自我介绍

雷 天 德

爱好：

编程、茶、咖啡、听听歌、运动（骑行、打球）

阶段分析：

c语言：

linux及c高级

数据结构: 数据关系：双向链表

io/进程

网络编程课程大纲：

day1_网络编程基础内容

网络的发展史：

协议

应用程序设计模型

专业术语：

路由、路由器、下一跳

基本知识：

套接字

ip

端口

字节序

服务器/客户端搭建流程

day2_TCP/UDP

day3_广播、组播

day4_并发（多进程、多线程服务器、io多路复用等）

day5_sqlite3(数据库等)

day6_复习、做项目

```c
#include <stdio>

int main()
{
    printf("hello world");  //1024   \n  fflush();
    while(1);

    return 0;
}
```

今日课程内容

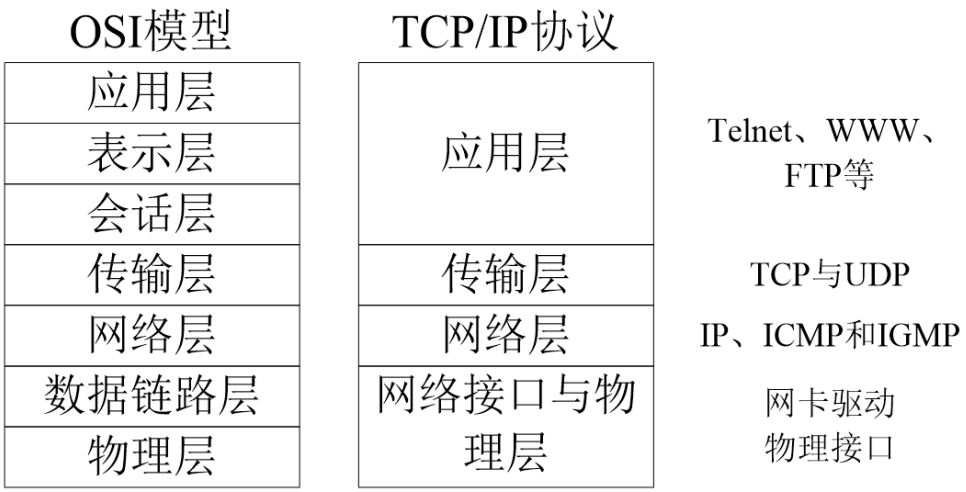
1、网络发展背景

1. 生活中用到网络的有哪些？
2. 用到网络的设备有哪些？
    1. 手机
    2. 电脑
    3. 智能手表
    4. 平板
    5. 电视
    6. 路由器
    7. 无人机
    8. 智能车（车载系统）
    9. 智能家居
    10. 智能管家
    11. 各种智能门禁
    12. 各种移动支付
3. 为什么用qq发的消息对方依然用qq接收？ ----端口号
4. 为什么发的消息 可以指定发给对象，而不是室友？ ------ip

2、网络分层模型： OSI 模型及 TCP/IP工业标准模型

1. OSI 7 层模型（理想化模型）：物、数、网、传、会、表、应
2. TCP/IP 4 层模型（工业化标准，实际应用的模型）：网、网、传、应

TCP/IP与OSI参考模型的对应关系



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，到达目的地址后再进行重组。常常把这一层数据叫做段。

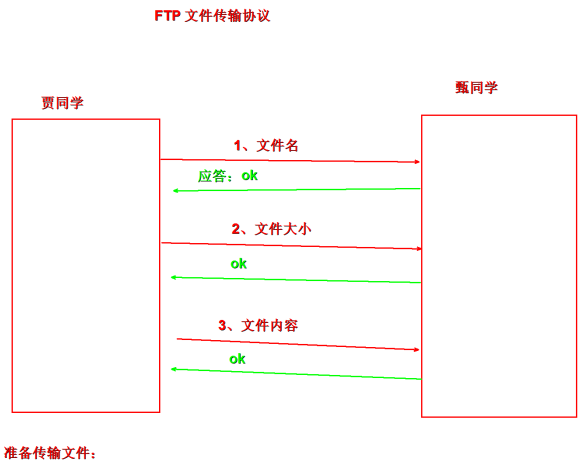
5. **会话层**：通过传输层(端口号：传输端口与接收端口)建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求（设备之间需要互相认识可以是IP也可以是MAC或者是主机名）。

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。

3、协议

1. 什么是协议？
    1. 生活中其实就是双方的约定、规则。
    2. 那么在网络传输里面：协议表示数据传输和解释的规则
2. FTP协议。



eg：FTP协议

{

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。

}

> 特点：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

1）传输层 常见协议有TCP/UDP协议。

2）应用层 常见的协议有HTTP协议，FTP协议。

3）网络层 常见协议有IP协议、ICMP协议、IGMP协议。

4）网络接口层 常见协议有ARP协议、RARP协议。

//重点以下5-9 个协议

5）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。

6）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。

7）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。

8）FTP文件传输协议（File Transfer Protocol）

9）IP协议是因特网互联协议（Internet Protocol）

10）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。

11）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。

12）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。

13）RARP是反向地址转换协议，通过MAC地址确定IP地址。

4、应用程序设计模型

两种模型：b/s（Browser/Server） c/s（client/server）

C/S ：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

应用：LOL等大型3D游戏（缓存数据图像，环境包等）

优点： 1. 协议选用灵活。 （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。）

2. 将数据缓存至客户端，提高数据传输效率。

缺点： 1. 对用户的安全构成威胁 （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。）

2. 开发工作量较大，调试困难（服务器和客户端都需要团队开发）

B/S ：浏览器(Browser)/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。

应用：小型的2D游戏，偷菜等游戏。

优点： 1. 安全性更高（没有客户端在用户主机上，用的是第三方浏览器）

2. 开发工作量小（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）

3. 跨平台(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，linux都行)

缺点： 1. 网络应用支持受限 （使用第三方浏览器）

2.缓存数据不理想（没有客户端放到对方主机上）

3.采用标准http协议进行通信，协议选择不灵活 （由于使用浏览器，所以必须要用http协议）

5、网络编程基础知识预知：（必须掌握重点）

1. 套接字

是一个编程接口
是一种特殊的文件描述符 (everything in Unix is a file)
并不仅限于TCP/IP协议

套接字三种类型：

- 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)

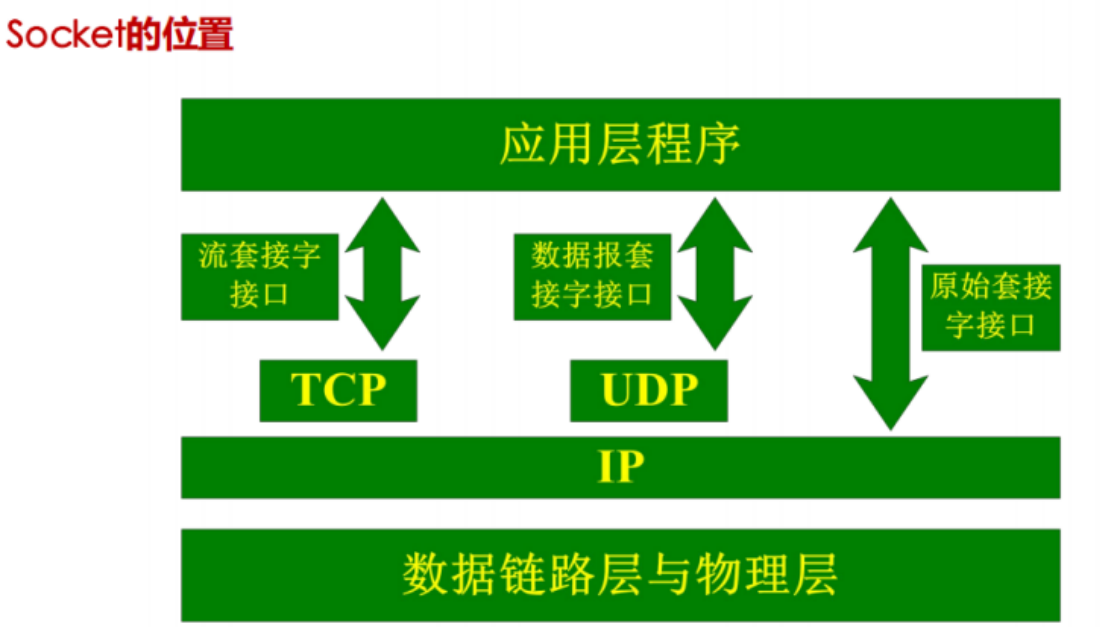
可以对较低层次协议如IP、ICMP直接访问

面向连接 (Transmission Control Protocol - TCP/IP)

无连接 (User Datagram Protocol -UDP 和 Inter-network Packet Exchange - IPX)

（文件描述符：1、非负整数、2、用于区分内核操作正在打开的一个文件 3、io阶段 0-1023）

2. socket的位置



3.

1. ip地址

1. 什么是ip、用来干嘛？

    1. 在因特网里面区分主机的唯一标识

2. ip有多少类型

    1. 公有地址

        公有地址（Public address）由Inter NIC（Internet Network Information Center因特网信息中心）负责。这些IP地址分配给注册并向Inter NIC提出申请的组织机构。通过它直接访问因特网。

    2. 私有地址

        私有地址（Private address）属于非注册地址，专门为组织机构内部使用。

        以下列出留用的内部私有地址

        A类 10.0.0.0--10.255.255.255 网络ip 主机ip 10.1.1.1 10.255.255.255

        B类 172.16.0.0--172.31.255.255

        C类 192.168.0.0--192.168.255.255

- D类：224
- E类：

假设：设置自定义可用的ip　　192.168.10. N

- 主机ip：　192.168.10.100

  掩码：255.255.255.0

- 网关：192.168.10.1

2. 端口

1. 为了区分一台主机接收到的数据包应该转交给哪个任务来进行处理，使用端口号来区别

2. 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1024–49151之间的端口号。大于49151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

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

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

    int i = 0x11223344;
    char *ptr = (char *)&i;
    printf("%#x, %p\n",*ptr, ptr);
    printf("%#x, %p\n",*(ptr+1), ptr+1);
    printf("%#x, %p\n",*(ptr+2), ptr+2);
    printf("%#x, %p\n",*(ptr+3), ptr+3);
    return 0;
}



//小端：低字节存低地址
//大端：低字节存高地址
//网络字节序 采用大端存储
```

```
farsight@ubuntu:~/01-myLesson/NET$ s
s: 未找到命令
farsight@ubuntu:~/01-myLesson/NET$ ls
1_demo.c  2_inet_addr.c  3_inet_aton.c  a.out
farsight@ubuntu:~/01-myLesson/NET$ gcc 1_demo.c
farsight@ubuntu:~/01-myLesson/NET$ ./a.out
0x44, 0xbfb34348
0x33, 0xbfb34349
0x22, 0xbfb3434a
0x11, 0xbfb3434b
farsight@ubuntu:~/01-myLesson/NET$
```

3. 字节序

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。

字节序就是为了 解决由于 cpu差异化导致的整型数据存放的兼容问题？

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。

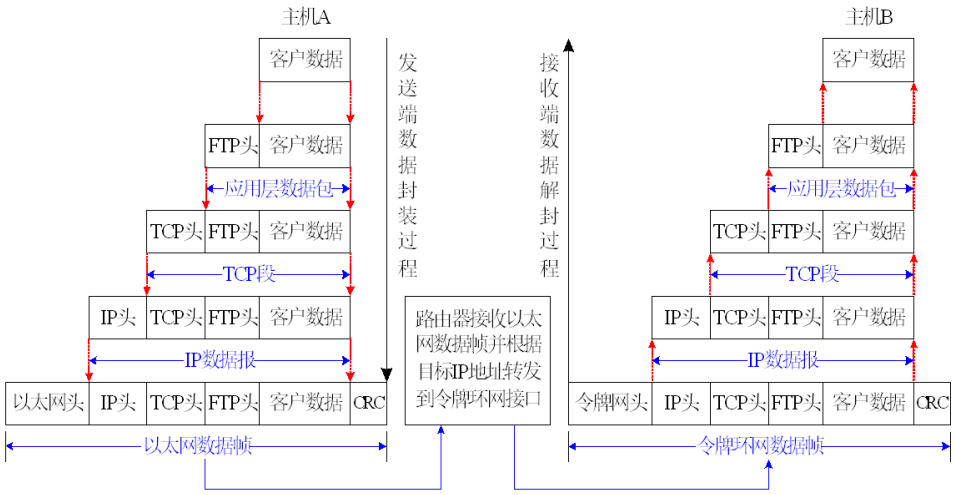
如果主机是小端字节序，这些函数将参数做相应的大小端转换然后返回，如果主机是大端字节序，这些函数不做转换，将参数原封不动地返回。

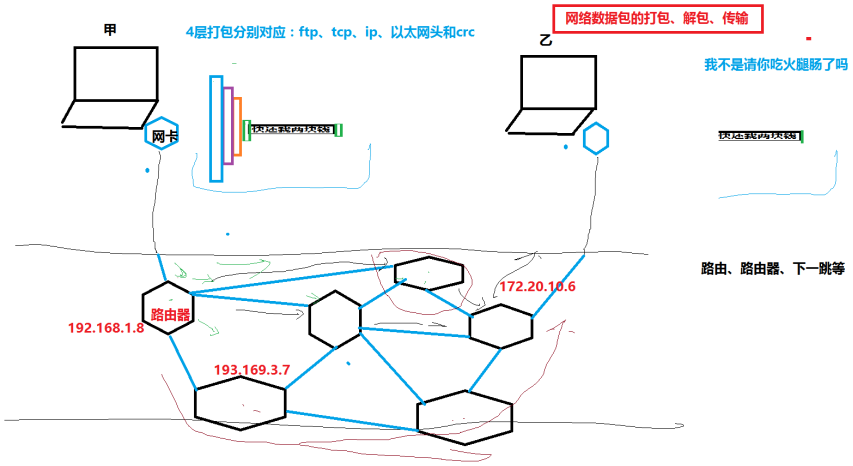
6、数据包的打包和解包

TCP/IP协议

OSI中的层	功能	TCP/IP协议族
应用层	文件传输，电子邮件，文件服务，虚拟终端	TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet
表示层	数据格式化，代码转换，数据加密	没有协议
会话层	解除或建立与别的节点的联系	没有协议
传输层	提供端对端的接口	TCP，UDP
网络层	为数据包选择路由	IP，ICMP，RIP，OSPF，BGP，IGMP
数据链路层	传输有地址的帧以及错误检测功能	SLIP，CSLIP，PPP，ARP，RARP，MTU
物理层	以二进制数据形式在物理媒体上传输数据	ISO2110，IEEE802.1,EEE802.2

数据的封装与传递过程

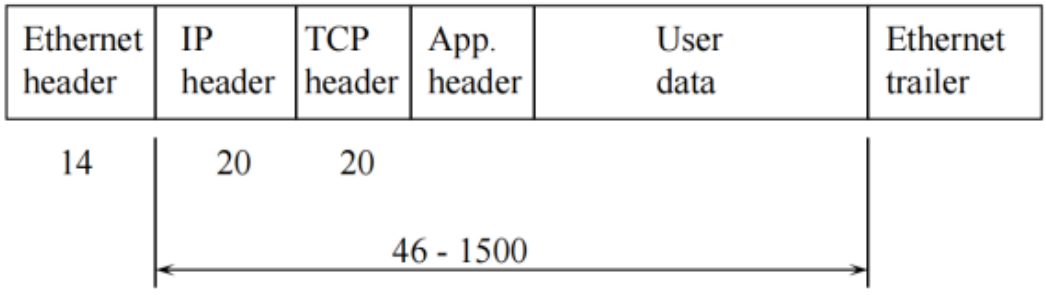


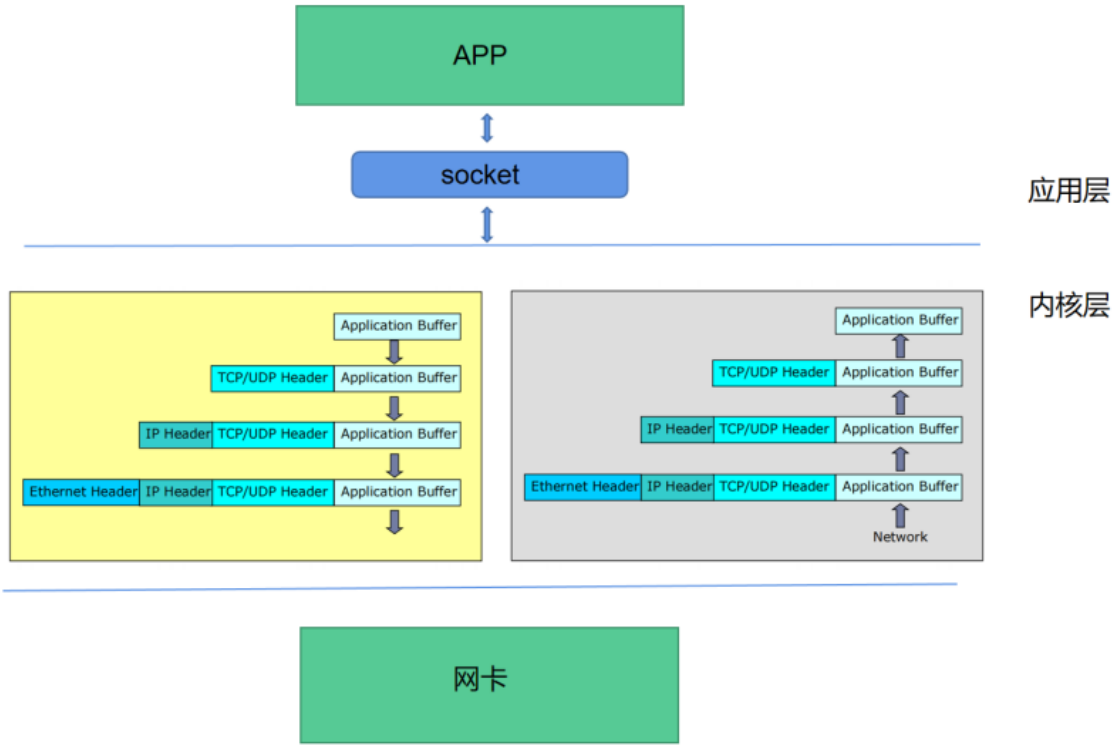


数据发送：由上层到下层，封装数据到网卡

数据接收：由下层到上层，解析数据到用户

TCP/IP协议下的数据包





== 数据封包和拆包由操作系统完成 ==

7、c/s模型服务器/客户端的搭建流程（框架）--重点

1. **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

2. **c/s 服务器搭建流程：**

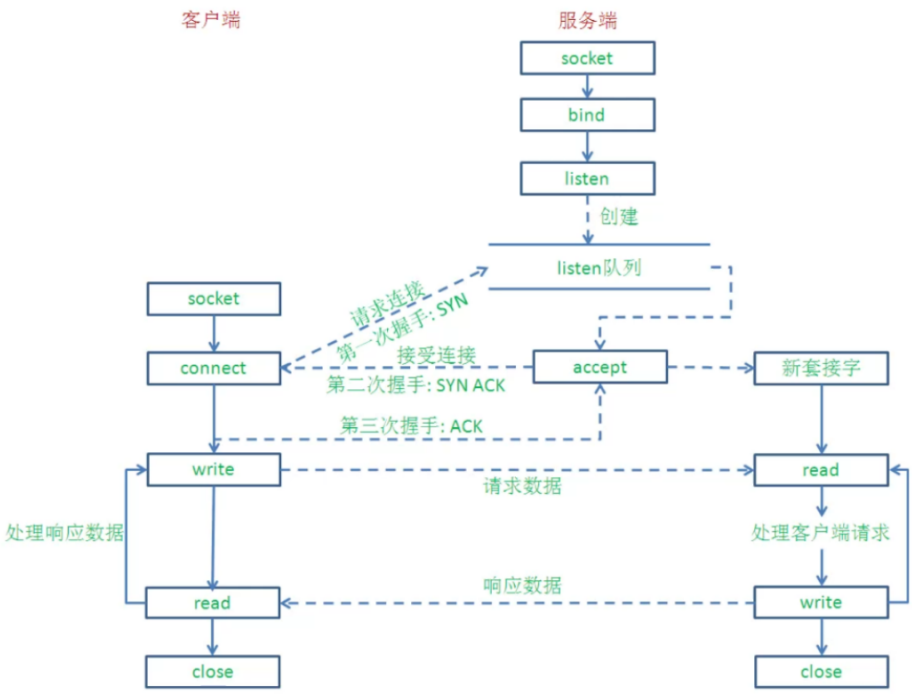
   1. 创建套接字　：socket
   2. 绑定ip和端口　：bind
   3. 建立监听　：listen
   4. 接收/处理客户端连接请求　：accept
   5. 正常通信
      1. read/recvfrom/recv
      2. write/sendto/send
   6. 通信完毕，关闭套接字

3. **c/s客户端搭建流程：**

   1. 创建套接字：socket
   2. 绑定：bind<可选>，默认隐式绑定

3. 主动连接服务器：connect
4. 正常通信
    1. read/recvfrom/recv
    2. write/sendto/send
5. 通信完毕，关闭套接字

4. **服务器客户端通信框架**

5. 

常用函数接口说明：

```
网络编程常用函数
{
    1>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。
            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。

}

2>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。

}

3>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次握手队列的链接数和

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。

}

4>accept函数

{

```
#include <sys/types.h>      /* See NOTES */
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

sockdf:

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。

}

5>connect函数

{

```
#include <sys/types.h>                  /* See NOTES */
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

sockdf:

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。

    }

}
```

5、c/s模型TcpServer.c

```
/*********************  版本 一  ********************/


#include <stdio.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main()
{
    int listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if(-1 == listenfd)
    {
        perror("socket");
        return -1;
    }
/*
    struct sockaddr_in serveraddr = {
        .sin_family = AF_INET,
        .sin_addr.s_addr = inet_addr("127.0.0.1");
        .sin_port = htons(8888);
    };
*/

    struct sockaddr_in serveraddr = {0};
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = INADDR_ANY;//inet_addr("0.0.0.0");////0
    serveraddr.sin_port = htons(8888);

    int len = sizeof(serveraddr);
    if(-1 == bind(listenfd, (struct sockaddr*)&serveraddr, len) )
    {
        perror("connect");
        return -1;
    }

    listen(listenfd, 10);//start to wait
    printf("wating ................ !\n");
```

```
    int clientfd = accept(listenfd, NULL, NULL);//if someone connected,then
create a new socket(clientfd) and return

    while(1)
    {
        char buf[100] = {0};
        int ret = read(clientfd, buf, 100);
        if(0 == ret)
            break;

        printf("recv[%d]: %s\n", ret, buf);
    }

    close(listenfd);
    close(clientfd);
}
```

```
/*********************  版本 二  ********************/
```

```
#include <stdio.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int server_init(char *ipaddr, unsigned short port,int backlog)//初始化服务器
{
/*****************************************
 AF_INET             IPv4 Internet protocols
 SOCK_STREAM         string socket
 0
*****************************************/
    int sockfd = socket(AF_INET,SOCK_STREAM,0);
    if(-1 == sockfd)
    {
        perror("socket");
        return -1;
    }
    printf("sockfd=%d\n",sockfd);

/*************************************************************
Internet协议地址结构“ /usr/include/netinet/in.h”
    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
```

```
    };
    struct in_addr{
        in_addr_t  s_addr;       // u32 network address
    };
*****************************************************************/
    struct sockaddr_in saddr;//定义Internet地址结构变量，保存服务器的ip及port
    memset(&saddr,0,sizeof(saddr));//bzero

    saddr.sin_family = AF_INET;//指定网络协议 IPV4
    saddr.sin_port = htons(port);//指定服务器的端口号 >=5001,由主机序转网络字节序

    //INADDR_ANY:任意ip地址
    saddr.sin_addr.s_addr = (NULL == ipaddr)?
(htonl(INADDR_ANY)):inet_addr(ipaddr);//指定服务器的ip地址，ip地址由点分式转32为无符号
网络字节序

    socklen_t slen = sizeof(saddr);

    int ret = bind(sockfd, (struct sockaddr *)&saddr,slen);//将服务器的ip和port与
sockfd绑定
    if(-1 == ret)
    {
        perror("bind");
        goto ERR_STEMP;
    }
    printf("bind success\n");

    ret = listen(sockfd,backlog);//sockfd变为监听套接字
    if(-1 == ret)
    {
        perror("listen");
        goto ERR_STEMP;
    }

    return sockfd;

ERR_STEMP:
        close(sockfd);
        return -1;

}

int main()
{
    int ret;

    int sockfd = server_init(NULL, 8000,1024);//初始化服务器
    if(-1 == sockfd)
    {
        printf("server_init error\n");
        return -1;
    }
    printf("listen....\n");


    struct sockaddr_in caddr;//保存客户端的ip及port
    memset(&caddr,0,sizeof(caddr));
```

```
    socklen_t clen = sizeof(caddr);

#if 0
    //int rws = accept(sockfd,NULL, NULL);//rws用于和客户端通信
#else
    int rws = accept(sockfd,(struct sockaddr *)&caddr, &clen);//rws用于和客户端通信
    if(-1 == rws)
    {
        perror("accept");
        close(sockfd);
        return -1;
    }
#endif
    printf("rws=%d\n",rws);

#define SIZE 128
    char buf[SIZE];

    while(1)
    {
        memset(buf,0,SIZE);
//      ret = read(rws,buf,sizeof(buf));//读取客户端发送的消息
        ret = recv(rws,buf,sizeof(buf),0);//读取客户端发送的消息
        if(-1 == ret)
        {
            perror("read");
            break;
        }
        else if(0 == ret)//客户端关闭
        {
            printf("client closed\n");
            break;
        }
        else
            fputs(buf,stdout);

        //ret = write(rws,buf,sizeof(buf));//给客户端回传消息
        ret = send(rws,buf,sizeof(buf),0);//给客户端回传消息
        if(-1 == ret)
        {
            perror("write");
            break;
        }
    }
    close(rws);
    close(sockfd);
    return 0;


}
```

6、c/s模型客户端 TcpClient.c

```
/**********************  版本 一  ********************/
#include <stdio.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
    if(3 != argc)
    {
        printf("Usage: <%s> <IP> <PORT>\n", argv[0]);
        return -1;
    }

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(-1 == sockfd)
    {
        perror("socket");
        return -1;
    }
/*
    struct sockaddr_in serveraddr = {
        .sin_family = AF_INET,
        .sin_addr.s_addr = inet_addr("127.0.0.1");
        .sin_port = htons(8888);
    };
*/

    struct sockaddr_in serveraddr = {0};
    serveraddr.sin_family = AF_INET,
    serveraddr.sin_addr.s_addr = inet_addr(argv[1]);
    serveraddr.sin_port = htons( atoi(argv[2]) );

    int len = sizeof(serveraddr);
    if(-1 == connect(sockfd, (struct sockaddr*)&serveraddr, len) )
    {
        perror("connect");
        return -1;
    }

    while(1)
    {
        write(sockfd, "hello", 5);
        sleep(1);
    }

    close(sockfd);
```

```
}


/**********************  版本二  ********************/

#include <stdio.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>


int main()
{
/*****************************************
 AF_INET              IPv4 Internet protocols
 SOCK_STREAM          string socket
 0
*****************************************/
    int sockfd = socket(AF_INET,SOCK_STREAM,0);
    if(-1 == sockfd)
    {
        perror("socket");
        return -1;
    }
    printf("sockfd=%d\n",sockfd);

/*************************************************************
Internet协议地址结构“ /usr/include/netinet/in.h”
    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
    };
    struct in_addr{
        in_addr_t  s_addr;       // u32 network address
    };
*************************************************************/
    struct sockaddr_in saddr;//定义Internet地址结构变量，保存服务器的ip及port
    memset(&saddr,0,sizeof(saddr));//bzero

    saddr.sin_family = AF_INET;//指定网络协议 IPV4
    saddr.sin_port = htons(8000);//指定服务器的端口号 >=5001,由主机序转网络字节序
    saddr.sin_addr.s_addr = inet_addr("192.168.2.173");//指定服务器的ip地址，ip地址
由点分式转32为无符号网络字节序

    socklen_t slen = sizeof(saddr);

    int ret = connect(sockfd, (struct sockaddr *)&saddr,slen);//将服务器的ip和port
与sockfd绑定
    if(-1 == ret)
    {
        perror("connect");
        goto ERR_STEMP;
    }
```

```
    printf("connect success\n");


#define SIZE 128
    char buf[SIZE];
    do
    {
        memset(buf,0,SIZE);
        fgets(buf,SIZE-1,stdin);//从标准输入获取字符串
        ret = write(sockfd,buf,sizeof(buf));//给服务器写消息
        if(-1 == ret)
        {
            perror("write");
            break;
        }
        memset(buf,0,SIZE);
        ret = read(sockfd,buf,SIZE);
        if(-1 == ret)
        {
            perror("read");
            break;
        }
        else if(0 == ret)//服务器关闭
        {
            printf("server closed\n");
            break;
        }
        else
            fputs(buf,stdout);
    }while(strncmp(buf,"quit",4) != 0);

    close(sockfd);
    return 0;


ERR_STEMP:
        close(sockfd);
        return -1;
}
```