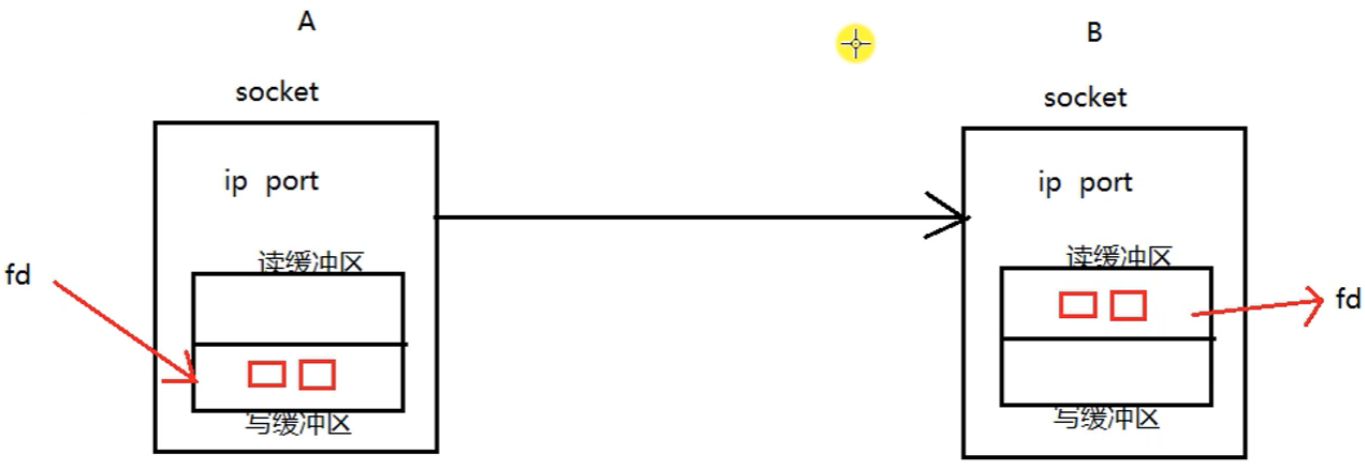
1. socket 介绍

> 所谓socket（套接字），就是对网络中不同主机上的应用进程之间进行双向通信的端点的抽象。一个套接字就是网络上进程通信的一端，提供了应用层进程利用网络协议交换数据的机制。从所处的地位来讲，套接字上联应用进程，下联网络协议栈，是应用程序通过网络协议进行通信的接口，是应用程序与网络协议根进行交互的接口。
>
> socket 可以看成是两个网络应用程序进行通信时，各自通信连接中的端点，这是一个逻辑上的概念。它是网络环境中进程间通信的 API，也是可以被命名和寻址的通信端点，使用中的每一个套接字都有其类型和一个与之相连进程。通信时其中一个网络应用程序将要传输的一段信息写入它所在主机的 socket 中，该 socket 通过与网络接口卡（NIC）相连的传输介质将这段信息送到另外一台主机的 socket 中，使对方能够接收到这段信息。socket 是由 IP 地址和端口结合的，提供向应用层进程传送数据包的机制。
>
> linux下一切皆文件
>
> socket 本身有"插座"的意思，在 Linux 环境下，用于表示进程间网络通信的特殊文件类型。本质为内核借助缓冲区形成的伪文件。既然是文件，那么理所当然的，我们可以使用文件描述符引用套接字。与管道类似的，Linux 系统将其封装成文件的目的是为了统一接口，使得读写套接字和读写文件的操作一致。区别是管道主要应用于本地进程间通信，而套接字多应用于网络进程间数据的传递。




```
// 套接字通信分两部分：
- 服务器端：被动接受连接，一般不会主动发起连接
- 客户端：主动向服务器发起连接

socket是一套通信的接口，Linux 和 Windows 都有，但是有一些细微的差别。
```

2. 字节序

简介

> 现代 CPU 的累加器一次都能装载（至少）4 字节（这里考虑 32 位机），即一个整数。那么这 4 字节在内存中排列的顺序将影响它被累加器装载成的整数的值，这就是字节序问题。在各种计算机体系结构中，对于字节、字等的存储机制有所不同，因而引发了计算机通信领域中一个很重要的问题，即通信双方交流的信息单元（比特、字节、字、双字等等）应该以什么样的顺序进行传送。如

> 果不达成一致的规则，通信双方将无法进行正确的编码/译码从而导致通信失败。
>
> **字节序，顾名思义字节的顺序，就是大于一个字节类型的数据在内存中的存放顺序(一个字节的数据当然就无需谈顺序的问题了)。**
>
> 字节序分为大端字节序（Big-Endian） 和小端字节序（Little-Endian）。大端字节序是指一个整数的最高位字节（23 ~ 31 bit）存储在内存的低地址处，低位字节（0 ~ 7 bit）存储在内存的高地址处；小端字节序则是指整数的高位字节存储在内存的高地址处，而低位字节则存储在内存的低地址处。 大部分计算机采用小端存储

字节序举例

相同

- 小端字节序： 整数的高位存高位，低位存低位

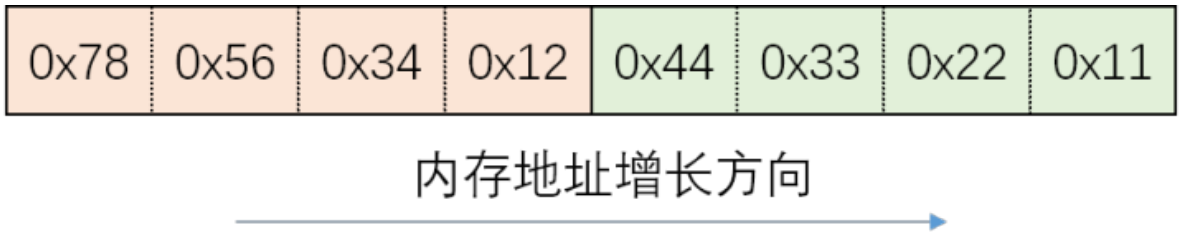
  0x 01 02 03 04 - ff = 255

  内存的方向 ----->

  内存的低位 -----> 内存的高位

  04 03 02 01

  0x 11 22 33 44 12 34 56 78



- 大端字节序 相反

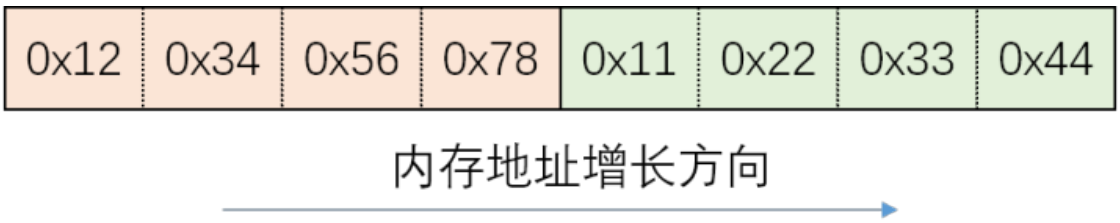
  0x 01 02 03 04 整数的高位存低位，低位存高位

  内存的方向 ----->

  内存的低位 -----> 内存的高位

  01 02 03 04

  0x 12 34 56 78 11 22 33 44



字节序转换函数

当格式化的数据在两台使用不同字节序的主机之间直接传递时，接收端必然错误的解释之。解决问题的方法是：发送端总是把要发送的数据转换成大端字节序数据后再发送，而接收端知道对方传送过来的数据总是采用大端字节序，所以接收端可以根据自身采用的字节序决定是否对接收到的数据进行转换（小端机转换，大端机不转换）。

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，网络字节顺序采用大端排序方式。

BSD Socket提供了封装好的转换接口，方便程序员使用。包括从主机字节序到网络字节序的转换函数：htons、htonl；从网络字节序到主机字节序的转换函数：ntohs、ntohl。

```
h   - host 主机，主机字节序
to  - 转换成什么
n   - network  网络字节序
s   - short  unsigned short
l   - long   unsigned int
```

```
#include <arpa/inet.h>
// 转换端口 short
uint16_t htons(uint16_t hostshort);     // 主机字节序 - 网络字节序
uint16_t ntohs(uint16_t netshort);      // 主机字节序 - 网络字节序

// 转IP long
uint32_t htonl(uint32_t hostlong);      // 主机字节序 - 网络字节序
uint32_t ntohl(uint32_t netlong);       // 主机字节序 - 网络字节序
```

3. socket 地址

```
// socket地址其实是一个结构体，封装端口号和IP等信息。后面的socket相关的api中需要使用到这个socket地址。
// 客户端 ->  服务器（IP, Port）
```

通用 socket 地址

socket 网络编程接口中表示 socket 地址的是结构体 sockaddr，其定义如下：

```
#include <bits/socket.h>
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
};

typedef unsigned short int sa_family_t;
```

sa_family 成员是地址族类型（sa_family_t）的变量。地址族类型通常与协议族类型对应。常见的协议族（protocol family，也称 domain）和对应的地址族如下所示：

协议族	地址族	描述
PF_UNIX	AF_UNIX	UNIX本地域协议族
PF_INET	AF_INET	TCP/IPv4协议族
PF_INET6	AF_INET6	TCP/IPv6协议族

宏 PF_* 和 AF_* 都定义在 bits/socket.h 头文件中，且后者与前者有完全相同的值，所以二者通常混用。

sa_data 成员用于存放 socket 地址值。但是，不同的协议族的地址值具有不同的含义和长度，如下所示：

协议族	地址值含义和长度
PF_UNIX	文件的路径名，长度可达到108字节
PF_INET	16 bit 端口号和 32 bit IPv4 地址，共 6 字节
PF_INET6	16 bit 端口号，32 bit 流标识，128 bit IPv6 地址，32 bit 范围 ID，共 26 字节

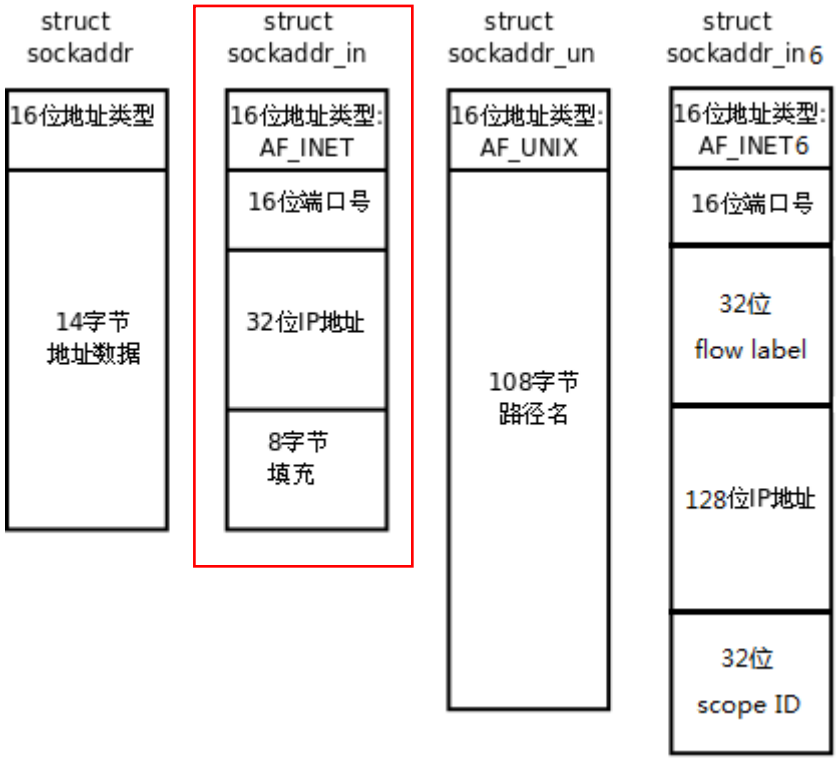
由上表可知，14 字节的 sa_data 根本无法容纳多数协议族的地址值。因此，Linux 定义了下面这个新的通用的 socket 地址结构体，这个结构体不仅提供了足够大的空间用于存放地址值，而且是内存对齐的。

```
#include <bits/socket.h>
struct sockaddr_storage
{
    sa_family_t sa_family;
    unsigned long int __ss_align;
    char __ss_padding[ 128 - sizeof(__ss_align) ];
};

typedef unsigned short int sa_family_t;
```

专用 socket 地址

很多网络编程函数诞生早于 IPv4 协议，那时候都使用的是 struct 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。



UNIX 本地域协议族使用如下专用的 socket 地址结构体：

```
#include <sys/un.h>
struct sockaddr_un
{
    sa_family_t sin_family;
    char sun_path[108];
};
```

TCP/IP 协议族有 sockaddr_in 和 sockaddr_in6 两个专用的 socket 地址结构体，它们分别用于 IPv4 和 IPv6：

```
#include <netinet/in.h>
struct sockaddr_in
{
    sa_family_t sin_family;     /* __SOCKADDR_COMMON(sin_) */  协议族
    in_port_t sin_port;         /* Port number.  */   端口号
    struct in_addr sin_addr;    /* Internet address.  */ IP地址
    /* Pad to size of `struct sockaddr'. */
    unsigned char sin_zero[sizeof (struct sockaddr) - __SOCKADDR_COMMON_SIZE -
               sizeof (in_port_t) - sizeof (struct in_addr)];
};

struct in_addr
{
    in_addr_t s_addr;
};                          int类型的数据，占4个字节

struct sockaddr_in6
{
    sa_family_t sin6_family;
    in_port_t sin6_port;    /* Transport layer port # */
    uint32_t sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr;  /* IPv6 address */
    uint32_t sin6_scope_id; /* IPv6 scope-id */
 };

typedef unsigned short  uint16_t;
typedef unsigned int    uint32_t;
typedef uint16_t in_port_t;
typedef uint32_t in_addr_t;
#define __SOCKADDR_COMMON_SIZE (sizeof (unsigned short int))
```

所有专用 socket 地址（以及 sockaddr_storage）类型的变量在实际使用时都需要转化为通用 socket 地址类型 sockaddr（强制转化即可），因为所有 socket 编程接口使用的地址参数类型都是 sockaddr。

4. IP地址转换（字符串ip-整数，主机、网络字节序的转换）

通常，人们习惯用可读性好的字符串来表示 IP 地址，比如用点分十进制字符串表示 IPv4 地址，以及用十六进制字符串表示 IPv6 地址。但编程中我们需要先把它们转化为整数（二进制数）方能使用。而记录日志时则相反，我们要把整数表示的 IP 地址转化为可读的字符串。下面 3 个函数可用于用点分十进制字符串表示的 IPv4 地址和用网络字节序整数表示的 IPv4 地址之间的转换：

只适用于IPV4

```
#include <arpa/inet.h>
in_addr_t inet_addr(const char *cp);
int inet_aton(const char *cp, struct in_addr *inp);
char *inet_ntoa(struct in_addr in);
```

下面这对更新的函数也能完成前面 3 个函数同样的功能，并且它们同时适用 IPv4 地址和 IPv6 地址：

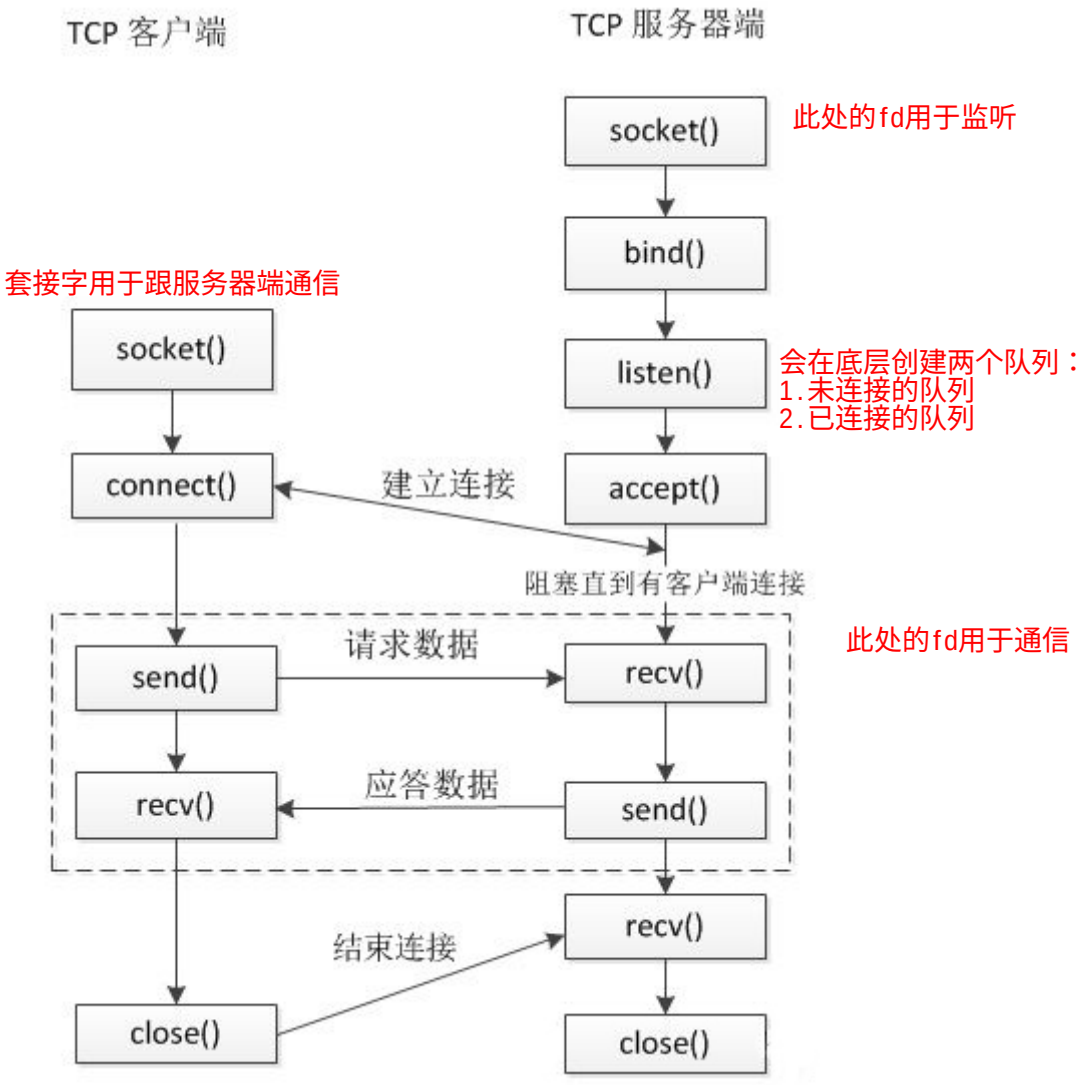
```
#include <arpa/inet.h>
// p:点分十进制的IP字符串，n:表示network，网络字节序的整数
int inet_pton(int af, const char *src, void *dst);
    af:地址族：  AF_INET   AF_INET6
    src:需要转换的点分十进制的IP字符串
    dst:转换后的结果保存在这个里面

// 将网络字节序的整数，转换成点分十进制的IP地址字符串
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
    af:地址族：  AF_INET   AF_INET6
    src：要转换的ip的整数的地址
    dst：转换成IP地址字符串保存的地方
    size：第三个参数的大小（数组的大小）
    返回值：返回转换后的数据的地址（字符串），和 dst 是一样的
```

5. TCP通信流程

```
// TCP 和 UDP  -> 传输层的协议
UDP(User Datagram Protoco):用户数据报协议，面向无连接，可以单播，多播，广播，  面向数据报，不可靠(无连接，不会备份数据)
TCP(Transmission Control Protocol):传输控制协议，面向连接的，可靠的，基于字节流，仅支持单播
```

传输	UDP	TCP
是否创建连接	无连接	面向连接
是否可靠	不可靠	可靠的
连接的对象个数	一对一、一对多、多对一、多对多	支持一对一
传输的方式	面向数据报	面向字节流
首部开销	8个字节	最少20个字节
适用场景	实时应用（视频会议，直播）	可靠性高的应用（文件传输）




```
// TCP 通信的流程
// 服务器端 （被动接受连接的角色）
1. 创建一个用于监听的套接字
    - 监听：监听有客户端的连接
    - 套接字：这个套接字其实就是一个文件描述符 fd
2. 将这个监听文件描述符和本地的IP和端口绑定（IP和端口就是服务器的地址信息）
    - 客户端连接服务器的时候使用的就是这个IP和端口
3. 设置监听，监听的fd开始工作
4. 阻塞等待，当有客户端发起连接，解除阻塞，接受客户端的连接，会得到一个和客户端通信的套接字（fd）
5. 通信
        - 接收数据
        - 发送数据
6. 通信结束，断开连接
```

```
// 客户端（主动连接的角色）
1. 创建一个用于通信的套接字（fd）
2. 连接服务器，需要指定连接的服务器的 IP 和 端口
3. 连接成功了，客户端可以直接和服务器通信

    - 接收数据
    - 发送数据
4. 通信结束，断开连接
```

6. 套接字函数

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>  // 包含了这个头文件，上面两个就可以省略

int socket(int domain, int type, int protocol);
    - 功能：创建一个套接字
    - 参数：
        - domain: 协议族
            AF_INET : ipv4
            AF_INET6 : ipv6
            AF_UNIX, AF_LOCAL : 本地套接字通信（进程间通信）
        - type: 通信过程中使用的协议类型
            SOCK_STREAM : 流式协议
            SOCK_DGRAM  : 报式协议
        - protocol : 具体的一个协议。一般写0
            - SOCK_STREAM : 流式协议默认使用 TCP
            - SOCK_DGRAM  : 报式协议默认使用 UDP
        - 返回值：
            - 成功：返回文件描述符，操作的就是内核缓冲区。
            - 失败：-1

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen); // socket命名
    - 功能：绑定，将fd 和本地的IP + 端口进行绑定
    - 参数：
            - sockfd : 通过socket函数得到的文件描述符
            - addr : 需要绑定的socket地址，这个地址封装了ip和端口号的信息
            - addrlen : 第二个参数结构体占的内存大小

int listen(int sockfd, int backlog);    // /proc/sys/net/core/somaxconn
    - 功能：监听这个socket上的连接
    - 参数：
        - sockfd : 通过socket()函数得到的文件描述符
        - backlog : 规 定 未连接的和已经连接的和的最大值， 一 般 指 定 5 即可

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
    - 功能：接收客户端连接，默认是一个阻塞的函数，阻塞等待客户端连接
    - 参数：
            - sockfd : 用于监听的文件描述符
            - addr : 传出参数，记录了连接成功后客户端的地址信息（ip，port）
            - addrlen : 指定第二个参数的对应的内存大小
    - 返回值：
            - 成功 ：用于通信的文件描述符
            - -1 ： 失败

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

accept是服务器端调用的

connect是客户端调用的

- 功能： 客户端连接服务器
- 参数：
    - sockfd ： 用于通信的文件描述符
    - addr ： 客户端要连接的服务器的地址信息
    - addrlen ： 第二个参数的内存大小
- 返回值：成功 0， 失败 -1

```
ssize_t write(int fd, const void *buf, size_t count);    // 写数据
ssize_t read(int fd, void *buf, size_t count);           // 读数据
```

7. TCP 三次握手

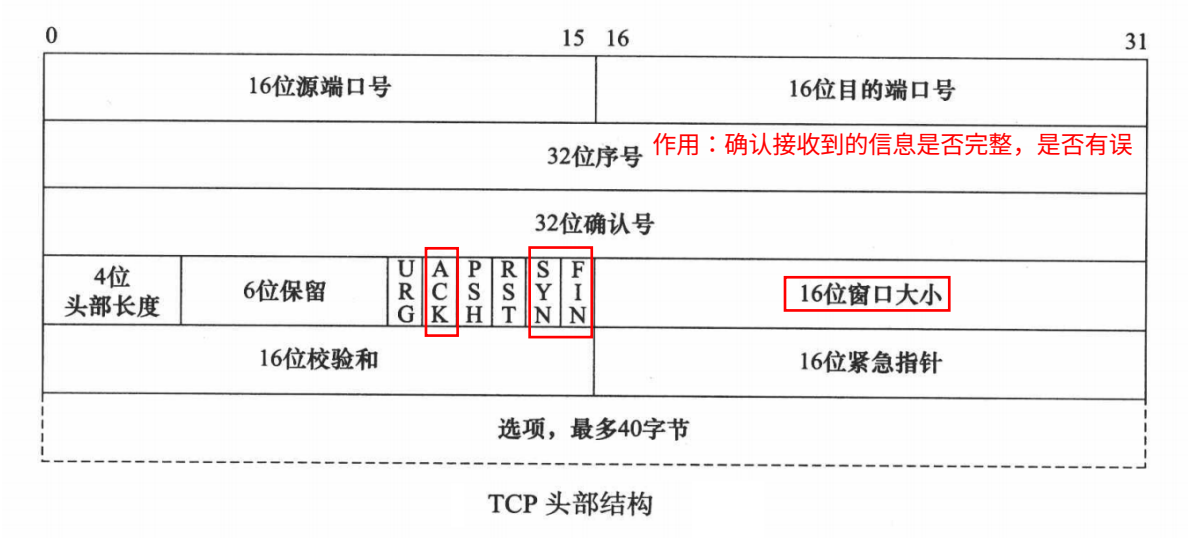
发生在客户端连接的时候，当调用connect()，底层会通过TCP协议进行三次握手

TCP 是一种面向连接的单播协议，在发送数据前，通信双方必须在彼此间建立一条连接。所谓的“连接”，其实是客户端和服务器的内存里保存的一份关于对方的信息，如 IP 地址、端口号等。

TCP 可以看成是一种字节流，它会处理 IP 层或以下的层的丢包、重复以及错误问题。在连接的建立过程中，双方需要交换一些连接的参数。这些参数可以放在 TCP 头部。

TCP 提供了一种可靠、面向连接、字节流、传输层的服务，采用三次握手建立一个连接。采用 四次挥手来关闭一个连接。

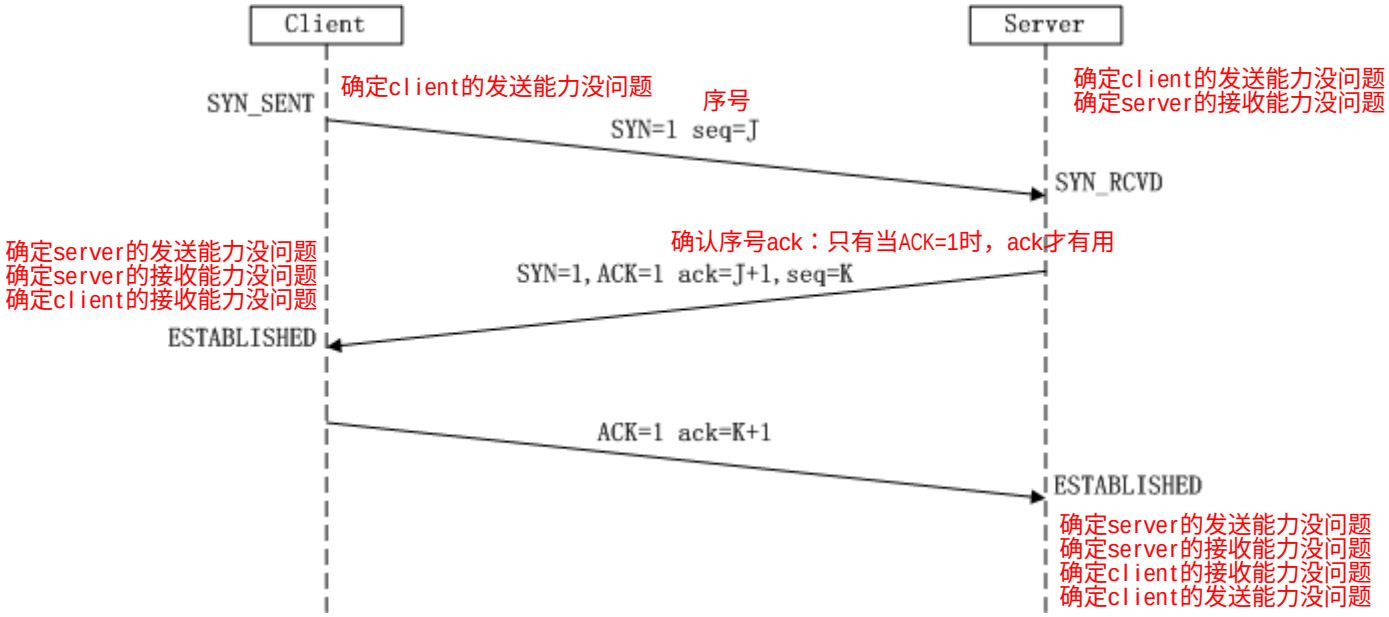
三次握手的目的就是保证通信双方互相建立了连接



TCP 头部结构

- 16 位端口号（port number）：告知主机报文段是来自哪里（源端口）以及传给哪个上层协议或应用程序（目的端口）的。进行 TCP 通信时，客户端通常使用系统自动选择的临时端口号。
- 32 位序号（sequence number）：一次 TCP 通信（从 TCP 连接建立到断开）过程中某一个传输方向上的字节流的每个字节的编号。假设主机 A 和主机 B 进行 TCP 通信，A 发送给 B 的第一个 TCP 报文段中，序号值被系统初始化为某个随机值 ISN（Initial Sequence Number，初始序号值）。那么在该传输方向上（从 A 到 B），后续的 TCP 报文段中序号值将被系统设置成 ISN 加上该报文段所携带数据的第一个字节在整个字节流中的偏移。例如，某个 TCP 报文段传送的数据是字节流中的第 1025 ~ 2048 字节，那么该报文段的序号值就是 ISN + 1025。另外一个传输方向（从 B 到 A）的 TCP 报文段的序号值也具有相同的含义。
- 32 位确认号（acknowledgement number）：用作对另一方发送来的 TCP 报文段的响应。其值是收到的 TCP 报文段的序号值 + 标志位长度（SYN，FIN） + 数据长度 。假设主机 A 和主机 B 进行 TCP 通信，那么 A 发送出的 TCP 报文段不仅携带自己的序号，而且包含对 B 发送来的 TCP 报文段的确认号。反之，B 发送出的 TCP 报文段也同样携带自己的序号和对 A 发送来的报文段的确认序号。
- 4 位头部长度（head length）：标识该 TCP 头部有多少个 32 bit(4 字节)。因为 4 位最大能表示 15，所以 TCP 头部最长是60 字节。

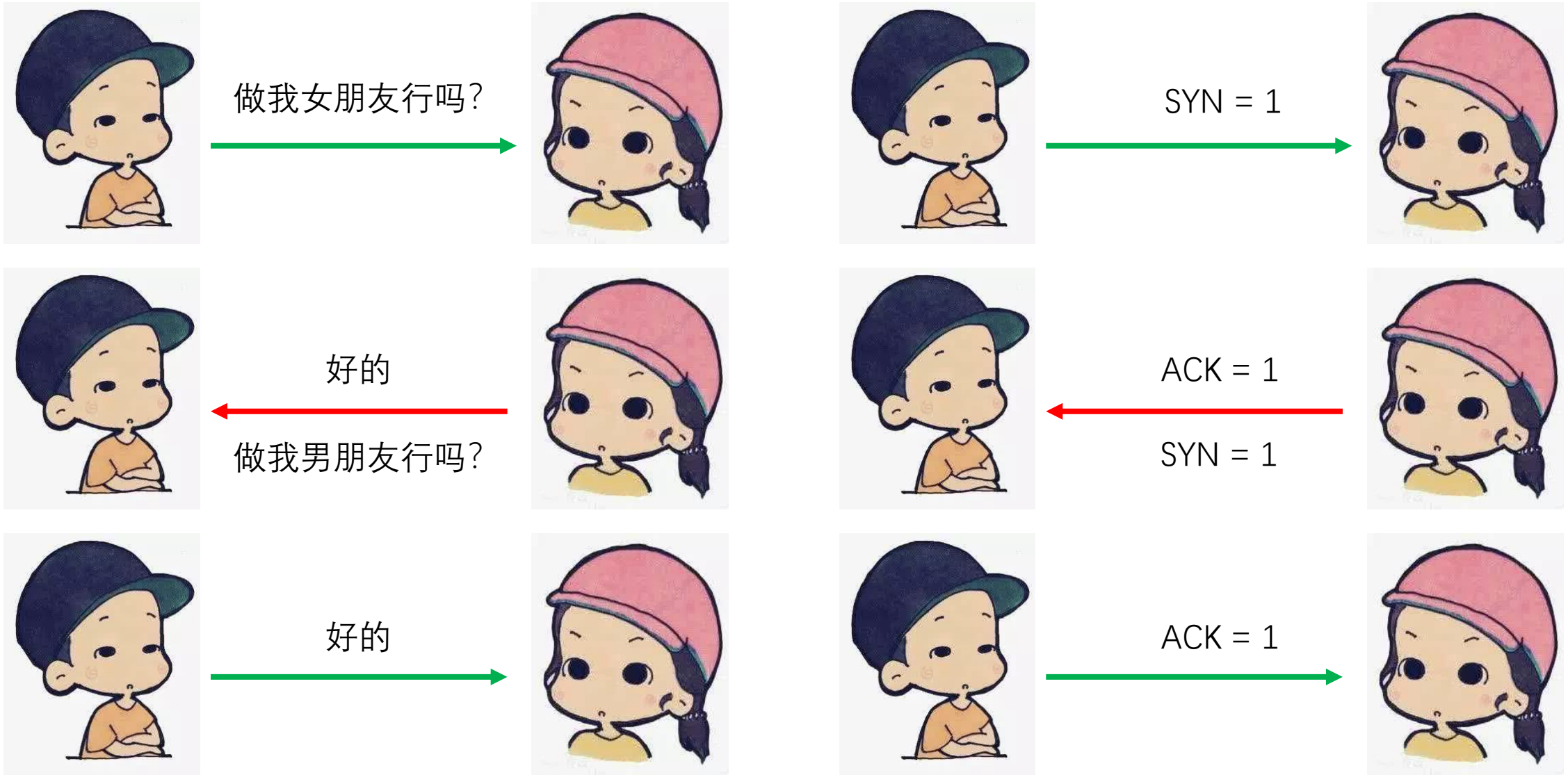
- 6 位标志位包含如下几项：
  - URG 标志，表示紧急指针（urgent pointer）是否有效。
  - ACK 标志，表示确认号是否有效。我们称携带 ACK 标志的 TCP 报文段为确认报文段。
  - PSH 标志，提示接收端应用程序应该立即从 TCP 接收缓冲区中读走数据，为接收后续数据腾出空间（如果应用程序不将接收到的数据读走，它们就会一直停留在 TCP 接收缓冲区中）。
  - RST 标志，表示要求对方重新建立连接。我们称携带 RST 标志的 TCP 报文段为复位报文段。
  - SYN 标志，表示请求建立一个连接。我们称携带 SYN 标志的 TCP 报文段为同步报文段。
  - FIN 标志，表示通知对方本端要关闭连接了。我们称携带 FIN 标志的 TCP 报文段为结束报文段。
  - 16 位窗口大小（window size）：是 TCP 流量控制的一个手段。这里说的窗口，指的是接收通告窗口（Receiver Window，RWND）。它告诉对方本端的 TCP 接收缓冲区还能容纳多少字节的数据，这样对方就可以控制发送数据的速度。
  - 16 位校验和（TCP checksum）：由发送端填充，接收端对 TCP 报文段执行 CRC 算法以校验 TCP 报文段在传输过程中是否损坏。注意，这个校验不仅包括 TCP 头部，也包括数据部分。这也是 TCP 可靠传输的一个重要保障。
  - 16 位紧急指针（urgent pointer）：是一个正的偏移量。它和序号字段的值相加表示最后一个紧急数据的下一个字节的序号。因此，确切地说，这个字段是紧急指针相对当前序号的偏移，不妨称之为紧急偏移。TCP 的紧急指针是发送端向接收端发送紧急数据的方法。



第一次握手：
1.客户端将SYN标志位设为 1
2.生成一个随机的32位序号，这个序号之后可以携带数据(或数据的大小)

第二次握手：
1.服务器端接收客户端的连接 ACK = 1
2.服务器会回发一个确认序号：ack = 客户端的序号 + 数据长度 + SYN/FIN（按一个字节算）
3.服务器端向客户端发送连接请求 SYN置为1
4.服务器端会生成一个随机序号：seq = k

第三次握手：
1.客户端应答服务器的连接 ACK = 1
2.客户端回复收到了服务器的数据：ack = 服务器端的序号 + 数据长度 + SYN/FIN（按一个字节算）


做我女朋友行吗?
SYN = 1
好的
做我男朋友行吗?
ACK = 1
SYN = 1
好的
ACK = 1

客户端

SYN=1 cseq=1000

服务端

只有当接收到SYN或FIN为１时，确认序号才会加１

ACK=1 sack=1001

SYN=1 sseq=2000

ACK=1 cack=2001

发送100个数据，cack没有用到因为没有服务端没有再发送数据（只有当接收到SYN或FIN为１时，确认序号才会加１）

1001(100) cack=2001

sack=1101

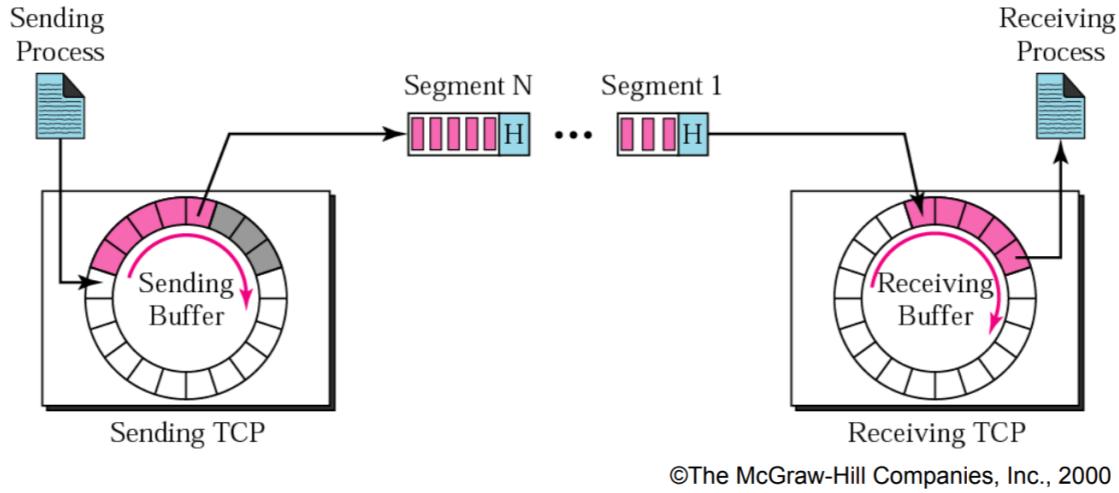
1101(200) cack=2001

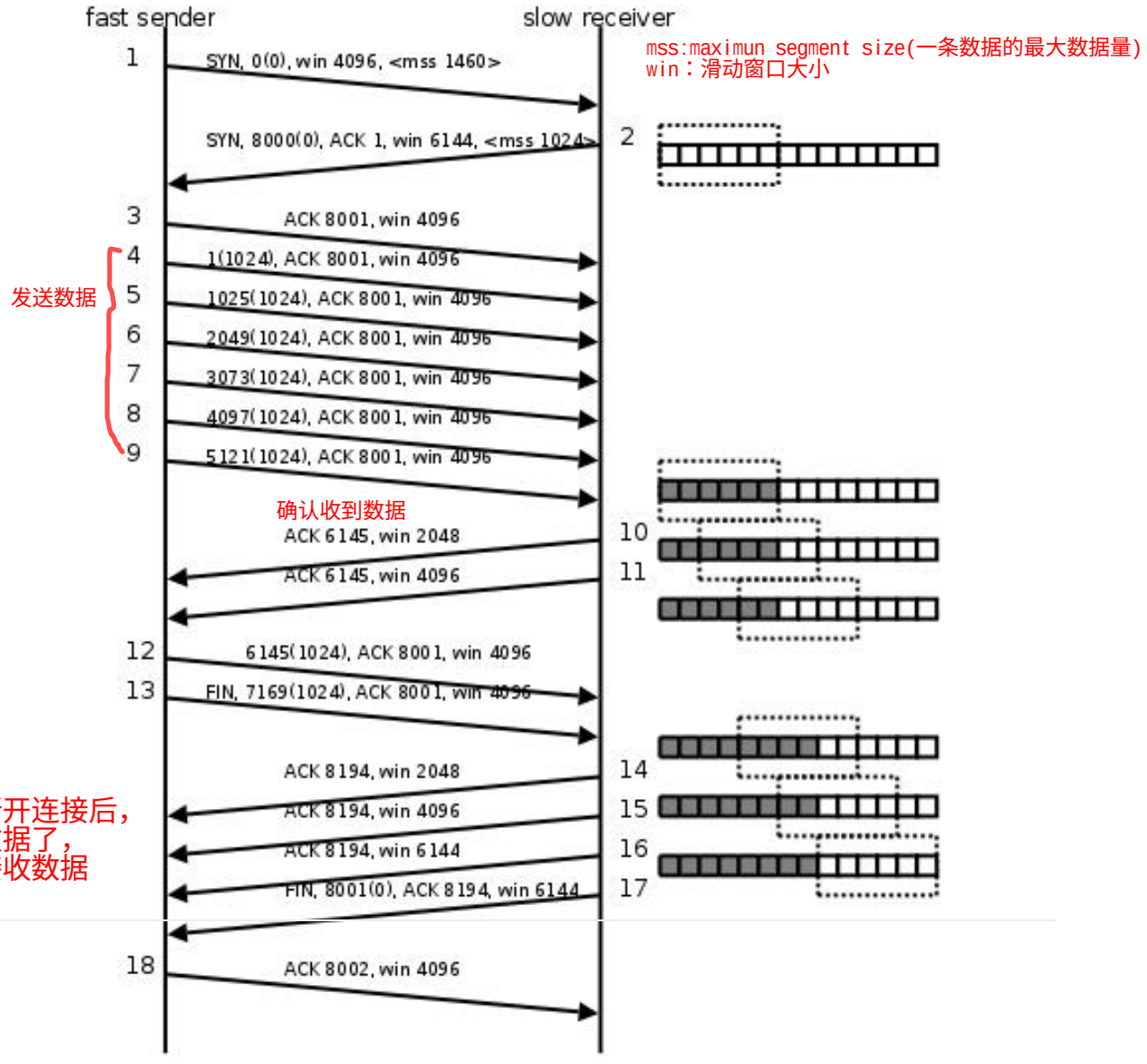
sack=1301

8. TCP 滑动窗口

滑动窗口（Sliding window）是一种流量控制技术。早期的网络通信中，通信双方不会考虑网络的拥挤情况直接发送数据。由于大家不知道网络拥塞状况，同时发送数据，导致中间节点阻塞掉包，谁也发不了数据，所以就有了滑动窗口机制来解决此问题。滑动窗口协议是用来改善吞吐量的一种技术，即容许发送方在接收任何应答之前传送附加的包。接收方告诉发送方在某一时刻能送多少包（称窗口尺寸）。TCP 中采用滑动窗口来进行传输控制，滑动窗口的大小意味着接收方还有多大的缓冲区可以用于接收数据。发送方可以通过滑动窗口的大小来确定应该发送多少字节的数据。当滑动窗口为 0时，发送方一般不能再发送数据报。

滑动窗口是 TCP 中实现诸如 ACK 确认、流量控制、拥塞控制的承载结构。





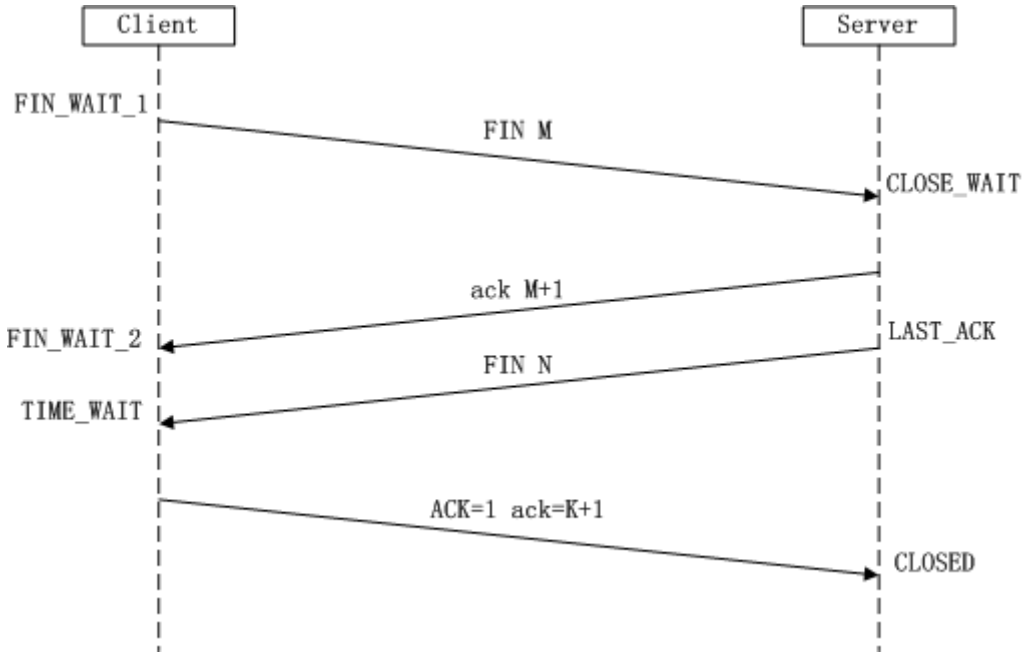
```
# mss: Maximum Segment Size(一条数据的最大的数据量)
# win: 滑动窗口
1. 客户端向服务器发起连接，客户单的滑动窗口是4096，一次发送的最大数据量是1460
2. 服务器接收连接情况，告诉客户端服务器的窗口大小是6144，一次发送的最大数据量是1024
3. 第三次握手
4. 4-9 客户端连续给服务器发送了6k的数据，每次发送1k
5. 第10次，服务器告诉客户端：发送的6k数据以及接收到，存储在缓冲区中，缓冲区数据已经处理了2k,窗口大小是2k
6. 第11次，服务器告诉客户端：发送的6k数据以及接收到，存储在缓冲区中，缓冲区数据已经处理了4k,窗口大小是4k
7. 第12次，客户端给服务器发送了1k的数据
8. 第13次，客户端主动请求和服务器断开连接，并且给服务器发送了1k的数据
9. 第14次，服务器回复ACK 8194, a:同意断开连接的请求 b:告诉客户端已经接受到方才发的2k的数据 c:滑动窗口2k
10.第15、16次，通知客户端滑动窗口的大小
11.第17次，第三次挥手，服务器端给客户端发送FIN,请求断开连接
12.第18次，第四次回收，客户端同意了服务器端的断开请求。
```

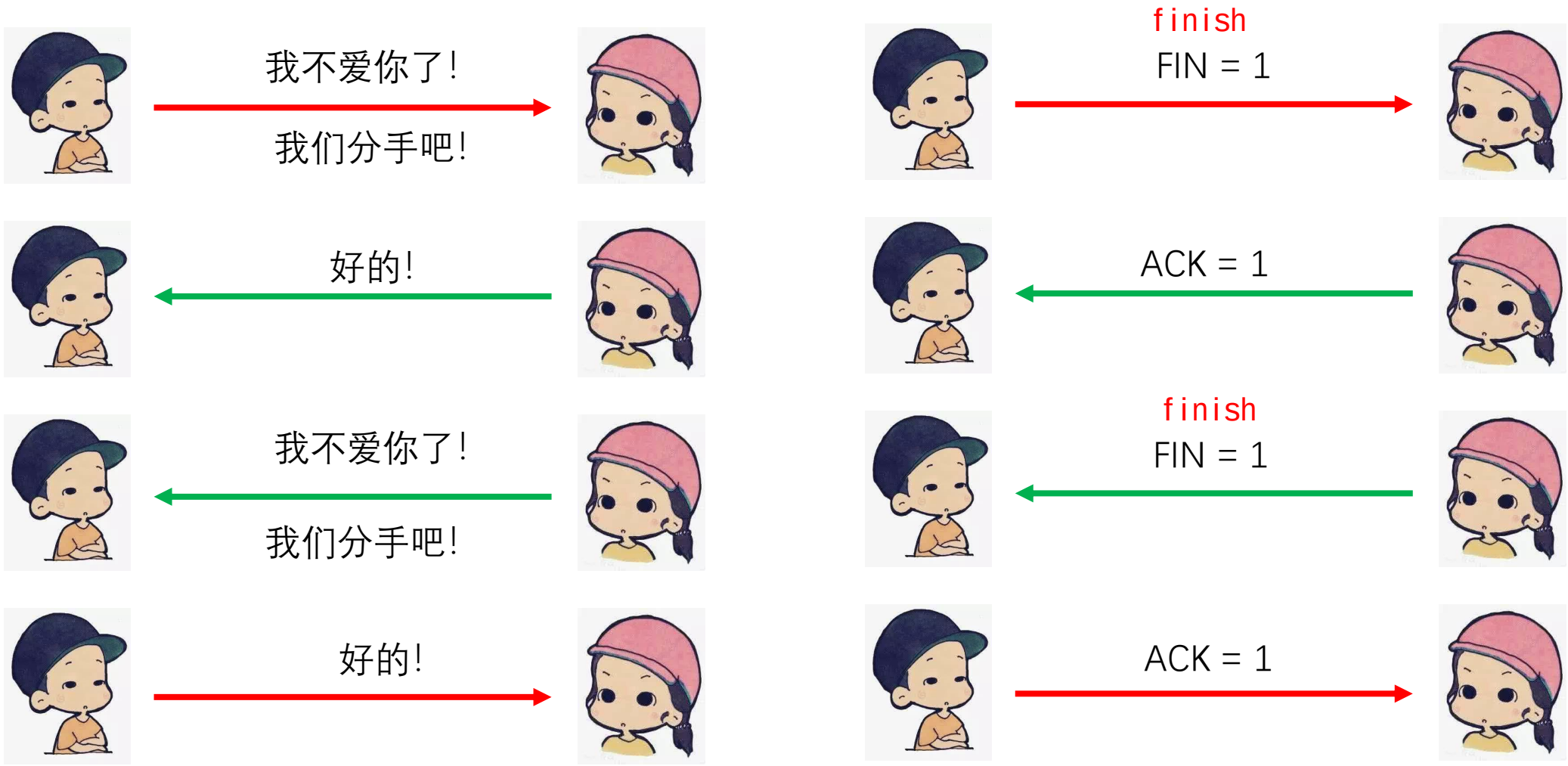
9. TCP 四次挥手

发生在断开连接的时候，在程序中调用了close()会使用TCP协议进行四次挥手。

客户端和服务器端都可以主动发起断开连接，谁先调用close()谁就是发起。

因为TCP连接的时候采用三次握手建立的连接是双向的，所以在四次挥手断开的时候也是双向断开。




我不爱你了!
我们分手吧!
好的!
我不爱你了!
我们分手吧!
好的!
finish
FIN = 1
ACK = 1
finish
FIN = 1
ACK = 1

10. TCP 通信并发

要实现TCP通信服务器处理并发的任务，要使用多线程/进程解决。

思路：

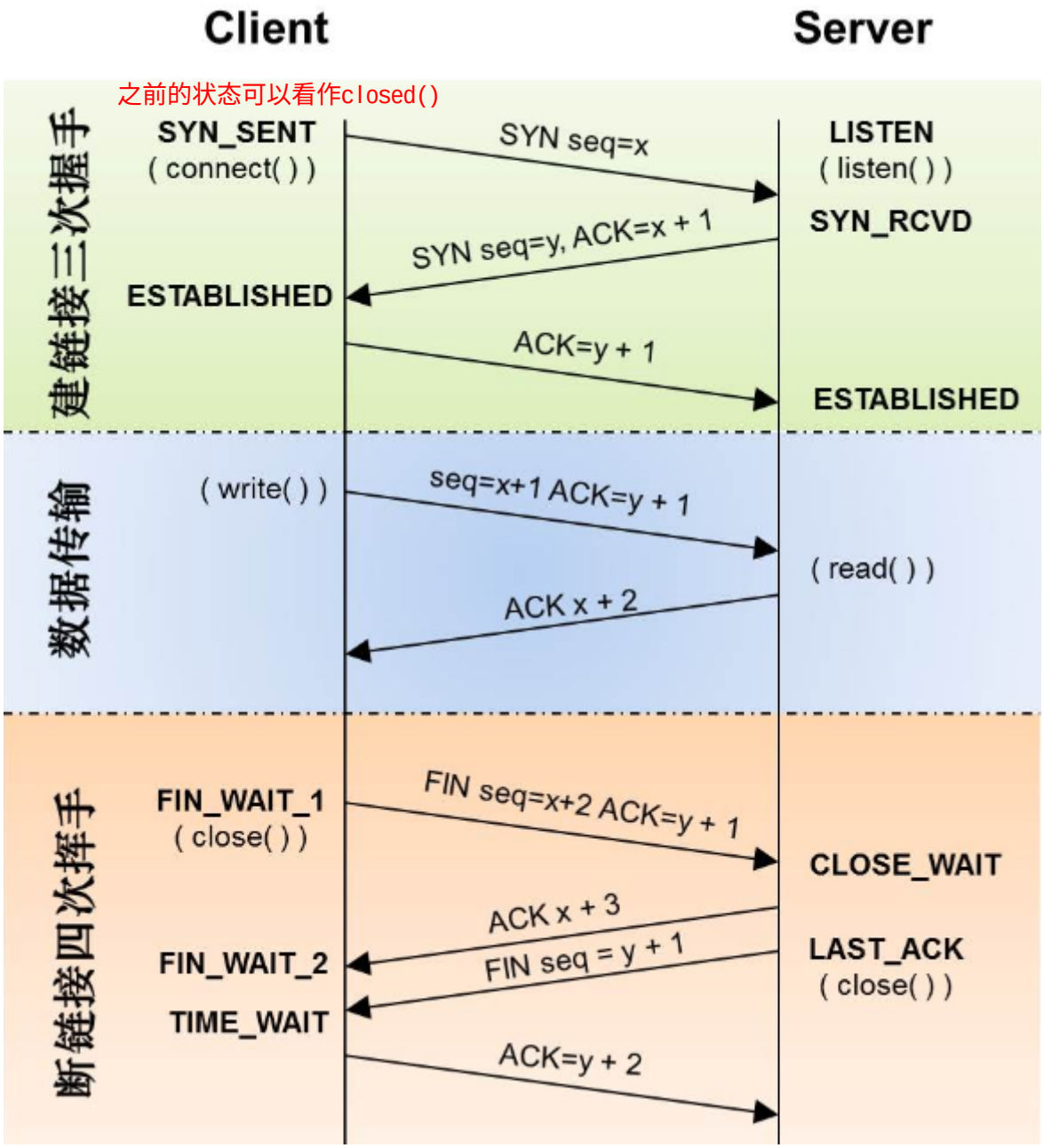
1.一个父进程，多个子进程

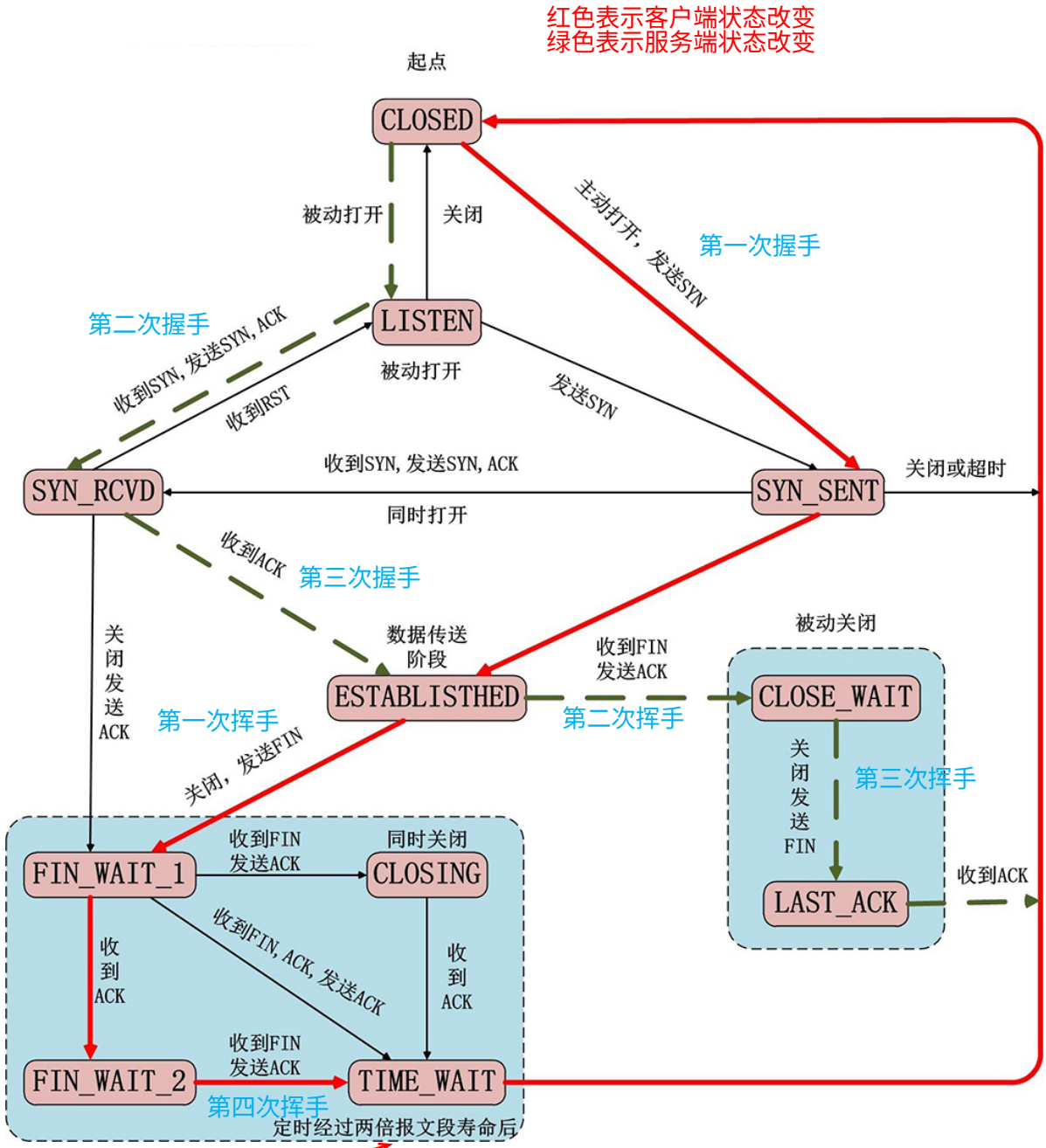
2.父进程负责等待并接受客户端连接

3.子进程：完成通信，接受一个客户端连接，就创建一个子进程用于通信

11. TCP 状态转换

发生在TCP三次握手与四次挥手的过程中
数据传输的过程中状态不会改变





在第四次挥手结束后，被动方可能未收到主动方的ACK信息，其状态可能无法结束。
所以2MSL的存在是为了保证断开连接的被动方接收到主动方的ACK信息

- 2MSL（Maximum Segment Lifetime）

  主动断开连接的一方, 最后进出入一个 TIME_WAIT状态, 这个状态会持续: 2msl

  - msl: 官方建议: 2分钟, 实际是30s

  > 当 TCP 连接主动关闭方接收到被动关闭方发送的 FIN 和最终的 ACK 后，连接的主动关闭方必须处于TIME_WAIT 状态并持续 2MSL 时间。
  >
  > 这样就能够让 TCP 连接的主动关闭方在它发送的 ACK 丢失的情况下重新发送最终的 ACK。
  >
  > 主动关闭方重新发送的最终 ACK 并不是因为被动关闭方重传了 ACK（它们并不消耗序列号，被动关闭方也不会重传），而是因为被动关闭方重传了它的 FIN。事实上，被动关闭方总是重传 FIN 直到它收到一个最终的 ACK。

- 半关闭

  > 当 TCP 链接中 A 向 B 发送 FIN 请求关闭，另一端 B 回应 ACK 之后（A 端进入 FIN_WAIT_2 状态），并没有立即发送 FIN 给 A，A 方处于半连接状态（半开关），此时 A 可以接收 B 发送的数据，但是 A 已经不能再向 B 发送数据。

从程序的角度，可以使用 API 来控制实现半连接状态：

```
#include <sys/socket.h>
int shutdown(int sockfd, int how);
sockfd: 需要关闭的socket的描述符
how:    允许为shutdown操作选择以下几种方式:
    SHUT_RD(0):  关闭sockfd上的读功能，此选项将不允许sockfd进行读操作。
                 该套接字不再接收数据，任何当前在套接字接受缓冲区的数据将被无声的丢弃掉。
    SHUT_WR(1):  关闭sockfd的写功能，此选项将不允许sockfd进行写操作。进程不能在对此套接字发出写操作。
    SHUT_RDWR(2):关闭sockfd的读写功能。相当于调用shutdown两次：首先是以SHUT_RD,然后以SHUT_WR。 相当于close()功能
```

使用 close 中止一个连接，但它只是减少描述符的引用计数，并不直接关闭连接，只有当描述符的引用计数为 0 时才关闭连接。shutdown 不考虑描述符的引用计数，直接关闭描述符。也可选择中止一个方向的连接，只中止读或只中止写。
注意:

1. 如果有多个进程共享一个套接字，close 每被调用一次，计数减 1 ，直到计数为 0 时，也就是所用进程都调用了 close，套接字将被释放。
2. 在多进程中如果一个进程调用了 shutdown(sfd, SHUT_RDWR) 后，其它的进程将无法进行通信。但如果一个进程 close(sfd) 将不会影响到其它进程。

12. 端口复用

> 端口复用最常用的用途是:
>
> - 防止服务器重启时之前绑定的端口还未释放
> - 程序突然退出而系统没有释放端口

```
#include <sys/types.h>
#include <sys/socket.h>
# 设置套接字的属性(不仅仅能设置端口复用)
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

参数：

```
-sockfd: 要操作的文件描述符
-level: 级别 - SOL_SOCKET (端口复用的级别)
-optname：选项的名称
    -SO_REUSEADDR
    -SO_REUSEPORT
-optval: 端口复用的值(整形)
    -1 ： 可以复用
    -0 ： 不可以复用
-optlen: optval的大小
```

端口复用的设置时机是在服务器绑定端口之前

查看网络相关信息的命令：

netstat

参数：

-a 所有的socket
-p 显示正在使用socket的程序的名称
-n 直接使用IP地址，而不通过域名服务器