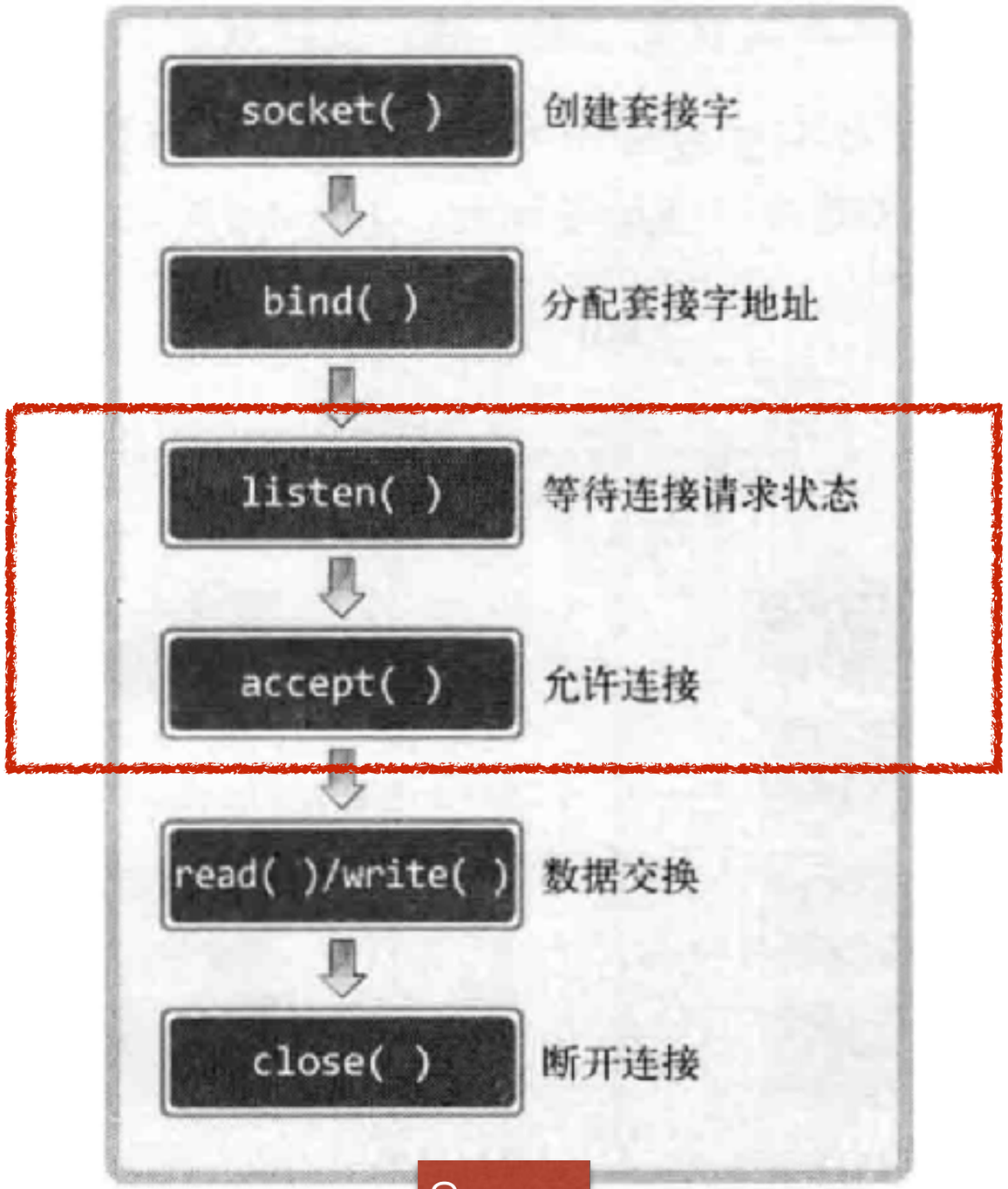
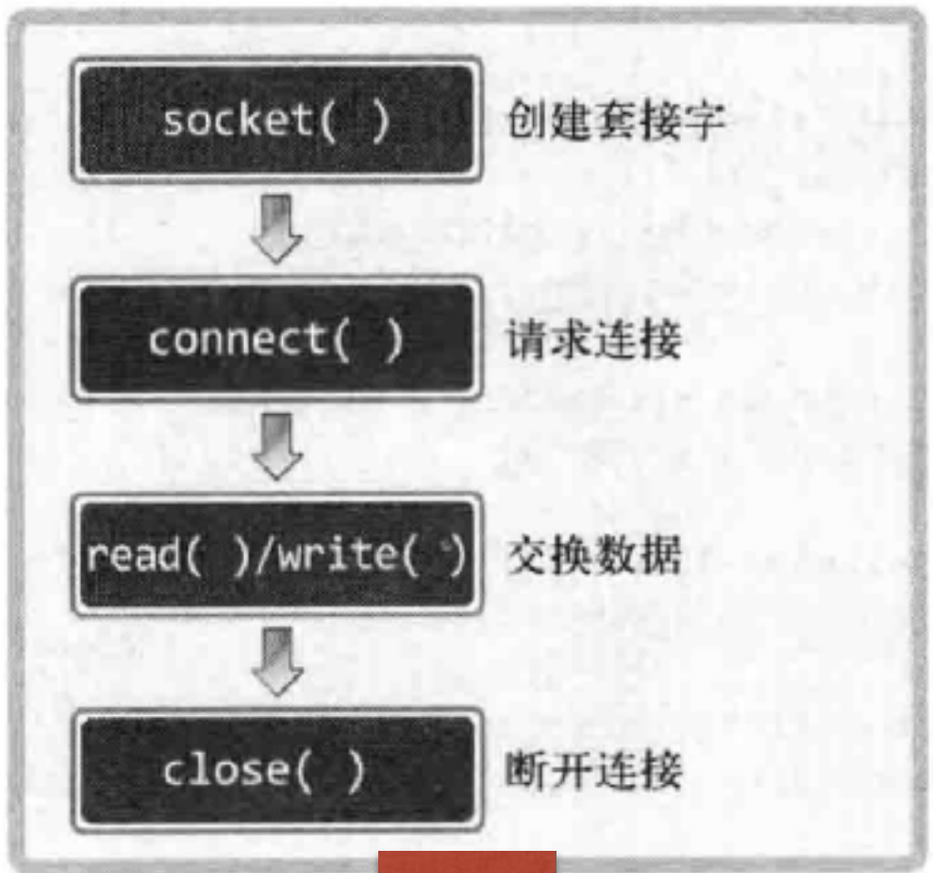
基于TCP与UDP
的服务器端/客户端程序开发

基于TCP的C/S程序-服务器端分析

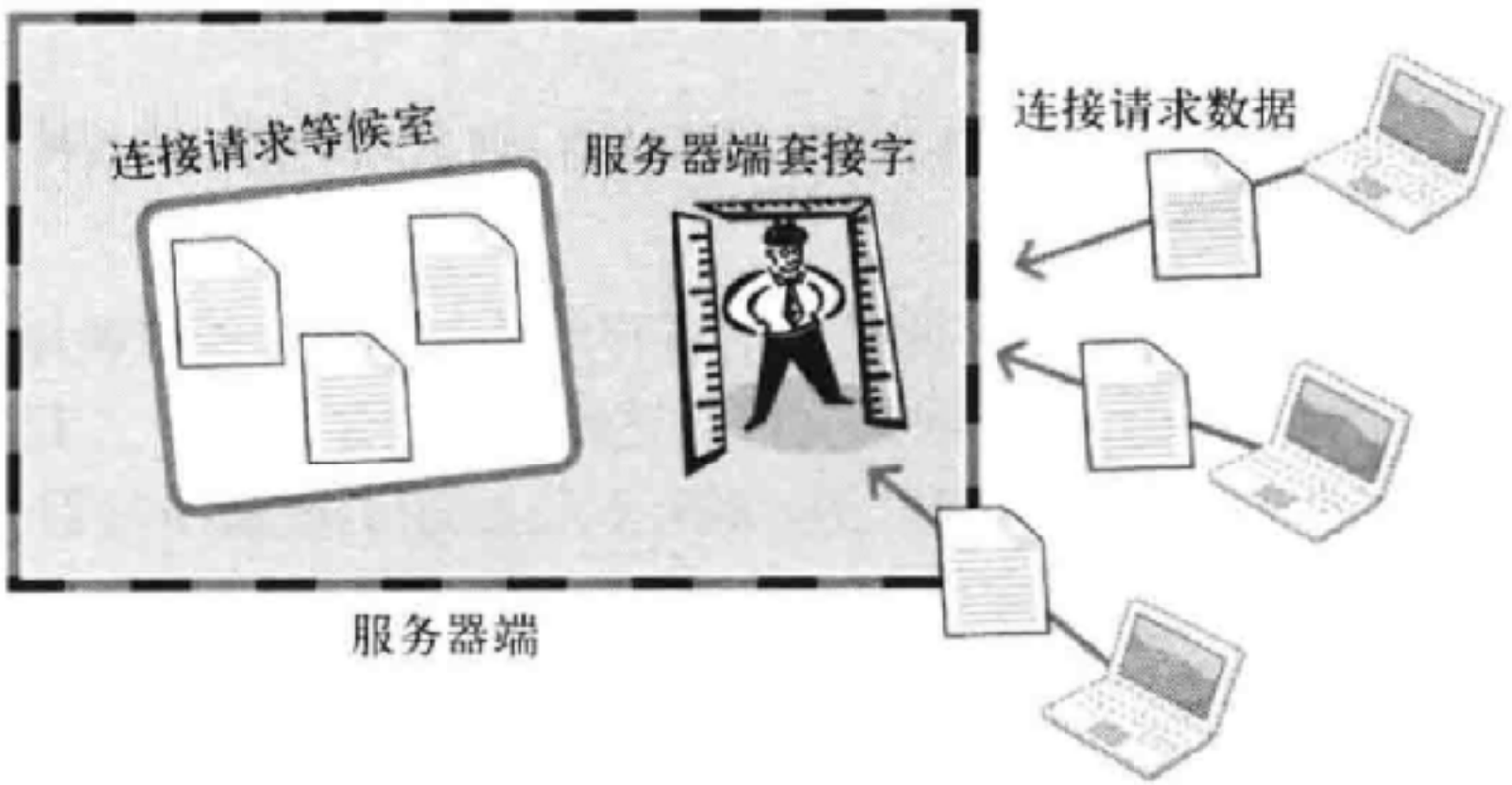




listen()的工作过程

- 调用listen()函数进入等待连接请求状态
- 在listen()之后，客户端的connect()调用才有作用
- listen(int sock, int backlog)
- 成功返回0，失败返回-1

listen()：连接请求等待队列



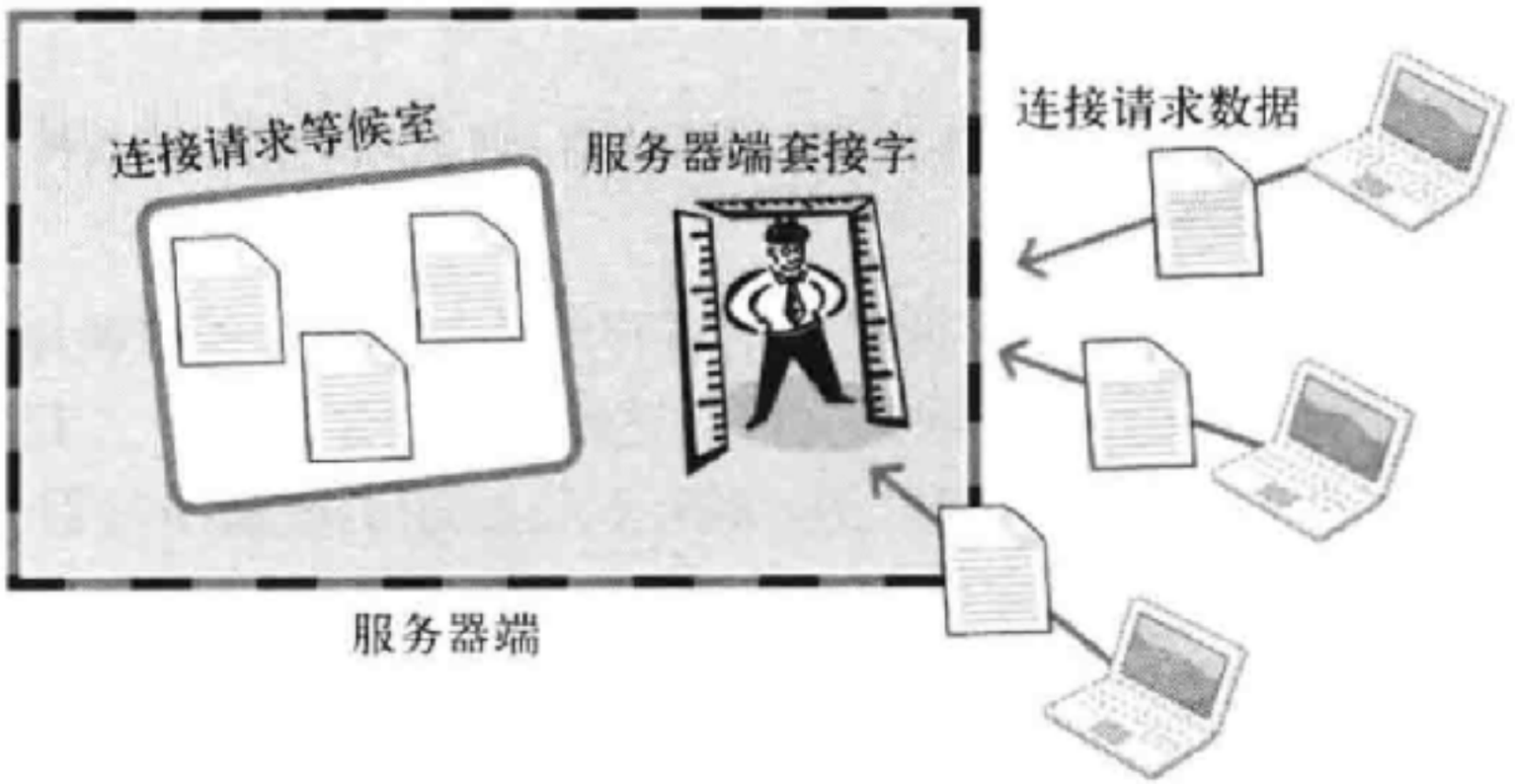
- 客户端请求连接时，服务器受理连接前，（客户端）连接一直处于等待状态

- listen(int sock, int backlog)函数两个参数的形象比喻
  - sock: 门卫、大门、传达室
    - 客户请求建立连接时，先咨询门卫：是否可以建立连接?

- listen(int sock, int backlog)函数两个参数的形象比喻
  - backlog: 等待室、休息室
    - 门卫同意用户连接后，用户进入后需要在休息室等待
    - 休息室的容量是有限的

- 再谈backlog参数
  - 根据服务器端的特性确定
    - 例如，请求频繁的Web服务器可设为15
- 如何确定：始终是按照实验来确定

accept()的工作过程



- 队列中等待的请求会被服务器最终处理
- 处理时由哪个套接字来处理?

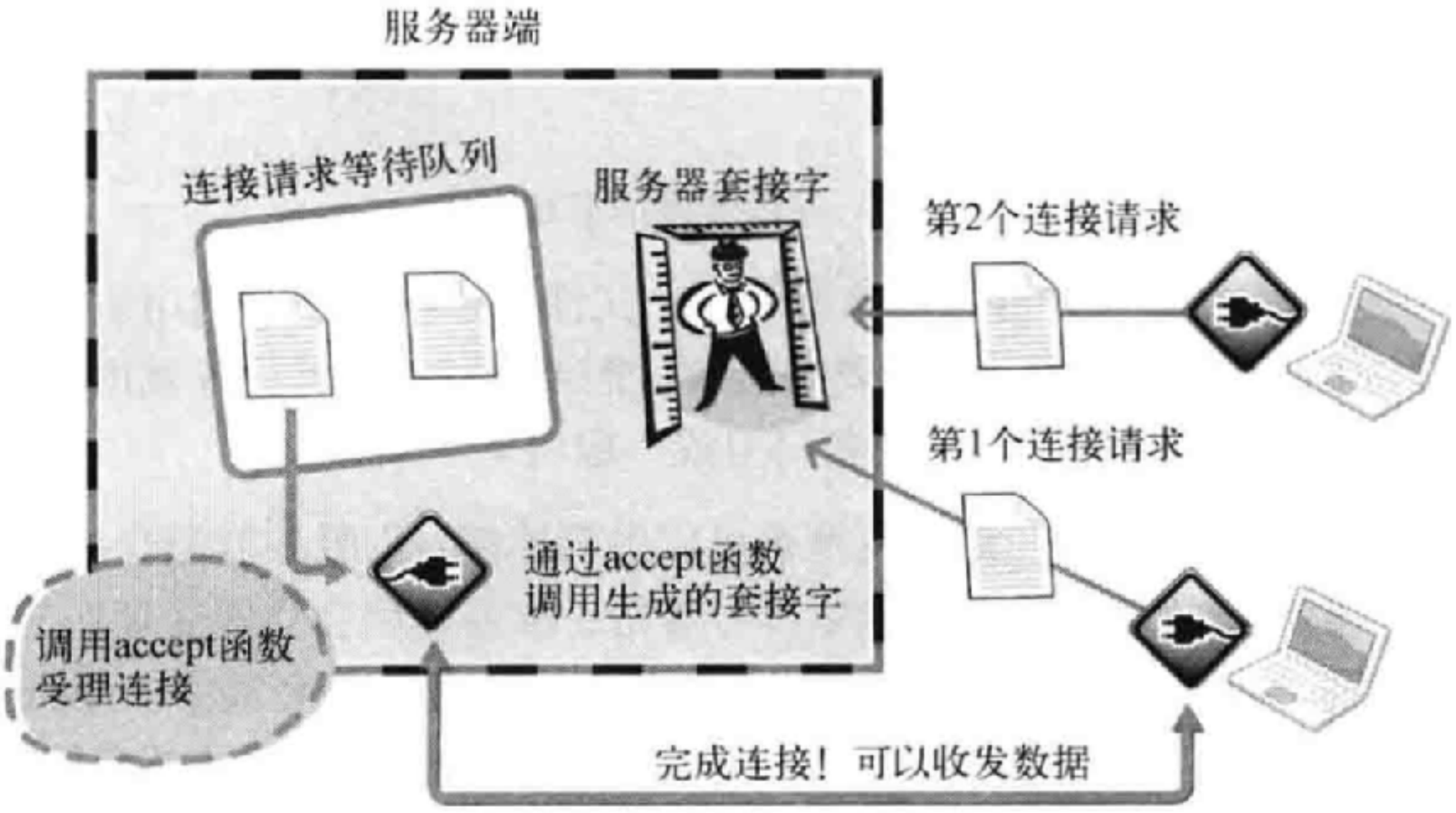
- int accept(int sock, struct sockaddr * addr, socklen_t * addrlen)
  - sock：服务器套接字的文件描述符（fd）
  - addr：保存发起连接请求的客户端的地址信息
  - addrlen：第二个参数的长度
- 为何第二、三个参数传指针?

```
9.  int main(int argc, char *argv[])
10. {
11.     int serv_sock;
12.     int clnt_sock;
13.
14.     struct sockaddr_in serv_addr;
15.     struct sockaddr_in clnt_addr;
16.     socklen_t clnt_addr_size;
17.
18.     char message[]="Hello World!";
19.
20.     if(argc!=2)
21.     {
22.         printf("Usage : %s <port>\n", argv[0]);
23.         exit(1);
24.     }
25.
26.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
27.     if(serv_sock == -1)
28.         error_handling("socket() error");
29.
```

```
30.     memset(&serv_addr, 0, sizeof(serv_addr));
31.     serv_addr.sin_family=AF_INET;
32.     serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
33.     serv_addr.sin_port=htons(atoi(argv[1]));
34.
35.     if(bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr))==-1)
36.         error_handling("bind() error");
37.
38.     if(listen(serv_sock, 5)==-1)
39.         error_handling("listen() error");
```

```
40.
41.     clnt_addr_size=sizeof(clnt_addr);
42.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
43.     if(clnt_sock==-1)
44.         error_handling("accept() error");
45.
46.     write(clnt_sock, message, sizeof(message));
47.     close(clnt_sock);
48.     close(serv_sock);
49.     return 0;
50. }
```

- accept()工作过程
  - 受理连接请求等待队列里的客户端连接请求
  - 若函数调用成功，则函数内部产生用于与客户端通信（I/O）的套接字，返回其文件描述符（fd）
- 注意：套接字是自动创建的，与服务器的套接字并不同

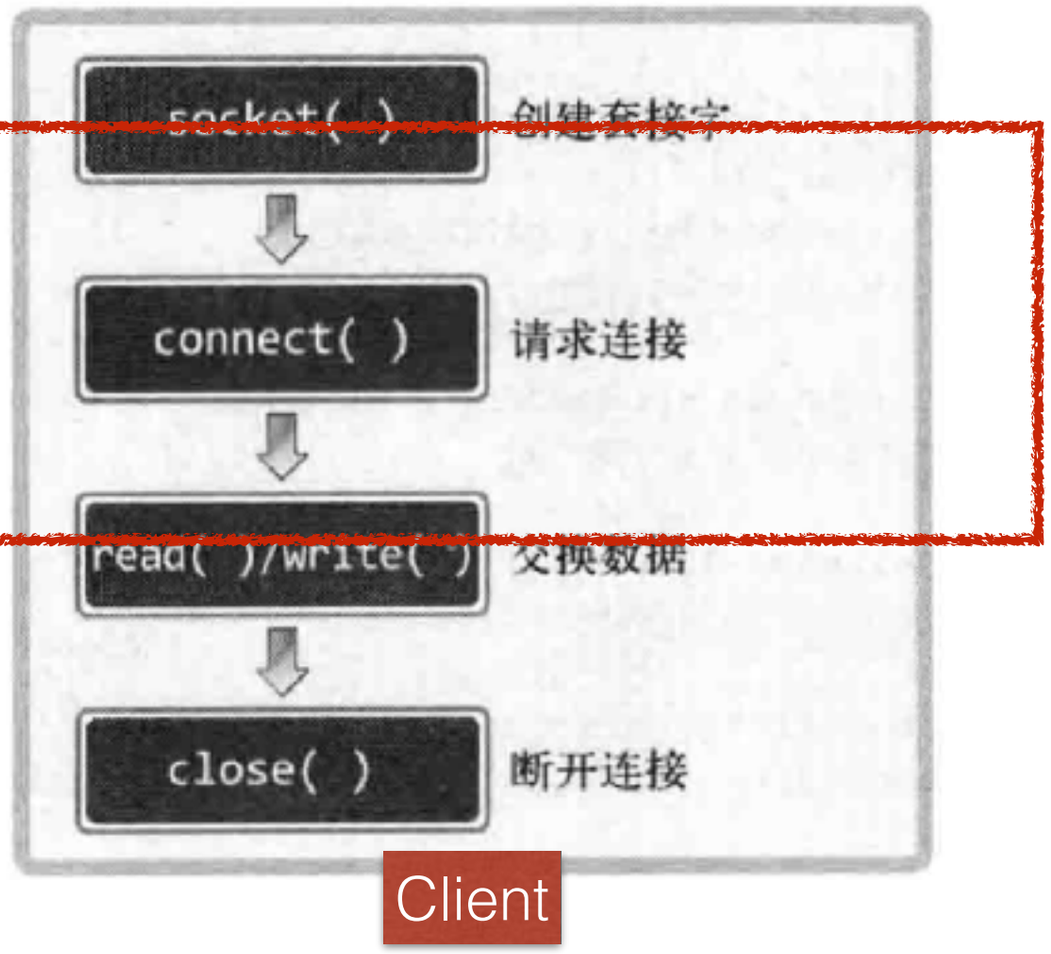

服务器端
连接请求等待队列
服务器套接字
第2个连接请求
第1个连接请求
通过accept函数
调用生成的套接字
调用accept函数
受理连接
完成连接！可以收发数据

代码分析： Hello socket

```
21.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
22.     if(serv_sock == -1)
23.         error_handling("socket() error");
24.
25.     memset(&serv_addr, 0, sizeof(serv_addr));
26.     serv_addr.sin_family=AF_INET;
27.     serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
28.     serv_addr.sin_port=htons(atoi(argv[1]));
29.
30.     if(bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr))==-1)
31.         error_handling("bind() error");
32.
33.     if(listen(serv_sock, 5)==-1)
34.         error_handling("listen() error");
35.
36.     clnt_addr_size=sizeof(clnt_addr);
37.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr,&clnt_addr_size);
38.     if(clnt_sock==-1)
39.         error_handling("accept() error");
40.
41.     write(clnt_sock, message, sizeof(message));
42.     close(clnt_sock);
43.     close(serv_sock);
```

Server

基于TCP的C/S程序-客户端分析



connect()

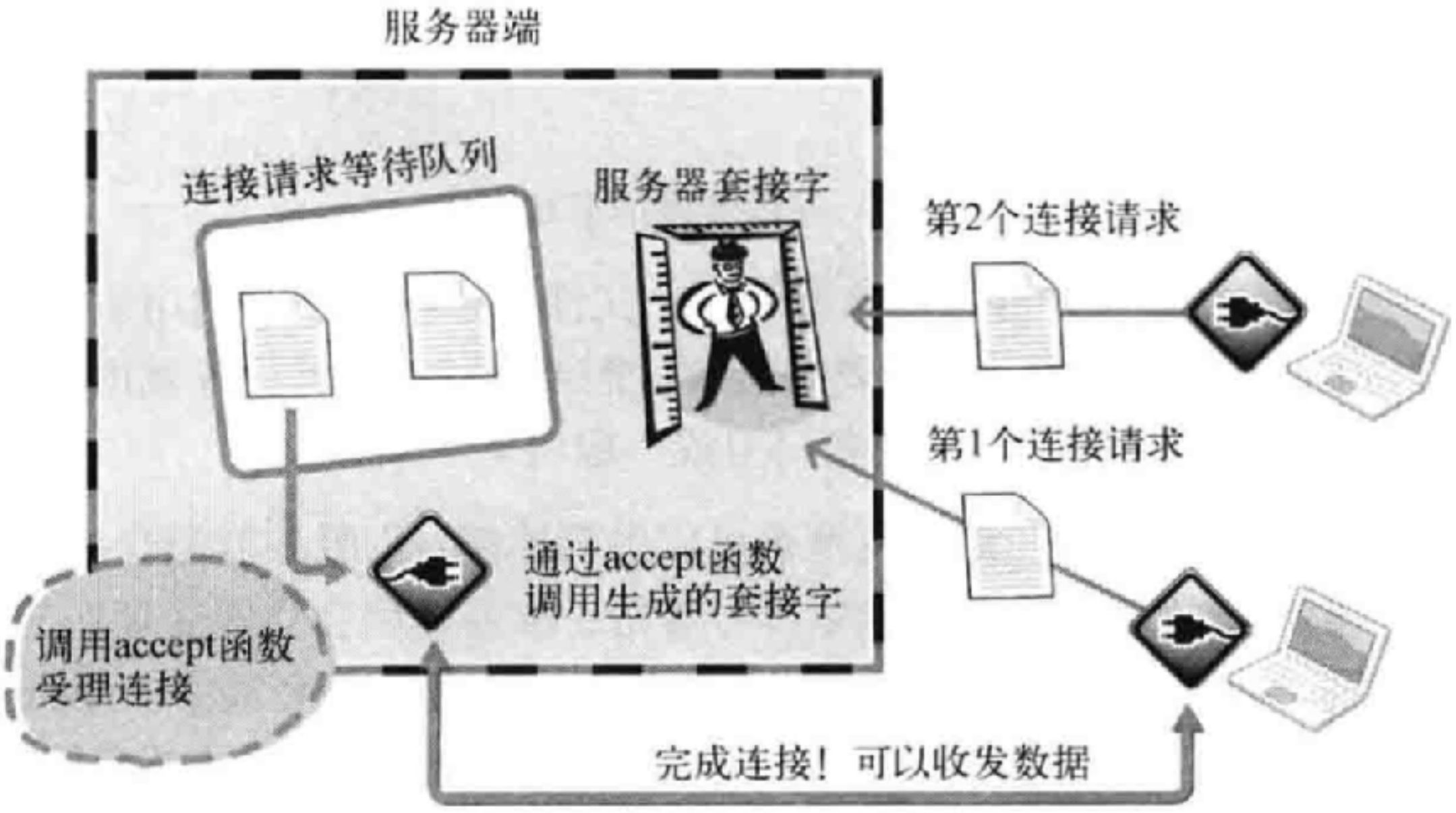
- 客户端与服务器端的显著区别：
  - 服务器端“等待连接”
  - 客户端“请求连接”
- 服务器端调用listen()之后，客户端就可以请求了

- int connect(int sock, struct sockaddr * addr, socklen_t * addrlen)
  - sock：服务器套接字的文件描述符（fd）
  - addr：保存可建立起连接的服务器端的地址信息
  - addrlen：第二个参数的长度

- 调用connect函数以后，客户端将一直等待返回结果，直到下面情况的发生
  - 服务器端接收连接，客户端获得返回结果
  - 发生异常，中断连接请求

- connect()调用后的数据交换
  - 服务器端的listen()函数需要一直等待客户端，connect()也需要等待服务器端接收连接请求。他们的等待有何联系和先后次序?
- connect调用后不必马上交换数据

- 客户端的套接字地址信息在哪里?
  - 一个TCP/IP应用层程序需要至少对应一个<IP地址, 端口号>
  - 服务器端运行时，知道自己的<IP地址,端口号>，显示地用bind来绑定到本机
- 客户端没有bind。其IP地址和端口号在connect函数处确定
  - 本机IP地址
  - 随机分配端口号

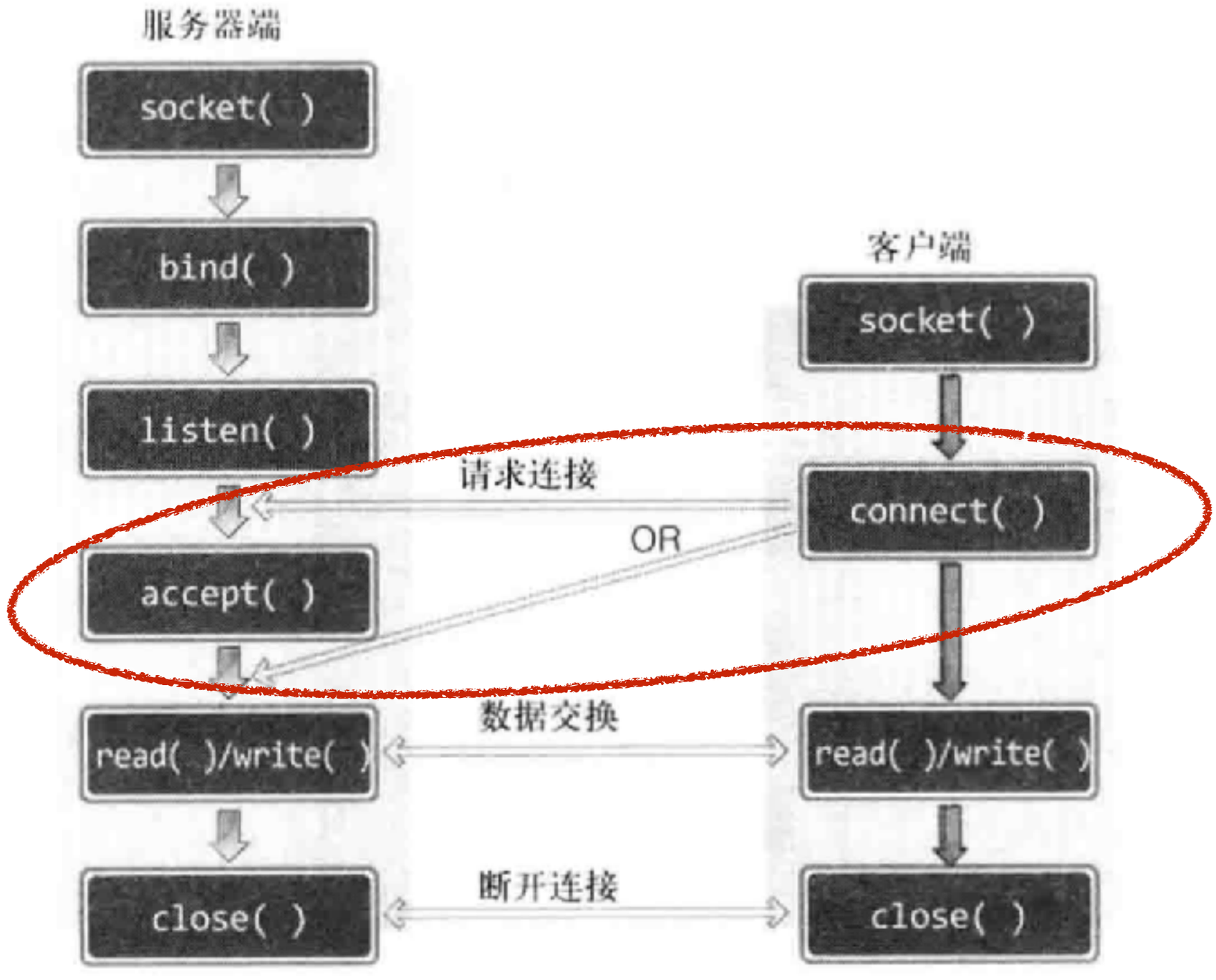

服务器端
连接请求等待队列
服务器套接字
第2个连接请求
第1个连接请求
通过accept函数
调用生成的套接字
调用accept函数
受理连接
完成连接！可以收发数据

代码分析： Hello socket

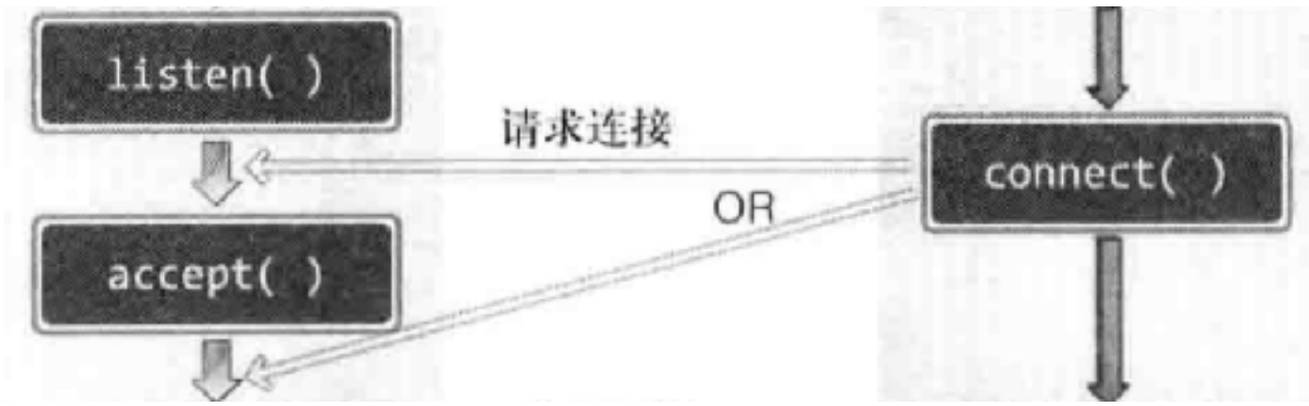
```
17.     sock=socket(PF_INET, SOCK_STREAM, 0);
18.     if(sock == -1)
19.         error_handling("socket() error");
20.
21.     memset(&serv_addr, 0, sizeof(serv_addr));
22.     serv_addr.sin_family=AF_INET;
23.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
24.     serv_addr.sin_port=htons(atoi(argv[2]));
25.
26.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
27.         error_handling("connect() error!");
28.
29.     str_len=read(sock, message, sizeof(message)-1);
30.     if(str_len==-1)
31.         error_handling("read() error!");
32.
33.     printf("Message from server : %s \n", message);
34.     close(sock);
35.     return 0;
```

Client

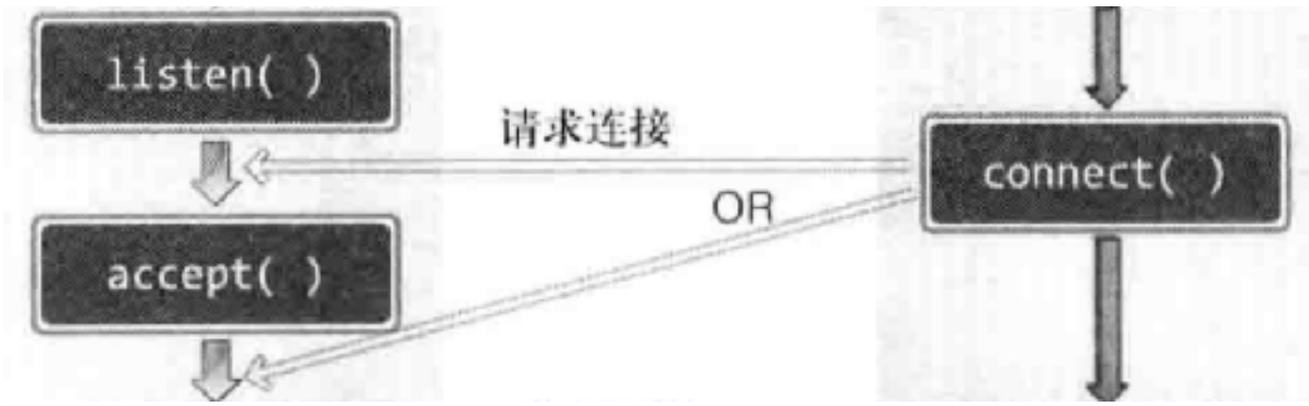
总结：基于TCP的C/S程序函数调用关系



总结：基于TCP的C/S程序函数调用关系



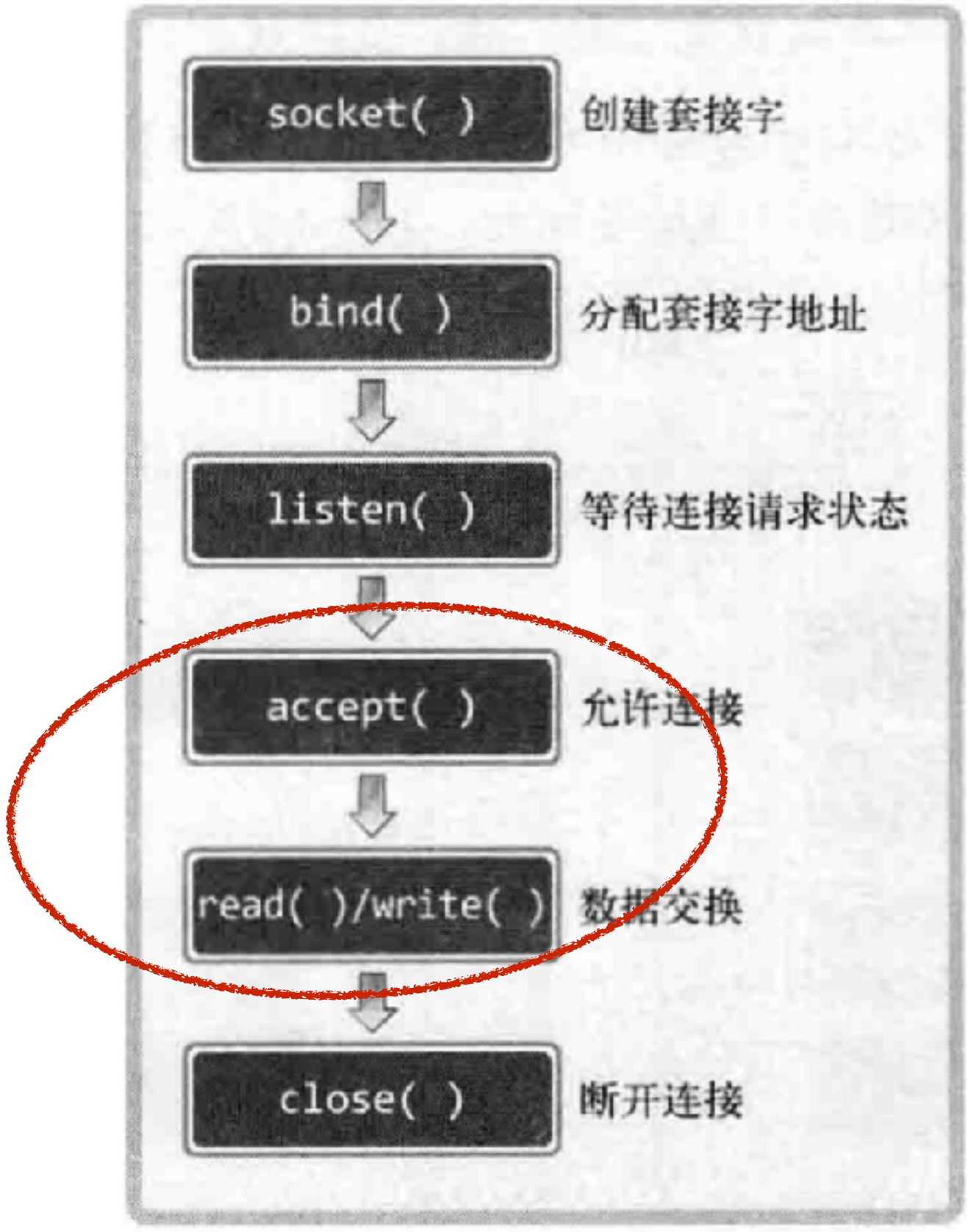
- 客户端的connect()
  - 必须要在listen()之后才可以调用
  - 调用后等待accept()

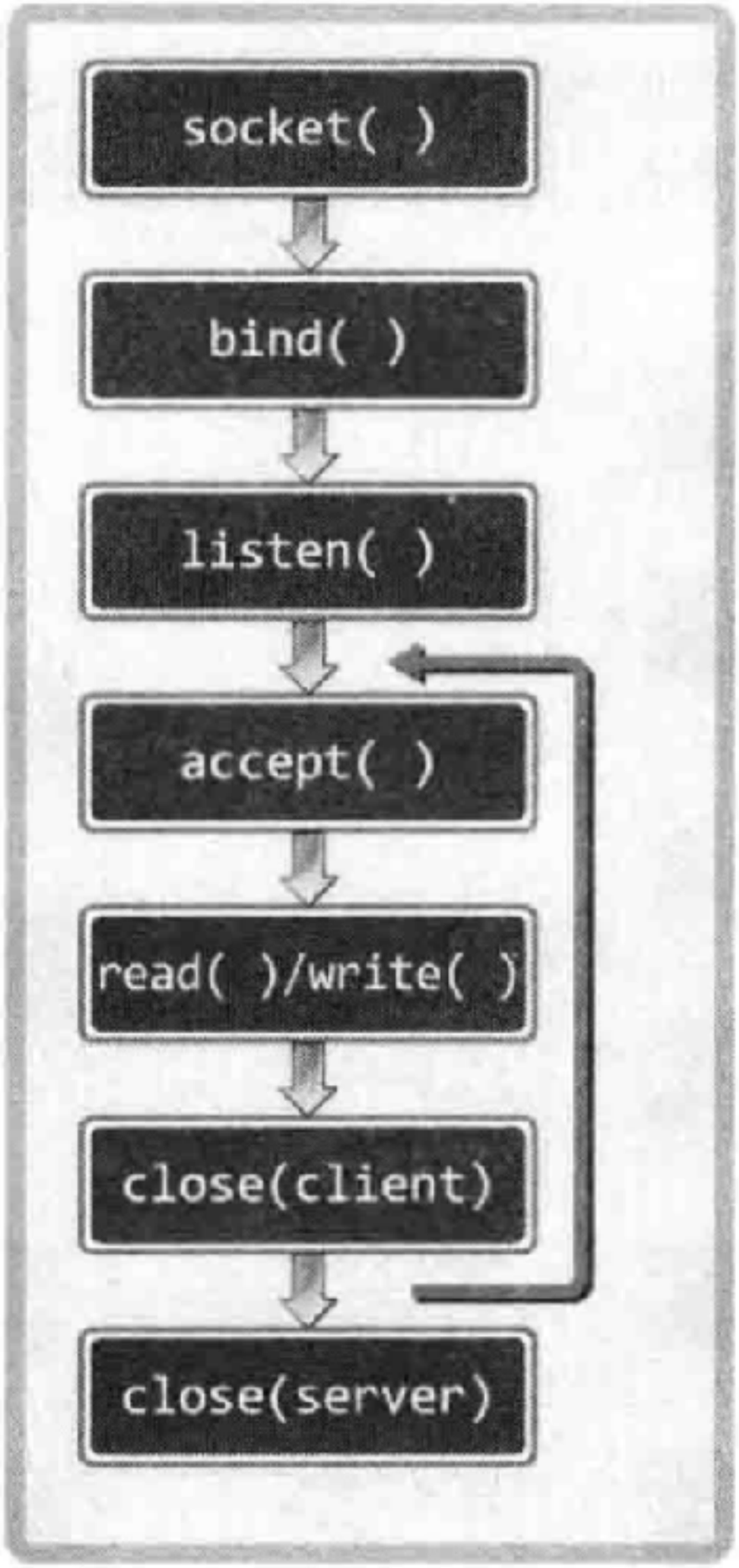


- 服务器端的listen()和accept()
  - 调用listen()后开始接收客户端请求
  - accept()的调用与客户端请求没有严格的顺序要求
    - 先来请求，再accept
    - 直接先accept？再来请求也可以

迭代执行：更实用的C/S程序

- 已经讨论的程序将每个函数执行一遍，close()了socket以后就退出运行了
- 实际的持续提供服务的服务器不可能执行一次程序就关闭了
- 需要有持续运行的基于TCP的C/S程序
- 用循环实现即可


socket( )
创建套接字
bind( )
分配套接字地址
listen( )
等待连接请求状态
accept( )
允许连接
read( )/write( )
数据交换
close( )
断开连接


socket( )
bind( )
listen( )
accept( )
read( )/write( )
close(client)
close(server)

迭代的回声服务器

- “回声”（echo）服务器
  - 定义
- 满足如下（比较简化的）设计要求：
  - 服务器端同一时刻只处理一个客户端请求
  - 服务器端依次向五个客户端提供服务，然后退出
  - 客户端发送文本给服务器，服务器回传（echo back）
  - 客户端输入退出指令，结束一个客户端的请求

Server

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.
8.  #define BUF_SIZE 1024
9.  void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int serv_sock, clnt_sock;
14.     char message[BUF_SIZE];
15.     int str_len, i;
16.
17.     struct sockaddr_in serv_adr, clnt_adr;
18.     socklen_t clnt_adr_sz;
19.
20.     if(argc!=2) {
21.         printf("Usage : %s <port>\n", argv[0]);
22.         exit(1);
23.     }
```

```
25.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
26.     if(serv_sock==-1)
27.         error_handling("socket() error");
28.
29.     memset(&serv_adr, 0, sizeof(serv_adr));
30.     serv_adr.sin_family=AF_INET;
31.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
32.     serv_adr.sin_port=htons(atoi(argv[1]));
33.
34.     if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
35.         error_handling("bind() error");
36.
37.     if(listen(serv_sock, 5)==-1)
38.         error_handling("listen() error");
39.
40.     clnt_adr_sz=sizeof(clnt_adr);
41.
```

Server

```
42.     for(i=0; i<5; i++)
43.     {
44.         clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
45.         if(clnt_sock==-1)
46.             error_handling("accept() error");
47.         else
48.             printf("Connected client %d \n", i+1);
49.
50.         while((str_len=read(clnt_sock, message, BUF_SIZE))!=0)
51.             write(clnt_sock, message, str_len);
52.
53.         close(clnt_sock);
54.     }
55.     close(serv_sock);
56.     return 0;
57. }
```

Server

```
23.     sock=socket(PF_INET, SOCK_STREAM, 0);
24.     if(sock==-1)
25.         error_handling("socket() error");
26.
27.     memset(&serv_adr, 0, sizeof(serv_adr));
28.     serv_adr.sin_family=AF_INET;
29.     serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
30.     serv_adr.sin_port=htons(atoi(argv[2]));
31.
32.     if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
33.         error_handling("connect() error!");
34.     else
35.         puts("Connected.....");
```

Client

```
37.     while(1)
38.     {
39.         fputs("Input message(Q to quit): ", stdout);
40.         fgets(message, BUF_SIZE, stdin);
41.
42.         if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
43.             break;
44.
45.         write(sock, message, strlen(message));
46.         str_len=read(sock, message, BUF_SIZE-1);
47.         message[str_len]=0;
48.         printf("Message from server: %s", message);
49.     }
50.     close(sock);
51.     return 0;
```

跳出循环，结束socket，导致Server端的50-51行while中断

Client

回声程序存在的问题思考

```
37.     while(1)
38.     {
39.         fputs("Input message(Q to quit): ", stdout);
40.         fgets(message, BUF_SIZE, stdin);
41.
42.         if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
43.             break;
44.
45.         write(sock, message, strlen(message));
46.         str_len=read(sock, message, BUF_SIZE-1);
47.         message[str_len]=0;
48.         printf("Message from server: %s", message);
49.     }
50.     close(sock);
51.     return 0;
```

Client

- 回声程序存在的问题 - 如何解决?
  - 策略一：提前如果知道传递的数据长度
  - 策略二：约定一个分界符

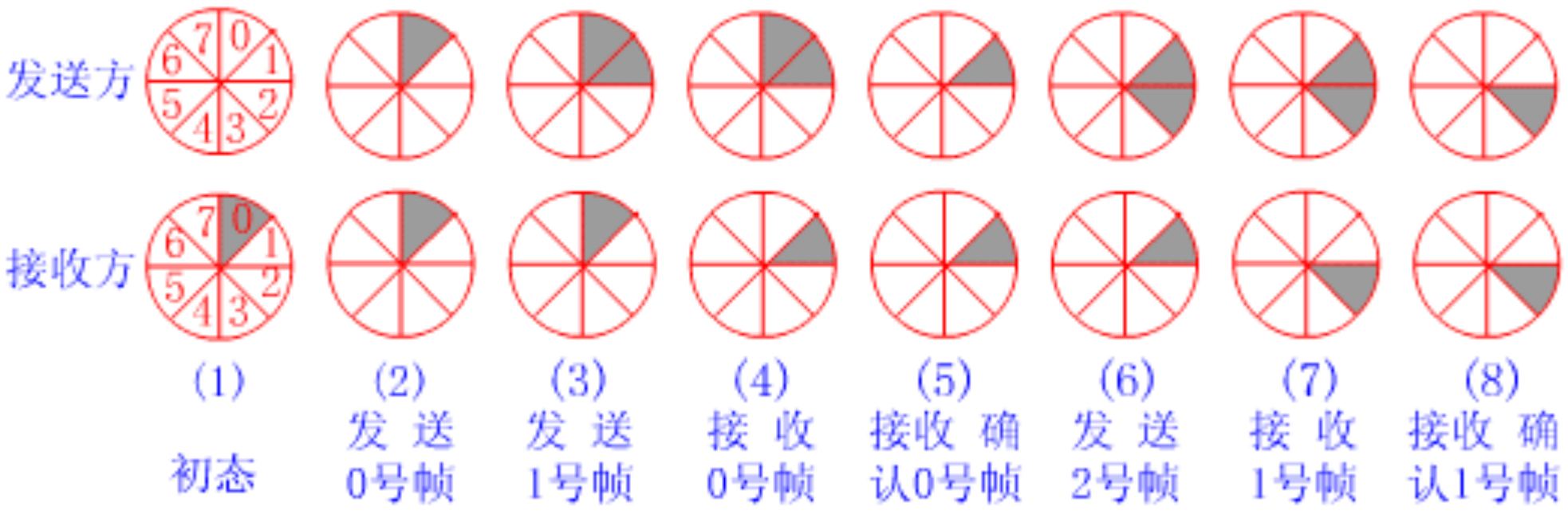
TCP原理：回顾

- TCP套接字的I/O缓存（缓冲）机制
  - 对于同样一段发送和接受的数据，发送过程和接受过程对于数据的操作次数是可以不一样的，例如一个长55字节的数据：
    - 可以传输11次，每次传输5个字节
    - 可以接收5次，每次接收11个字节
    - 可以传输/接收10次，传输的字节数1+2+...+9+10

- TCP套接字的I/O缓存（缓冲）机制续
  - 接收端来不及处理的数据都放在哪?
    - 发送端缓存
    - 接收端缓存
    - read/write函数只负责在调用的瞬间把数据读出缓存/加入缓存

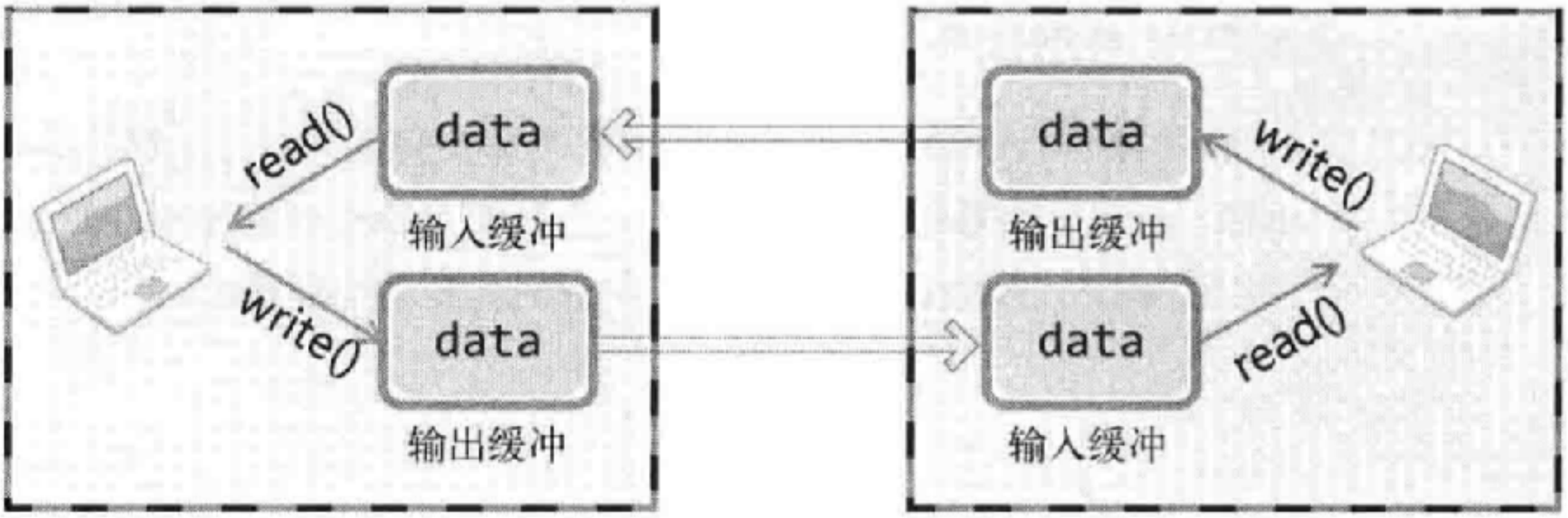
- TCP套接字的I/O缓存（缓冲）机制续
  - 输入缓存会不会“爆仓”?
    - TCP会控制数据流，不允许发生超过输入缓冲大小的数据传输
    - 滑动窗口（Sliding window）协议，

- TCP套接字的I/O缓存（缓冲）机制续
  - 滑动窗口协议



- TCP套接字的I/O缓存（缓冲）机制续
  - 套接字A：“你好，最多可以向我传递50字节”
  - 套接字B：“OK”
  - （发送小于50字节）
  - 套接字A：“我腾出了20字节的空间，最多可以收70字节”
  - 套接字B：“OK”

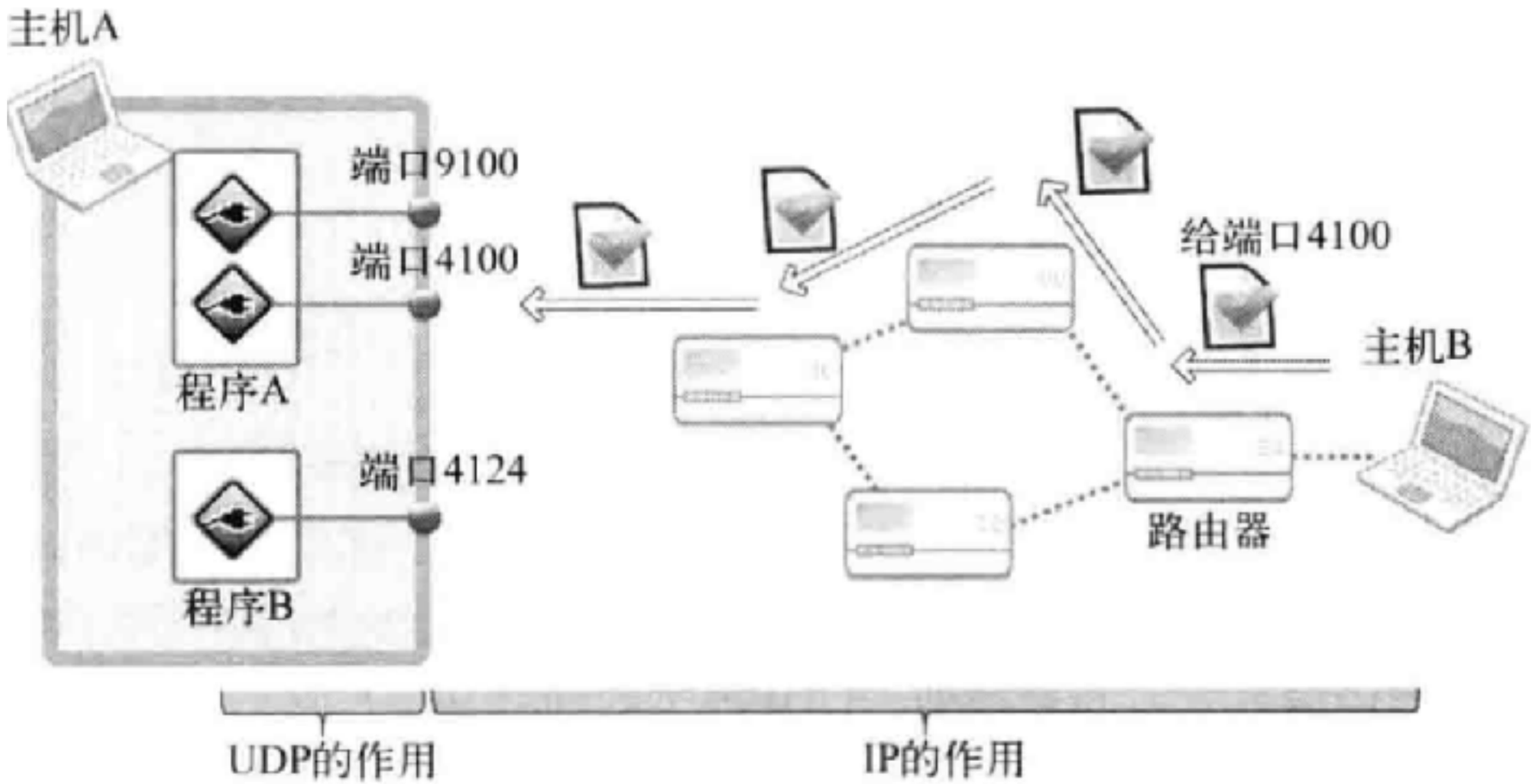
TCP的I/O过程


read()
data
输入缓冲
write()
data
输出缓冲
data
输出缓冲
write()
data
输入缓冲
read()

基于UDP的C/S程序

- TCP与UDP的最主要区别
  - 可靠性
  - 流控制

UDP的内部工作原理



- UDP的最重要作用就是根据端口号来将数据包交付给应用层程序

UDP的性能

- 之前的比喻：
  - TCP是流水线/UDP是快递
  - TCP是电话/UDP是邮件
- TCP比UDP快?
  - UDP是TCP的一个上界

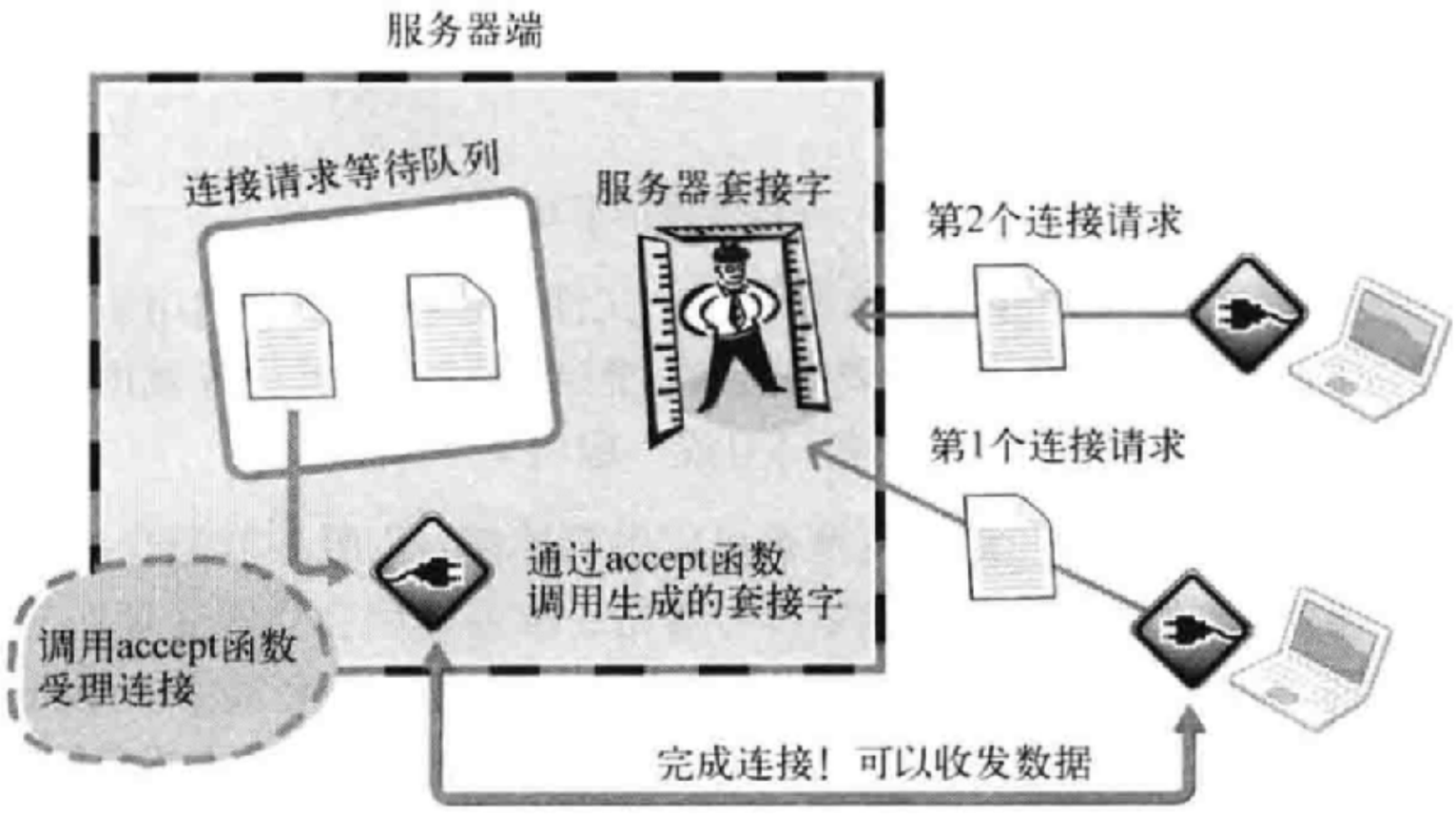
- UDP的性能续
  - 实际上UDP并非每次都比TCP传输快，原因可能为
    - 收发数据前后进行的连接设置与清除
    - UDP参与者两端为了保证可靠性而添加的流控制

基于UDP的C/S程序

- 在基于TCP的C/S程序上理解UDP
  - UDP中服务器端和客户端没有连接
    - 无需经过连接过程：listen()、accept()
    - UDP程序只有创建套接字的过程和数据交换的过程
    - 不分服务器端和客户端

UDP服务器/客户端的套接字

TCP



- UDP的服务器端和客户端均只需要一个套接字
  - 与TCP的“门卫”模式对比?

UDP

- UDP为“邮筒”模式
- 只需要一个套接字即可用来和多个主机通信
- 套接字是被“复用”的

UDP的I/O函数

- 基于TCP的传输不需要每次都指定对方地址信息，因为连接是保持的，TCP套接字知道对方地址信息
- UDP每次传输时都要传入目标地址的信息
  - 每次写信都要写地址
  - 邮筒不包含目标地址信息

- ssize_t **sendto**(int sock, void *buf, size_t nbytes, int flags, struct sockaddr *to, socklen_t addr_len);
  - buf：待传输数据缓冲的地址
  - nbytes：待传输数据长度
  - to：目标地址信息的指针

```
40.
41.     clnt_addr_size=sizeof(clnt_addr);
42.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
43.     if(clnt_sock==-1)
44.         error_handling("accept() error");
45.
46.     write(clnt_sock, message, sizeof(message));
47.     close(clnt_sock);
48.     close(serv_sock);
49.     return 0;
50. }
```

- ssize_t **recvfrom**(int sock, void *buf, size_t nbytes, int flags, struct sockaddr *to, socklen_t addr_len);
  - buf：待传输数据缓冲的地址
  - nbytes：待传输数据长度
  - to：目标地址信息的指针

基于UDP的C/S程序-代码分析

- 服务器端

```
8.  #define BUF_SIZE 30
9.  void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int serv_sock;
14.     char message[BUF_SIZE];
15.     int str_len;
16.     socklen_t clnt_adr_sz;
17.
```

Server

```
18.     struct sockaddr_in serv_adr, clnt_adr;
19.     if(argc!=2){
20.         printf("Usage : %s <port>\n", argv[0]);
21.         exit(1);
22.     }
23.
24.     serv_sock=socket(PF_INET, SOCK_DGRAM, 0);
25.     if(serv_sock == -1)
26.         error_handling("UDP socket creation error");
27.
28.     memset(&serv_adr, 0, sizeof(serv_adr));
29.     serv_adr.sin_family=AF_INET;
30.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
31.     serv_adr.sin_port=htons(atoi(argv[1]));
32.
33.     if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
34.         error_handling("bind() error");
35.
36.     while(1)
37.     {
38.         clnt_adr_sz=sizeof(clnt_adr);
39.         str_len=recvfrom(serv_sock, message, BUF_SIZE, 0,
40.                 (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
41.         sendto(serv_sock, message, str_len, 0,
42.                 (struct sockaddr*)&clnt_adr, clnt_adr_sz);
43.     }
44.     close(serv_sock);
45.     return 0;
```

Server

```
2.  #define BUF_SIZE 30
3.  void error_handling(char *message);
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int sock;
8.      char message[BUF_SIZE];
9.      int str_len;
10.     socklen_t adr_sz;
11.
12.     struct sockaddr_in serv_adr, from_adr;
13.     if(argc!=3){
14.         printf("Usage : %s <IP> <port>\n", argv[0]);
15.         exit(1);
16.     }
17.
18.     sock=socket(PF_INET, SOCK_DGRAM, 0);
19.     if(sock==-1)
20.         error_handling("socket() error");
21.
22.     memset(&serv_adr, 0, sizeof(serv_adr));
23.     serv_adr.sin_family=AF_INET;
24.     serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
25.     serv_adr.sin_port=htons(atoi(argv[2]));
```

客户端

Client

```
27.     while(1)
28.     {
29.         fputs("Insert message(q to quit): ", stdout);
30.         fgets(message, sizeof(message), stdin);
31.         if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
32.             break;
33.
34.         sendto(sock, message, strlen(message), 0,
35.             (struct sockaddr*)&serv_adr, sizeof(serv_adr));
36.         adr_sz=sizeof(from_adr);
37.         str_len=recvfrom(sock, message, BUF_SIZE, 0,
38.                 (struct sockaddr*)&from_adr, &adr_sz);
39.         message[str_len]=0;
40.         printf("Message from server: %s", message);
41.     }
42.     close(sock);
43.     return 0;
44. }
```

Client

看个栗子

回顾：流套接字通用编程模型

- 服务端：
  - 套接字的创建
  - 绑定套接字到指定的IP地址和端口号
  - 设置套接字进入监听状态
  - 接收连接请求
  - 收发数据
  - 关闭套接字
- 客户端：
  - 套接字创建
  - 申请建立连接
  - 收发数据
  - 断开连接
  - 关闭套接字

服务器端

Socket()建立套接字，返回套接字号s

bind()，套接字s与本地接口绑定

listen()，通知TCP服务器准备好接收连接

accept()，接收连接，等待客户端的请求

建立连接，accept()返回，得到新的套接字

recv()/send()，接收/发送数据

closesocket()，关闭套接字

客户端

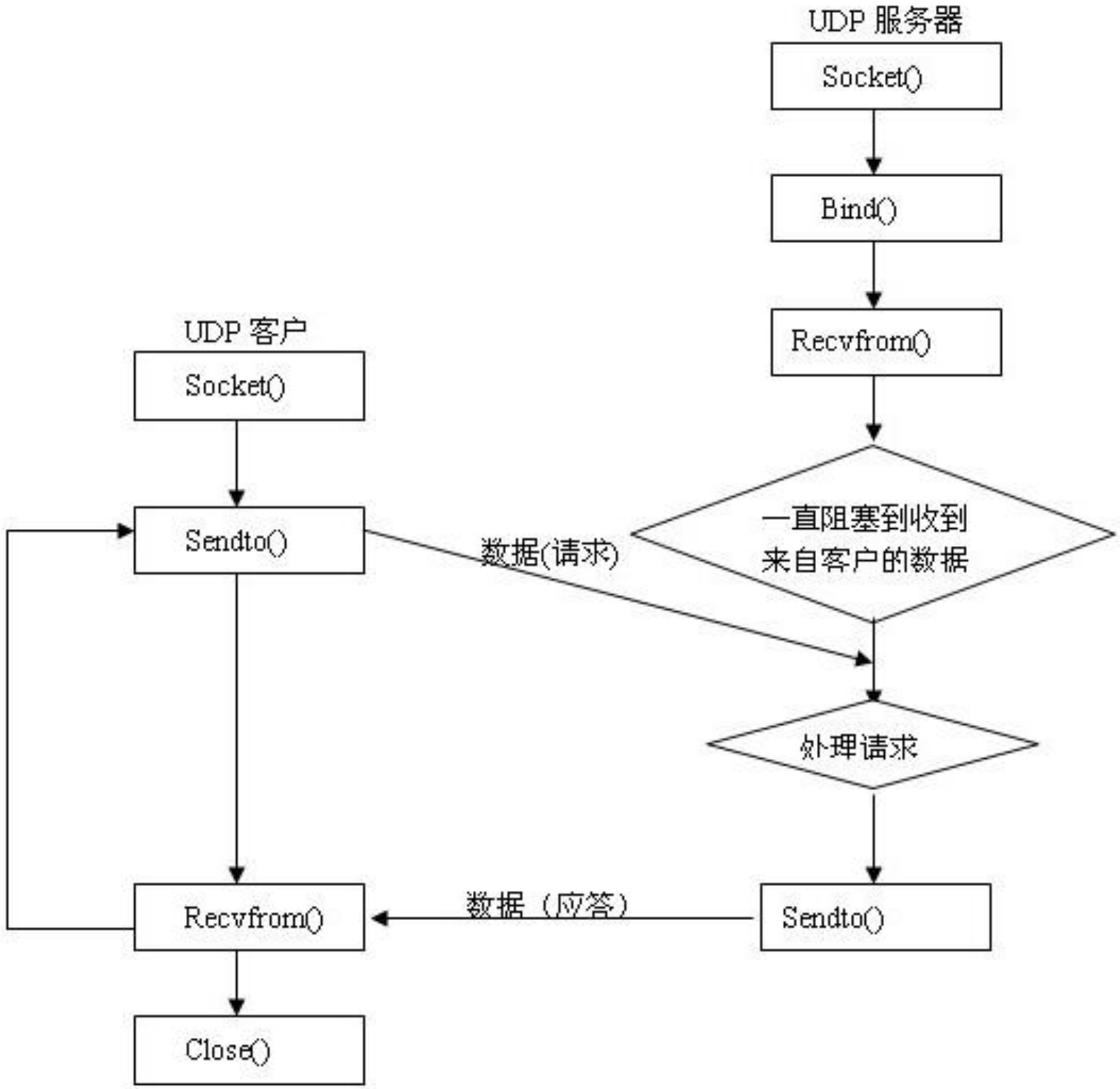
Socket()建立套接字，返回套接字号s

Connect()，主动发出请求，连接服务器

send()/recv()，发送/接收数据

回顾：数据报套接字通用编程模型

- 服务端：
  - 创建套接字
  - 绑定IP地址和端口
  - 收发数据
  - 关闭套接字
- 客户端：
  - 创建套接字
  - 收发数据
  - 关闭套接字


UDP 服务器
Socket()
Bind()
Recvfrom()
一直阻塞到收到
来自客户的数据
处理请求
Sendto()
UDP 客户
Socket()
Sendto()
数据(请求)
Recvfrom()
数据（应答）
Close()

UDP传输的I/O特性

- 回顾：TCP不存在数据边界，I/O是基于流的模式
  - TCP传输中调用I/O函数的次数无意义
- UDP是存在数据边界的
  - 调用多少次输入函数则必须调用多少次输出函数来配套

- 其中一端（服务器端）

```
30.     for(i=0; i<3; i++)
31.     {
32.         sleep(5);           // delay 5 sec.
33.         adr_sz=sizeof(your_adr);
34.         str_len=recvfrom(sock, message, BUF_SIZE, 0,
35.                 (struct sockaddr*)&your_adr, &adr_sz);
36.
37.         printf("Message %d: %s \n", i+1, message);
38.     }
39.     close(sock);
40.     return 0;
```

- 另一端（客户端）

```
28.     sendto(sock, msg1, sizeof(msg1), 0,
29.         (struct sockaddr*)&your_adr, sizeof(your_adr));
30.     sendto(sock, msg2, sizeof(msg2), 0,
31.         (struct sockaddr*)&your_adr, sizeof(your_adr));
32.     sendto(sock, msg3, sizeof(msg3), 0,
33.         (struct sockaddr*)&your_adr, sizeof(your_adr));
34.     close(sock);
35.     return 0;
```

TCP和UDP的bind()函数问题

- bind：把IP和端口信息绑定到本机的socket
- bind的意义：告诉操作系统该侦听哪个IP和哪个端口
- bind以后，该IP和该端口的消息都会被收到并处理
- bind的两个功能：
  - 绑定IP
  - 绑定端口

- bind函数对服务器端的意义
- 由于服务器端不是发起连接的一端，而是等待客户端来连接
- 因此服务器需要有一个明显的“目标”被客户端“找到”
  - 用bind来固定服务器端的IP地址和一个端口
  - 客户端可以找到

- bind之于客户端
  - 在TCP里，客户端的端口号是服务器端给分配的
    - 没有必要bind
      - 如果bind反而只限定了某端口和服务器进行通信
- 在UDP里，客户端只需知道服务器端的地址和端口
  - 若完全对等通信，不分C/S，双方都要bind

- bind的简单归纳：
- 无论服务器或客户端
  - 1. 需要在建立连接前就知道端口的话，需要 bind
  - 2. 需要通过指定的端口来通讯的话，需要 bind

客户端的IP地址与端口分配

- 服务器端，通过bind可以绑定本地服务器端socket的IP地址和端口

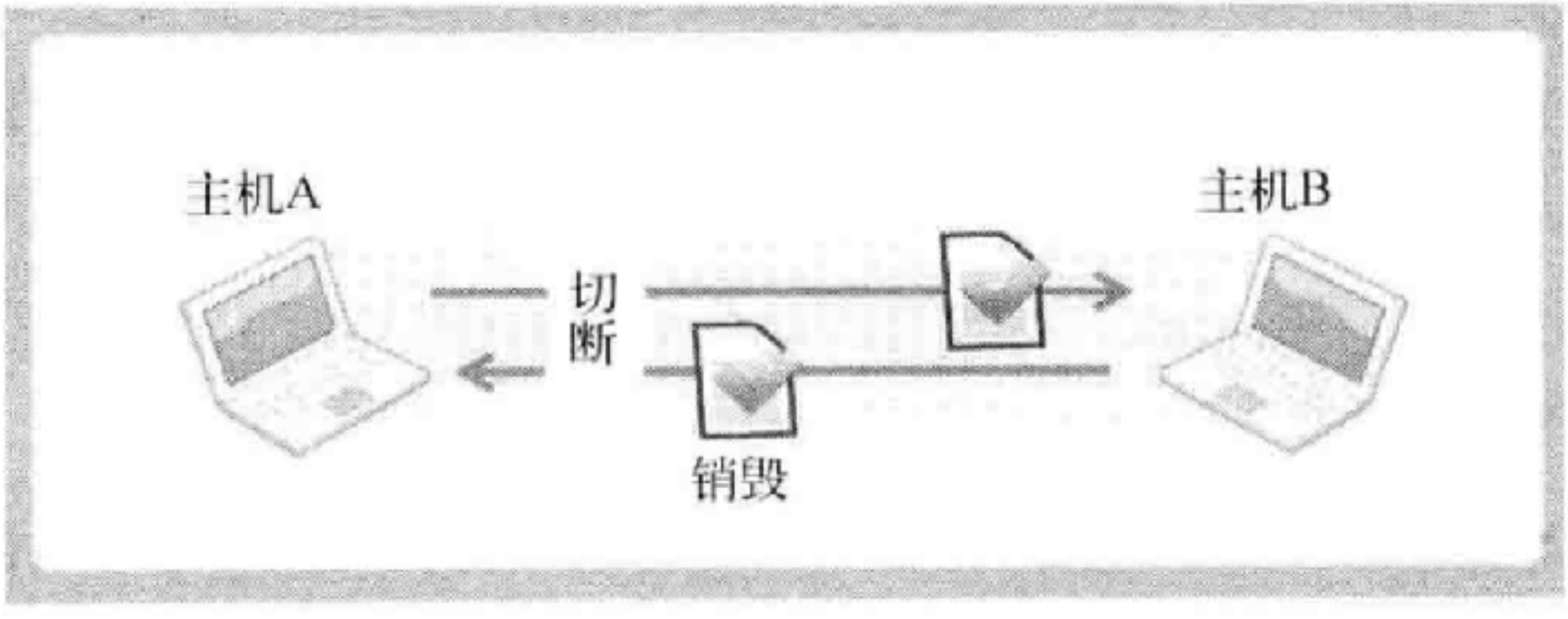
- 客户端如果指定用于socket通信的IP地址和端口?

- TCP中，调用connect函数的时候，如果服务器端运行accept了，则自动给客户端分配一个端口号用于通信

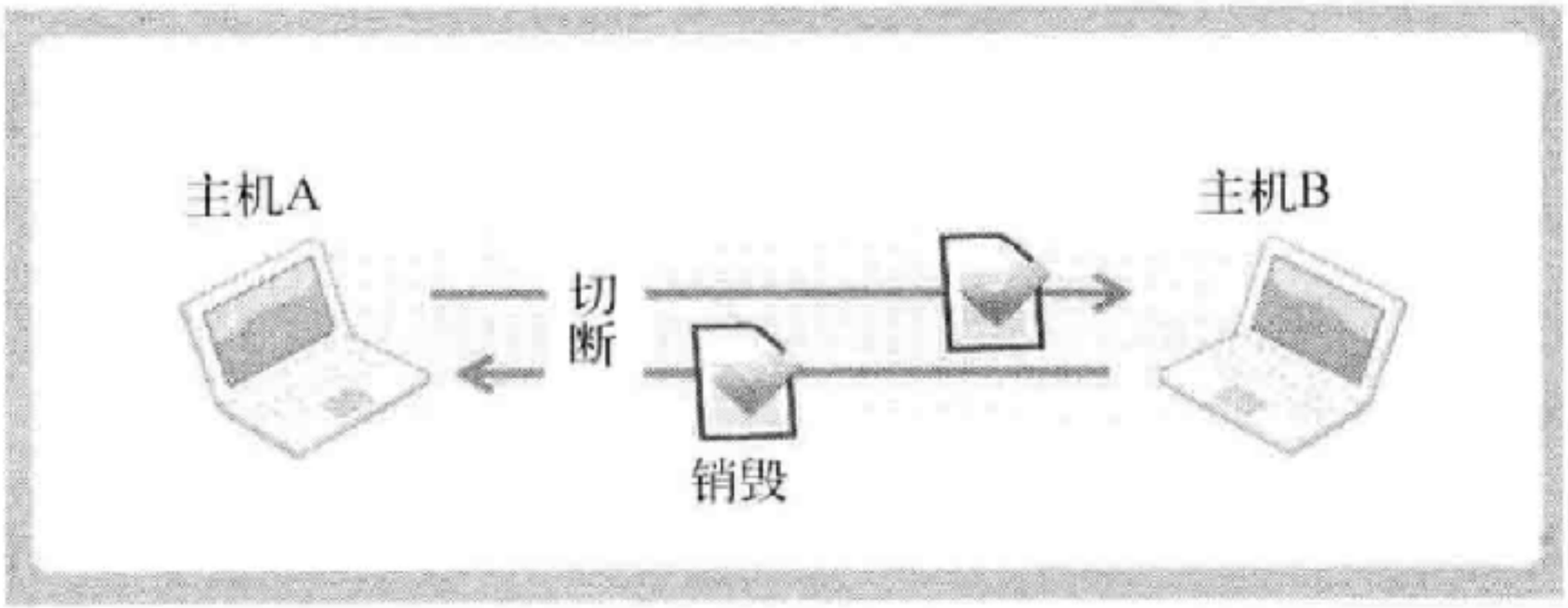
- UDP中，在第一次调用sendto函数时分配

TCP半关闭：优雅地断开套接字

- TCP中断开连接比建立连接过程更加关键
  - 连接过程一般不会出现太大问题
  - 断开连接的过程可能会较复杂，涉及到两个防线数据传输的断开
- 调用close()函数直接断开TCP连接，显得不够优雅?
  - close()直接将所有数据传输都断掉



- 令主机A调用close（或者winsock下的closesocket)
- 之后主机A无法调用任何发送与接收相关的函数
- 主机A不能再向B传送数据，同时主机B可能还有给主机A传数据的需求



- 解决方案
    - 只关闭一部分（Half-close）数据交换中使用的流
    - 可以传输数据，但无法接收，或，
    - 可以接收数据，但无法传输
- 半双工?

shutdown()函数

- int shutdown(int sock, int howto);
- 成功时返回0，失败时返回-1
  - sock需要断开的套接字文件描述符（fd）
  - 断开的方式

- shutdown()
  - 之前提到过，可以断开一边（传出或传入）
  - 通过传递第二个参数howto来实现
    - SHUT_RD：断开输入流
    - SHUT_WR：断开输出流
    - SHUT_RDWR：同时断开I/O流

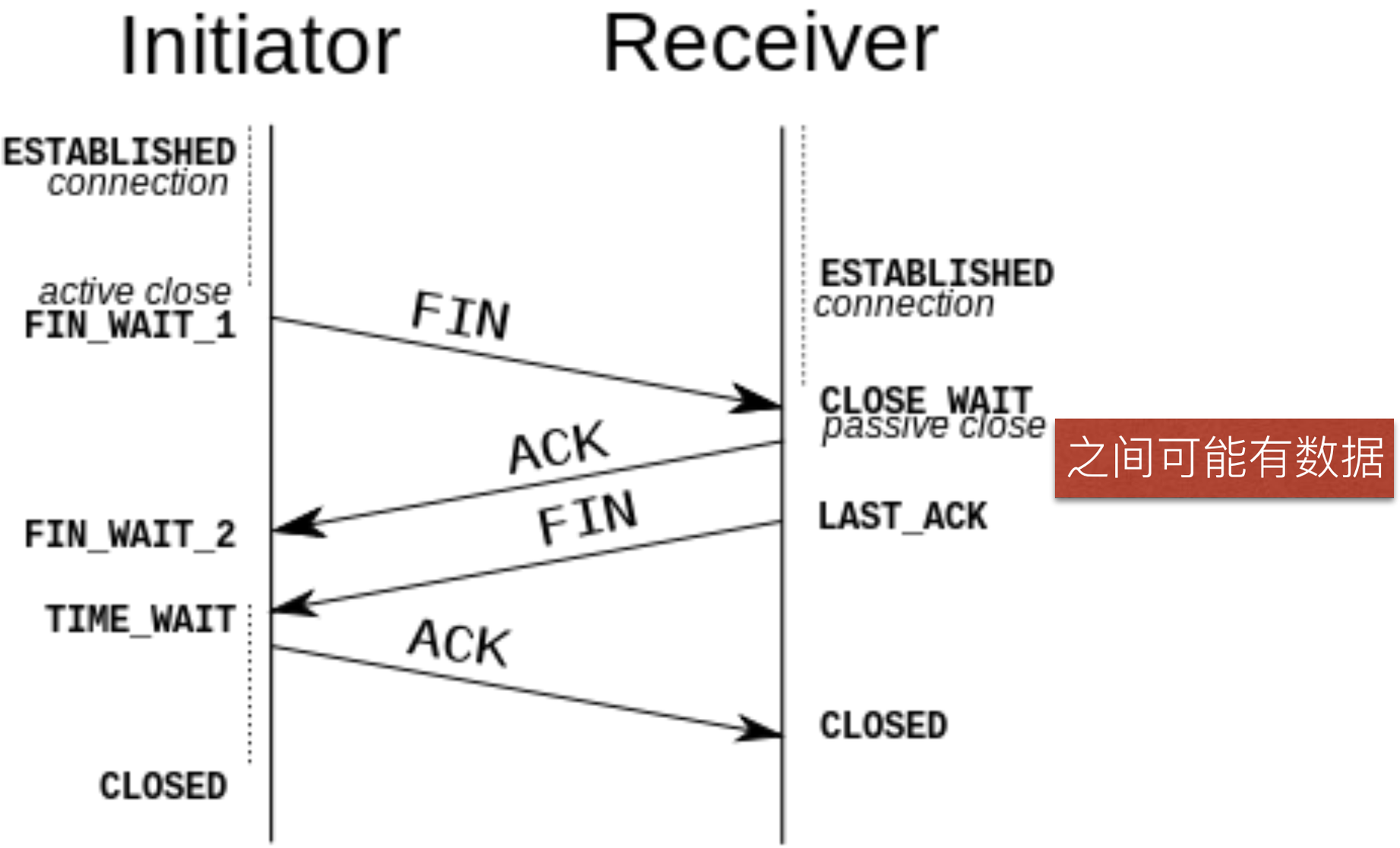
- SHUT_RD
  - 断开输入流，套接字无法接收数据
  - 及时输入缓冲收到数据，也会被抹去
- SHUT_WR
  - 断开输出流，无法传递数据
  - 但是，若输出缓存还有没发完的数据，则继续发送
- SHUT_RDWR
  - 相当于调用两次shutdown，一次参数为SHUT_RD，一次参数为SHUT_WR

为何要“优雅地”断开连接

- 为什么要有半关闭（half-close）
- 因为一方结束传输后另一方可能还需要保持数据传输
  - 可以等待足够长的时间等到完成数据交换即可?
  - 但这里“足够长”如何把握

- 两个考虑：
  - 客户端必须等到服务器端的数据传送完毕才能处理与回复
  - 服务器端需要等待客户端数据发送完成

半关闭和四次挥手


Initiator
Receiver
ESTABLISHED
connection
active close
FIN_WAIT_1
FIN
ESTABLISHED
connection
CLOSE_WAIT
passive close
ACK
之间可能有数据
FIN_WAIT_2
FIN
LAST_ACK
TIME_WAIT
ACK
CLOSED
CLOSED

半关闭可能的应用场景

- 尽快释放socket资源：一端的主机可能运行多线程程序，某些线程需要尽快释放socket以便充分利用网络资源

- Socket的安全：多个线程可能潜在地都能访问某个socket，若线程不希望socket被访问到，在操作完后可单方面关闭socket

- Socket传输的查错：在大量数据传输过程中，如果网络出错导致数据无法发出而传送方不知道的话，可以单方面调用shutdown，该函数只有在数据都发送完成时才能返回成功值0

半关闭代码分析-服务器端

```
8.  #define BUF_SIZE 30
9.  void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int serv_sd, clnt_sd;
14.     FILE * fp;
15.     char buf[BUF_SIZE];
16.     int read_cnt;
17.
18.     struct sockaddr_in serv_adr, clnt_adr;
19.     socklen_t clnt_adr_sz;
20.
21.     if(argc!=2) {
```

Server

```
22.         printf("Usage: %s <port>\n", argv[0]);
23.         exit(1);
24.     }
25.
26.     fp=fopen("file_server.c", "rb");
27.     serv_sd=socket(PF_INET, SOCK_STREAM, 0);
28.
29.     memset(&serv_adr, 0, sizeof(serv_adr));
30.     serv_adr.sin_family=AF_INET;
31.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
32.     serv_adr.sin_port=htons(atoi(argv[1]));
33.
34.     bind(serv_sd, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
35.     listen(serv_sd, 5);
36.
37.     clnt_adr_sz=sizeof(clnt_adr);
38.     clnt_sd=accept(serv_sd, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
39.
```

Server

```
40.     while(1)
41.     {
42.         read_cnt=fread((void*)buf, 1, BUF_SIZE, fp);
43.         if(read_cnt<BUF_SIZE)
44.         {
45.             write(clnt_sd, buf, read_cnt);
46.             break;
47.         }
48.         write(clnt_sd, buf, BUF_SIZE);
49.     }
50.
51.     shutdown(clnt_sd, SHUT_WR);
52.     read(clnt_sd, buf, BUF_SIZE);
53.     printf("Message from client: %s \n", buf);
54.
55.     fclose(fp);
56.     close(clnt_sd); close(serv_sd);
57.     return 0;
```

Server

半关闭代码分析-客户端

```
18.     fp=fopen("receive.dat", "wb");
19.     sd=socket(PF_INET, SOCK_STREAM, 0);
20.
21.     memset(&serv_adr, 0, sizeof(serv_adr));
22.     serv_adr.sin_family=AF_INET;
23.     serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
24.     serv_adr.sin_port=htons(atoi(argv[2]));
25.
26.     connect(sd, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
27.
28.     while((read_cnt=read(sd, buf, BUF_SIZE ))!=0)
29.         fwrite((void*)buf, 1, read_cnt, fp);
30.
31.     puts("Received file data");
32.     write(sd, "Thank you", 10);
33.     fclose(fp);
34.     close(sd);
35.     return 0;
```

Client