网络编程套接字

本节重点

> - 认识IP地址, 端口号, 网络字节序等网络编程中的基本概念;
> - 学习socket api的基本用法;
> - 能够实现一个简单的udp客户端/服务器;
> - 能够实现一个简单的tcp客户端/服务器(单连接版本, 多进程版本, 多线程版本);
> - 理解tcp服务器建立连接, 发送数据, 断开连接的流程;

预备知识

理解源IP地址和目的IP地址

唐僧例子1

在IP数据包头部中, 有两个IP地址, 分别叫做源IP地址, 和目的IP地址.

思考: 我们光有IP地址就可以完成通信了嘛? 想象一下发qq消息的例子, 有了IP地址能够把消息发送到对方的机器上, 但是还需要有一个其他的标识来区分出, 这个数据要给哪个程序进行解析.

认识端口号

端口号(port)是传输层协议的内容.

> - 端口号是一个2字节16位的整数;
> - 端口号用来标识一个进程, 告诉操作系统, 当前的这个数据要交给哪一个进程来处理;
> - IP地址 + 端口号能够标识网络上的某一台主机的某一个进程;
> - 一个端口号只能被一个进程占用.

理解 "端口号" 和 "进程ID"

我们之前在学习系统编程的时候, 学习了 pid 表示唯一一个进程; 此处我们的端口号也是唯一表示一个进程. 那么这两者之间是怎样的关系?

10086例子

另外, 一个进程可以绑定多个端口号; 但是一个端口号不能被多个进程绑定;

理解源端口号和目的端口号

唐僧例子2

送快递例子

传输层协议(TCP和UDP)的数据段中有两个端口号, 分别叫做源端口号和目的端口号. 就是在描述 "数据是谁发的, 要发给谁";

认识TCP协议

此处我们先对TCP(Transmission Control Protocol 传输控制协议)有一个直观的认识; 后面我们再详细讨论TCP的一些细节问题.

- 传输层协议
- 有连接
- 可靠传输
- 面向字节流

认识UDP协议

此处我们也是对UDP(User Datagram Protocol 用户数据报协议)有一个直观的认识; 后面再详细讨论.

- 传输层协议
- 无连接
- 不可靠传输
- 面向数据报

网络字节序

我们已经知道,内存中的多字节数据相对于内存地址有大端和小端之分, 磁盘文件中的多字节数据相对于文件中的偏移地址也有大端小端之分, 网络数据流同样有大端小端之分. 那么如何定义网络数据流的地址呢?

- 发送主机通常将发送缓冲区中的数据按内存地址从低到高的顺序发出;
- 接收主机把从网络上接到的字节依次保存在接收缓冲区中,也是按内存地址从低到高的顺序保存;
- 因此,网络数据流的地址应这样规定:先发出的数据是低地址,后发出的数据是高地址.
- 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.
- 不管这台主机是大端机还是小端机, 都会按照这个TCP/IP规定的网络字节序来发送/接收数据;
- 如果当前发送主机是小端, 就需要先将数据转成大端; 否则就忽略, 直接发送即可;

```
将0x1234abcd写入到以0x0000开始的内存中，则结果为

        big-endian    little-endian
0x0000      0x12          0xcd
0x0001      0x23          0xab
0x0002      0xab          0x34
0x0003      0xcd          0x12
```

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);
```

- 这些函数名很好记,h表示host,n表示network,l表示32位长整数,s表示16位短整数。
- 例如htonl表示将32位的长整数从主机字节序转换为网络字节序,例如将IP地址转换后准备发送。
- 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。

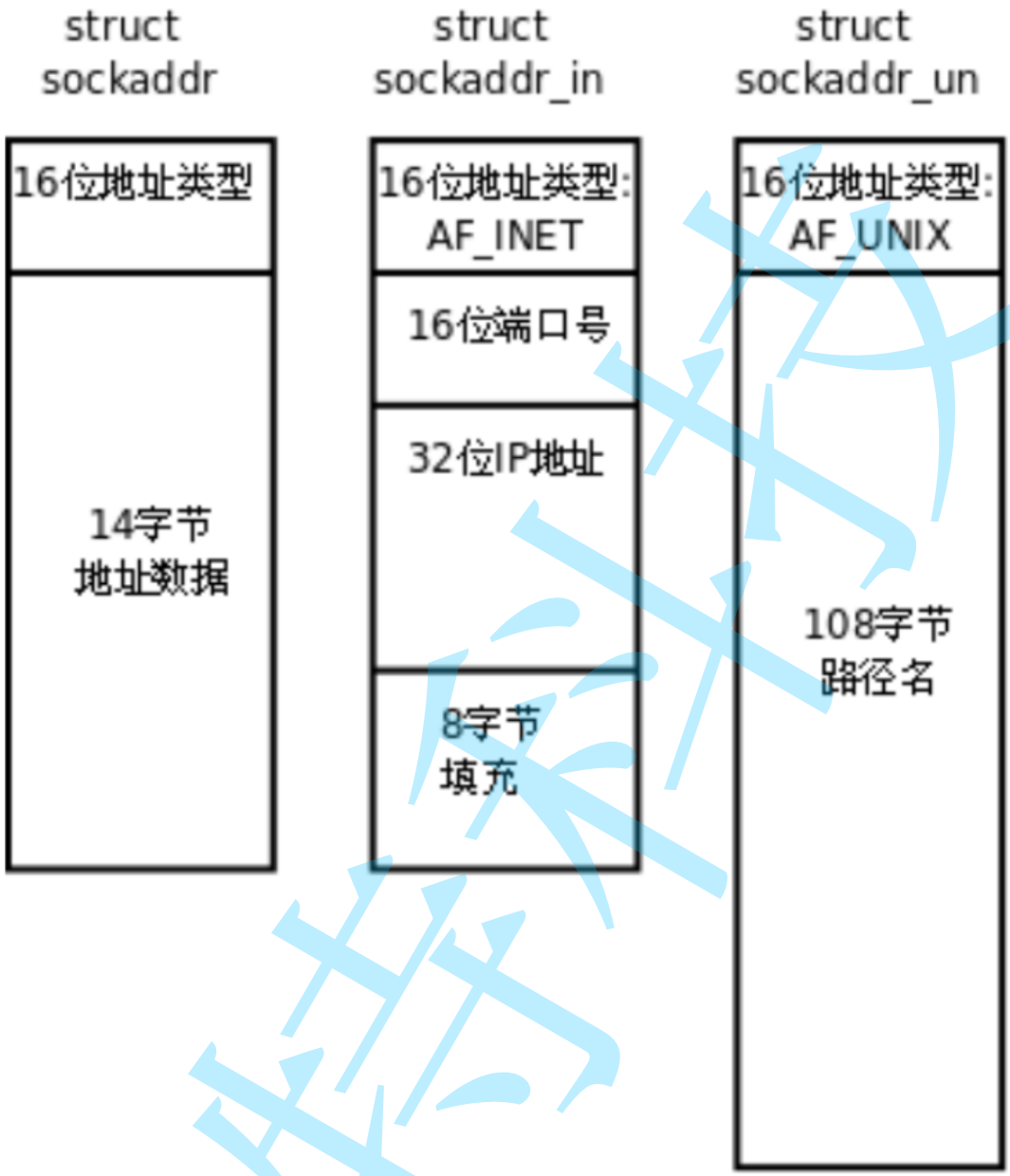
socket编程接口

socket 常见API

```
// 创建 socket 文件描述符 (TCP/UDP, 客户端 + 服务器)
int socket(int domain, int type, int protocol);

// 绑定端口号 (TCP/UDP, 服务器)
int bind(int socket, const struct sockaddr *address,
          socklen_t address_len);

// 开始监听socket (TCP, 服务器)
int listen(int socket, int backlog);

// 接收请求 (TCP, 服务器)
int accept(int socket, struct sockaddr* address,
          socklen_t* address_len);

// 建立连接 (TCP, 客户端)
int connect(int sockfd, const struct sockaddr *addr,
          socklen_t addrlen);
```

sockaddr结构

socket API是一层抽象的网络编程接口,适用于各种底层网络协议,如IPv4、IPv6,以及后面要讲的UNIX Domain Socket. 然而, 各种网络协议的地址格式并不相同.



- IPv4和IPv6的地址格式定义在netinet/in.h中,IPv4地址用sockaddr_in结构体表示,包括16位地址类型, 16位端口号和32位IP地址.
- IPv4、IPv6地址类型分别定义为常数AF_INET、AF_INET6. 这样,只要取得某种sockaddr结构体的首地址,不需要知道具体是哪种类型的sockaddr结构体,就可以根据地址类型字段确定结构体中的内容.
- socket API可以都用struct sockaddr *类型表示, 在使用的时候需要强制转化成sockaddr_in; 这样的好处是程序的通用性, 可以接收IPv4, IPv6, 以及UNIX Domain Socket各种类型的sockaddr结构体指针做为参数;

sockaddr 结构

```
148 struct sockaddr
149   {
150     __SOCKADDR_COMMON (sa_);  /* Common data: address family and length.  */
151     char sa_data[14];   /* Address data.  */
152   };
```

sockaddr_in 结构

```
237 /* Structure describing an Internet socket address.  */
238 struct sockaddr_in
239   {
240     __SOCKADDR_COMMON (sin_);
241     in_port_t sin_port;     /* Port number.  */
242     struct in_addr sin_addr;    /* Internet address.  */
243 
244     /* Pad to size of `struct sockaddr'.  */
245     unsigned char sin_zero[sizeof (struct sockaddr) -
246         __SOCKADDR_COMMON_SIZE -
247         sizeof (in_port_t) -
248         sizeof (struct in_addr)];
249   };
```

虽然socket api的接口是sockaddr, 但是我们真正在基于IPv4编程时, 使用的数据结构是sockaddr_in; 这个结构里主要有三部分信息: 地址类型, 端口号, IP地址.

in_addr结构

```
30 /* Internet address.  */
31 typedef uint32_t in_addr_t;
32 struct in_addr
33   {
34     in_addr_t s_addr;
35   };
36 
```

in_addr用来表示一个IPv4的IP地址. 其实就是一个32位的整数;

简单的UDP网络程序

实现一个简单的英译汉的功能

备注: 代码中会用到 地址转换函数 . 参考接下来的章节.

封装 UdpSocket

udp_socket.hpp

```
#pragma once
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <cassert>
#include <string>

#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
typedef struct sockaddr sockaddr;
typedef struct sockaddr_in sockaddr_in;

class UdpSocket {
public:

  UdpSocket() : fd_(-1) {
```

```
}

bool Socket() {
  fd_ = socket(AF_INET, SOCK_DGRAM, 0);
  if (fd_ < 0) {
    perror("socket");
    return false;
  }
  return true;
}

bool Close() {
  close(fd_);
  return true;
}

bool Bind(const std::string& ip, uint16_t port) {
  sockaddr_in addr;
  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = inet_addr(ip.c_str());
  addr.sin_port = htons(port);
  int ret = bind(fd_, (sockaddr*)&addr, sizeof(addr));
  if (ret < 0) {
    perror("bind");
    return false;
  }
  return true;
}

bool RecvFrom(std::string* buf, std::string* ip = NULL, uint16_t* port = NULL) {
  char tmp[1024 * 10] = {0};
  sockaddr_in peer;
  socklen_t len = sizeof(peer);
  ssize_t read_size = recvfrom(fd_, tmp,
                               sizeof(tmp) - 1, 0, (sockaddr*)&peer, &len);
  if (read_size < 0) {
    perror("recvfrom");
    return false;
  }
  // 将读到的缓冲区内容放到输出参数中
  buf->assign(tmp, read_size);
  if (ip != NULL) {
    *ip = inet_ntoa(peer.sin_addr);
  }
  if (port != NULL) {
    *port = ntohs(peer.sin_port);
  }
  return true;
}

bool SendTo(const std::string& buf, const std::string& ip, uint16_t port) {
  sockaddr_in addr;

  addr.sin_family = AF_INET;
```

```
    addr.sin_addr.s_addr = inet_addr(ip.c_str());
    addr.sin_port = htons(port);
    ssize_t write_size = sendto(fd_, buf.data(), buf.size(), 0, (sockaddr*)&addr, sizeof(addr));
    if (write_size < 0) {
      perror("sendto");
      return false;
    }
    return true;
  }

private:
  int fd_;
};
```

UDP通用服务器

udp_server.hpp

```
#pragma once
#include "udp_socket.hpp"

// C 式写法
// typedef void (*Handler)(const std::string& req, std::string* resp);
// C++ 11 式写法，能够兼容函数指针，仿函数，和 lamda
#include <functional>
typedef std::function<void (const std::string&, std::string* resp)> Handler;

class UdpServer {
public:
  UdpServer() {
    assert(sock_.Socket());
  }

  ~UdpServer() {
    sock_.Close();
  }

  bool Start(const std::string& ip, uint16_t port, Handler handler) {
    // 1. 创建 socket
    // 2. 绑定端口号
    bool ret = sock_.Bind(ip, port);
    if (!ret) {
      return false;
    }
    // 3. 进入事件循环
    for (;;) {
      // 4. 尝试读取请求
      std::string req;
      std::string remote_ip;
      uint16_t remote_port = 0;
      bool ret = sock_.RecvFrom(&req, &remote_ip, &remote_port);

      if (!ret) {
```

```
        continue;
      }
      std::string resp;
      // 5. 根据请求计算响应
      handler(req, &resp);
      // 6. 返回响应给客户端
      sock_.SendTo(resp, remote_ip, remote_port);
      printf("[%s:%d] req: %s, resp: %s\n", remote_ip.c_str(), remote_port,
            req.c_str(), resp.c_str());
    }
    sock_.Close();
    return true;
  }

private:
  UdpSocket sock_;
};
```

实现英译汉服务器

以上代码是对 udp 服务器进行通用接口的封装. 基于以上封装, 实现一个查字典的服务器就很容易了.

dict_server.cc

```
#include "udp_server.hpp"
#include <unordered_map>
#include <iostream>

std::unordered_map<std::string, std::string> g_dict;

void Translate(const std::string& req, std::string* resp) {
  auto it = g_dict.find(req);
  if (it == g_dict.end()) {
    *resp = "未查到!";
    return;
  }
  *resp = it->second;
}

int main(int argc, char* argv[]) {
  if (argc != 3) {
    printf("Usage ./dict_server [ip] [port]\n");
    return 1;
  }
  // 1. 数据初始化
  g_dict.insert(std::make_pair("hello", "你好"));
  g_dict.insert(std::make_pair("world", "世界"));
  g_dict.insert(std::make_pair("c++", "最好的编程语言"));
  g_dict.insert(std::make_pair("bit", "特别NB"));
  // 2. 启动服务器
  UdpServer server;

  server.Start(argv[1], atoi(argv[2]), Translate);
```

```
  return 0;
}
```

UDP通用客户端

udp_client.hpp

```
#pragma once
#include "udp_socket.hpp"

class UdpClient {
public:
  UdpClient(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {
    assert(sock_.Socket());
  }

  ~UdpClient() {
    sock_.Close();
  }

  bool RecvFrom(std::string* buf) {
    return sock_.RecvFrom(buf);
  }

  bool SendTo(const std::string& buf) {
    return sock_.SendTo(buf, ip_, port_);
  }
private:
  UdpSocket sock_;
  // 服务器端的 IP 和 端口号
  std::string ip_;
  uint16_t port_;
};
```

实现英译汉客户端

```
#include "udp_client.hpp"
#include <iostream>

int main(int argc, char* argv[]) {
  if (argc != 3) {
    printf("Usage ./dict_client [ip] [port]\n");
    return 1;
  }
  UdpClient client(argv[1], atoi(argv[2]));
  for (;;) {
    std::string word;
    std::cout << "请输入您要查的单词: ";
    std::cin >> word;
    if (!std::cin) {
      std::cout << "Good Bye" << std::endl;
```

```
      break;
    }
    client.SendTo(word);
    std::string result;
    client.RecvFrom(&result);
    std::cout << word << " 意思是 " << result << std::endl;
  }
  return 0;
}
```

地址转换函数

本节只介绍基于IPv4的socket网络编程,sockaddr_in中的成员struct in_addr sin_addr表示32位 的IP 地址
但是我们通常用点分十进制的字符串表示IP 地址,以下函数可以在字符串表示 和in_addr表示之间转换;

字符串转in_addr的函数:

```
#include <arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr);
in_addr_t inet_addr(const char *strptr);
int inet_pton(int family, const char *strptr, void *addrptr);
```

in_addr转字符串的函数:

```
char *inet_ntoa(struct in_addr inaddr);
const char *inet_ntop(int family, const void *addrptr, char *strptr,
size_t len);
```

其中inet_pton和inet_ntop不仅可以转换IPv4的in_addr,还可以转换IPv6的in6_addr,因此函数接口是void *addrptr。

代码示例:

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main() {
  struct sockaddr_in addr;
  inet_aton("127.0.0.1", &addr.sin_addr);
  uint32_t* ptr = (uint32_t*)(&addr.sin_addr);
  printf("addr: %x\n", *ptr);
  printf("addr_str: %s\n", inet_ntoa(addr.sin_addr));
  return 0;
}
```

关于inet_ntoa

inet_ntoa这个函数返回了一个char*, 很显然是这个函数自己在内部为我们申请了一块内存来保存ip的结果. 那么是否需要调用者手动释放呢?

```
The inet_ntoa() function converts the Internet host address in, given in network byte order, to a
string in IPv4 dotted-decimal notation. The string is returned in a statically allocated buffer, which
subsequent calls will overwrite.
```

man手册上说, inet_ntoa函数, 是把这个返回结果放到了静态存储区. 这个时候不需要我们手动进行释放.

那么问题来了, 如果我们调用多次这个函数, 会有什么样的效果呢? 参见如下代码:

```c
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main() {
  struct sockaddr_in addr1;
  struct sockaddr_in addr2;
  addr1.sin_addr.s_addr = 0;
  addr2.sin_addr.s_addr = 0xffffffff;
  char* ptr1 = inet_ntoa(addr1.sin_addr);
  char* ptr2 = inet_ntoa(addr2.sin_addr);
  printf("ptr1: %s, ptr2: %s\n", ptr1, ptr2);
  return 0;
}
```

运行结果如下:

```
[tangzhong@tz addr_convert]$ ./a.out
ptr1: 255.255.255.255, ptr2: 255.255.255.255
```

因为inet_ntoa把结果放到自己内部的一个静态存储区, 这样第二次调用时的结果会覆盖掉上一次的结果.

> - 思考: 如果有多个线程调用 inet_ntoa, 是否会出现异常情况呢?
> - 在APUE中, 明确提出inet_ntoa不是线程安全的函数;
> - 但是在centos7上测试, 并没有出现问题, 可能内部的实现加了互斥锁;
> - 同学们课后自己写程序验证一下在自己的机器上inet_ntoa是否会出现多线程的问题;
> - 在多线程环境下, 推荐使用inet_ntop, 这个函数由调用者提供一个缓冲区保存结果, 可以规避线程安全问题;

多线程调用inet_ntoa代码示例如下(同学们课后自己测试):

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>

void* Func1(void* p) {
  struct sockaddr_in* addr = (struct sockaddr_in*)p;
  while (1) {
    char* ptr = inet_ntoa(addr->sin_addr);
    printf("addr1: %s\n", ptr);
  }
  return NULL;
}

void* Func2(void* p) {
  struct sockaddr_in* addr = (struct sockaddr_in*)p;
  while (1) {

    char* ptr = inet_ntoa(addr->sin_addr);
```

```
    printf("addr2: %s\n", ptr);
  }
  return NULL;
}

int main() {
  pthread_t tid1 = 0;
  struct sockaddr_in addr1;
  struct sockaddr_in addr2;
  addr1.sin_addr.s_addr = 0;
  addr2.sin_addr.s_addr = 0xffffffff;
  pthread_create(&tid1, NULL, Func1, &addr1);
  pthread_t tid2 = 0;
  pthread_create(&tid2, NULL, Func2, &addr2);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  return 0;
}
```

简单的TCP网络程序

和刚才UDP类似. 实现一个简单的英译汉的功能

TCP socket API 详解

下面介绍程序中用到的socket API,这些函数都在sys/socket.h中。

socket():

```
NAME
       socket - create an endpoint for communication

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

       int socket(int domain, int type, int protocol);
```

- 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, family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
- protocol参数的介绍从略,指定为0即可。

bind():

```
NAME
       bind - bind a name to a socket

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

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

- 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; 服务器需要调用bind绑定一个固定的网络地址和端口号;
- bind()成功返回0,失败返回-1。
- bind()的作用是将参数sockfd和myaddr绑定在一起, 使sockfd这个用于网络通讯的文件描述符监听myaddr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,myaddr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;

我们的程序中对myaddr参数是这样初始化的:

```
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);
```

1. 将整个结构体清零;
2. 设置地址类型为AF_INET;
3. 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;
4. 端口号为SERV_PORT, 我们定义为9999;

listen():

```
NAME
       listen - listen for connections on a socket

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

       int listen(int sockfd, int backlog);
```

- listen()声明sockfd处于监听状态, 并且最多允许有backlog个客户端处于连接等待状态, 如果接收到更多的连接请求就忽略, 这里设置不会太大(一般是5), 具体细节同学们课后深入研究;
- listen()成功返回0,失败返回-1;

accept():

```
NAME
       accept - accept a connection on a socket

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

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

- 三次握手完成后, 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;
- 如果给addr 参数传NULL,表示不关心客户端的地址;
- 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);

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);
}
```

理解accecpt的返回值: 饭店拉客例子

connect

```
NAME
       connect - initiate a connection on a socket

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

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

- 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;

封装 TCP socket

tcp_socket.hpp

```
#pragma once
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <string>
#include <cassert>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
typedef struct sockaddr sockaddr;
typedef struct sockaddr_in sockaddr_in;

#define CHECK_RET(exp) if (!(exp)) {\
  return false;\

}
```

```cpp
class TcpSocket {
public:
  TcpSocket() : fd_(-1) {  }
  TcpSocket(int fd) : fd_(fd) {  }

  bool Socket() {
    fd_ = socket(AF_INET, SOCK_STREAM, 0);
    if (fd_ < 0) {
      perror("socket");
      return false;
    }
    printf("open fd = %d\n", fd_);
    return true;
  }

  bool Close() const {
    close(fd_);
    printf("close fd = %d\n", fd_);
    return true;
  }

  bool Bind(const std::string& ip, uint16_t port) const {
    sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(ip.c_str());
    addr.sin_port = htons(port);
    int ret = bind(fd_, (sockaddr*)&addr, sizeof(addr));
    if (ret < 0) {
      perror("bind");
      return false;
    }
    return true;
  }

  bool Listen(int num) const {
    int ret = listen(fd_, num);
    if (ret < 0) {
      perror("listen");
      return false;
    }
    return true;
  }

  bool Accept(TcpSocket* peer, std::string* ip = NULL, uint16_t* port = NULL) const {
    sockaddr_in peer_addr;
    socklen_t len = sizeof(peer_addr);
    int new_sock = accept(fd_, (sockaddr*)&peer_addr, &len);
    if (new_sock < 0) {
      perror("accept");
      return false;
    }

    printf("accept fd = %d\n", new_sock);
```

```
    peer->fd_ = new_sock;
    if (ip != NULL) {
      *ip = inet_ntoa(peer_addr.sin_addr);
    }
    if (port != NULL) {
      *port = ntohs(peer_addr.sin_port);
    }
    return true;
  }

  bool Recv(std::string* buf) const {
    buf->clear();
    char tmp[1024 * 10] = {0};
    // [注意!] 这里的读并不算很严谨, 因为一次 recv 并不能保证把所有的数据都全部读完
    // 参考 man 手册 MSG_WAITALL 节.
    ssize_t read_size = recv(fd_, tmp, sizeof(tmp), 0);
    if (read_size < 0) {
      perror("recv");
      return false;
    }
    if (read_size == 0) {
      return false;
    }
    buf->assign(tmp, read_size);
    return true;
  }

  bool Send(const std::string& buf) const {
    ssize_t write_size = send(fd_, buf.data(), buf.size(), 0);
    if (write_size < 0) {
      perror("send");
      return false;
    }
    return true;
  }

  bool Connect(const std::string& ip, uint16_t port) const {
    sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(ip.c_str());
    addr.sin_port = htons(port);
    int ret = connect(fd_, (sockaddr*)&addr, sizeof(addr));
    if (ret < 0) {
      perror("connect");
      return false;
    }
    return true;
  }

  int GetFd() const {
    return fd_;
  }
```

```
private:
  int fd_;
};
```

TCP通用服务器

tcp_server.hpp

```
#pragma once
#include <functional>
#include "tcp_socket.hpp"

typedef std::function<void (const std::string& req, std::string* resp)> Handler;

class TcpServer {
public:
  TcpServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {
  }

  bool Start(Handler handler) {
    // 1. 创建 socket;
    CHECK_RET(listen_sock_.Socket());
    // 2. 绑定端口号
    CHECK_RET(listen_sock_.Bind(ip_, port_));
    // 3. 进行监听
    CHECK_RET(listen_sock_.Listen(5));
    // 4. 进入事件循环
    for (;;) {
      // 5. 进行 accept
      TcpSocket new_sock;
      std::string ip;
      uint16_t port = 0;
      if (!listen_sock_.Accept(&new_sock, &ip, &port)) {
        continue;
      }
      printf("[client %s:%d] connect!\n", ip.c_str(), port);
      // 6. 进行循环读写
      for (;;) {
        std::string req;
        // 7. 读取请求. 读取失败则结束循环
        bool ret = new_sock.Recv(&req);
        if (!ret) {
          printf("[client %s:%d] disconnect!\n", ip.c_str(), port);
          // [注意!] 需要关闭 socket
          new_sock.Close();
          break;
        }
        // 8. 计算响应
        std::string resp;
        handler(req, &resp);
        // 9. 写回响应

        new_sock.Send(resp);
```

```
        printf("[%s:%d] req: %s, resp: %s\n", ip.c_str(), port,
               req.c_str(), resp.c_str());
      }
    }
    return true;
  }
private:
  TcpSocket listen_sock_;
  std::string ip_;
  uint64_t port_;
};
```

英译汉服务器

```
#include <unordered_map>
#include "tcp_server.hpp"

std::unordered_map<std::string, std::string> g_dict;

void Translate(const std::string& req, std::string* resp) {
  auto it = g_dict.find(req);
  if (it == g_dict.end()) {
    *resp = "未找到";
    return;
  }
  *resp = it->second;
  return;
}

int main(int argc, char* argv[]) {
  if (argc != 3) {
    printf("Usage ./dict_server [ip] [port]\n");
    return 1;
  }
  // 1. 初始化词典
  g_dict.insert(std::make_pair("hello", "你好"));
  g_dict.insert(std::make_pair("world", "世界"));
  g_dict.insert(std::make_pair("bit", "贼NB"));
  // 2. 启动服务器
  TcpServer server(argv[1], atoi(argv[2]));
  server.Start(Translate);
  return 0;
}
```

TCP通用客户端

tcp_client.hpp

```
#pragma once
#include "tcp_socket.hpp"

class TcpClient {
public:
  TcpClient(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {
    // [注意!!] 需要先创建好 socket
    sock_.Socket();
  }

  ~TcpClient() {
    sock_.Close();
  }

  bool Connect() {
    return sock_.Connect(ip_, port_);
  }

  bool Recv(std::string* buf) {
    return sock_.Recv(buf);
  }

  bool Send(const std::string& buf) {
    return sock_.Send(buf);
  }

private:
  TcpSocket sock_;
  std::string ip_;
  uint16_t port_;
};
```

英译汉客户端

dict_client.cc

```
#include "tcp_client.hpp"
#include <iostream>

int main(int argc, char* argv[]) {
  if (argc != 3) {
    printf("Usage ./dict_client [ip] [port]\n");
    return 1;
  }
  TcpClient client(argv[1], atoi(argv[2]));
  bool ret = client.Connect();
  if (!ret) {
    return 1;
  }
  for (;;) {
```

```
    std::cout << "请输入要查询的单词:" << std::endl;
    std::string word;
    std::cin >> word;
    if (!std::cin) {
      break;
    }
    client.Send(word);
    std::string result;
    client.Recv(&result);
    std::cout << result << std::endl;
  }
  return 0;
}
```

由于客户端不需要固定的端口号,因此不必调用bind(),客户端的端口号由内核自动分配.

注意:

> - 客户端不是不允许调用bind(), 只是没有必要调用bind()固定一个端口号. 否则如果在同一台机器上启动多个客户端, 就会出现端口号被占用导致不能正确建立连接;
> - 服务器也不是必须调用bind(), 但如果服务器不调用bind(), 内核会自动给服务器分配监听端口, 每次启动服务器时端口号都不一样, 客户端要连接服务器就会遇到麻烦;

测试多个连接的情况

再启动一个客户端, 尝试连接服务器, 发现第二个客户端, 不能正确的和服务器进行通信.
分析原因, 是因为我们accecpt了一个请求之后, 就在一直while循环尝试read, 没有继续调用到accecpt, 导致不能接受新的请求.

我们当前的这个TCP, 只能处理一个连接, 这是不科学的.

简单的TCP网络程序(多进程版本)

通过每个请求, 创建子进程的方式来支持多连接;

tcp_process_server.hpp

```
#pragma once
#include <functional>
#include <signal.h>
#include "tcp_socket.hpp"

typedef std::function<void (const std::string& req, std::string* resp)> Handler;

// 多进程版本的 Tcp 服务器
class TcpProcessServer {
public:
  TcpProcessServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {
    // 需要处理子进程
    signal(SIGCHLD, SIG_IGN);
  }

  void ProcessConnect(const TcpSocket& new_sock, const std::string& ip, uint16_t port,
```

```
                      Handler handler) {
  int ret = fork();
  if (ret > 0) {
    // father
    // 父进程不需要做额外的操作，直接返回即可.
    // 思考，这里能否使用 wait 进行进程等待?
    // 如果使用 wait ，会导致父进程不能快速再次调用到 accept，仍然没法处理多个请求
    // [注意!!] 父进程需要关闭 new_sock
    new_sock.Close();
    return;
  } else if (ret == 0) {
    // child
    // 处理具体的连接过程. 每个连接一个子进程
    for (;;) {
      std::string req;
      bool ret = new_sock.Recv(&req);
      if (!ret) {
        // 当前的请求处理完了，可以退出子进程了. 注意，socket 的关闭在析构函数中就完成了
        printf("[client %s:%d] disconnected!\n", ip.c_str(), port);
        exit(0);
      }
      std::string resp;
      handler(req, &resp);
      new_sock.Send(resp);
      printf("[client %s:%d] req: %s, resp: %s\n", ip.c_str(), port,
          req.c_str(), resp.c_str());
    }
  } else {
    perror("fork");
  }
}

bool Start(Handler handler) {
  // 1. 创建 socket;
  CHECK_RET(listen_sock_.Socket());
  // 2. 绑定端口号
  CHECK_RET(listen_sock_.Bind(ip_, port_));
  // 3. 进行监听
  CHECK_RET(listen_sock_.Listen(5));
  // 4. 进入事件循环
  for (;;) {
    // 5. 进行 accept
    TcpSocket new_sock;
    std::string ip;
    uint16_t port = 0;
    if (!listen_sock_.Accept(&new_sock, &ip, &port)) {
      continue;
    }
    printf("[client %s:%d] connect!\n", ip.c_str(), port);
    ProcessConnect(new_sock, ip, port, handler);
  }
  return true;

}
```

```
private:
  TcpSocket listen_sock_;
  std::string ip_;
  uint64_t port_;
};
```

dict_server.cc 稍加修改

将 TcpServer 类改成 TcpProcessServer 类即可

简单的TCP网络程序(多线程版本)

通过每个请求, 创建一个线程的方式来支持多连接;

tcp_thread_server.hpp

```
#pragma once
#include <functional>
#include <pthread.h>
#include "tcp_socket.hpp"

typedef std::function<void (const std::string&, std::string*)> Handler;

struct ThreadArg {
  TcpSocket new_sock;
  std::string ip;
  uint16_t port;
  Handler handler;
};

class TcpThreadServer {
public:
  TcpThreadServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {

  }

  bool Start(Handler handler) {
    // 1. 创建 socket;
    CHECK_RET(listen_sock_.Socket());
    // 2. 绑定端口号
    CHECK_RET(listen_sock_.Bind(ip_, port_));
    // 3. 进行监听
    CHECK_RET(listen_sock_.Listen(5));
    // 4. 进入循环
    for (;;) {
      // 5. 进行 accept
      ThreadArg* arg = new ThreadArg();
      arg->handler = handler;
      bool ret = listen_sock_.Accept(&arg->new_sock, &arg->ip, &arg->port);
      if (!ret) {
        continue;
```

```
      }
      printf("[client %s:%d] connect\n", arg->ip.c_str(), arg->port);
      // 6. 创建新的线程完成具体操作
      pthread_t tid;
      pthread_create(&tid, NULL, ThreadEntry, arg);
      pthread_detach(tid);
    }
    return true;
  }

  // 这里的成员函数为啥非得是 static?
  static void* ThreadEntry(void* arg) {
    // C++ 的四种类型转换都是什么?
    ThreadArg* p = reinterpret_cast<ThreadArg*>(arg);
    ProcessConnect(p);
    // 一定要记得释放内存!!! 也要记得关闭文件描述符
    p->new_sock.Close();
    delete p;
    return NULL;
  }

  // 处理单次连接. 这个函数也得是 static
  static void ProcessConnect(ThreadArg* arg) {
    // 1. 循环进行读写
    for (;;) {
      std::string req;
      // 2. 读取请求
      bool ret = arg->new_sock.Recv(&req);
      if (!ret) {
        printf("[client %s:%d] disconnected!\n", arg->ip.c_str(), arg->port);
        break;
      }
      std::string resp;
      // 3. 根据请求计算响应
      arg->handler(req, &resp);
      // 4. 发送响应
      arg->new_sock.Send(resp);
      printf("[client %s:%d] req: %s, resp: %s\n", arg->ip.c_str(),
             arg->port, req.c_str(), resp.c_str());
    }
  }
private:
  TcpSocket listen_sock_;
  std::string ip_;
  uint16_t port_;
};
```

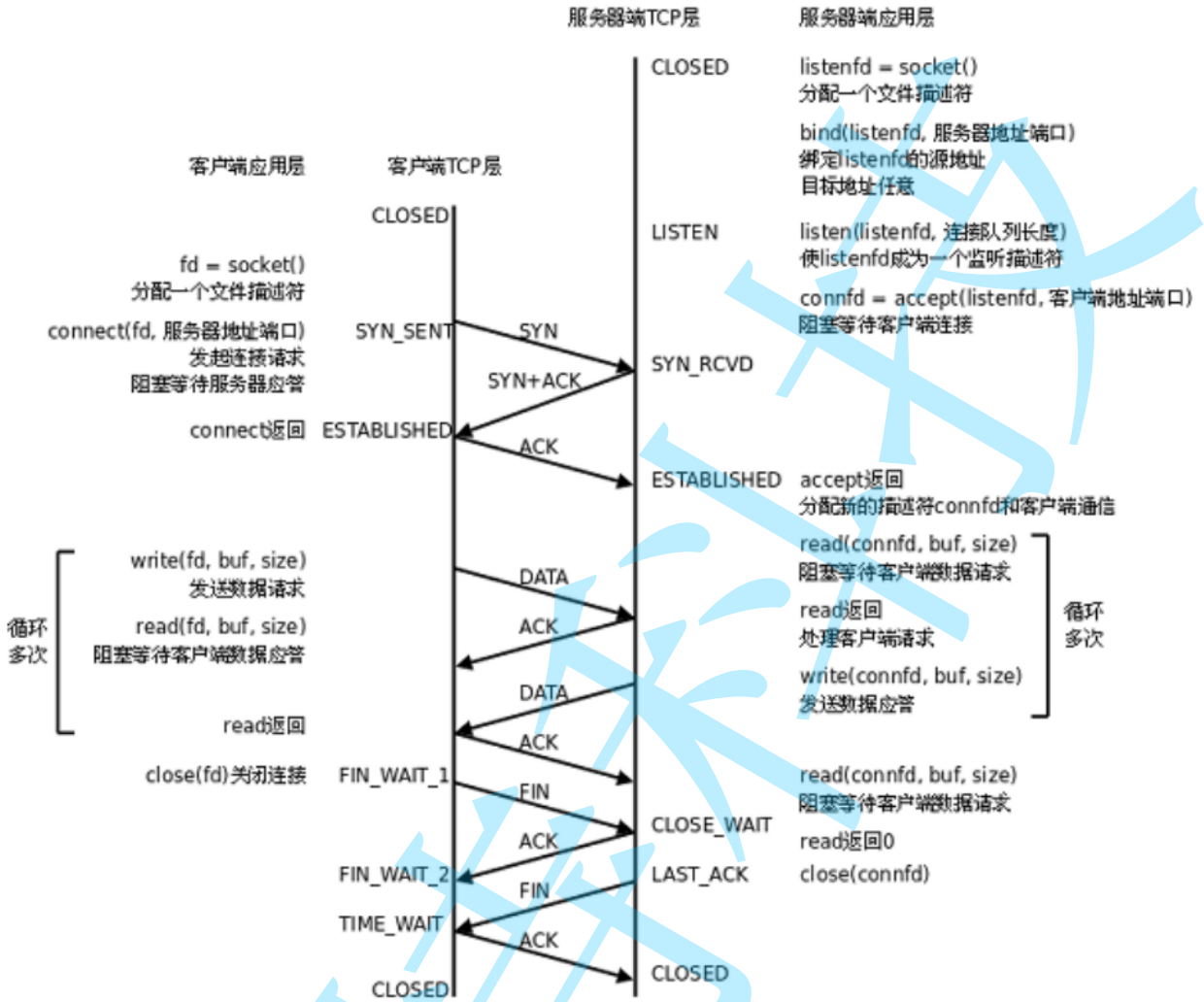
线程池版本的 TCP 服务器

同学们根据之前学过的线程池, 自行修改服务器为线程池版本.

TCP协议通讯流程

谈恋爱例子

下图是基于TCP协议的客户端/服务器程序的一般流程:



服务器初始化:

> - 调用socket, 创建文件描述符;
> - 调用bind, 将当前的文件描述符和ip/port绑定在一起; 如果这个端口已经被其他进程占用了, 就会bind失败;
> - 调用listen, 声明当前这个文件描述符作为一个服务器的文件描述符, 为后面的accept做好准备;
> - 调用accecpt, 并阻塞, 等待客户端连接过来;

建立连接的过程:

> - 调用socket, 创建文件描述符;
> - 调用connect, 向服务器发起连接请求;
> - connect会发出SYN段并阻塞等待服务器应答; (第一次)
> - 服务器收到客户端的SYN, 会应答一个SYN-ACK段表示"同意建立连接"; (第二次)
> - 客户端收到SYN-ACK后会从connect()返回, 同时应答一个ACK段; (第三次)

这个建立连接的过程, 通常称为 **三次握手**;

数据传输的过程

- 建立连接后,TCP协议提供全双工的通信服务; 所谓全双工的意思是, 在同一条连接中, 同一时刻, 通信双方可以同时写数据; 相对的概念叫做半双工, 同一条连接在同一时刻, 只能由一方来写数据;
- 服务器从accept()返回后立刻调 用read(), 读socket就像读管道一样, 如果没有数据到达就阻塞等待;
- 这时客户端调用write()发送请求给服务器, 服务器收到后从read()返回,对客户端的请求进行处理, 在此期间客户端调用read()阻塞等待服务器的应答;
- 服务器调用write()将处理结果发回给客户端, 再次调用read()阻塞等待下一条请求;
- 客户端收到后从read()返回, 发送下一条请求,如此循环下去;

断开连接的过程:

- 如果客户端没有更多的请求了, 就调用close()关闭连接, 客户端会向服务器发送FIN段(第一次);
- 此时服务器收到FIN后, 会回应一个ACK, 同时read会返回0 (第二次);
- read返回之后, 服务器就知道客户端关闭了连接, 也调用close关闭连接, 这个时候服务器会向客户端发送一个FIN; (第三次)
- 客户端收到FIN, 再返回一个ACK给服务器; (第四次)

这个断开连接的过程, 通常称为 **四次挥手**

在学习socket API时要注意应用程序和TCP协议层是如何交互的:

- 应用程序调用某个socket函数时TCP协议层完成什么动作,比如调用connect()会发出SYN段
- 应用程序如何知道TCP协议层的状态变化,比如从某个阻塞的socket函数返回就表明TCP协议收到了某些段,再比如read()返回0就表明收到了FIN段

TCP 和 UDP 对比

- 可靠传输 vs 不可靠传输
- 有连接 vs 无连接
- 字节流 vs 数据报