套接字接口函数使用示例

一、头文件及类型定义

- linux

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

- windows

```
#include <winsock2.h>     // 某些编译器可能需要指定参数：-lws2_32，以链接 ws2_32.lib
#include <windows.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

typedef int socklen_t;     // windows没有socklen_t
```

二、创建 socket 端点

socket(协议族，套接字类型，协议)

- 参数说明：
  - 协议类型：PF_INET（互联网协议族）
  - 套接字类型：SOCK_STREAM（流式，面向连接），SOCK_DGRAM（数据报，无连接）
  - 协议：IPPROTO_TCP（TCP协议），IPPROTO_UDP（UDP协议）
  - 套接字类型和协议要匹配，即TCP协议是面向连接的，UDP是无连接的
- 成功时，函数返回大于0的套接字句柄

In [ ]:

```
// 异常退出
void exitWithErr(const char err[])
{
    printf("Fatal Error: %s\n", err);
    exit(-1);
}

// 初始化
int init(int type)
{
    /* WINDOWS 还需要加入以下代码
    WORD sockVersion = MAKEWORD(2, 2);
    WSADATA wsaData;
    if (WSAStartup(sockVersion, &wsaData) != 0)
        exitWithErr("Start Winsocket Failed");
    */

    int s;
    if (type == 1)  // TCP
        s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    else  // UDP
        s = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);

    if (s < 0) exitWithErr("unable to init socket");
    return s;
}
```

三、设置 socket 选项

setsockopt(套接字句柄，选项类型，选项名，数据指针，数据大小)

- 参数说明：
  - 套接字句柄：初始化 socket 成功返回的套接字句柄
  - 选项类型：
    - SOL_SOCKET: 关于套接字的
    - SOL_IPPROTO_IP: 关于IP协议的
    - SOL_IPPRO_TCP: 关于TCP协议的
  - 选项名：
    - SO_REUSEADDR: 允许重用地址（服务端连接释放后，端口无需等待一段时间即可重复使用）
    - SO_RCVBUF: 设置接收缓冲区大小
    - SO_SNDBUF: 设置发送缓冲区大小
    - SO_RCVTIMEO: 设置接收超时时间

        - SO_SNDTIMEO: 设置发送超时时间
    - 数组指针：指向选项值地址的指针
    - 数据大小：选项值所占的字节大小
- 返回0表示成功，返回-1表示失败

In [ ]:
```
// 设置套接字选项
void setOptions(int s)
{
    int on = 1;
    setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on));

    struct timeval t;
    t.tv_sec = 20;   // 秒
    t.tv_usec = 0;   // 毫秒
    setsockopt(s, SOL_SOCKET, SO_SNDTIMEO, (char *)&t, sizeof(t));
}
```

四、绑定IP地址和端口

bind(套接字句柄，地址指针，地址大小)

- 参数说明：
    - 套接字句柄：初始化 socket 成功返回的套接字句柄
    - 地址指针：指向 socketaddr 结构类型的指针
    - 地址大小：socketadr 结构的字节大小
- 绑定成功返回 0，错误返回-1
- sockaddr 和 sockaddr_in 大小相同，后者更精确定义了结构，方便设置和读取。两者可以相互转换
- sockaddr_in 结构说明：
    - sin_family: 地址族，互联网地址族填写 AF_INET
    - sin_addr: 网络地址
        - s_addr: 32位IP地址，可以调用 inet_addr() 将字符型IP地址转换成32位IP地址
    - sin_port: 端口，16位整数类型，需调用 htons 转换成网络顺序
- IP地址为 0 （“0.0.0.0”）表示绑定本机的全部地址
- inet_addr()将字符串形式的IP地址转换成32位整数形式，inet_ntoa()将32位整数形式转换为字符串形式
- htons()将端口（2个字节的整数）转成网络字节顺序，而ntohs()则将网络字节顺序转成主机字节顺序

In [ ]:
```
// 绑定地址和端口
void bindAddress(int s, const char ip[], int port)
{
    struct sockaddr_in channel;
    memset(&channel, 0, sizeof(channel));
    channel.sin_family = AF_INET;
    channel.sin_addr.s_addr = inet_addr(ip);
    channel.sin_port = htons(port);
    int rc = bind(s, (struct sockaddr *)&channel, sizeof(channel));
    if (rc < 0) exitWithErr("bind failed");
}
```

五、启动监听

listen(套接字句柄，队列大小)

- 服务端调用，适合 TCP
- 参数说明：
    - 套接字句柄：初始化 socket 成功返回的套接字句柄
    - 队列大小：等待处理的连接数（包括正在进行握手的）
- 启动成功返回 0，错误返回-1

In [ ]:
```
// 启动监听
#define QUEUE_SIZE 20
void startListen(int s)
{
    rc = listen(s, QUEUE_SIZE);
    if (rc < 0) exitWithErr("listen failed");
}
```

六、接受客户端连接

accept(套接字句柄，地址指针，地址大小指针)

- 服务端调用。适合TCP
- 阻塞操作，一直等到队列中存在已完成握手的连接才返回与此客户端相关的套接字句柄
- 出现错误，返回小于 0 的值
- 参数：
    - 套接字句柄：初始化 socket 成功返回的套接字句柄
    - 地址指针：返回的socketaddr地址指针，通常设为 NULL

- 地址大小指针：返回的socketadr地址大小指针，通常设为 NULL

七、连接

connect(套接字句柄，地址指针，地址大小)

- 客户端调用。适合TCP
- 连接成功返回 0，失败返回-1，失败原因保存在 errno
- 参数说明：
  - 套接字句柄：初始化 socket 成功返回的套接字句柄
  - 地址指针：指向 socketaddr 结构类型的指针
  - 地址大小：socketadr 结构的字节大小

In [ ]:

```
void startConnect(int s, char ip[], int port)
{
    struct sockaddr_in channel;
    memset(&channel, 0, sizeof(channel));
    channel.sin_family = AF_INET;
    channel.sin_addr.s_addr = inet_addr(ip);
    channel.sin_port = htons(port);
    int rc = connect(s, (struct sockaddr *)&channel, sizeof(channel));
    if (rc < 0) exitWithErr("connect failed");
}
```

八、接收数据

recv(套接字句柄，缓冲区指针，缓冲区大小，标志)

- 适用于 TCP
- 参数说明：
  - 套接字句柄：初始化 socket 成功返回的套接字句柄
  - 缓冲区指针：用于保存接收数据的缓冲区地址
  - 缓冲区大小：缓冲区的字节大小
  - 标志：一般设置为0
- 默认为阻塞状态，有数据时返回读取的字节数量（最多读取缓冲区大小的数量）
- 失败时返回小于或等于0的值，成功时返回读取的字节数
- 也可以使用: read(套接字句柄，缓冲区指针，缓冲区大小)

recvfrom(套接字句柄，缓冲区指针，缓冲区大小，标记，地址指针，地址大小)

- 适用于 UDP
- 参数说明：
  - 套接字句柄：初始化 socket 成功返回的套接字句柄
  - 缓冲区指针：用于保存接收数据的缓冲区地址
  - 缓冲区大小：缓冲区的字节大小
  - 标志：一般设置为0
  - 地址指针：用于保存发送方地址 sockaddr 的指针
  - 地址大小：地址的字节数
- 默认为阻塞状态，有数据时返回读取的字节数量（最多读取缓冲区大小的数量）
- 失败时返回SOCKET_ERROR（小于0），成功时返回读取的字节数

九、发送数据

send(套接字句柄，数据指针，数据大小，标志)

- 适用于 TCP
- 参数说明：
  - 套接字句柄：初始化 socket 成功返回的套接字句柄
  - 数据指针：指向待发送数据的地址
  - 数据大小：数据的字节大小
  - 标志：一般设置为0
- 默认为非阻塞状态，把数据写入运输层发送缓冲区即返回，不等待对方接收确认
- 失败时返回SOCKET_ERROR（小于0），成功时返回发送的字节数
- 也可以使用: write(套接字句柄，缓冲区指针，缓冲区大小)

sendto(套接字句柄，数据指针，数据大小，标志，地址指针，地址大小)

- 适用于 UDP
- 参数说明：
  - 套接字句柄：初始化 socket 成功返回的套接字句柄
  - 数据指针：指向待发送数据的地址
  - 数据大小：数据的字节大小
  - 标志：一般设置为0

    - 地址指针：接收方地址 sockaddr 的指针
    - 地址大小：地址的字节数
- 默认为非阻塞状态，把数据写入运输层发送缓冲区即返回，不等待对方接收确认
- 失败时返回SOCKET_ERROR（小于0），成功时返回发送的字节数

十、服务端 socket 主线程

1）创建 socket 主线程

- 线程的工作函数为 worker
- 对于TCP，worker会调用 waitAndAccept 函数，后者为每一个连接创建一个单独的线程
- 对于UDP，worker循环调用 recvfrom 函数，同时处理多个客户端发来的数据
- 对于创建线程，Windows和Linux实现略有不同

In [ ]:
```
// Windows实现
DWORD WINAPI Thread_Socket(LPVOID lpParam)
{
    int s = (long long)lpParam & 0xffffffff;
    worker(s);    // 调用工作函数（下面定义）
    return 0;
}

void startSocketThread(int s)
{
    DWORD t_id;  // 返回的线程id

    // 传入套接字句柄s
    CreateThead(NULL, 0, Thread_Socket, (LPVOID *)(long long)s, 0, &t_id);
}
```

In [ ]:
```
// Linux实现
void *Thread_Socket(void *arg)
{
    int s = (long)arg & 0xffffffff;
    worker(s);    // 调用工作函数（下面定义）
    return NULL;
}

void startSocketThread(int s)
{
    pthread_t t_id;  // 返回的线程id

    // 传入套接字句柄s
    pthread_create(&t_id, 0, Thread_Socket, (void *)(long)s);
}
```

2）适合TCP的服务端

worker 函数

- 工作在 socket 主线程中，调用 waitAndAccept 函数

In [ ]:
```
// 适合 TCP 的服务端 worker 函数
void worker(int s)
{
    waitAndAccept(s);
}
```

waitAndAccept 函数

- 循环等待客户端连接（调用accept），
- 为每个客户端创建单独的线程，当客户端连接成功时自动创建

In [ ]:
```
// 服务端等待客户端完成握手，然后启动单独的线程处理该客户端的请求
void waitAndAccept(int s)
{
    while (1) {
        int cs = accept(s, NULL, NULL);
        if (cs < 0) exitWithErr("accept failed");

        printf("New client connected. socket=%d\n", cs);

        // 创建处理本次客户端连接的线程(下面定义)
        startClientThread(cs);
    }
}
```

startClientThread 函数

- 为某个客户连接创建单独处理线程
- 创建线程的方式，Windows和Linux实现略有不同

In [ ]:

```
// Windows实现
DWORD WINAPI Thread_Client(LPVOID lpParam)
{
    int cs = (long long)lpParam & 0xffffffff;   // 从参数指针中提取客户端套接字句柄
    process(cs);
    return 0;
}

void startClientThread(int cs)
{
    int t_id;  // 返回的线程id

    // 传入客户端套接字句柄cs
    CreateThead(NULL, 0, Thread_Client, (LPVOID *)(long long)cs, 0, &t_id);

    // 保存客户端连接信息
    saveConnectInfo(cs, t_id);
}
```

In [ ]:

```
// Linux实现
void *Thread_Client(void *arg)
{
    int cs = (long)arg & 0xffffffff;  // 从参数指针中提取客户端套接字句柄
    process(cs);
    return NULL;
}

void startClientThread(int cs)
{
    pthread_t t_id;  // 返回的线程id

    // 传入客户端套接字句柄cs
    pthread_create(&t_id, 0, Thread_Client, (void *)(long)cs);

    // 保存客户端连接信息
    saveConnectInfo(cs, t_id);
}
```

saveConnectInfo 函数

- 保存客户端socket句柄
- 保存客户端的IP地址和端口号
- 保存客户端处理线程id
- 自动关闭同一客户端的其他连接（IP地址和端口号均相同为同一客户端）

In [ ]:

```
// 保存客户端连接信息的数组结构
struct CLIENT {
    int stat;           // 0: 未连接, 1: 已连接
    int sock;           // TCP连接句柄
    in_addr_t ip;       // IP地址
    in_port_t port;     // 端口
    unsigned long thread_id;      // 线程id
} Clients[256];

void saveConnectInfo(int cs, int thread_id)
{
    struct sockaddr_in addr;
    socklen_t len = sizeof(addr);

    getpeername(cs, (struct sockaddr *)&addr, &len);

    in_addr_t ip = addr.sin_addr;
    in_port_t port = addr.sin_port;

    int i, j;
    for (i=0; i<256; i++) {
        if (Clients[i].stat == 0) {
            Clients[i].stat = 1;
            Clients[i].sock = sock;
            Clients[i].ip = ip;
            Clients[i].port = port;
            Clients[i].thread_id = thread_id;
            break;
        }
    }

    // 关闭同一客户端的其他连接
    for (j=0; j<256; j++) {
        if (j != i && Clients[i].ip == ip && Clients[i].port == port) {
            // 关闭 socket 句柄
            close(Clients[i].sock);

            // 关闭线程(Windows换成TerminateThread)
            pthread_cancle(Clients[i].thread_id);

            // 清除记录状态
            memset(&(Clients[i]), 0, sizeof(struct CLIENT));
```

```
        }
    }
}
```

process 函数

- 处理客户请求的主要工作由函数 process 执行

In [ ]:

```
// 处理客户端的请求，并发回响应
void process(int cs)
{
    char buf[4096];

    while (1) {
        ssize_t rc = recv(cs, buf, sizeof(buf), 0);
        if (rc <= 0) break;

        printf("Rx from client (socket=%d): %s\n", cs, buf);

        // 此处仅为演示：如果收到对方打招呼，发回响应
        if (strcmp(buf, "hello") == 0) {
            send(cs, "HELLO", 6, 0);   // 加一个 '\0' 字符
        }
        else if (strcmp(buf, "bye") == 0) {
            send(cs, "BYE", 4, 0);
            break;
        }
        // 其他...
        else send(cs, "UNKOWN", 7, 0);
    }
    close(cs);
    printf("Client (socket=%d) closed.\n", cs);
}
```

3）适合UDP的服务端

worker 函数

- 循环等待客户端请求，发回响应

In [ ]:

```
// 适合 UDP 的服务端工作函数
void worker(int s)
{
    char buf[4096];
    struct sockaddr_in addr;
    socklen_t len = sizeof(addr);   // 必须设置，否则得不到对方地址;

    while (1) {
        ssize_t rc = recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr *)&addr, &len);
        if (rc <= 0) break;

        printf("Rx from client (ip=%s, port=%d): %s\n",
               inet_ntoa(addr.sin_addr),    // inet_ntoa()将32位整数形式IP地址转换为字符串形式
               ntohs(addr.sin_port), buf);  // ntohs()将端口2个字节的整数转成主机顺序

        // 此处仅为演示：如果收到对方请求，发回响应
        if (strcmp(buf, "connect") == 0) {
            // 加一个 '\0' 字符
            sendto(s, "CONNECTED", 10, 0, (struct sockaddr *)&addr, sizeof(addr));

            // 记录客户端连接状态
            saveConnectInfo(addr.sin_addr, addr.sin_port, 1);
        }
        else if (strcmp(buf, "bye") == 0) {
            sendto(s, "BYE", 4, 0, (struct sockaddr *)&addr, sizeof(addr));

            // 记录客户端连接状态
            saveConnectInfo(addr.sin_addr, addr.sin_port, 0);
        }
        // 其他...
        else {
            // 检查客户端连接状态，如果未连接，发回错误信息
            // ...

            // 如果已连接，根据请求发回响应
            if (strcmp(buf, "hello") == 0) {
                // 加一个 '\0' 字符
                sendto(s, "HELLO", 6, 0, (struct sockaddr *)&addr, sizeof(addr));
            }
            else sendto(s, "UNKOWN", 7, 0, (struct sockaddr *)&addr, sizeof(addr));
        }
    }
    close(s);
    printf("Worker quit and socket closed.\n");
}
```

In [ ]:

```
// 保存客户端连接信息的数组结构
struct CLIENT {
```

```
    int stat;           // 0: 未连接, 1: 已连接
    in_addr_t ip;       // IP地址
    in_port_t port;     // 端口
} Clients[256];

void saveConnectInfo(in_addr_t ip, in_port_t port, int stat)
{
    int i, j;
    if (stat == 1) {  // 新连接建立
        for (i=0; i<256; i++) {
            if (Clients[i].stat == 0) {
                Clients[i].stat = 1;
                Clients[i].ip = ip;
                Clients[i].port = port;
                break;
            }
        }
    }
    else i = 256;  // 用于关闭连接

    // 关闭同一客户端的其他连接
    for (j=0; j<256; j++) {
        if (j != i && Clients[i].ip == ip && Clients[i].port == port) {
            // 通知客户端断开连接
            // ...

            // 清除记录状态
            memset(&(Clients[i]), 0, sizeof(struct CLIENT));
        }
    }
}
```

十一、服务端主程序

1）适合 TCP 的服务端主程序

1. 初始化 socket
2. 设置 socket 选项
3. 绑定地址、端口
4. 启动 socket 主线程，循环等待连接建立，然后创建客户端线程
5. 循环接收键盘输入命令，如果命令为quit，则退出程序

In [ ]:

```
// 适合 TCP 的服务端主程序
void server_main()
{
    int s = init(1);
    setOptions(s);

    char ip[] = "0.0.0.0";
    int port = 1234;
    bindAddress(s, ip, port);
    startListen(s);
    printf("Server listen at %s:%d\n", ip, port);

    // 启动 socket 主线程
    startSocketThread(s);

    // 主线程等待用户键盘命令
    char cmd[100];
    while (1) {
        printf("Input quit to exit server: ");
        gets(cmd);
        if (strlen(buff) == 0) continue;

        if (strcmp(cmd, "quit") == 0) break;
    }
}
```

2）适合 UDP 的服务端主程序

1. 初始化 socket
2. 设置 socket 选项
3. 绑定地址、端口
4. 启动 socket 主线程，循环等待数据到达，处理客户端请求，发回响应
5. 循环接收键盘输入命令，如果命令为quit，则退出程序

In [ ]:

```
// 适合 UDP 的服务端主程序
void server_main()
{
    int s = init(0);
    setOptions(s);

    char ip[] = "0.0.0.0";
    int port = 1234;
    bindAddress(s, ip, port);
```

```
    printf("Server bind at %s:%d\n", ip, port);

    // 启动 socket 主线程
    startSocketThread(s);

    // 主线程等待用户键盘命令
    char cmd[100];
    while (1) {
        printf("Input quit to exit server: ");
        gets(cmd);
        if (strlen(buff) == 0) continue;

        if (strcmp(cmd, "quit") == 0) break;
    }
}
```

十二、客户端主程序

1）创建 socket 主线程

- Windows和Linux实现略有不同

In [ ]:

```
// Windows实现
DWORD WINAPI Thread(LPVOID lpParam)
{
    int s = (long long)lpParam & 0xffffffff;
    worker(s);    // 调用工作函数（下面定义）
    return 0;
}

void startSocketThread(int s)
{
    int t_id;  // 返回的线程id

    // 传入套接字句柄s
    CreateThead(NULL, 0, Thread, (LPVOID *)(long long)s, 0, &t_id);
}
```

In [ ]:

```
// Linux实现
void *Thread(void *arg)
{
    int s = (long)arg & 0xffffffff;
    worker(s);    // 调用工作函数（下面定义）
    return NULL;
}

void startSocketThread(int s)
{
    pthread_t t_id;  // 返回的线程id

    // 传入套接字句柄s
    pthread_create(&t_id, 0, Thread, (void *)(long)s);
}
```

2）适合TCP的客户端 worker 函数

- 循环接收客户端发送的数据，给出响应

In [ ]:

```
// 适合 TCP 的客户端工作函数
void worker(int s)
{
    char buff[1024];
    ssize_t rc;

    while (1) {
        memset(buff, 0, sizeof(buff));
        rc = recv(s, buff, sizeof(buff), 0);

        if (rc <= 0) {
            perror("receive failed.");
            close(s);
            break;
        }
        else printf("Rx: %s\n", buff);

        if (strcmp(buff, "BYE")==0) {
            close(s);
            break;
        }
    }
}
```

3）适合UDP的客户端 worker 函数

- 循环接收客户端发送的数据，给出响应

In [ ]:

```
// 适合 UDP 的客户端工作函数
void worker(int s)
{
    char buff[1024];
    ssize_t rc;
    struct sockaddr_in addr;
    socklen_t len= sizeof(addr);   // 必须设置，否则得不到对方地址;

    while (1) {
        memset(buff, 0, sizeof(buff));
        rc = recvfrom(s, buff, sizeof(buff), 0, (struct sockaddr *)&addr, &len);
        if (rc <= 0) {
            perror("receive failed.");
            close(s);
            break;
        }

        // 判断 addr 的地址是否来自服务器，不是则忽略
        // ...

        printf("Rx: %s\n", buff);

        if (strcmp(buff, "BYE")==0) {
            printf("Server disconnected\n");

            // 记录连接状态为已断开
            // ...
        }
        else if (strcmp(buff, "CONNECTED")==0) {
            printf("Server connected\n");

            // 记录连接状态为已连接
            // ...
        }
    }
}
```

4）适合 TCP 的客户端主程序

1. 初始化 socket
2. 设置 socket 选项（可选）
3. 连接服务端
4. 启动socket接收线程，接收服务端返回的数据，打印
5. 循环等待键盘输入命令，如果命令是quit，则退出程序
6. 否则，将命令发送到服务端

In [ ]:

```
// 适用于 TCP 的客户端主程序
void client_main()
{
    int s = init(1);
    // 可选
    setOptions(s);

    char serverIP[20] = "127.0.0.1";
    int serverPort=1234;
    startConnect(s, serverIP, serverPort);
    printf("Server connected\n");

    startSocketThread(s);

    char buff[500];
    while (1) {
        printf("Input: ");
        memset(buff, 0, sizeof(buff));
        gets(buff);
        if (strlen(buff) == 0) continue;

        if (strcmp(buff, "quit")==0) {   // 强制退出
            close(s);
            exit(0);
        }

        ssize_t rc = send(s, buff, strlen(buff)+1, 0);  // 多传1个空字符'\0'
        if (rc < 0) perror("Send failed");
    }
}
```

5）适合 UDP 的客户端主程序

1. 初始化 socket
2. 设置 socket 选项（可选）
3. 绑定本地IP地址和端口
4. 启动 socket 接收线程, 接收服务端返回的数据，打印
5. 循环等待键盘输入命令，如果命令是quit，则退出程序

6. 否则，将命令发送到服务端

In [ ]:

```
// 适用于 UDP 的客户端主程序
void client_main()
{
    int s = init(1);
    // 可选
    setOptions(s);

    char serverIP[] = "127.0.0.1";
    int serverPort = 1234;
    int localPort = 4321;

    bindAddress(s, "0.0.0.0", localPort);

    startSocketThread(s);

    struct sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(serverIP);
    addr.sin_port = htons(serverPort);

    char buff[500];
    while (1) {
        printf("Input: ");
        memset(buff, 0, sizeof(buff));
        gets(buff);
        if (strlen(buff) == 0) continue;
        if (strcmp(buff, "quit")==0) {   // 强制退出
            close(s);
            exit(0);
        }

        if (strcmp(buff, "connect")!=0) {
            // 判断连接状态，如果未连接，则不允许发送其他命令
            // ...
        }

        // 多传1个空字符'\0'
        ssize_t rc = sendto(s, buff, strlen(buff)+1, 0,
                            (struct sockaddr *)&addr, sizeof(addr));
        if (rc < 0) perror("Send failed");
    }
}
```