Socket Programming

王昕兆

2019 年 12 月 12 日

Outline 1

因特网域名

- 产生原因：为了人们更好地记忆 IP 地址

因特网域名

- 产生原因：为了人们更好地记忆 IP 地址
- 是一个层次结构

因特网域名

- 产生原因：为了人们更好地记忆 IP 地址
- 是一个层次结构
- 域名到 IP 地址的映射关系利用分散在世界各地的数据库（Domain Name System）进行维护，可以通过 nslookup+ 域名进行查询

因特网域名

- 产生原因：为了人们更好地记忆 IP 地址
- 是一个层次结构
- 域名到 IP 地址的映射关系利用分散在世界各地的数据库（Domain Name System）进行维护，可以通过 nslookup+ 域名进行查询
- 并非严格的一一对应关系

因特网连接

- 三个性质：点对点、双工、可靠

因特网连接

- 三个性质：点对点、双工、可靠
- 连接的端点：套接字（一种 linux 文件，是一种进程间的通讯机制，目的是让网络文件读写调用像本地文件一样便捷），其地址格式为 (32 位 IP 地址:16 位端口)

因特网连接

- 三个性质：点对点、双工、可靠
- 连接的端点：套接字（一种 linux 文件，是一种进程间的通讯机制，目的是让网络文件读写调用像本地文件一样便捷），其地址格式为 (32 位 IP 地址:16 位端口)
- 一对套接字地址唯一确定了一个连接

因特网连接

- 三个性质：点对点、双工、可靠
- 连接的端点：套接字（一种 linux 文件，是一种进程间的通讯机制，目的是让网络文件读写调用像本地文件一样便捷），其地址格式为 (32 位 IP 地址:16 位端口)
- 一对套接字地址唯一确定了一个连接
- 端口，套接字，进程的关系？主要是有时候会认为套接字前面代表主机，后面是进程。端口是 tcp/ip 协议本身就有的概念，就是为了让一台主机提供多种不同的服务，套接字作为对各种网络读写操作的封装，自然要适应底层的协议

因特网连接

连接一定是两个套接字之间，而非主机，即使对端口并没有特殊要求

- 客户端的端口由内核自动分配（因为没有特殊需求）

因特网连接

连接一定是两个套接字之间，而非主机，即使对端口并没有特殊要求

- 客户端的端口由内核自动分配（因为没有特殊需求）
- 服务器端口通常是知名端口，和提供的服务相对应，且同一个的服务在不同的服务器上有相同的端口号
  1. Port 7: Echo server
  2. Port 23: Talent server
  3. Port 25: Mail server
  4. Port 80: Web server

Outline 2

先区分几个概念

- 套接字（socket）：一种 linux 文件

先区分几个概念

- 套接字（socket）：一种 linux 文件
- 套接字描述符 (socketfd)：和其他文件一样都有的整数标志
- 套接字地址：确定套接字的（IP 地址: 端口号）序列（概念上）

先区分几个概念

- 套接字（socket）：一种 linux 文件
- 套接字描述符 (socketfd)：和其他文件一样都有的整数标志
- 套接字地址：确定套接字的（IP 地址: 端口号）序列（概念上）
- 套接字地址结构体 (socketaddr or SA)：c 中通用的的存储套接字地址的结构体，之后创建套接字的参数都是这个（实现上）它提供了不同协议中的套接字地址结构体要有的共同特征：
  1. 前 2byte 存储标示协议类型的变量
  2. 后面必须恰有 14byte 的空间

  例如我们常用的 IP 套接字地址结构体 (地址按大端法存放)

```
struct sockaddr_in{
    uint16_t sin_family;
    uint16_t sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

先区分几个概念

- 套接字（socket）：一种 linux 文件
- 套接字描述符 (socketfd)：和其他文件一样都有的整数标志
- 套接字地址：确定套接字的（IP 地址: 端口号）序列（概念上）
- 套接字地址结构体 (socketaddr or SA)：c 中通用的的存储套接字地址的结构体，之后创建套接字的参数都是这个（实现上）它提供了不同协议中的套接字地址结构体要有的共同特征：
  1. 前 2byte 存储标示协议类型的变量
  2. 后面必须恰有 14byte 的空间

  例如我们常用的 IP 套接字地址结构体 (地址按大端法存放)

```
struct sockaddr_in{
    uint16_t sin_family;
    uint16_t sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

- addrinfo 结构：是后面为了简化通过域名和服务名建立和服务器连接的过程定义的一种结构

创建套接字

服务器和客户端的套接字的行为是不同的，因此他们的创建函数也不同

1. 用户端：open_clientfd(char * hostname, char *port);
2. 服务器：open_listenfd(char * port);

在正确返回的情况下，open_clientfd 返回的是一个已经和服务器建立好联系的套接字描述符，而 open_listenfd 返回的是一个正在监听某个端口的套接字描述符，真正建立联系需要再调用 accept 函数

echo client

```c
int main(int argc, char **argv)
{
    int clientfd;
    char *host, *port, buf[MAXLINE];
    rio_t rio;

    host = argv[1];
    port = argv[2];

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

哪个地方会出现阻塞?

echoclient.c

echo server

```c
#include "csapp.h"
void echo(int connfd);

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr; /* Enough room for any addr */
    char client_hostname[MAXLINE], client_port[MAXLINE];

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage); /* Important! */
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Getnameinfo((SA *) &clientaddr, clientlen,
                    client_hostname, MAXLINE, client_port, MAXLINE, 0)
        printf("Connected to (%s, %s)\n", client_hostname, client_port)
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

为什么不用listenfd通讯?

echoserver

Outline 3

创建套接字的过程拆分

open_clienfd 和 open_listenfd 实际上还有几个更加具体的步骤

- open_clientfd:
  1. 创建套接字描述符 socket （这时 socket 还没有和某个端口相关联）
  2. 将某个套接字描述符和指定服务器端口建立因特网连接 connect

创建套接字的过程拆分

open_clienfd 和 open_listenfd 实际上还有几个更加具体的步骤

- open_clientfd:
  1. 创建套接字描述符 socket （这时 socket 还没有和某个端口相关联）
  2. 将某个套接字描述符和指定服务器端口建立因特网连接 connect
- open_listenfd:
  1. 创建套接字描述符 socket
  2. 将某个套接字描述符和指定服务器端口联系起来（内核完成）bind
  3. 告诉内核这个 socket 是监听套接字（要在 bind 之后调用）listen

参数说明

```
int socket(int domain, int type, int protocol);
int connet(int clientfd, const struct sockaddr *addr, socklen_t addrlen);
int bind (int sockfd, const struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int listenfd, struct sockaddr *addr, int *addrlen);
```

IP 协议版本无关代码

open_clientfd 和 open_listenfd 都是 ip 协议无关的，但却没有对 ip 协议进行判断，这得益于所有的分辨协议版本并作相应处理的操作都集中在了 getaddrinfo 和 getnameinfo 中

```
int getaddrinfo(const char *host,              /* Hostname or address */
                const char *service,           /* Port or service name */
                const struct addrinfo *hints,/* Input parameters */
                struct addrinfo **result);    /* Output linked list */

void freeaddrinfo(struct addrinfo *result);   /* Free linked list */
const char *gai_strerror(int errcode);        /* Return error msg */
```

```
struct addrinfo {
    int              ai_flags;     /* Hints argument flags */
    int              ai_family;    /* First arg to socket function */
    int              ai_socktype;  /* Second arg to socket function */
    int              ai_protocol;  /* Third arg to socket function  */
    char            *ai_canonname; /* Canonical host name */
    size_t           ai_addrlen;   /* Size of ai_addr struct */
    struct sockaddr *ai_addr;      /* Ptr to socket address structure */
    struct addrinfo *ai_next;      /* Ptr to next item in linked list */
};
```

为什么要用一个链表？域名和 ip 地址并非一一对应 getaddrinfo 的默认行为返回的并不只是可用于连接的套接字，默认情况中，对 host 关联的每个地址，最多返回三个不同类型的套接字地址。

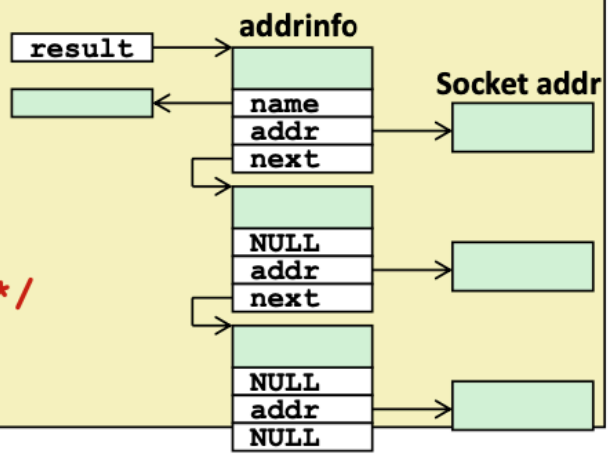
利用 getaddrinfo 写版本无关代码

之后所有套接字地址相关的参数都来自 getaddrinfo 生成的链表，版本相关的操作都封装在 getaddrinfo 函数中

```
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_socktype - SOCK_STREAM;
hints.ai_flags = AI_NUMERCONFIG;
hints.ai_flags |= AI_ADDRCONFIG;
Getaddrinfo(hostname, port, &hints, &listp);
```

open_clientfd 的实现

```
    /* Walk the list for one that we can successfully connect to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((clientfd = socket(p->ai_family, p->ai_socktype,
                               p->ai_protocol)) < 0)
            continue; /* Socket failed, try the next */

        /* Connect to the server */
        if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
            break; /* Success */
        Close(clientfd); /* Connect failed, try another */
    }

    /* Clean up */
    Freeaddrinfo(listp);
    if (!p) /* All connects failed */
        return -1;
    else    /* The last connect succeeded */
        return clientfd;
}
```



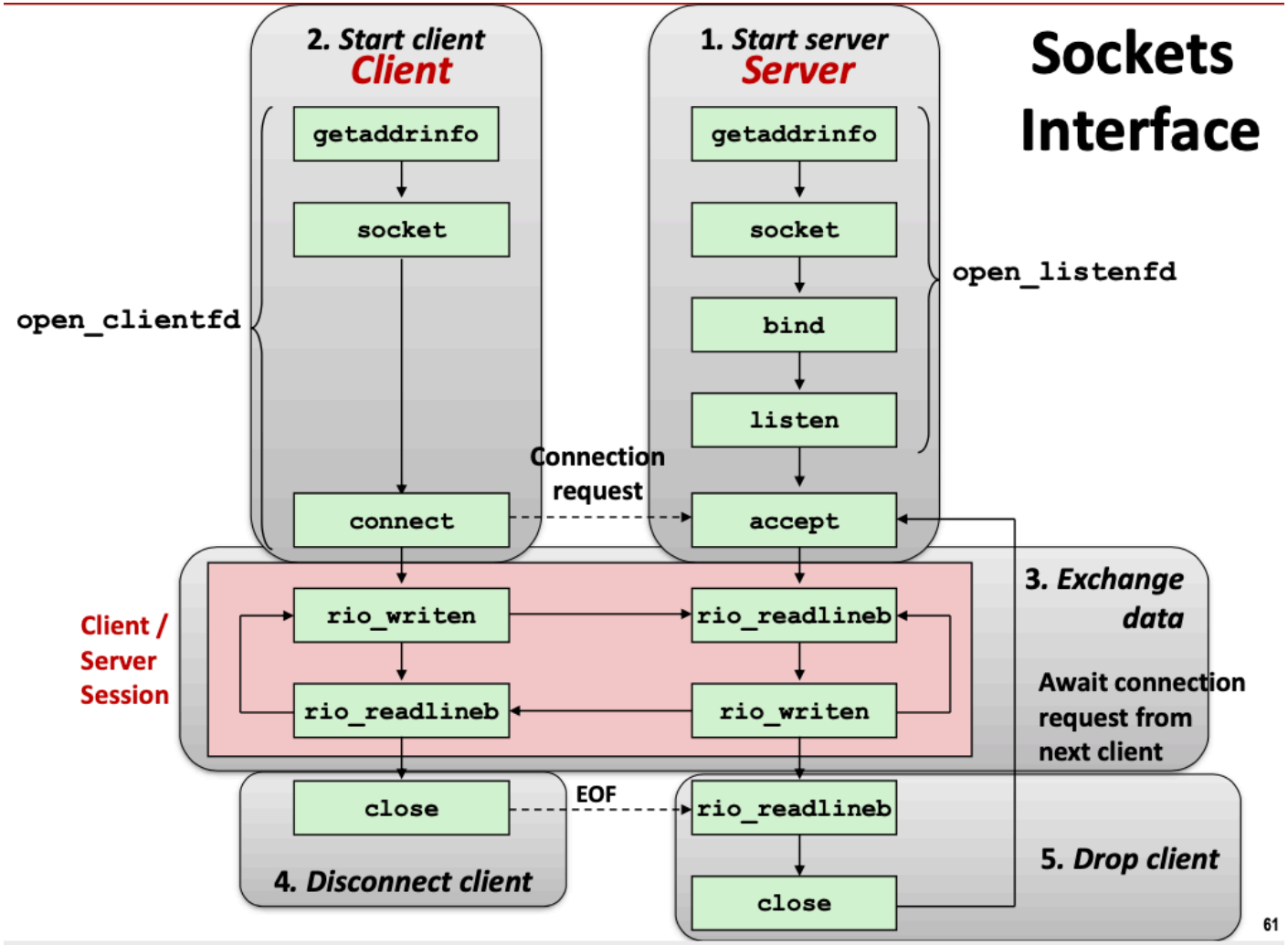
open_listenfd 的实现

```
    /* Walk the list for one that we can bind to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((listenfd = socket(p->ai_family, p->ai_socktype,
                               p->ai_protocol)) < 0)
            continue;  /* Socket failed, try the next */

        /* Eliminates "Address already in use" error from bind */
        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int));

        /* Bind the descriptor to the address */
        if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
            break; /* Success */
        Close(listenfd); /* Bind failed, try the next */
    }
```

csapp.c

```
    /* Clean up */
    Freeaddrinfo(listp);
    if (!p) /* No address worked */
        return -1;

    /* Make it a listening socket ready to accept conn. requests */
    if (listen(listenfd, LISTENQ) < 0) {
        Close(listenfd);
        return -1;
    }
    return listenfd;
}
```

csapp.c


Sockets Interface
2. Start client
Client
1. Start server
Server
getaddrinfo
socket
bind
listen
connect
accept
open_clientfd
open_listenfd
Connection request
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
3. Exchange data
Await connection request from next client
close
EOF
rio_readlineb
close
4. Disconnect client
5. Drop client
61