机字节序。为了避免不同类别主机之间在数据交换时由于对于字节序的不同而导致的差错，引入了网络字节序。

- 主机字节序到网络字节序
  - `u_long htonl (u_long hostlong);`
  - `u_short htons (u_short short);`

- 网络字节序到主机字节序
  - `u_long ntohl (u_long hostlong);`
  - `u_short ntohs (u_short short);`

网络编程相关API

- 网络编程常用函数
  - socket()　创建套接字
  - bind()　绑定本机地址和端口
  - connect()　建立连接
  - listen()　设置监听套接字
  - accept()　接受TCP连接
  - recv(), read(), recvfrom()　数据接收
  - send(), write(), sendto()　数据发送
  - close(), shutdown()　关闭套接字

socket

- int socket (int domain, int type, int protocol);
  - domain 是地址族

    PF_INET // internet 协议

    PF_UNIX // unix internal协议

    PF_NS   // Xerox NS协议

    PF_IMPLINK // Interface Message协议

  - type // 套接字类型

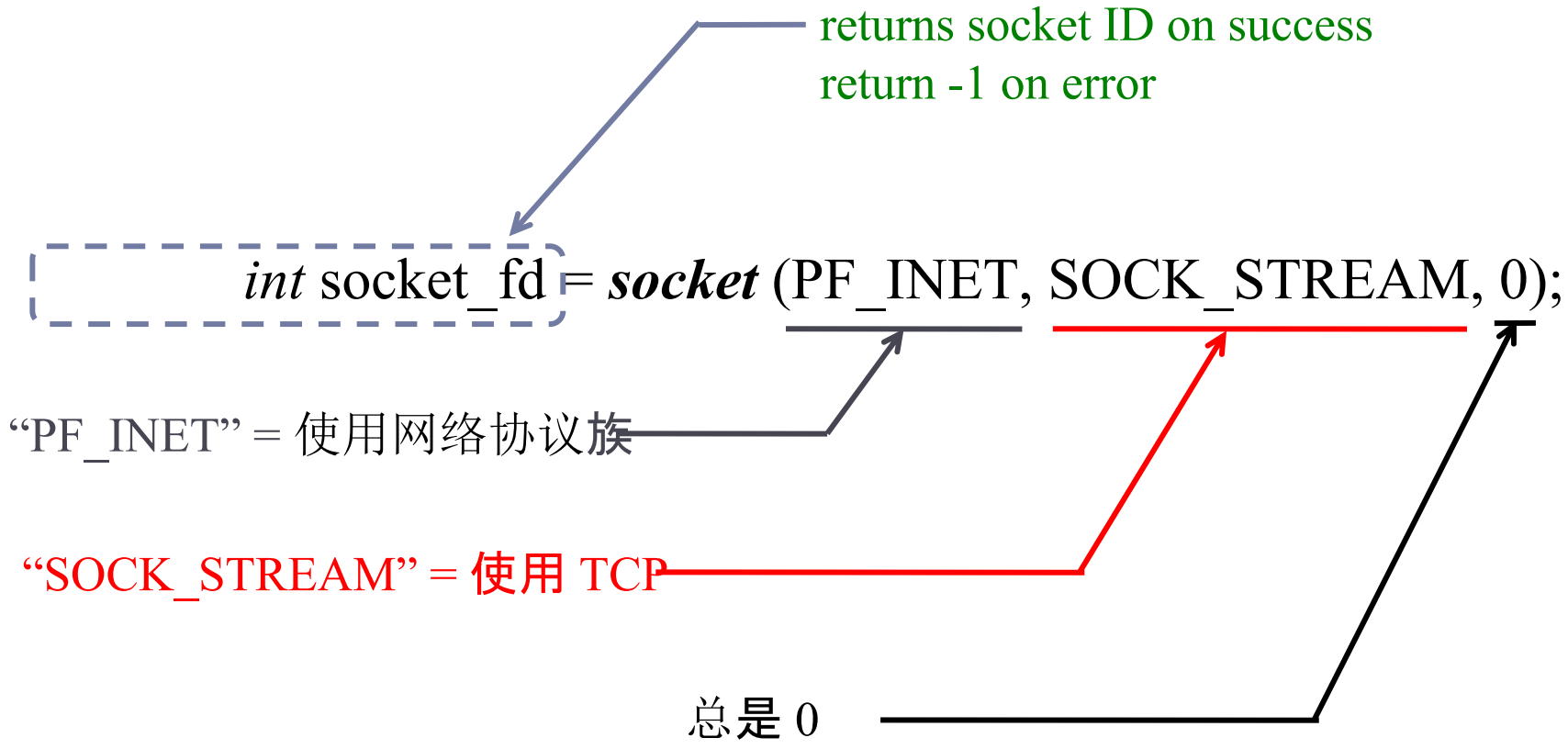
    SOCK_STREAM   // 流式套接字

    SOCK_DGRAM   // 数据报套接字

    SOCK_RAW   // 原始套接字

  - protocol 参数通常置为0

socket


returns socket ID on success
return -1 on error
int socket_fd = socket (PF_INET, SOCK_STREAM, 0);
“PF_INET” = 使用网络协议族
“SOCK_STREAM” = 使用 TCP
总是 0

地址相关的数据结构

- *通用地址结构*

```
struct sockaddr
{
    u_short  sa_family;   // 地址族, AF_xxx
    char  sa_data[14];    // 14字节协议地址
};
```

- *Internet协议地址结构*

```
struct sockaddr_in
{
    u_short sin_family;      // 地址族, AF_INET，2 bytes
    u_short sin_port;        // 端口，2 bytes
    struct in_addr sin_addr; // IPV4地址，4 bytes
    char sin_zero[8];        // 8 bytes unused，作为填充
};
```

地址相关的数据结构

- *IPv4地址结构*

```
// internet address
struct in_addr
{
    in_addr_t  s_addr;          // u32 network address
};
```

bind ()

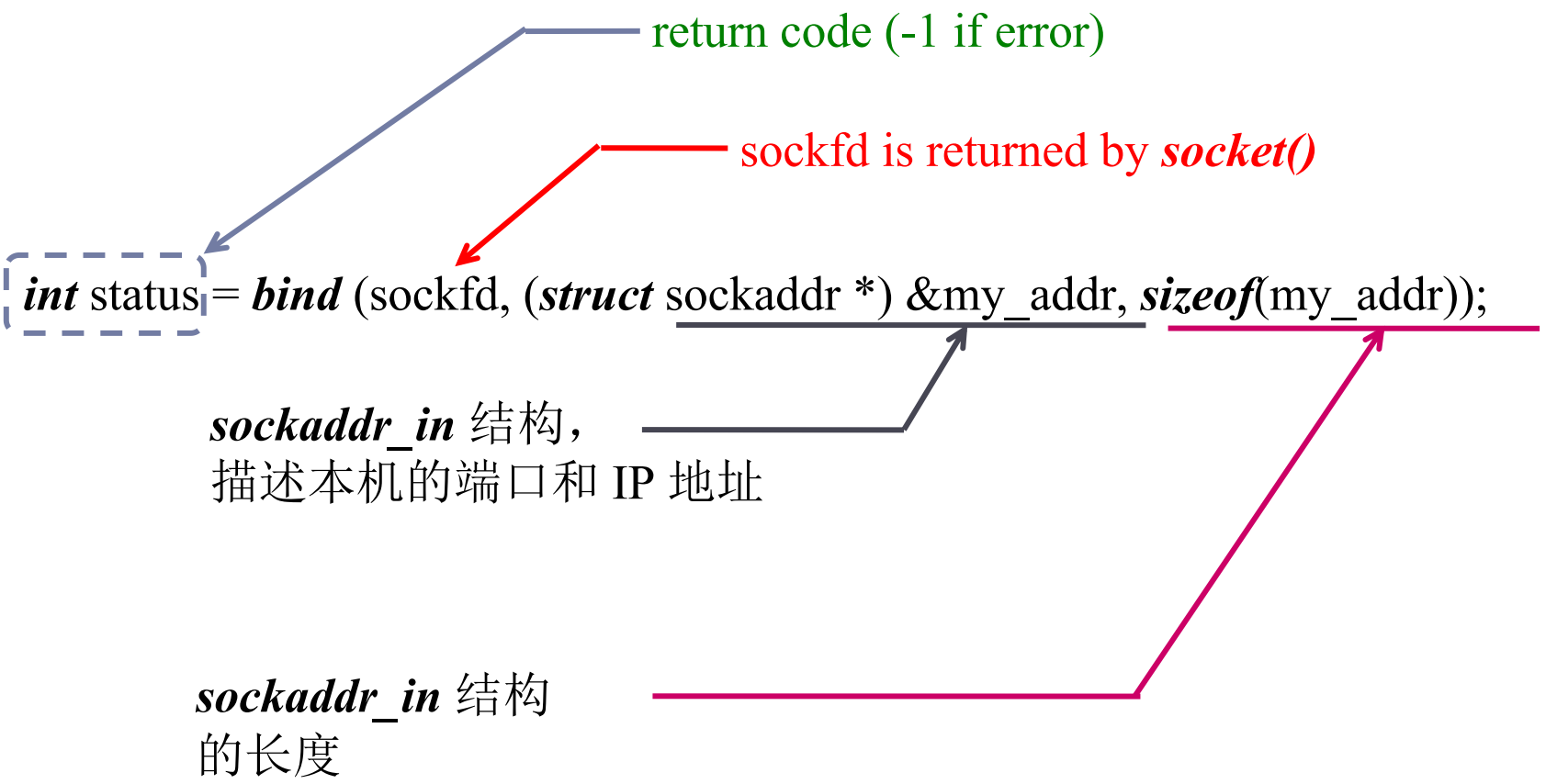
- int bind(int sockfd, struct sockaddr *my_addr, int addrlen) ;
  - 头文件:

    #include <sys/types.h>

    #include <sys/socket.h>

  - sockfd: socket调用返回的文件描述符
  - addrlen: sockaddr地址结构的长度
  - 返回值：0或-1

bind ()


return code (-1 if error)
sockfd is returned by socket()
int status = bind (sockfd, (struct sockaddr *) &my_addr, sizeof(my_addr));
sockaddr_in 结构，
描述本机的端口和 IP 地址
sockaddr_in 结构
的长度

bind

int bind (int sockfd, struct sockaddr* addr, int addrLen);

- **sockfd** 由socket() 调用返回

- **addr** 是指向 sockaddr_in 结构的指针，包含本机IP地址和端口号

```
struct sockaddr_in
    u_short sin_family // protocol family
    u_short sin_port     // port number
    struct in_addr  sin_addr  //IP address (32-bits)
```

- **addrLen** : sizeof (struct sockaddr_in)

how to fill addr info

Step 1: 初始化该数据结构

```
struct sockaddr_in   my_addr;        /* My (client) Internet address      */
```

```
/* Set My(client's) IP Address ------------------------------------ */
my_addr.sin_family = PF_INET;         /* Protocol Family To Be Used */
my_addr.sin_port  = htons (6666);     /* Port number to use */
my_addr.sin_addr.s_addr = inet_addr("192.168.1.100");      /* My IP address */
```

Step 2: 填充信息

地址结构的一般用法

1. 定义一个struct sockaddr_in类型的变量并清空

```
struct sockaddr_in myaddr;
memset(&myaddr, 0, sizeof(myaddr));
```

2. 填充地址信息

```
myaddr.sin_family = PF_INET;
myaddr.sin_port = htons(8888);
myaddr.sin_addr.s_addr = inet_addr("192.168.1.100");
```

3. 将该变量强制转换为struct sockaddr类型在函数中使用

```
bind(listenfd, (struct sockaddr*)(&myaddr), sizeof(myaddr));
```

地址转换函数

- **unsigned long inet_addr(char *address);**

  address是以NULL结尾的点分IPv4字符串。该函数返回32位的地址。如果字符串包含的不是合法的IP地址，则函数返回-1。例如：

  ```
  struct in_addr addr;
  addr.s_addr = inet_addr(" 192.168.1.100 ");
  ```

- **char* inet_ntoa(struct in_addr address);**

  address是IPv4地址结构，函数返回一指向包含点分IP地址的静态存储区字符指针。如果错误则函数返回NULL

listen

int listen (int sockfd, int backlog);

- sockfd:监听连接的套接字
- backlog
  - 指定了正在等待连接的最大队列长度，它的作用在于处理可能同时出现的几个连接请求。
  - DoS(拒绝服务)攻击即利用了这个原理，非法的连接占用了全部的连接数，造成正常的连接请求被拒绝。
- 返回值： 0 或 -1

完成listen()调用后，socket变成了监听socket(listening socket).

accept()

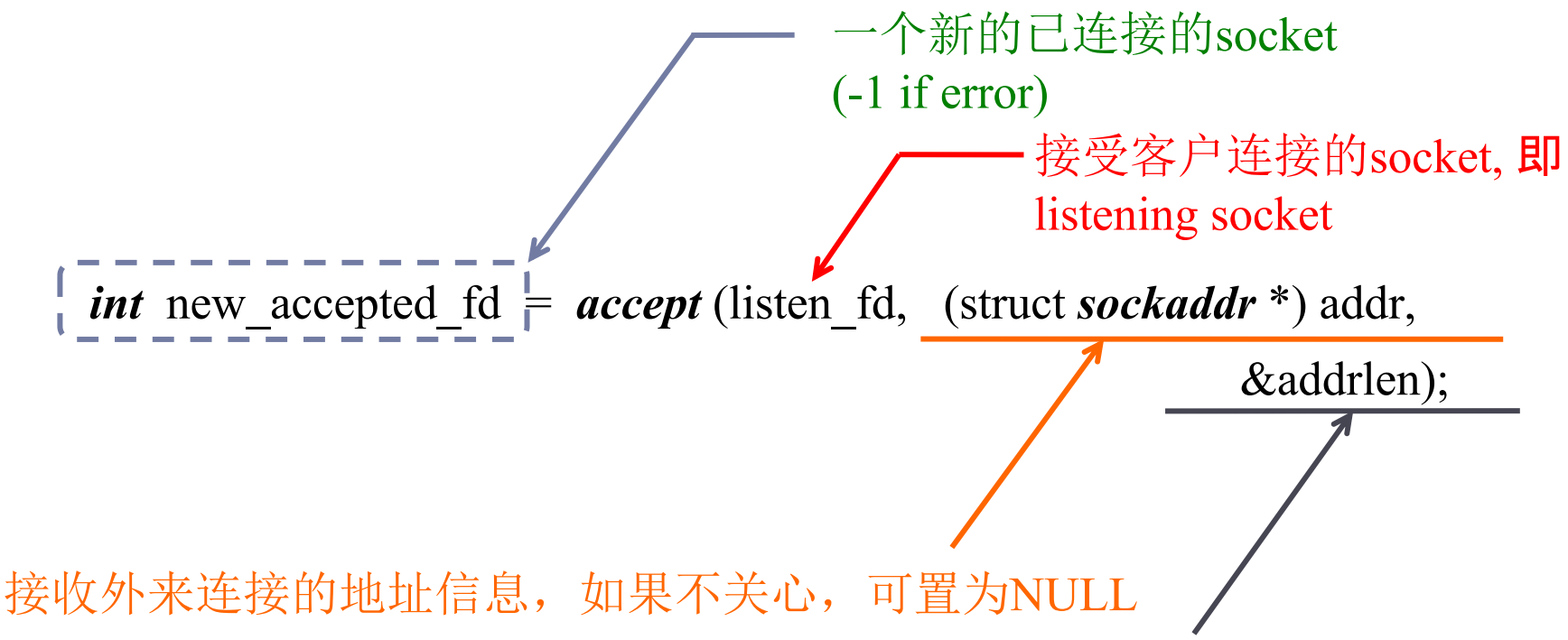
- int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen) ;
  - 返回值：已建立好连接的套接字或-1
  - 头文件

    #include <sys/types.h>

    #include <sys/socket.h>
  - sockfd : 监听套接字
  - addr : 对方地址
  - addrlen：地址长度

listen()和accept()是TCP服务器端使用的函数

accept ( ) 函数


一个新的已连接的socket
(-1 if error)
接受客户连接的socket, 即
listening socket
int new_accepted_fd = accept (listen_fd, (struct sockaddr *) addr,
&addrlen);
接收外来连接的地址信息，如果不关心，可置为NULL
传递结构体addr的长度并返回对方地址的长度

connect()

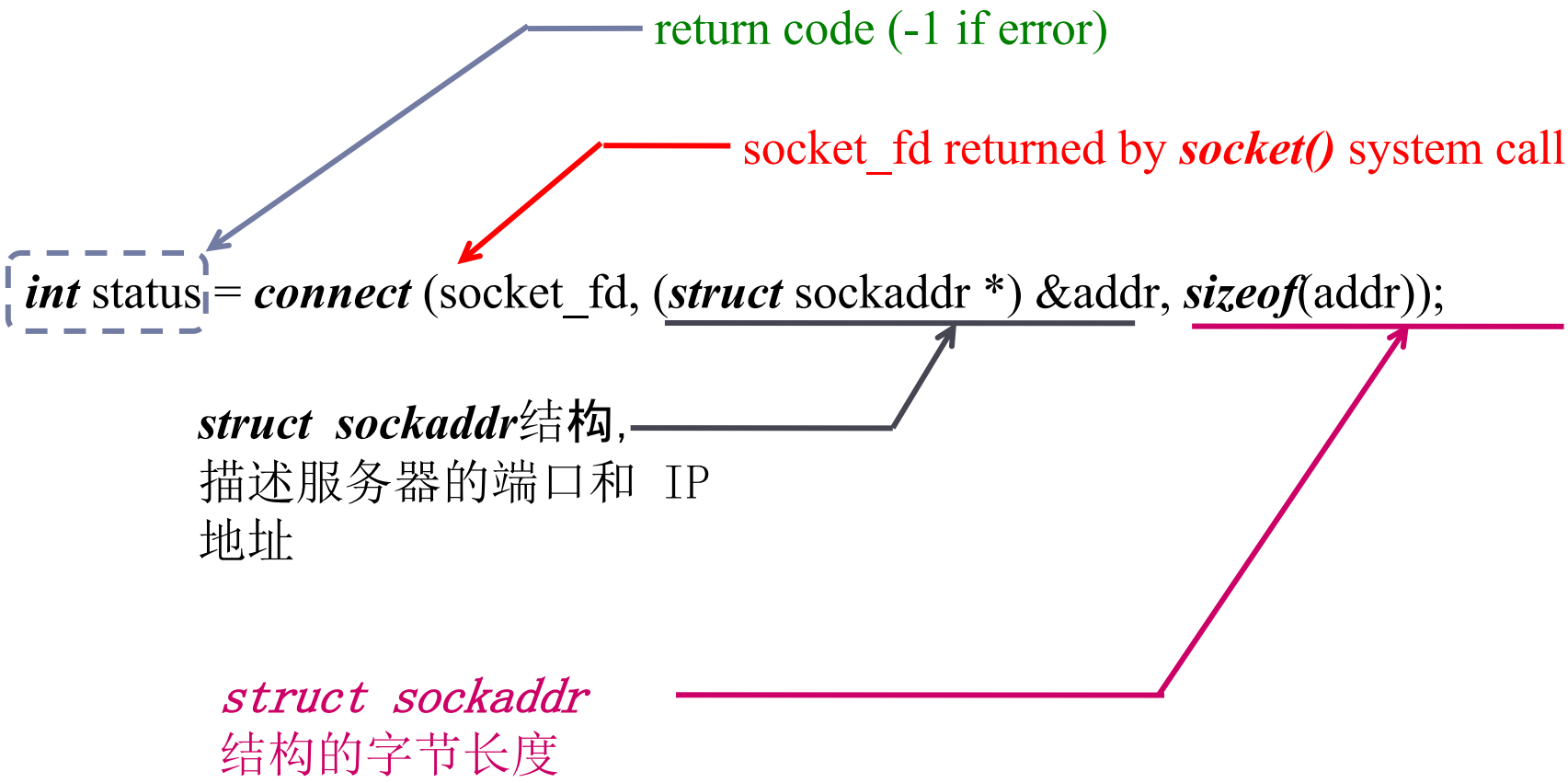
- int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
  - 返回值：0 或 -1
  - 头文件：

    #include <sys/types.h>

    #include <sys/socket.h>
  - sockfd : socket返回的文件描述符
  - serv_addr : 服务器端的地址信息
  - addrlen : serv_addr的长度

connect()是客户端使用的系统调用。

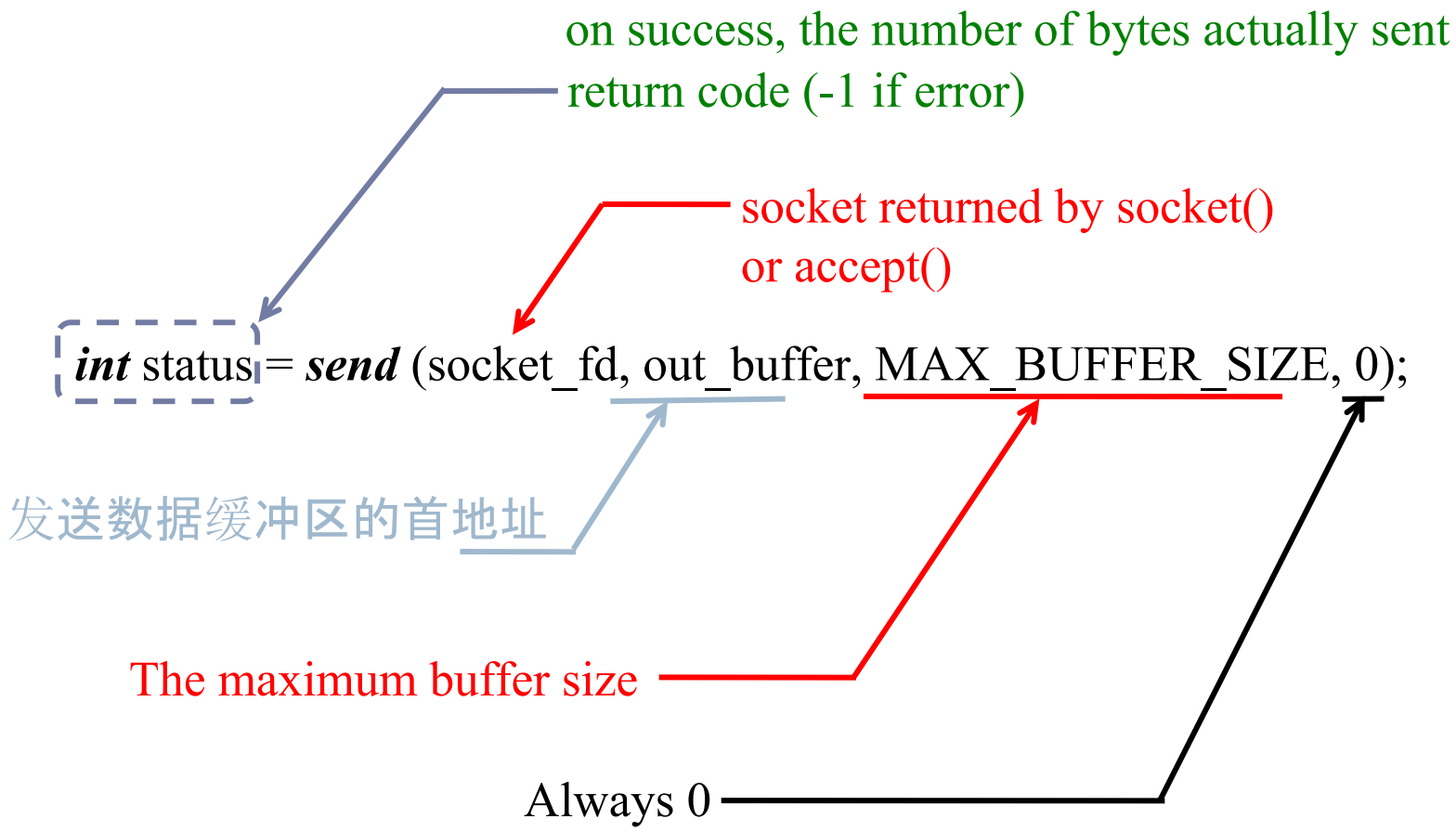
connect( ) 函数


return code (-1 if error)
socket_fd returned by *socket()* system call
int status = *connect* (socket_fd, (*struct* sockaddr *) &addr, *sizeof*(addr));
*struct sockaddr*结构,
描述服务器的端口和 IP
地址
struct sockaddr
结构的字节长度

send()

- ssize_t send(int socket, const void *buffer, size_t length, int flags);
  - 返回值：
    - 成功：实际发送的字节数
    - 失败：-1, 并设置errno
  - 头文件：
    - #include <sys/socket.h>
  - buffer : 发送缓冲区首地址
  - length : 发送的字节数
  - flags : 发送方式（通常为0）

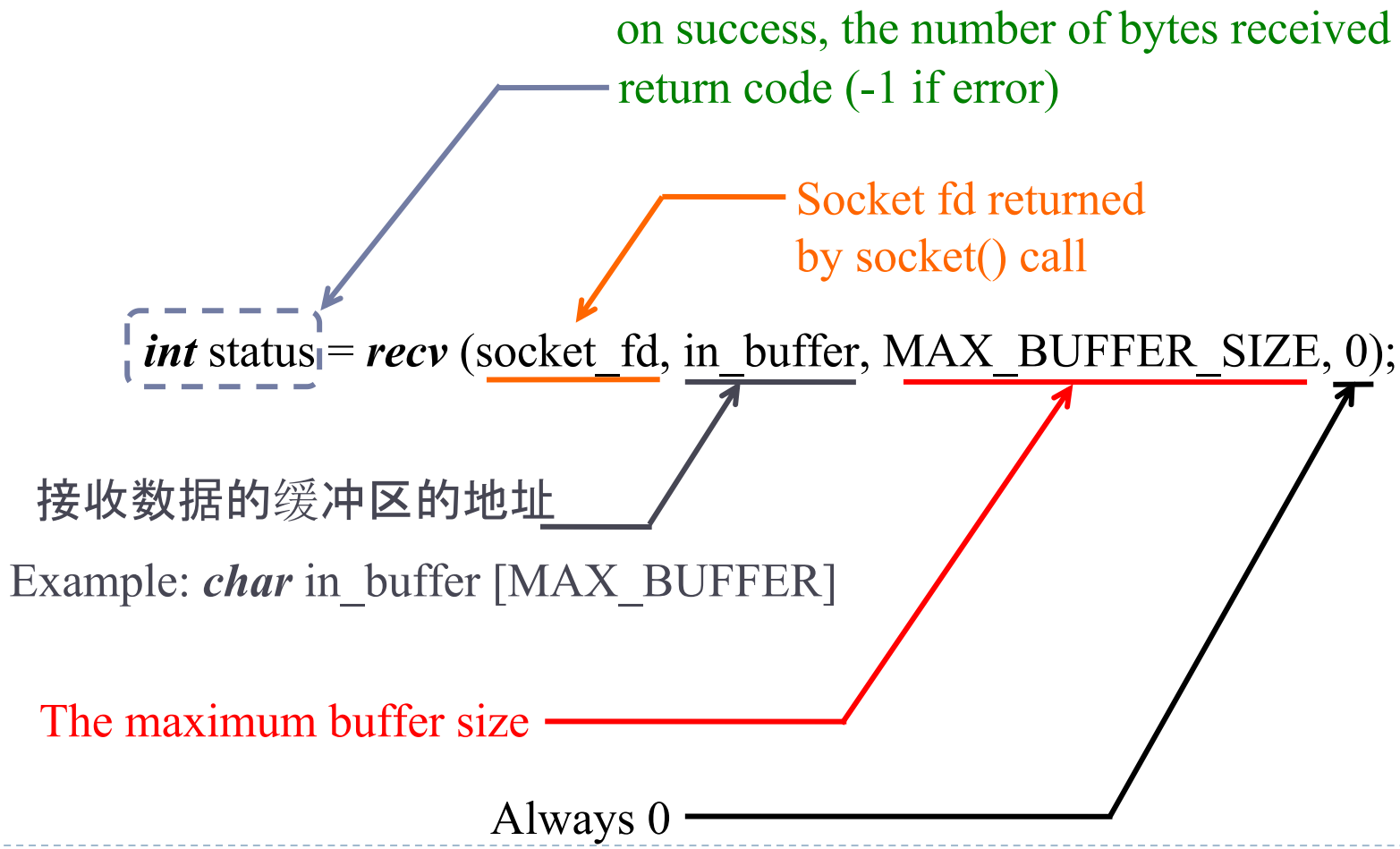
send ( ) 函数


on success, the number of bytes actually sent
return code (-1 if error)
socket returned by socket()
or accept()
int status = send (socket_fd, out_buffer, MAX_BUFFER_SIZE, 0);
发送数据缓冲区的首地址
The maximum buffer size
Always 0

recv()

- ssize_t recv(int socket, const void *buffer, size_t length, int flags);
  - 返回值：
    - 成功：实际接收的字节数
    - 失败：-1, 并设置errno
  - 头文件：
    - #include <sys/socket.h>
  - buffer : 发送缓冲区首地址
  - length : 发送的字节数
  - flags : 接收方式（通常为0）

recv ( ) 函数


on success, the number of bytes received
return code (-1 if error)
Socket fd returned
by socket() call
int status = recv (socket_fd, in_buffer, MAX_BUFFER_SIZE, 0);
接收数据的缓冲区的地址
Example: char in_buffer [MAX_BUFFER]
The maximum buffer size
Always 0

read()/write()

- ssize_t read(int fd, void *buf, size_t count);
- ssize_t write(int fd, const void *buf, size_t count);

  - read()和write()经常会代替recv()和send()，通常情况下，看程序员的偏好
  - 使用read()/write()和recv()/send()时最好统一

套接字的关闭

- int close(int sockfd);
  - 关闭双向通讯
- int shutdown(int sockfd, int howto);
  - TCP连接是双向的(是可读写的)，当我们使用close时,会把读写通道都关闭，有时侯我们希望只关闭一个方向，这个时候我们可以使用shutdown。
  - 针对不同的howto，系统回采取不同的关闭方式。

shutdown()的howto参数

- howto = 0

关闭读通道，但是可以继续往套接字写数据。

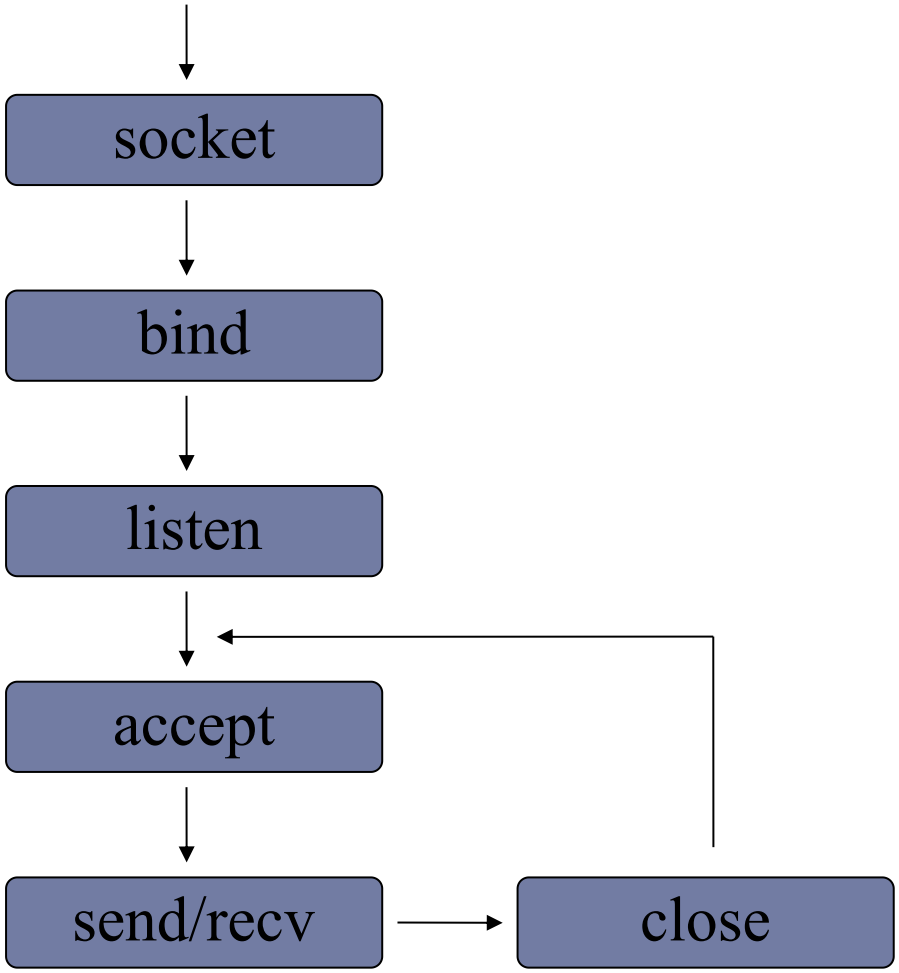
- howto = 1

和上面相反，关闭写通道。只能从套接字读取数据。

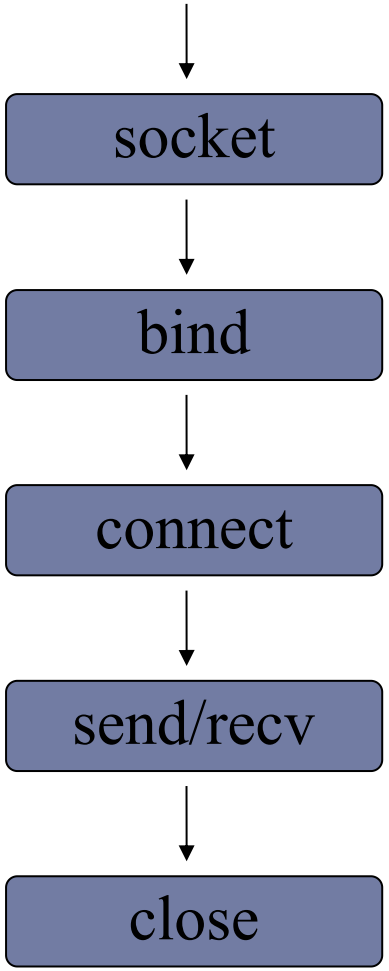
- howto = 2

关闭读写通道，和close()一样

TCP服务器端流程



TCP客户端流程



<可选>

循环服务器模型

华清远见 FARSIGHT 嵌入式培训专家

- **TCP服务器**
  - TCP服务器端运行后等待客户端的连接请求。
  - TCP服务器接受一个客户端的连接后开始处理，完成了客户的所有请求后断开连接。
  - TCP循环服务器一次只能处理一个客户端的请求。
  - 只有在当前客户的所有请求都完成后，服务器才能处理下一个客户的连接/服务请求。
  - 如果某个客户端一直占用服务器资源，那么其它的客户端都不能被处理。TCP服务器一般很少采用循环服务器模型。

TCP循环服务器

- 流程如下:

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

并发服务器模型

- **TCP服务器**
  - 为了弥补TCP循环服务器的缺陷，人们又设计了并发服务器的模型。

    并发服务器的设计思想是服务器接受客户端的连接请求后创建子进程来为客户端服务
  - TCP并发服务器可以避免TCP循环服务器中客户端独占服务器的情况。
  - 为了响应客户机的请求,服务器要创建子进程来处理。 如果有多个客户端的话，服务器端需要创建多个子进程。过多的子进程会影响服务器端的运行效率

服务器模型

- **I/O多路复用并发服务器**

```
初始化(socket -> bind -> listen);
 while(1) {
    设置监听读写文件描述符集合;
    调用poll()/select();
    如果是监听套接字就绪,说明有新的连接请求 {
        建立连接(accept);
        加入到监听文件描述符集合;
    }
    否则说明是一个已经连接过的描述符 {
        进行操作(send或者recv);
    }
 }
```