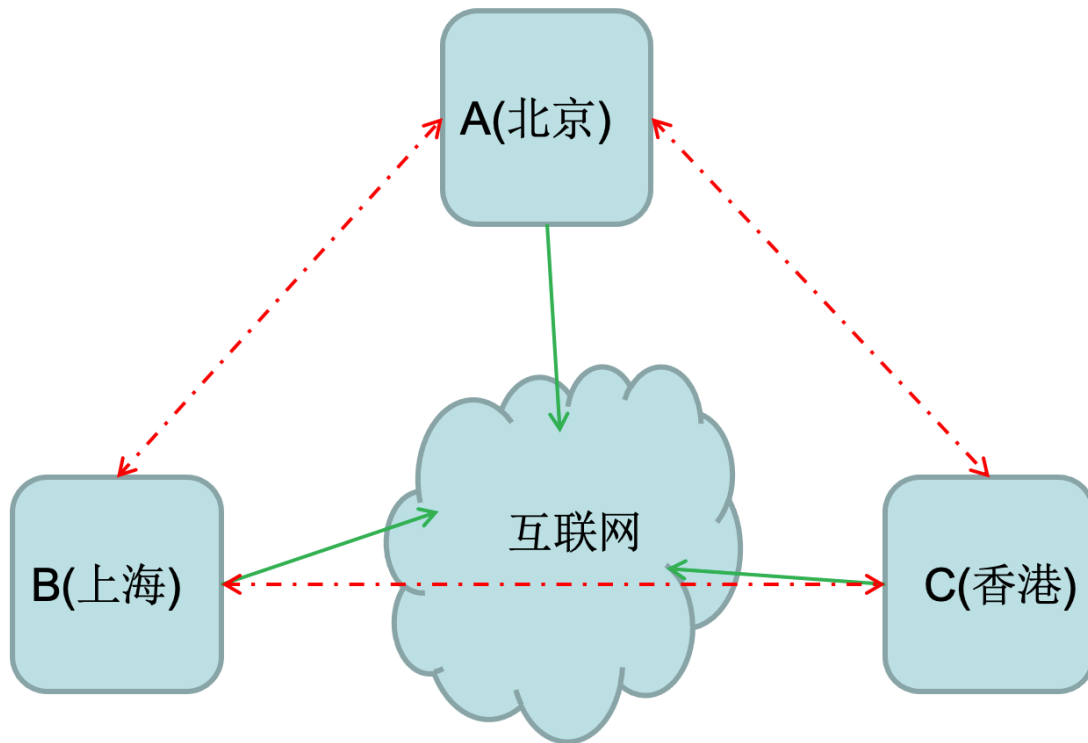


1 基本概念扫盲

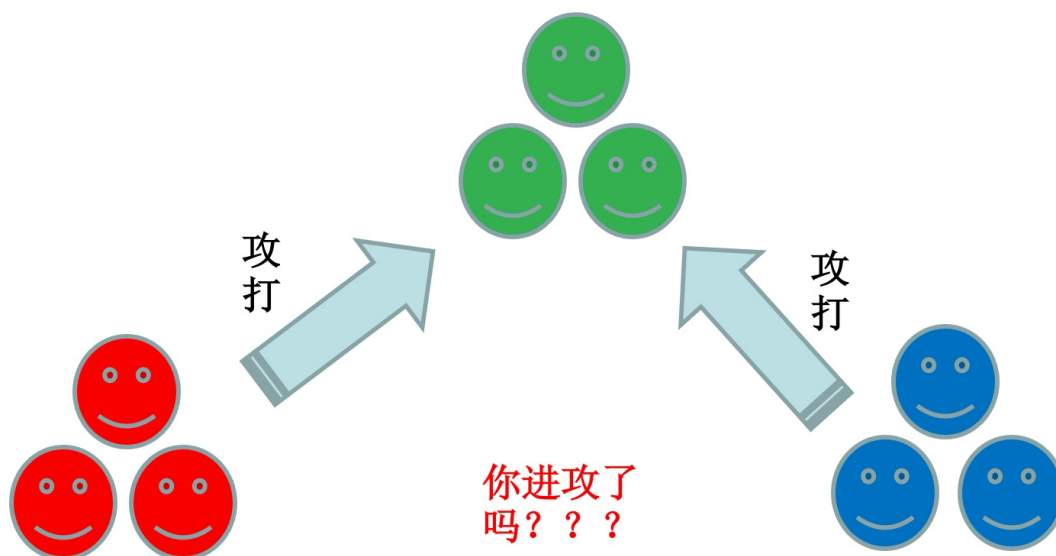
1.1 为什么需要计算机网络

如下图所示，A、B、C三个不同地域的主机要想进行通信不是凭空就可以通信的，而是需要基于互联网进行互相连接、通信。



1.2 为什么需要协议

如下图所示，红和蓝是联合攻打绿，它们以烽火为信号出动攻打绿，那么这时候就需要一个约定，比如红先点烽火，然后蓝看见了狼烟再点烽火，红看见了蓝的狼烟之后熄灭烽火，以此表示自己看见了，而蓝看见了红熄灭烽火之后也熄灭自己的烽火以此表示自己知道红看见了此信号，而后两人就需要再约定信号一起整顿出军以确保没有失误。

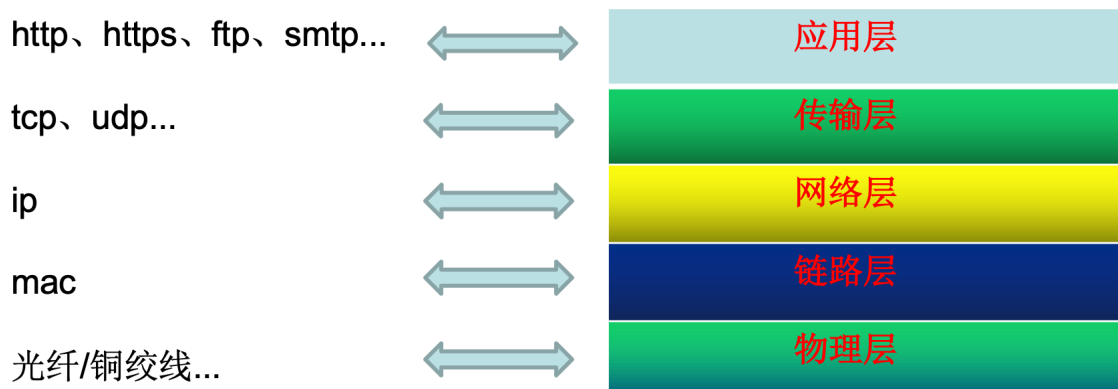


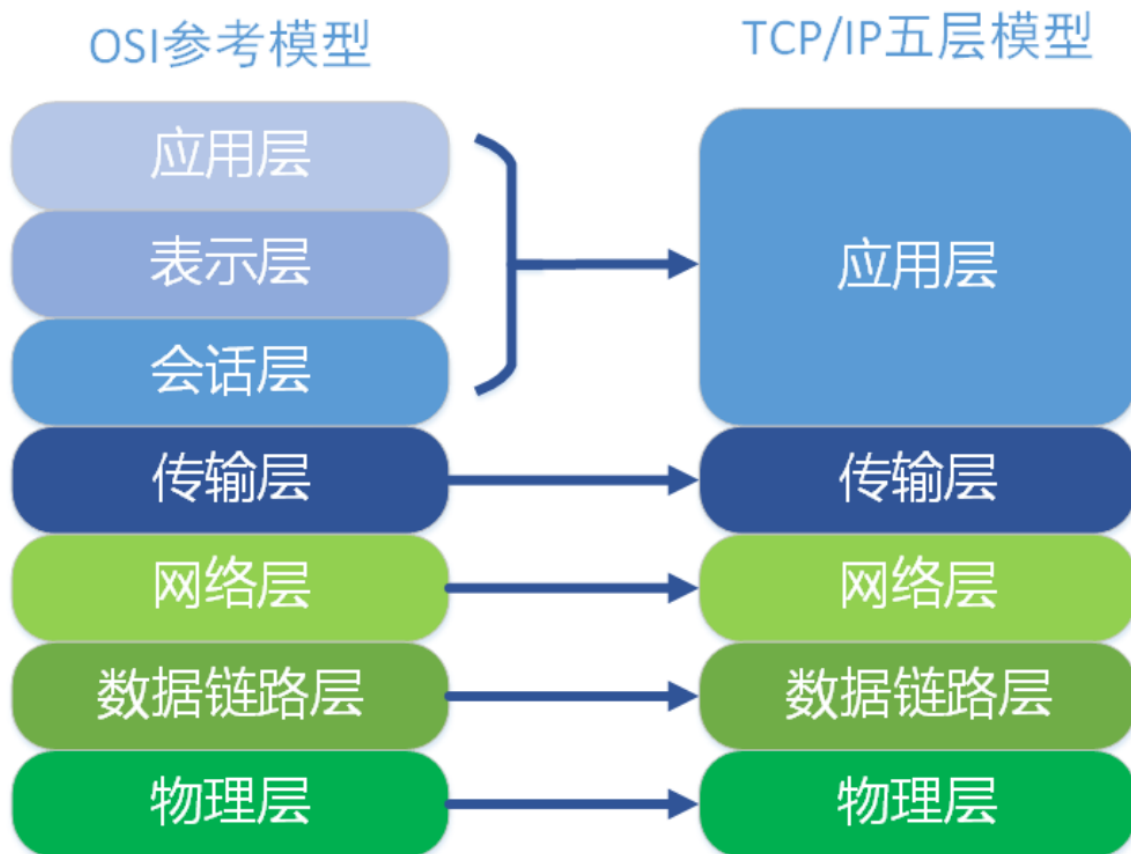
所以我们知道蓝和红之间的通信不能保证100%成功，但是要尽量保证没有失误的话就需要一遍又一遍的去确认，而这些一次又一次的确定就是双方定下的协议；由此我们可以清楚的认识到我们网络通信中是必须要有协议的存在。

1.3 为什么需要这么多协议

上文中我们举了两军协同作战，他们之间有个作战协议，而一旦作战成功，夺下对方城池那就需要另外一个瓜分战果的协议，所以每个不同的场景都会有对应的协议，这是有这么多协议的原因。

如下图所示，我们的计算机网络也有很多协议，下面是分为五层，如果你了解过计算机网络协议应该会知道七层模型、五层模型，但本章节不讲七层模型而是选择五层模型，因为七层模型是一种理想化的模型，实际应用我们用到的是五层模型。





1.4 如何定位互联网上的终端

首先我们熟知的系统是通过线程ID、进程ID知道对应的线程和进程的，在每个国家公民都是有身份证号码的，这也用来定位你这个人；在互联网上同样也有这样一个标识去确认终端，这就是**IP地址**。

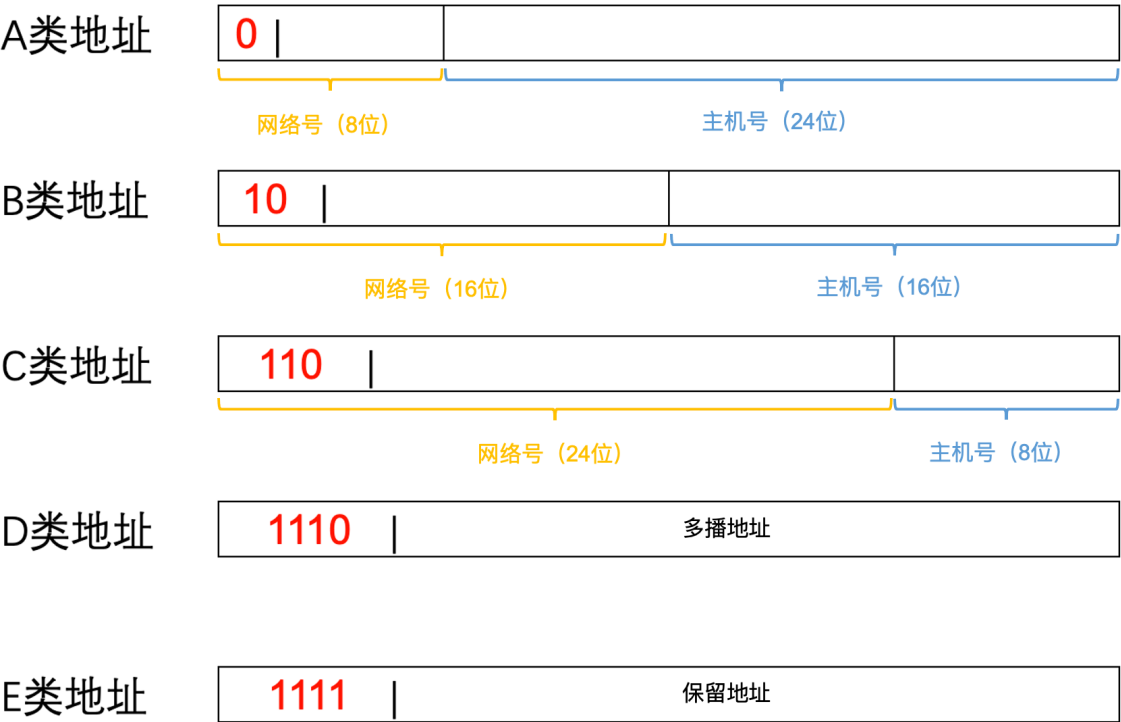
IP地址以"."符号分割，一共有四组，例如：120.120.120.120，每一组都是的区间都是0到255，IP地址的组成是网络号加上主机号，而具体的界定我们可以查看下文。

IP地址分为5类，其分别如下所示：

类型	起始地址	结束地址
A类	0.0.0.0	127.255.255.255
B类	128.0.0.0	191.255.255.255
C类	192.0.0.0	223.255.255.255
D类	224.0.0.0	239.255.255.255

类型	起始地址	结束地址
E类	240.0.0.0	247.255.255.255

我们不需要死记硬背，需要的时候自己查下就可以，具体含义网上很多，这里不过多赘述。



1.5 如何区分出网络号、主机号

如上图中我们可以知道IP地址分成了网络号和主机号两部分，通过子网掩码可以从IP地址中区分出网络号，其运算规则是：**网络号 = IP地址 & (按位与) 子网掩码**。

我们查看自己本机的IP地址和子网掩码来计算：

配置 IPv4:

使用 DHCP

IP 地址:

192.168.8.117

子网掩码:

255.255.255.0

IP地址：**192.168.8.117**，子网掩码：**255.255.255.0**，将这两个转为二进制则为：

1	11000000.10101000.00001000.01110101
---	-------------------------------------

2	11111111.11111111.11111111.00000000
---	-------------------------------------

我们进行按位与运算，结果就是：

1	11000000.10101000.00001000.00000000
2	C0.A8.08.00
3	192.168.8.0

那么在这里**192.168.8.0**就是其网络号，同样我们可以根据子网掩码来获取主机号，其运算规则是：**主机号 = IP地址 & (按位与) ~ (取反) 子网掩码**。

1	~ 11111111.11111111.11111111.00000000 // 取反子网掩码
2	00000000.00000000.00000000.11111111
3	& 11000000.10101000.00001000.01110101 // 按位与
4	00000000.00000000.00000000.01110101
5	Dec -> 0.0.0.117 // 十进制结果

最终结果我们知道了其主机号为**0.0.0.117**。

子网掩码本质上是32位的二进制，只不过是为了看着直观一些就转为了十进制，子网掩码1所对应的位为网络号位而0所对应的位为主机号位，其用来区分有几个子网，例如这里我们的**255.255.255.0**，转为二进制实际上前24位是网络位，后8位是主机位，那也就表示我们只有一个子网，在这里我们的子网地址范围就是：

192.168.8.0-192.168.8.255，可用的主机号计算公式就是**2的8（主机位）次方-2**，这里结果也就是254，为什么我们还需要减去2，这是因为根据计算方法，**192.168.8.0就是网络号（代表当前网络）**，同时根据定义，**主机号位全为1的地址为此网段的广播地址**，此时的广播地址为**192.168.8.255**，去掉网络地址和广播地址，也就是254个主机号可用。

而如果我们子网掩码为**255.255.255.192**，转为二进制就是**11111111.11111111.11111111.11000000**，可以看见在我们的原先的后8位主机位中占用了2位作为网络位，现在有26个1，那么根据二进制非0即1，其表现方式就有**11000000、10000000、00000000、01000000**，也就是说我们将原有的**192.168.8.0**这个网络分成了四份，即4个子网，也可以理解为这里就是**2的2（后8位主机位中占用了2位）次方**，现在我们将它们转换成10进制就分别是**0、64、128、192**，那么这4段网络的范围如下所示：

1	192.168.8.0 - 192.168.8.63
2	192.168.8.64 - 192.168.8.127
3	192.168.8.128 - 192.168.8.191
4	192.168.8.192 - 192.168.8.255

1.6 端口号是什么

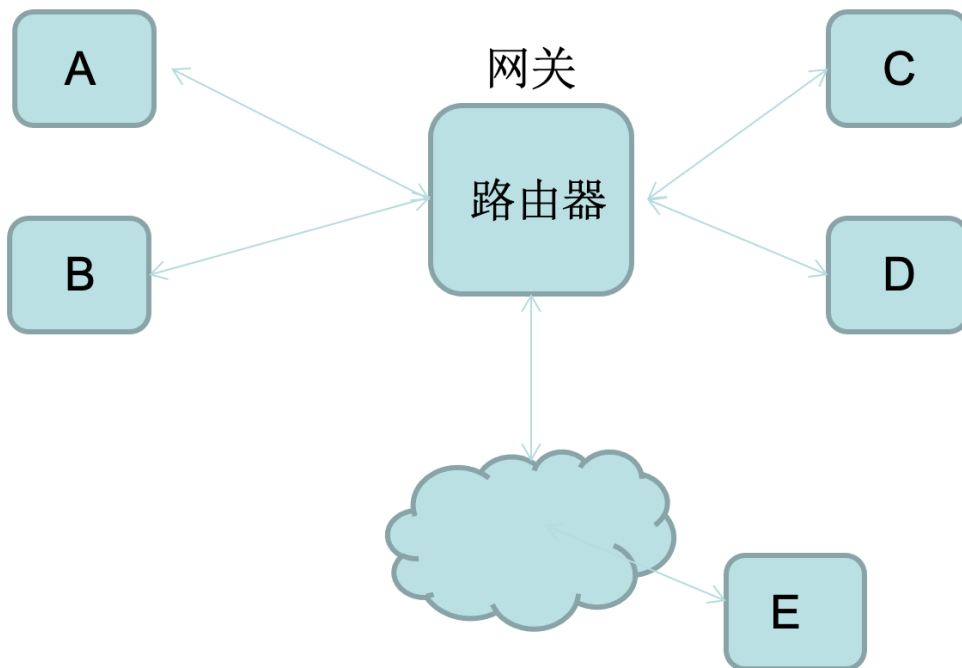
问题：系统中有很多个进程连着网，比如QQ、微信、迅雷...那么系统是如何区分出数据包应该分给哪个进程呢？

答案：系统是根据端口号来区分出数据包应该分给哪个进程，每个联网的进程都会分配一个系统唯一的ID，发送数据包的时候这个ID也会放进去，接受数据包的时候就可以根据这个ID来分别出对应进程，这个ID也就是端口号。

注意：端口号的范围就是0-65535

1.7 网关是什么

如下图所示，路由器就是一个网关，网关就相当于是一扇门，关内是一个网络，A、B、C、D都可以在这个网内进行通信，就不需要网关了，而如果A想跟E进行通信就需要通过网关将你的请求转发去通信，这是因为E不在关内。



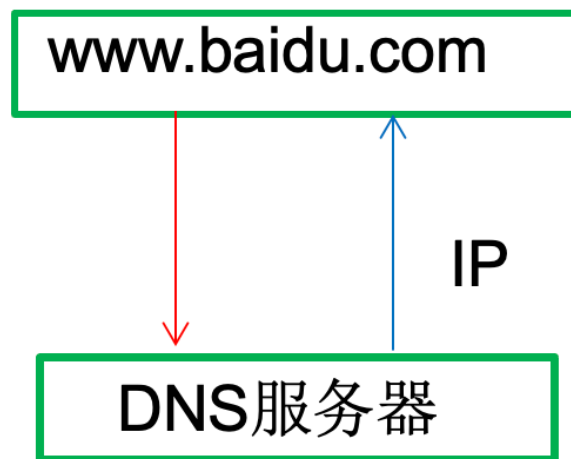
1.8 DNS是什么

假设你访问的是`www.baidu.com`，这是一个域名，但是这个域名你想要去访问到真正的那些展示给你的资源其背后对应的正是某个服务器的IP，根据这个IP和对应的端口你才可以访问到资源，而将域名和IP进行关联的正是DNS。

DNS服务器通过记录域名和IP的关联，当你想要去访问某个域名的时候，就需要给DNS服务器发送请求，而后DNS服务器接收到你的请求，将请求中想要查询的域名在DNS服务器本身的记录中去搜索找到对应的IP，最后返回给你。

台面上的：www.baidu.com

台面下的：



2 TCP客户端和服务端编程架构

2.1 什么是TCP

TCP，英文全称是Transmission Control Protocol，中文为传输**控制**协议，在我们之前所说的五层还是七层模型中，TCP都属于传输层。

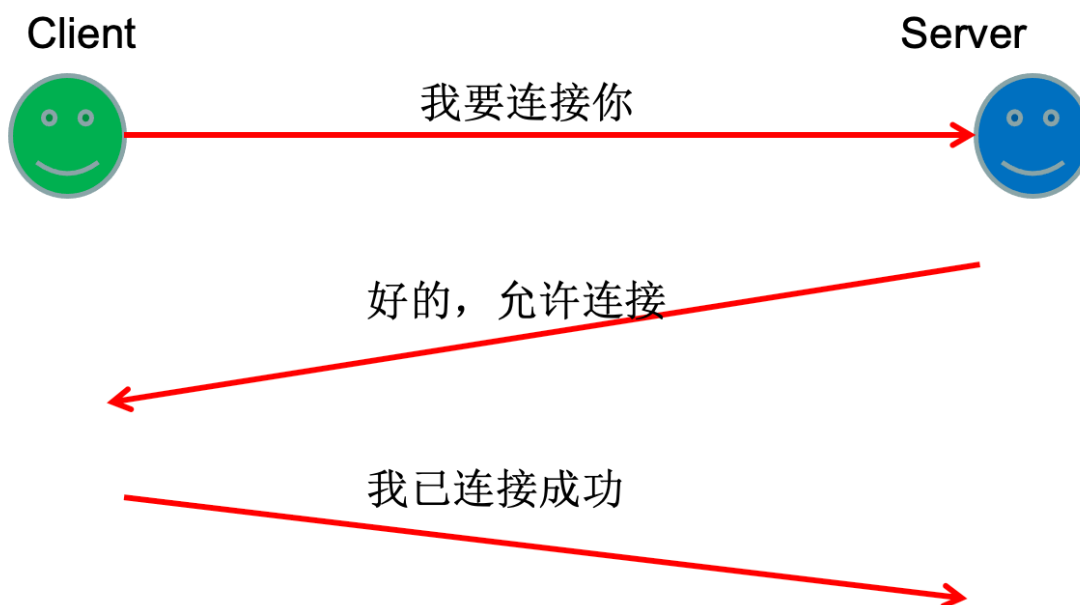
如下图所示，A和B基于TCP协议进行传输控制，该协议可以控制协议传输或者说保证传输过程中的数据是正确的：



2.2 面向连接

之前我们说到TCP协议可以保证传输过程中的数据是正确的，这是因为其是**面向连接**的网络协议。

如下图所示，客户端和服务端基于TCP进行传输通信，首先客户端要跟服务器端说（发送请求）我要跟你进行连接，其次服务器端要回应（发送请求）允许客户端进行连接，而后客户端才会在发送一个请求正式连接，这就是**三次握手**的特点。



当客户端和服务端连起来之后，才是会进入传输。

2.3 服务器端编程框架

了解了理论之后就要付诸于行动，在编程的时候我们的服务器端要有七个步骤去完成：

- | | |
|---|-------------|
| 1 | 1. 创建套接字 |
| 2 | 2. 绑定套接字 |
| 3 | 3. 监听套接字 |
| 4 | 4. 等待连接 |
| 5 | 5. 收发数据 |
| 6 | 6. 断开连接(被动) |
| 7 | 7. 关闭套接字 |

这时候就有一个新的东西，就是套接字，这是系统给你打包好的，你可以理解这是网络通信过程中端点的抽象表示，而想要客户端去连接服务器端，就需要一对套接字，一个运行在服务器端，一个运行在客户端；如果概念无法很清晰的去了解，没关系，在实际编程中你就会有所体会。

2.3.1 按顺序编写代码

首先我们创建一个Win32控制台应用的项目，其次在头部包含文件和调用lib：

1	#include <WSOCK2.H>
2	#pragma comment(lib, "ws2_32.lib")

接着我们就需要按照顺序去编写代码，首先第一步是创建套接字，这个需要用到一个函数socket，其语法如下：

1	SOCKET socket(
2	int af, // 地址族规范：常见有IPv6 (AF_INET6) 或IPv4 (AF_INET)
3	int type, // 套接字类型：原始套接字SOCKET_RAW（对较低层次的协议直接访问，例如IP、ICMP协议）、SOCK_STREAM面向连接（TCP/IP协议）、SOCK_DGRAM面向无连接（UDP协议）
4	int protocol // 使用的协议：这里我们可以直接写0，这样操作系统就会根据前面两个选项推断出你想用的协议
5);
6	
7	// 实现代码
8	
9	SOCKET sSocket = socket(AF_INET, SOCK_STREAM, 0);

接下来我们的就需要绑定套接字，使用函数bind，其语法如下：

1	int bind(
2	SOCKET s, // 套接字：将创建的套接字变量名字写上去
3	const struct sockaddr FAR *name, // 网络地址信息：包含通信所需要的相关信息，传递的应该是一个sockaddr结构体，在具体传参的时候，会用该结构体的变体sockaddr_in形式去初始化相关字段

```

4     int namelen                // sockaddr_in结构体的长度
5 );

```

sockaddr_in结构体的定义如下：

```

1  /*
2   * Socket address, internet style.
3   */
4  struct sockaddr_in {
5      short sin_family; // 地址族规范：与创建套接字时候所使用的一致即可
6      u_short sin_port; // 端口
7      struct in_addr sin_addr; // IP地址
8      char sin_zero[8]; // 无特殊的含义，只是为了与sockaddr结构体一致，因为
                          // 在给套接字分配网络地址的时候会调用bind函数，其中的参数会把sockaddr_in结构体转化为
                          // sockaddr结构体
9  };

```

我们只需要关注前三个成员即可，最后一个不用管，可以看见IP地址又是一个结构体，我们接着看看in_addr结构体：

```

1  /*
2   * Internet address (old style... should be updated)
3   */
4  struct in_addr {
5      union {
6          struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
7          struct { u_short s_w1,s_w2; } S_un_w;
8          u_long S_addr;
9      } S_un;
10 #define s_addr S_un.S_addr
11                                     /* can be used for most tcp & ip code */
12 #define s_host S_un.S_un_b.s_b2
13                                     /* host on imp */
14 #define s_net S_un.S_un_b.s_b1
15                                     /* network */
16 #define s_imp S_un.S_un_w.s_w2
17                                     /* imp */
18 #define s_impno S_un.S_un_b.s_b4
19                                     /* imp # */
20 #define s_lh S_un.S_un_b.s_b3
21                                     /* logical host */
22 };

```

这个结构体里面又是一个联合体，联合体和结构体是差不多的，区别在于联合体用于覆盖使用而结构体是不覆盖使用；

并且我们通过代码可以看见这就是一个u_long类型的地址，我们可以使用函数inet_addr来按照网络字节序转换：

```
1    inet_addr("192.168.1.1");
```

最终，我们在赋值的时候还是要选择某个成员去赋值，代码实现如下（需要注意的是，这里的IP地址是不可以乱写的需要通过命令行或其他方式获取本机的IP地址）：

```
1    sockaddr_in sockAddrInfo = {0};    // 初始化
2    sockAddrInfo.sin_addr.S_un.S_addr = inet_addr("192.168.1.1"); // 地址
3    sockAddrInfo.sin_port = htons(2118); // 端口需要按照网络字节序，所以需要使用
    htons函数
4    sockAddrInfo.sin_family = AF_INET; // 地址族规范
5
6    bind(sSocket, (sockaddr*)&sockAddrInfo, sizeof(sockAddrInfo));
```

第三步就是监听套接字，使用函数listen，其语法如下：

```
1    int listen(
2        SOCKET s,    // 套接字：将创建的套接字变量名字写上去
3        int backlog // 待处理连接队列的最大长度：表示队列中最多同时有多少个连接请求
4    );
5
6    // 实现代码
7
8    listen(sSocket, 1);
```

第四步等待连接，使用函数accept，其语法如下：

```
1    SOCKET accept(
2        SOCKET s, // 套接字：将创建的套接字变量名字写上去
3        struct sockaddr FAR *addr, // 输出参数，需要传入一个sockaddr结构体的地址
4        int FAR *addrlen // 输出参数，需要传入一个sockaddr结构体长度的地址
5    );
6
7    // 实现代码，accept返回的也是一个SOCKET，我们需要赋值一下
8
9    sockaddr_in acceptSockAddrInfo = {0};    // 初始化
10   int acceptSockAddrLen = 0;
11   SOCKET aSocket = accept(sSocket, (sockaddr*)&acceptSockAddrInfo,
    &acceptSockAddrLen);
```

第五步收发数据，首先我们看下收数据，使用到函数recv，其语法如下：

```
1    int recv(
2        SOCKET s,    // 套接字：将accept返回的套接字变量名字写上去
3        char FAR *buf, // 输出参数，数据缓冲区，接收到的数据
4        int len,    // 缓冲区大小
5        int flags    // 指定调用方式的标志，这个我们就直接写0即可
6    );
7
```

```

8 // 实现代码
9
10 char buf[100] = {0};
11 recv(aSocket, buf, 100, 0);
12 printf("Recv data: %s\n", buf);

```

接着我们看下发数据，使用函数send，其语法如下：

```

1 int send(
2     SOCKET s,           // 套接字：将accept返回的套接字变量名字写上去
3     const char FAR *buf, // 传输数据的缓冲区
4     int len,            // 缓冲区大小
5     int flags            // 指定调用方式的标志，这个我们就直接写0即可
6 );
7
8 // 实现代码
9
10 send(aSocket, buf, strlen(buf)+1, 0);

```

第六步断开连接，我们使用shutdown函数，其语法如下：

```

1 int shutdown(
2     SOCKET s, // 套接字：将accept返回的套接字变量名字写上去
3     int how   // 断开连接的形式：SD_SEND不再发送数据、SD_RECEIVE不再接受数据、
                SD_BOTH不再收发数据
4 );
5
6 // 实现代码
7
8 shutdown(aSocket, SD_SEND);

```

第七步也是最后一步，关闭套接字（这里有2个都要关闭），使用函数closesocket，其语法如下：

```

1 int closesocket(
2     SOCKET s // 套接字：将accept返回的套接字变量名字写上去
3 );
4
5 // 实现代码
6
7 closesocket(aSocket);
8 closesocket(sSocket);

```

这时候还没有结束，需要使用函数WSAStartup进行Winsock的初始化，其语法格式如下：

```

1 int WSAStartup(
2     WORD wVersionRequested, // 版本号，指定所需的Windows Sockets版本，我们可以使用
                              MAKEWORD去创建一个版本号

```

```

3     LPWSADATA lpWSaData // 指向WSADATA数据结构的指针，用于接收Windows Sockets实现的
4     细节
    );

```

实现代码如下：

```

1     WORD wsVersion = MAKEWORD(2, 2);
2     WSADATA wsaData = {0};
3     WSASStartup(wsVersion, &wsaData);

```

最终我们实现了服务器端的功能，完整代码如下：

```

1     int main(int argc, char* argv[])
2     {
3         // 0. 初始化
4         WORD wsVersion = MAKEWORD(2, 2);
5         WSADATA wsaData = {0};
6         WSASStartup(wsVersion, &wsaData);
7         // 1. 创建套接字
8         SOCKET sSocket = socket(AF_INET, SOCK_STREAM, 0);
9         if (SOCKET_ERROR == sSocket) {
10            printf("套接字创建失败!\n" );
11        }
12        else {
13            printf("套接字创建成功!\n" );
14        }
15        // 2. 绑定套接字
16        sockaddr_in sockAddrInfo = {0}; // 初始化
17        sockAddrInfo.sin_addr.S_un.S_addr = inet_addr("172.16.176.5");
18        sockAddrInfo.sin_port = htons(2118); // 端口
19        sockAddrInfo.sin_family = AF_INET; // 地址族规范
20
21        int bRes = bind(sSocket, (sockaddr*)&sockAddrInfo,
22            sizeof(sockAddrInfo));
23
24        if (SOCKET_ERROR == bRes) {
25            printf("绑定失败!\n");
26        }
27        else {
28            printf("绑定成功!\n");
29        }
30        // 3. 监听套接字
31        int lRes = listen(sSocket, 1);
32        if (SOCKET_ERROR == lRes) {
33            printf("监听失败!\n");
34        }
35        else {
36            printf("监听成功!\n");
37        }
38        // 4. 等待连接
39        sockaddr_in acceptSockAddrInfo = {0}; // 初始化

```

```

39     int acceptSockAddrLen = sizeof(acceptSockAddrInfo);
40     SOCKET aSocket = accept(sSocket, (sockaddr*)&acceptSockAddrInfo,
    &acceptSockAddrLen);
41     if (INVALID_SOCKET == aSocket) {
42         printf("服务端等待连接失败!\n");
43     }
44     else {
45         printf("服务端等待连接成功!\n");
46     }
47     // 5. 收发数据
48     char buf[100] = {0};
49     // 循环
50     while (true) {
51         int ret = recv(aSocket, buf, 100, 0);
52         if (ret == 0) {
53             // 如果recv返回为0则表示客户端要断开连接, 就跳出循环断开连接
54             break;
55         }
56         printf("Recv data: %s\n", buf);
57         send(aSocket, buf, strlen(buf)+1, 0);
58         memset(buf, 0, 100);
59     }
60
61     // 6. 断开连接(被动)
62     shutdown(aSocket, SD_SEND);
63     // 7. 关闭套接字
64     closesocket(aSocket);
65     closesocket(sSocket);
66
67     WSACleanup();
68     return 0;
69 }

```

最后, 如果你不使用了这个扩展就需要使用WSACleanup函数去终止使用; 建议在实际编程过程中, 应该将函数的返回值存储下来并做判断。

2.4 客户端编程框架

客户端编程框架的步骤就简单了一些, 只有六个步骤:

- 1 1. 创建套接字
- 2 2. 绑定套接字
- 3 3. 连接服务器
- 4 4. 收发数据
- 5 5. 断开连接(主动)
- 6 6. 关闭套接字

了解了服务器端如何编写, 客户端也就了如指掌的, 实现代码如下:

```

1 int main(int argc, char* argv[])

```

```

2  {
3      // 0. 初始化
4      WORD wsVersion = MAKEWORD(2, 2);
5      WSADATA wsaData = {0};
6      WSStartup(wsVersion, &wsaData);
7      // 1. 创建套接字
8      SOCKET cSocket = socket(AF_INET, SOCK_STREAM, 0);
9      if (SOCKET_ERROR == cSocket) {
10         printf("套接字创建失败!\n");
11     }
12     else {
13         printf("套接字创建成功!\n");
14     }
15     // 2. 绑定套接字
16     sockaddr_in sockAddrInfo = {0};    // 初始化
17     sockAddrInfo.sin_addr.S_un.S_addr = inet_addr("172.16.176.12");
18     sockAddrInfo.sin_port = htons(2119); // 端口
19     sockAddrInfo.sin_family = AF_INET; // 地址族规范
20
21     int bRes = bind(cSocket, (sockaddr*)&sockAddrInfo,
22         sizeof(sockAddrInfo));
23
24     if (SOCKET_ERROR == bRes) {
25         printf("绑定失败!\n");
26     }
27     else {
28         printf("绑定成功!\n");
29     }
30     // 3. 连接服务器
31     sockaddr_in serverSockAddrInfo = {0};    // 初始化
32     serverSockAddrInfo.sin_addr.S_un.S_addr = inet_addr("172.16.176.5");
33     serverSockAddrInfo.sin_port = htons(2118); // 端口
34     serverSockAddrInfo.sin_family = AF_INET; // 地址族规范
35     int cRes = connect(cSocket, (sockaddr*)&serverSockAddrInfo,
36         sizeof(serverSockAddrInfo));
37     if (SOCKET_ERROR == cRes) {
38         printf("与服务器连接失败!\n");
39     }
40     else {
41         printf("与服务器连接成功!\n");
42     }
43     // 4. 收发数据
44     printf("Input: ");
45     char sendData[100];
46     scanf("%s", sendData);
47     send(cSocket, sendData, strlen(sendData)+1, 0);
48     char buf[100] = {0};
49     recv(cSocket, buf, 100, 0);
50     printf("Recv data: %s\n", buf);
51     // 5. 断开连接(主动)
52     shutdown(cSocket, SD_SEND);
53     // 6. 关闭套接字
54     closesocket(cSocket);

```

```

53
54     WSACleanup();
55     return 0;
56 }

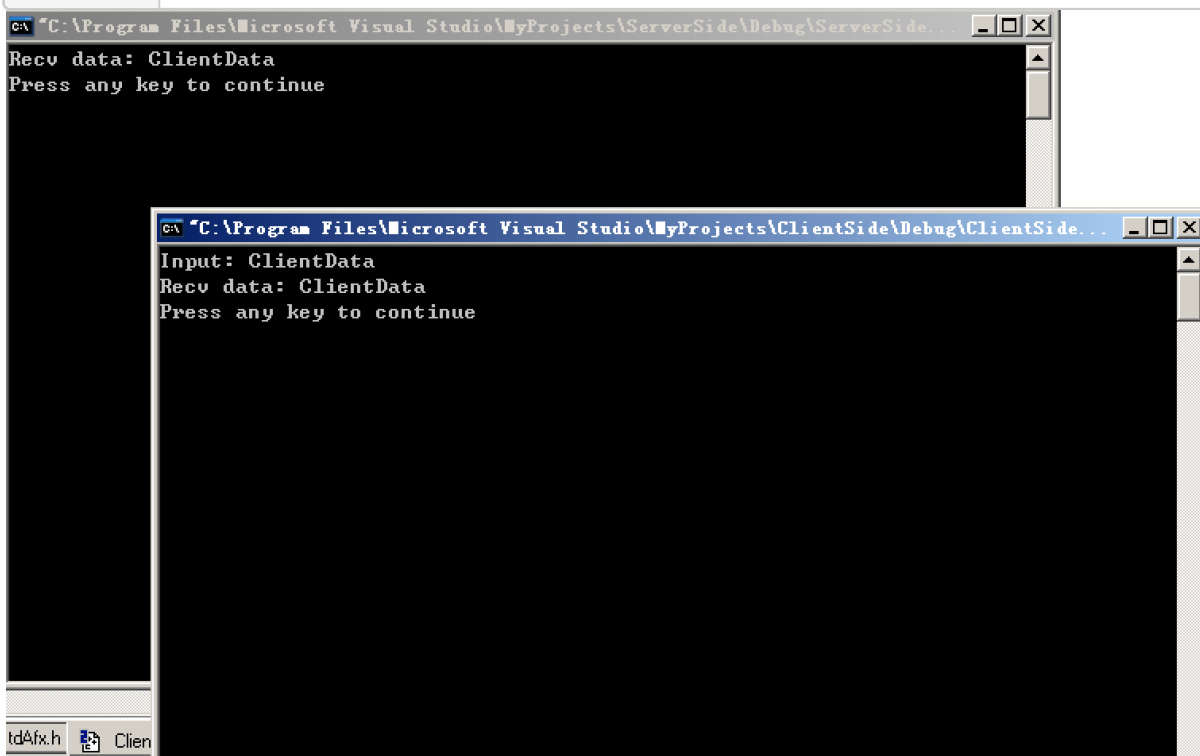
```

与服务器端不同的是，客户端需要连接服务器端，同样也是通过sockaddr_in结构体去指定服务器端的地址和端口，使用到了一个新的函数connect，该函数语法如下：

```

1  int connect(
2      SOCKET s,                // 套接字：
3      const struct sockaddr FAR *name, // sockaddr 结构体
4      int namelen              // 结构体长度
5  );

```



2.5 需要补充的背景知识

补充的背景知识实际上是之前也了解过的存储数据的大、小端的存储模式，这里就不再过多赘述。

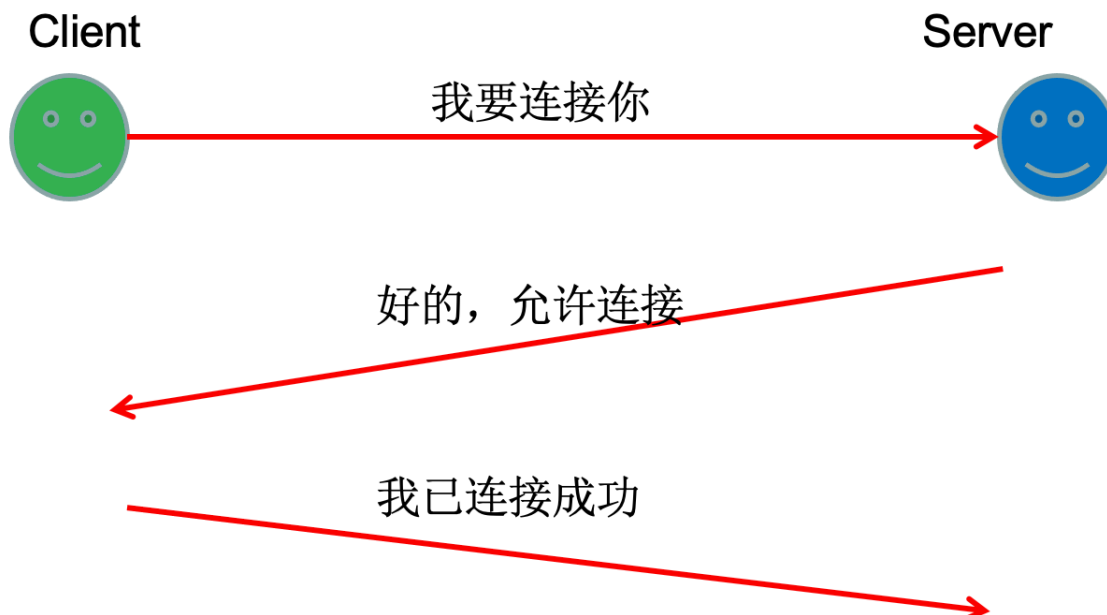
- 1 字节序：字节与存储位置的关系
- 2
- 3 小端：将低序字节存储在起始地址
- 4
- 5 大端：将高序字节存储在起始地址
- 6

7 网络字节序：顾名思义就是网络上的字节序

我们需要重点的是网络字节序，其顾名思义就是网络上的字节序，比如说一个数据在你电脑上存储的是小端存储，而可能在网络传输的时候就是大端模式，所以你就需要将你电脑上的存储的数据转换。

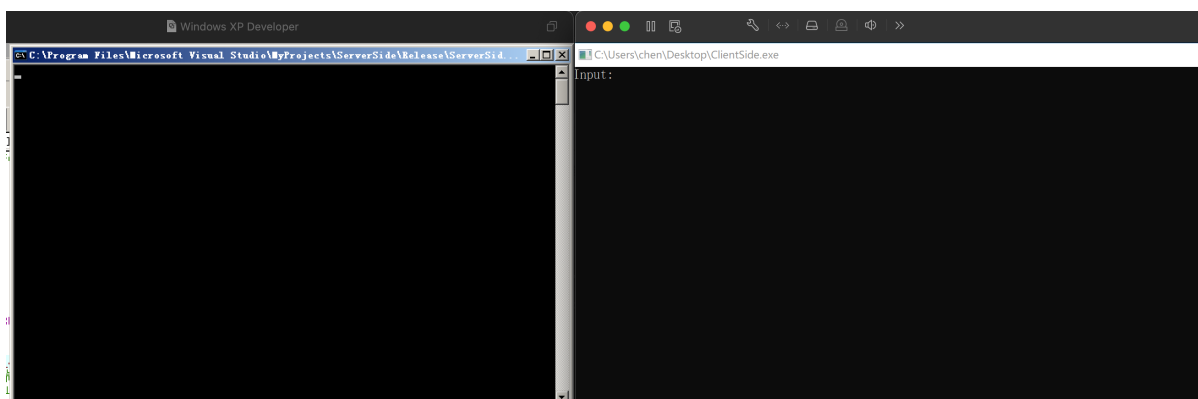
3 TCP三次握手与抓包分析

虽然我们已经了解过TCP三次握手的流程，但是里面具体的细节，我们并不了解，了解这些细节也有便于我们去深入了解TCP协议以及解决今后细节性的问题。



在这里，我们使用Wireshark这个抓包工具，下载地址：<https://www.wireshark.org/>

接下来我们使用两台机器作为服务器端和客户端，在客户端机器上安装Wireshark软件进行抓包，如下图（左服务器端，右客户端）：



Wireshark直接抓网卡的流量即可，用过滤语法来过滤一下：**(ip.dst==172.16.176.5 and tcp.port==2118) or (ip.src==172.16.176.5)**，172.16.176.5地址为服务器端的IP地址，172.16.176.12地址为客户端的IP地址。

如下图所示，我们一共抓到了三个包，这三个包就是我们所说的三次握手对应的包。

正在捕获 Ethernet0

文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)

(ip.dst==172.16.176.5 and tcp.port==2118) or (ip.src==172.16.176.5)

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.176.12	172.16.176.5	TCP	66	2119 → 2118 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256
2	0.000283	172.16.176.5	172.16.176.12	TCP	66	2118 → 2119 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=
3	0.000356	172.16.176.12	172.16.176.5	TCP	54	2119 → 2118 [ACK] Seq=1 Ack=1 Win=2102272 Len=0

首先我们可以看到第一个包，就是客户端向服务器端发送SYN（同步序列编号）包，我们可以看见其有一个seq=0，这表示一个序号，是随机的；

接着服务器端响应该请求，返回了SYN+ACK（确认字符）包表示允许连接，同样也有一个seq=0，并且多出了一个ack=1，同样这里的seq是随机表示的，而ack则是由第一个包的seq=0这个值+1的结果；

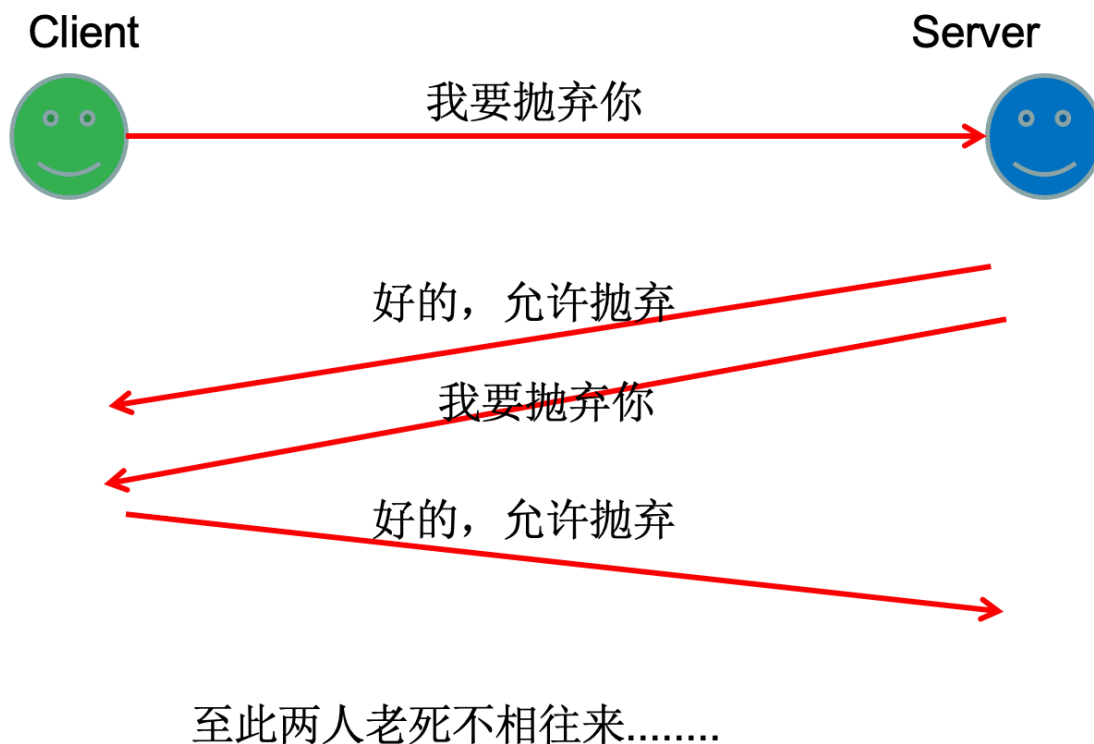
最后客户端收到确认请求，回应服务器端的请求表示连接成功，发送了ACK包，seq我们就不用管了，这里ack的值也是第二个包的seq=0这个值+1的结果。

4 TCP四次握手与抓包分析

4.1 了解四次握手

之前我们所了解的三次握手，是在建立连接时的，而我们现在所需要了解的四次握手，则是在断开连接时的；你可以思考一下为什么在这里连接时需要三次握手，而在断开连接时，却需要四次握手呢？

我们可以看见如下图，假设服务器端和客户端是两个相亲相爱的恋人，客户端提出要跟服务器端分手，需要得到服务器端的确认，而不仅仅是单方面的分手，服务器端也要提出跟客户端分手，客户端也要确认。那么为什么需要这样呢？

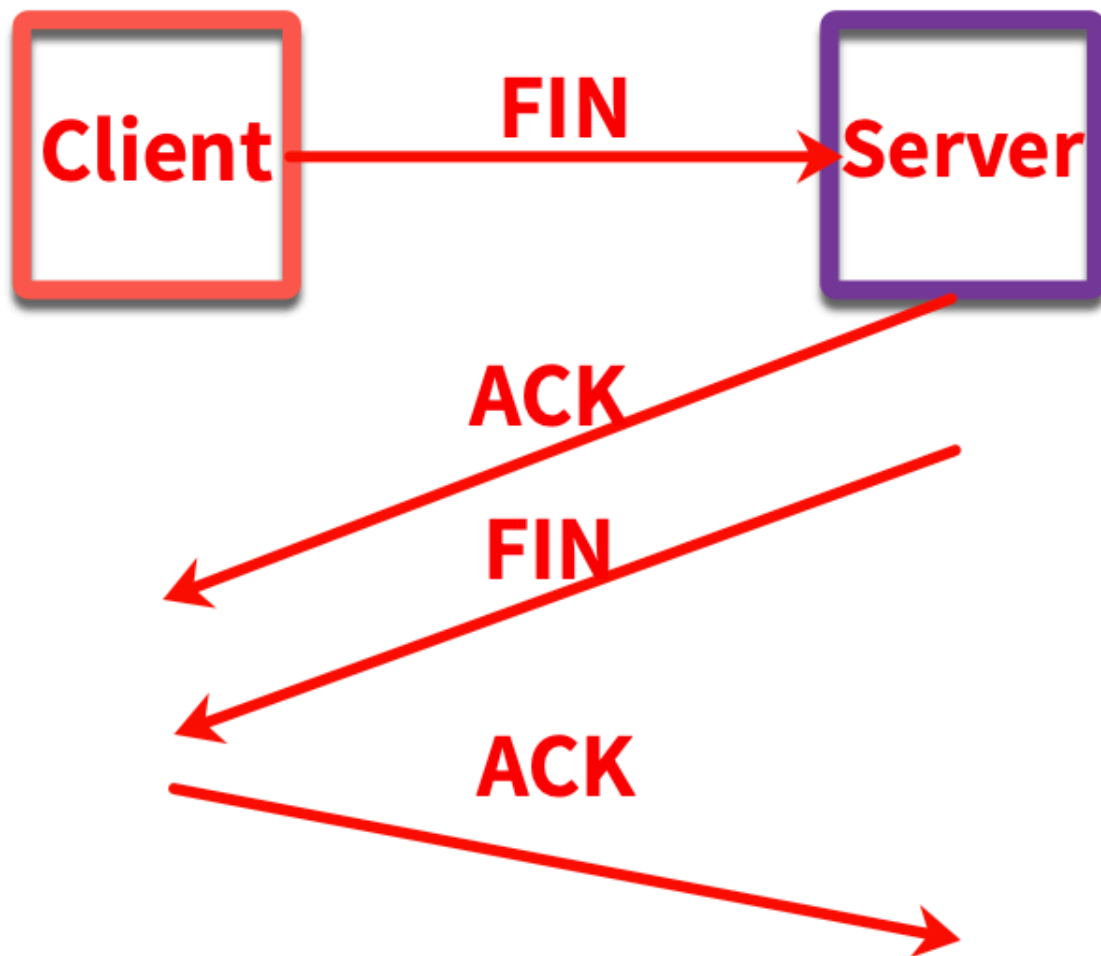


这时候我们需要了解通信中的三种通信模式：

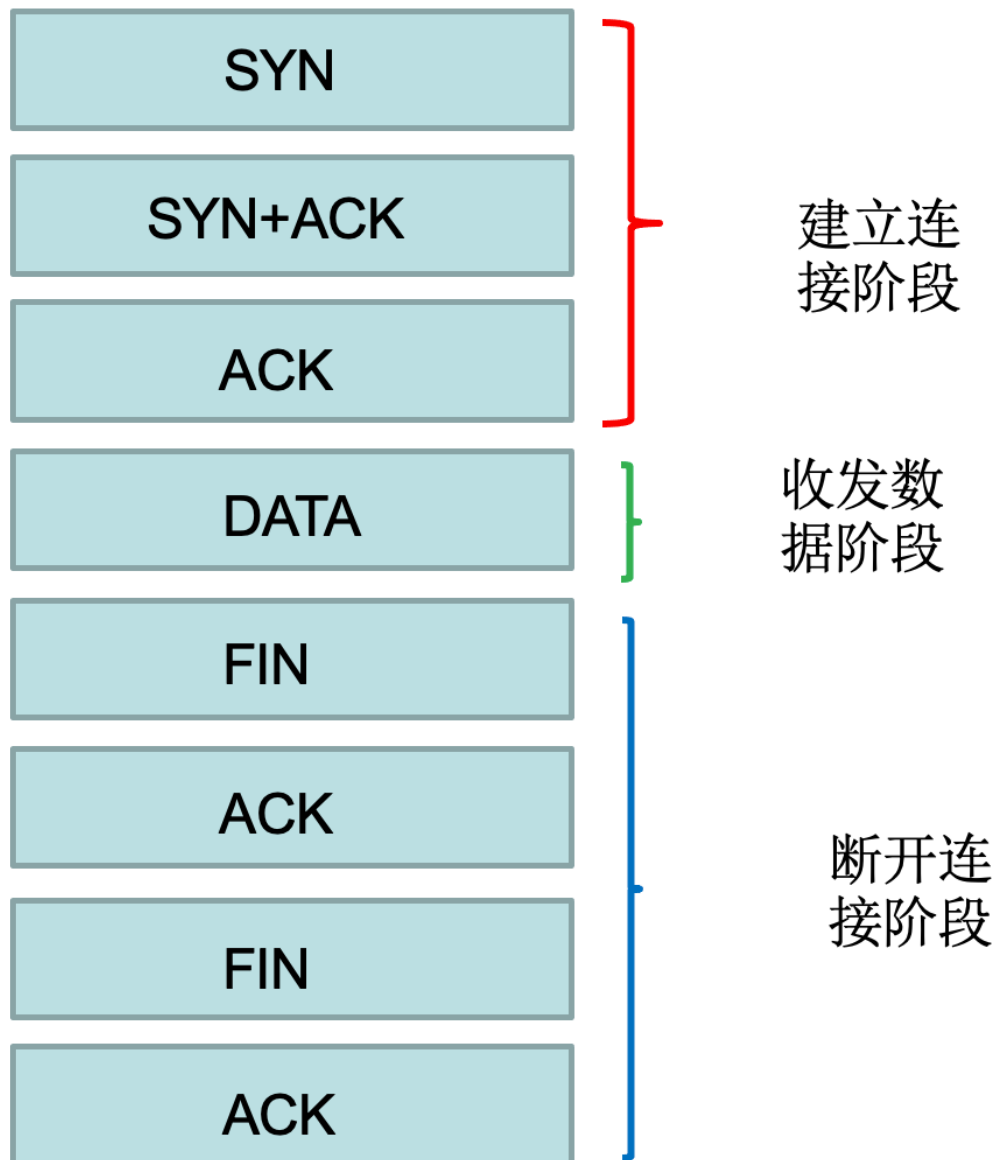
1. 单工：A和B进行通信，只能有一方发送，一方接收，只有一条通信线路；
2. 双工：A和B进行通信，双方可以同时互相发送接收，有两条通信线路；
3. 准双工：A和B进行通信，双方可以互相发送接收，但不可以同时进行，只有一条通信线路。

在互联网中采用了这种双工的通信模式，所以说当客户端给服务器端说要断开通信的时候，实际上只断开了一条通信线路，还有一条通信线路需要服务器端给客户端说要断开通信才会断开。

如下图所示，发送断开连接就是先发送FIN包，等待对方发送ACK包回应：



我们可以来看下TCP建立连接、通信、断开连接全貌：



4.2 抓包分析

在抓包之前，我们需要改一下服务器端的代码，在收发数据时，我们应该写一个死循环，然后判断recv函数的返回值是否为0，为0则表示客户端要断开连接，就跳出循环从而进入断开连接的代码：

```
// 5. 收发数据
char buf[100] = {0};
// 循环
while (true) {
    int ret = recv(aSocket, buf, 100, 0);
    if (ret == 0) {
        // 如果recv返回为0则表示客户端要断开连接，就跳出循环断开连接
        break;
    }
    printf("Recv data: %s\n", buf);
    send(aSocket, buf, strlen(buf)+1, 0);
    memset(buf, 0, 100);
}
```

最后，来使用Wireshark抓包测试一下看看这个全貌：

No.	Time	Source	Destination	Protocol	Length	Info
450	1685.756402	172.16.176.12	172.16.176.5	TCP	66	2119 → 2118 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256
453	1685.760507	172.16.176.5	172.16.176.12	TCP	66	2118 → 2119 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=
454	1685.760629	172.16.176.12	172.16.176.5	TCP	54	2119 → 2118 [ACK] Seq=1 Ack=1 Win=262656 Len=0
476	1701.123739	172.16.176.12	172.16.176.5	TCP	57	2119 → 2118 [PSH, ACK] Seq=1 Ack=1 Win=262656 Len=3
477	1701.124282	172.16.176.5	172.16.176.12	TCP	60	2118 → 2119 [PSH, ACK] Seq=1 Ack=4 Win=64237 Len=4
478	1701.124805	172.16.176.12	172.16.176.5	TCP	54	2119 → 2118 [FIN, ACK] Seq=4 Ack=5 Win=262656 Len=0
479	1701.125416	172.16.176.5	172.16.176.12	TCP	60	2118 → 2119 [ACK] Seq=5 Ack=5 Win=64237 Len=0
480	1701.125416	172.16.176.5	172.16.176.12	TCP	60	2118 → 2119 [FIN, ACK] Seq=5 Ack=5 Win=64237 Len=0
481	1701.125483	172.16.176.12	172.16.176.5	TCP	54	2119 → 2118 [ACK] Seq=5 Ack=6 Win=262656 Len=0

可以看见果然如我们所了解的在断开连接的时候发送了FIN以及ACK包，这里的seq的值和ack的值与我们之前所说的是一样的，seq是随机的，但ack的值是根据上一条的seq的值+1所得出来的。

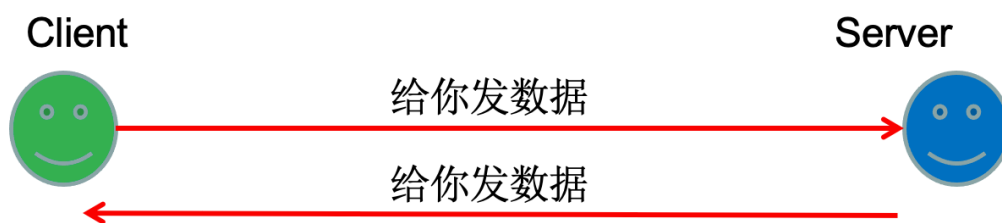
5 UDP客户端和服务端编程架构

5.1 什么是UDP

UDP是User Datagram Protocol的首字母简写，翻译过来就是标识用户数据报协议。UDP也是属于传输层的协议，从名字上来看，TCP是传输控制，而UDP是用户数据报，其实也就说明了UDP协议并不会去控制传输。

5.2 面向无连接

面向无连接，就是不用去询问服务器允不允许发送数据，他不管你怎么办，直接就给你发送数据了。



这样做的好处就是非常高效，但是却没办法保证数据是否正确地传递过去了。

5.3 服务器端编程框架

在UDP协议中服务器端和客户端的概念被弱化了，很难界定客户端与服务端，所以一般在UDP协议中，我们不以服务器端和客户端概念去做称谓，而是以端对端这种概念。

接着就是服务器端编程框架：

- | | |
|---|----------|
| 1 | 1. 创建套接字 |
| 2 | 2. 绑定套接字 |
| 3 | 3. 收发数据 |
| 4 | 4. 关闭套接字 |

实际代码如下：

```
1  int main(int argc, char* argv[])
2  {
3      // 0. 初始化
4      WORD wsVersion = MAKEWORD(2, 2);
5      WSADATA wsaData = {0};
6      WSStartup(wsVersion, &wsaData);
7      // 1. 创建套接字
8      SOCKET sSocket = socket(AF_INET, SOCK_DGRAM, 0);
9      if (SOCKET_ERROR == sSocket) {
10         printf("套接字创建失败!\n" );
```



```

11     }
12     else {
13         printf("套接字创建成功!\n" );
14     }
15     // 2. 绑定套接字
16     sockaddr_in sockAddrInfo = {0}; // 初始化
17     sockAddrInfo.sin_addr.S_un.S_addr = inet_addr("172.16.176.5");
18     sockAddrInfo.sin_port = htons(2118); // 端口
19     sockAddrInfo.sin_family = AF_INET; // 地址族规范
20
21     int bRes = bind(sSocket, (sockaddr*)&sockAddrInfo,
sizeof(sockAddrInfo));
22
23     if (SOCKET_ERROR == bRes) {
24         printf("绑定失败!\n");
25     }
26     else {
27         printf("绑定成功!\n");
28     }
29     // 3. 收发数据
30     char recvBuf[100] = {0};
31     while (true) {
32         recvfrom(sSocket, recvBuf, 100, 0, NULL, NULL);
33         if (strlen(recvBuf) != 0){
34             printf("Recv Data: %s \n", recvBuf);
35         }
36         memset(recvBuff, 0, 100);
37     }
38     // 4. 关闭套接字
39     closesocket(sSocket);
40     WSACleanup();
41     return 0;
42 }

```

相比较TCP协议我们少了很多代码，并且在创建套接字的时候参数变成了SOCK_DGRAM，我们还需要了解一个新函数recvfrom，这个函数是用来收信息的，其语法如下：

```

1 int recvfrom(
2     SOCKET s, // 套接字
3     char FAR* buf, // 输出参数，接收数据的缓冲区
4     int len, // 缓冲区长度
5     int flags, // 指定调用方式的标志，这个我们就直接写0即可
6     struct sockaddr FAR *from, // 输出参数（可选，我们可以直接写NULL），sockaddr
// 结构体
7     int FAR *fromlen // 输入输出共用参数（可选，我们可以直接写NULL），
// sockaddr结构体的大小，注意这里需要传入实际的大小
8 );

```

5.4 客户端编程框架

了解了，服务端变成框架之后我们再来看客户端会发现，它的步骤更加简单：

- 1 1. 创建套接字
- 2 2. 收发数据
- 3 3. 关闭套接字

实际代码如下：

```

1  int main(int argc, char* argv[])
2  {
3      // 0. 初始化
4      WORD wsVersion = MAKEWORD(2, 2);
5      WSADATA wsaData = {0};
6      WSStartup(wsVersion, &wsaData);
7      // 1. 创建套接字
8      SOCKET cSocket = socket(AF_INET, SOCK_DGRAM, 0);
9      if (SOCKET_ERROR == cSocket) {
10         printf("套接字创建失败!\n");
11     }
12     else {
13         printf("套接字创建成功!\n");
14     }
15     // 2. 收发数据
16     printf("Input: ");
17     char sendBuf[100] = {0};
18     scanf("%s", sendBuf);
19     sockaddr_in sockAddrInfo = {0};
20     sockAddrInfo.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
21     sockAddrInfo.sin_port = htons(2118);
22     sockAddrInfo.sin_family = AF_INET;
23     sendto(cSocket, sendBuf, 100, 0, (sockaddr*)&sockAddrInfo,
24     sizeof(sockAddrInfo));
25     // 3. 关闭套接字
26     closesocket(cSocket);
27     WSACleanup();
28     return 0;
29 }
```

与服务端意义，这里有一个新函数sendto，其语法如下：

```

1  int sendto(
2      SOCKET s,                // 套接字
3      const char FAR *buf,      // 发送的数据
4      int len,                  // 发送的数据长度
5      int flags,                // 指定调用方式的标志，这个我们就直接写0即可
6      const struct sockaddr FAR *to, // sockaddr 结构体，表示发送数据给谁

```

```

7   int tolen                                // sockaddr 结构体长度
8   );

```

5.5 TCP和UDP的比较

如下就是TCP协议和UDP协议的优缺点，我们可以根据实际场景情况，并根据两个协议的优缺点去选择适合当前场景的协议。

协议	优点	缺点
TCP	传输可靠	慢、低效、流程繁琐
UDP	传输效率高	传输不可靠

5.6 实现端对端互相收发

我们已经知道了，在UDP中不存在客户端和服务端着两个概念，所有传输都是端对端的，不会区分，现在我们实现端对端互相收发数据，并且在发出或收到数据内容为close的时候关闭连接：

```

1   int main(int argc, char* argv[])
2   {
3       // 0. 初始化
4       WORD wsVersion = MAKEWORD(2, 2);
5       WSADATA wsaData = {0};
6       WSStartup(wsVersion, &wsaData);
7       // 1. 创建套接字
8       SOCKET cSocket = socket(AF_INET, SOCK_DGRAM, 0);
9       if (SOCKET_ERROR == cSocket) {
10          printf("套接字创建失败!\n" );
11      }
12      else {
13          printf("套接字创建成功!\n" );
14      }
15      // 2. 绑定套接字
16      sockaddr_in sockAddrInfo = {0};    // 初始化
17      sockAddrInfo.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
18      sockAddrInfo.sin_port = htons(2119); // 端口
19      sockAddrInfo.sin_family = AF_INET; // 地址族规范
20
21      int bRes = bind(cSocket, (sockaddr*)&sockAddrInfo,
22                      sizeof(sockAddrInfo));
23
24      if (SOCKET_ERROR == bRes) {
25          printf("绑定失败!\n");
26      }
27      else {
28          printf("绑定成功!\n");

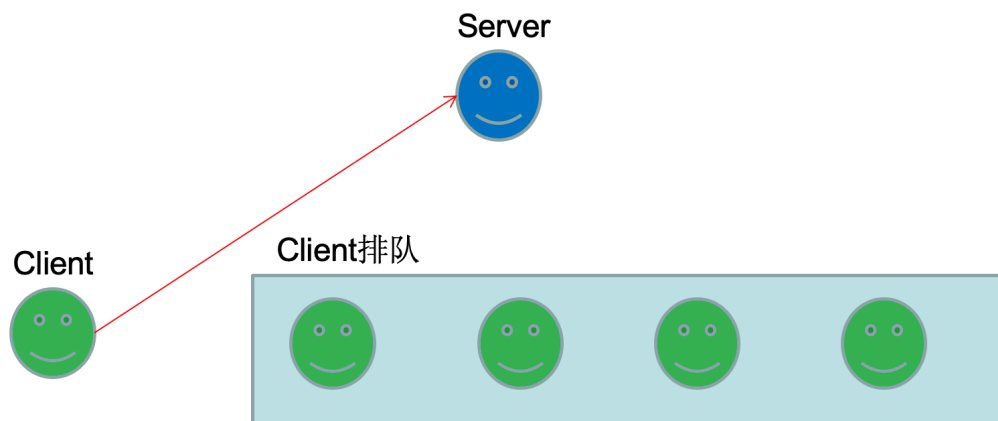
```

```
28     }
29     // 3. 收发数据
30     char recvBuf[100] = {0};
31     char sendBuf[100] = {0};
32     sockaddr sockclient = {0};
33
34     sockaddr_in sockAddrInfoS = {0};
35     sockAddrInfoS.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
36     sockAddrInfoS.sin_port = htons(2118);
37     sockAddrInfoS.sin_family = AF_INET;
38     while (true) {
39         printf("Input: ");
40         gets(sendBuf); // 这里使用了gets函数替换了scanf函数, 是因为scanf函数在获
                        // 取数据的时候遇到了空格, 就不会再去管空格后面的数据了, 相当于截断了
41         sendto(cSocket, sendBuf, strlen(sendBuf)+1, 0,
                (sockaddr*)&sockAddrInfoS, sizeof(sockAddrInfoS));
42         recvfrom(cSocket, recvBuf, sizeof(recvBuf), 0, NULL, NULL);
43         if (strlen(recvBuf) != 0){
44             printf("Recv Data: %s \n", recvBuf);
45         }
46         if ((strcmp(recvBuf, "close")==0) || (strcmp(sendBuf, "close")==0)
47     )
48     {
49         break;
50     }
51     memset(recvBuf, 0, 100);
52 }
53 // 4. 关闭套接字
54 closesocket(cSocket);
55 WSACleanup();
56 return 0;
57 }
```

6 多连接之多线程解决方案

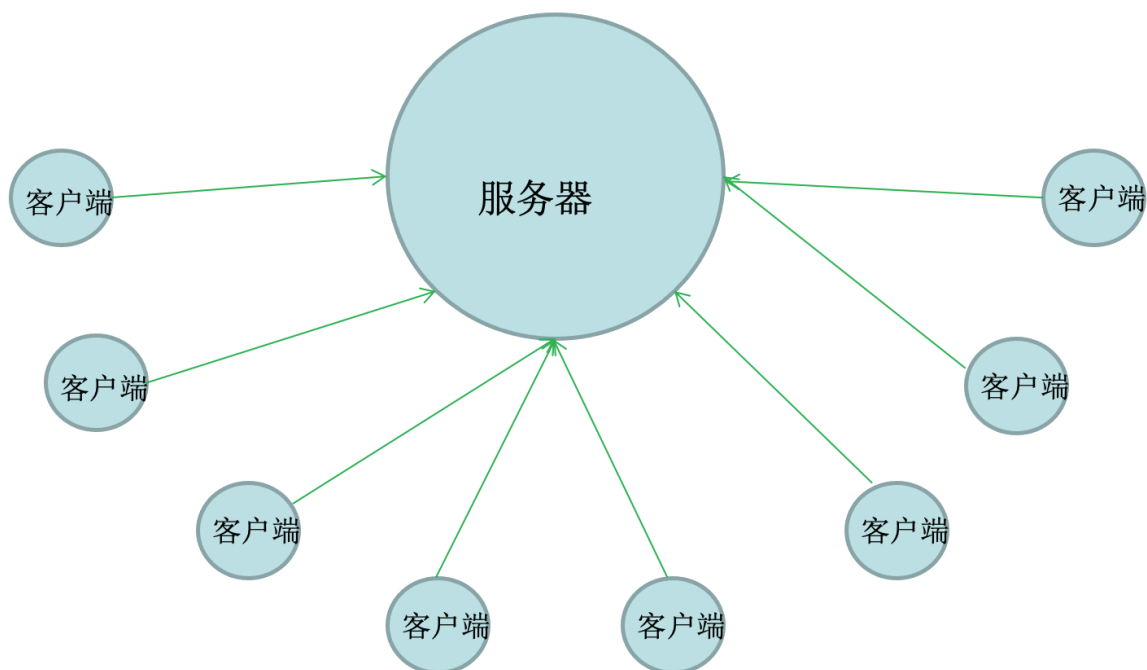
6.1 单连接面临的窘境

单连接如下图所示当一个客户端和服务进行通信时，其它客户端就需要排队，那如果是一个电商网站，你想想一下有人买东西会先看很久，然后再对比，然后再挑选，当他这一系列动作完成之后可能已经几个小时过去了，那么请问你能忍受得了等这么几个小时的队伍吗？这也是当年结面临的窘境。



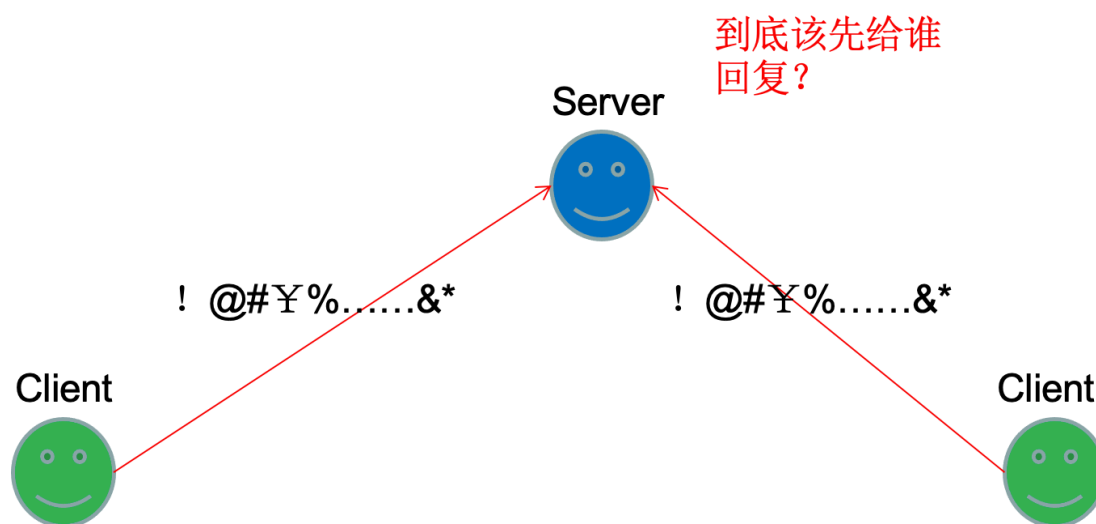
6.2 什么是多连接

多连接就是解决单连接所面临的窘境，使得多个客户端可以同时跟一个服务器进行通信。



6.3 多连接所面临的问题

多连接解决了多客户端与服务器通信排队的问题，但同样也会出现新的问题，当多个客户端连接过来时，服务器应该先给谁回复？这时候就会有一个顺序逻辑的问题。



6.4 多线程解决方案

为了解决多连接所面临的问题，这时候就需要用到多线程的解决方案，为每个客户端连接开一个线程，主线程只管负责监听客户端连接请求，而真正负责通信的任务转交为工作线程。

同样一个东西总会有好也有坏，在这里多线程解决方案优点很明显，但缺点同样也很致命。

优点：整个逻辑非常清楚，编程实现及维护都相对容易；

缺点：占用系统资源太严重，客户端数量上升到一定程度，容易造成系统瘫痪（资源用光了）。

6.4.1 实际代码编写

学过Win32的话应该了解到当我们打开一个进程的时候，会默认启用一个线程（每个进程至少需要一个线程），那么我们可以用这个线程作为主线程，负责监听客户端连接请求，再用之前学习到的**CreateThread**函数创建新的工作线程负责通信（收发信息）。

```

1  DWORD WINAPI ThreadProc(LPVOID lpParameter);
2
3  int main(int argc, char* argv[])
4  {
5      // 0. 初始化
6      WORD wsVersion = MAKEWORD(2, 2);
7      WSADATA wsaData = {0};
8      WSStartup(wsVersion, &wsaData);
9      // 1. 创建套接字
10     SOCKET sSocket = socket(AF_INET, SOCK_STREAM, 0);
11     if (SOCKET_ERROR == sSocket) {
12         printf("套接字创建失败!\n");
13     }
14     else {
15         printf("套接字创建成功!\n");
16     }
17     // 2. 绑定套接字
18     sockaddr_in sockAddrInfo = {0}; // 初始化
19     sockAddrInfo.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
20     sockAddrInfo.sin_port = htons(2118); // 端口
21     sockAddrInfo.sin_family = AF_INET; // 地址族规范
22
23     int bRes = bind(sSocket, (sockaddr*)&sockAddrInfo,
24                     sizeof(sockAddrInfo));
25
26     if (SOCKET_ERROR == bRes) {
27         printf("绑定失败!\n");
28     }
29     else {
30         printf("绑定成功!\n");
31     }
32     // 3. 监听套接字
33     int lRes = listen(sSocket, 1);
34     if (SOCKET_ERROR == lRes) {
35         printf("监听失败!\n");
36     }
37     else {
38         printf("监听成功!\n");
39     }

```

```

40     // 循环
41     while (true) {
42         // 4. 等待连接
43         sockaddr_in acceptSockAddrInfo = {0};    // 初始化
44         int acceptSockAddrLen = sizeof(acceptSockAddrInfo);
45         SOCKET aSocket = accept(sSocket, (sockaddr*)&acceptSockAddrInfo,
46 &acceptSockAddrLen);
47         if (INVALID_SOCKET != aSocket) {
48             HANDLE hThread = CreateThread(NULL, NULL, ThreadProc,
49 (LPVOID)aSocket, 0, NULL);
50             CloseHandle(hThread);
51         }
52     }
53     // 7. 关闭套接字
54     closesocket(sSocket);
55     WSACleanup();
56     return 0;
57 }
58
59 DWORD WINAPI ThreadProc(LPVOID lpParameter) {
60     int ret = 1;
61     SOCKET aSocket = (SOCKET)lpParameter;
62     char buf[100] = {0};
63     // 收发数据
64     do
65     {
66         ret = recv(aSocket, buf, 100, 0);
67         if (strcmp(buf, "close") == 0) {
68             break;
69         }
70         printf("Recv data: %s \n", buf);
71         send(aSocket, buf, strlen(buf)+1, 0);
72         memset(buf, 0, 100);
73     } while (ret != 0);
74
75     // 断开连接(被动)
76     shutdown(aSocket, SD_BOTH);
77     // 关闭套接字
78     closesocket(aSocket);
79     return 0;
80 }

```


<pre>C:\Program Files\Microsoft V 套接字创建成功! 绑定成功! 监听成功! Recv data: 123 Recv data: 456 Recv data: 789 Recv data: bbb Recv data: ccc</pre>	<pre>C:\Program Files\Microsoft Visual Studio\MyProjects\ClientSide\Debug\ClientSide.exe 套接字创建成功! 绑定失败! 与服务器连接成功! Input: 789 Recv data: 789 Input: bbb Recv data: bbb Input:</pre>
<pre>C:\Program Files\Microsoft 套接字创建成功! 绑定失败! 与服务器连接成功! Input: 123 Recv data: 123 Input: close Recv data: Input: aaa Recv data: Input:</pre>	<pre>C:\Program Files\Microsoft Visual Studio\MyProjects\ClientSide\Debug\ClientSide.exe 套接字创建成功! 绑定成功! 与服务器连接成功! Input: 456 Recv data: 456 Input: ccc Recv data: ccc Input: _</pre>

7 多连接之select模型

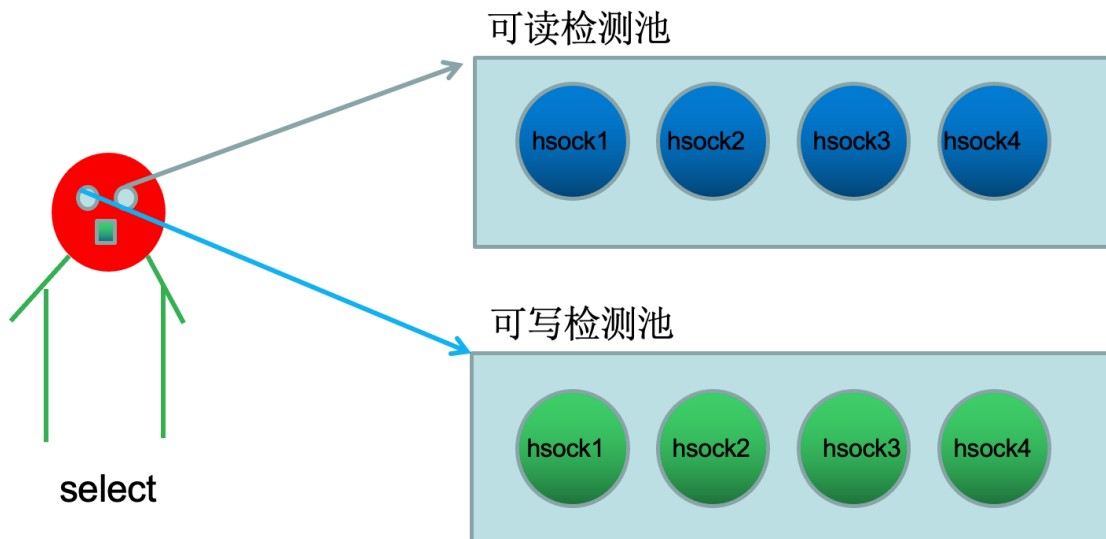
多连接需求使用多线程的方式去满足，但是同样也造成了当客户端连接数量到达一定程度，就会导致占用系统资源太过于严重，所以我们可以使用select模型解决方案处理多连接，并节省系统资源。

7.1 什么是select模型

7.1.1 select模型的本质

select的中文意思是选择，也就表示是从某些东西中去选择自己想要的，如下图所示，有两个地方可供我们选择，也就是可读检测池、可写检测池，在这两个池子里的是我们的套接字句柄（SOCKET）。

从字面意思理解，**select**可从诸多连接中检测出可读的（**accept**函数），也就是有响应的连接；也可以从诸多连接中检测出可写的（**recv**、**send**函数），也就是可以发送消息的连接。



7.1.2 select模型逻辑

select模型逻辑步骤如下：

1. 将所有的socket装进一个数组中
2. 通过select函数遍历socket数组，取出有响应（可读、可写）的socket放进另一个数组
3. 对存入有响应的socket数组处理

7.1.3 select函数

select模型实际上就是调用了select这个函数，其语法如下：

```

1  int select (
2      int nfd, // 填0
3      fd_set *readfds, // 输入输出共用参数，检查/输出可读的socket
4      fd_set *writefds, // 输入输出共用参数，检查/输出可写的socket

```

```
5     fd_set *exceptfds, // 输入输出共用参数, 检查/输出socket上的异常错误
6     const struct timeval *timeout // struct timeval结构体, 最大等待时间, 当
7     socket没有响应时, 要等待的时间
    );
```

该函数的作用就是检测出可读、可写的socket，它的返回值有这些：

- 1 0：指定等待时间内没有socket响应，continue进行下一次等待
- 2 大于0：有socket响应
- 3 SOCKET_ERROR(宏)：发送错误

除了需要了解这个函数以外，我们可以看见其成员是一个fd_set结构体，我们还需要了解一下这个结构体和几个常用的宏。

7.1.4 fd_set结构体

fd_set是用来装socket的结构体，默认情况下，它可以装64个socket：

```
1 #ifndef FD_SETSIZE
2 #define FD_SETSIZE 64
3 #endif /* FD_SETSIZE */
4
5 typedef struct fd_set {
6     u_int fd_count; // 统计的数量
7     SOCKET fd_array[FD_SETSIZE]; // 存放SOCKET的数组
8 } fd_set;
```

如果你想装更多的socket，可以通过在winsock2.h头文件前声明宏，给一个更大的值：

```
1 #define FD_SETSIZE 128
2 #include <WinSock2.h>
```

select的原理就是不停的检测，越多的socket效率就越低，延迟就越大，所以说这个模型只适合小用户量去访问，应该自己选择一个合适的大小。

7.1.5 四个操作fd_set的操作宏

操作宏	作用	代码
FD_ZERO	将客户端socket集合清零	FD_ZERO(&clientSockets);
FD_SET	添加一个socket(超过默认值大小不再处理)	FD_SET(socketListen, &clientSockets);

操作宏	作用	代码
FD_CLR	从集合中删除指定的socket一定要close, 手动释放	<pre>FD_CLR(socketListen, &clientSockets); closesocket(socketListen);</pre>
FD_ISSET	查询socket是否在集合中, 不存在返回0, 存在返回非0	<pre>FD_ISSET(socketListen, &clientSockets);</pre>

7.2 实际代码编写

按照select模型逻辑编写代码即可：

```

1  int main(int argc, char* argv[])
2  {
3      // 0. 初始化
4      WORD wsVersion = MAKEWORD(2, 2);
5      WSADATA wsaData = {0};
6      WSStartup(wsVersion, &wsaData);
7      // 1. 创建套接字
8      SOCKET sSocket = socket(AF_INET, SOCK_STREAM, 0);
9      if (SOCKET_ERROR == sSocket) {
10         printf("套接字创建失败!\n");
11     }
12     else {
13         printf("套接字创建成功!\n");
14     }
15     // 2. 绑定套接字
16     sockaddr_in sockAddrInfo = {0};    // 初始化
17     sockAddrInfo.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
18     sockAddrInfo.sin_port = htons(2118); // 端口
19     sockAddrInfo.sin_family = AF_INET;  // 地址族规范
20
21     int bRes = bind(sSocket, (sockaddr*)&sockAddrInfo,
sizeof(sockAddrInfo));
22
23     if (SOCKET_ERROR == bRes) {
24         printf("绑定失败!\n");
25     }
26     else {
27         printf("绑定成功!\n");
28     }
29     // 3. 监听套接字
30     int lRes = listen(sSocket, 5);
31     if (SOCKET_ERROR == lRes) {
32         printf("监听失败!\n");

```

```

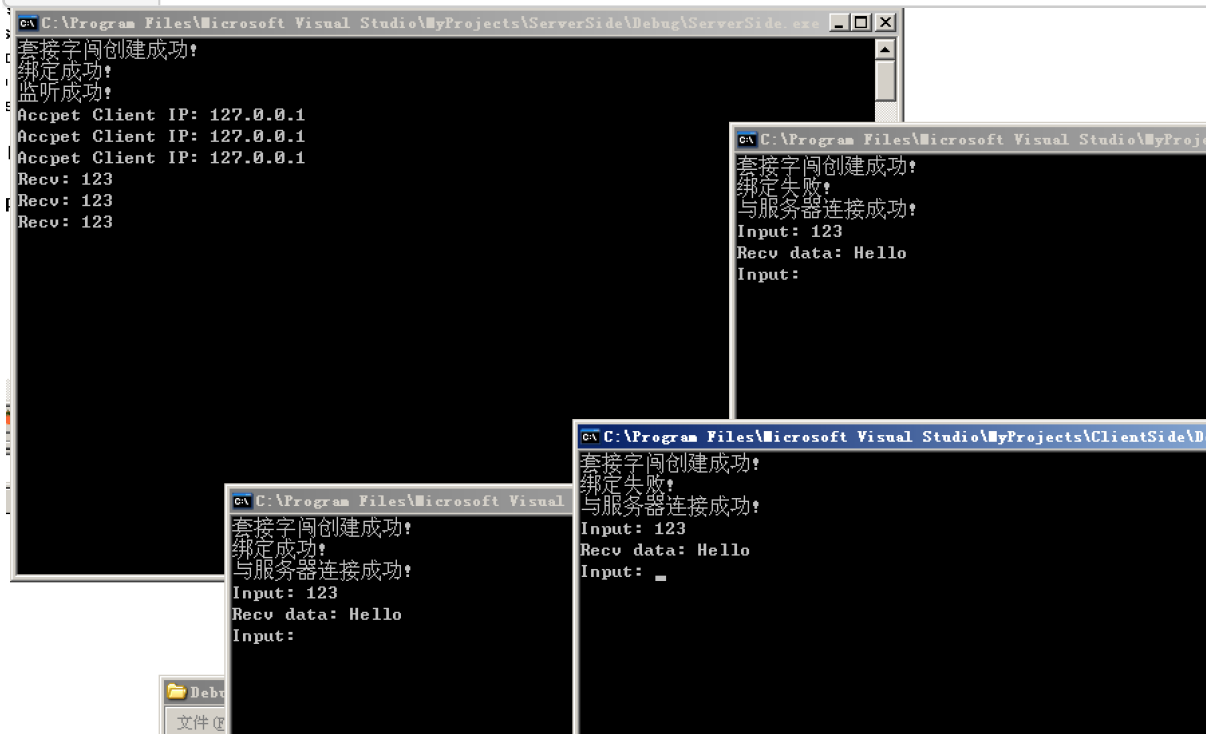
33     }
34     else {
35         printf("监听成功!\n");
36     }
37
38     fd_set fdSocket; // 定义
39     FD_ZERO(&fdSocket); // 初始化
40     FD_SET(sSocket, &fdSocket); // 将当前服务器创建的socket放入集合中
41
42     while (true) {
43         fd_set readfds = fdSocket; // 定义可读的集合
44         fd_set writefds = fdSocket; // 定义可写的集合
45         // fd_set exceptfds = fdSocket;
46
47         // 获取select函数的返回值
48         int iRes = select(0, &readfds, &writefds, NULL, NULL);
49         // 如果返回值大于0则说明不存在无响应、错误的情况,继续向下
50         if (iRes > 0) {
51             // 遍历可写集合,给每个socket发送Hello
52             for (u_int i = 0; i < writefds.fd_count; i++) {
53                 send(readfds.fd_array[i], "Hello", 6, 0);
54             }
55
56             // 遍历可读集合
57             for (i = 0; i < readfds.fd_count; i++) {
58                 // 如果socket为当前服务器创建的socket则进入accept等待消息
59                 if (readfds.fd_array[i] == sSocket) {
60                     sockaddr_in s = {0};
61                     int l = sizeof(s);
62                     SOCKET aSocket = accept(sSocket, (sockaddr*)&s, &l);
63                     if (INVALID_SOCKET == aSocket)
64                     {
65                         continue;
66                     }
67                     FD_SET(aSocket, &fdSocket);
68                     // inet_ntoa获取IP
69                     printf("Accpet Client IP: %s \n",
inet_ntoa(s.sin_addr));
70                     // 如果不是,则进入接收消息
71                     } else {
72                         char buf[100] = {0};
73                         int iRecv = recv(readfds.fd_array[i], buf,
sizeof(buf), 0);
74
75                         // 判断接收消息的返回值,大于0则表示接收成功。
76                         if (iRecv > 0) {
77                             printf("Recv: %s \n", buf);
78                             // 否则就关闭连接、关闭套接字
79                         } else {
80                             SOCKET tSocket = readfds.fd_array[i];
81                             FD_CLR(tSocket, &fdSocket);
82                             shutdown(tSocket, SD_BOTH);
83                             closesocket(tSocket);
84                         }
85                     }
86                 }
87             }
88         }
89     }
90 }

```

```

84         }
85     }
86     } else {
87         continue;
88     }
89 }
90
91 closesocket(sSocket);
92
93 WSACleanup();
94 return 0;
95 }

```



8 遍历物理网卡

如果你要抓网络的包，首先你要知道你要处理的是哪个网卡，那么你就要知道这个网卡的相关信息，所以本章就遍历物理网卡并获取相关信息。

我们都知道在Windows操作系统上，我们没有办法直接去操作硬件层的东西，因为这就涉及到驱动内核了，我们想要去用的话就需要通过Windows系统封装好的库，在这里我们使用到的就WinPcap库。

8.1 WinPcap介绍

WinPcap(Windows Packet Capture)是Windows平台下一个免费的、公共的库。开发WinPcap这个项目的目的在于为Win32 App提供访问网络底层的能力。

8.1.1 常用功能

WinPcap常用的功能如下所示：

1. 捕获原始数据包，无论它是发往某台机器的，还是在其他设备(共享媒介)上进行交换的；
2. 在数据包发送给某应用程序前，根据用户指定的规则过滤数据包；
3. 将原始数据包通过网络发送出去；
4. 收集并统计网络流量信息。

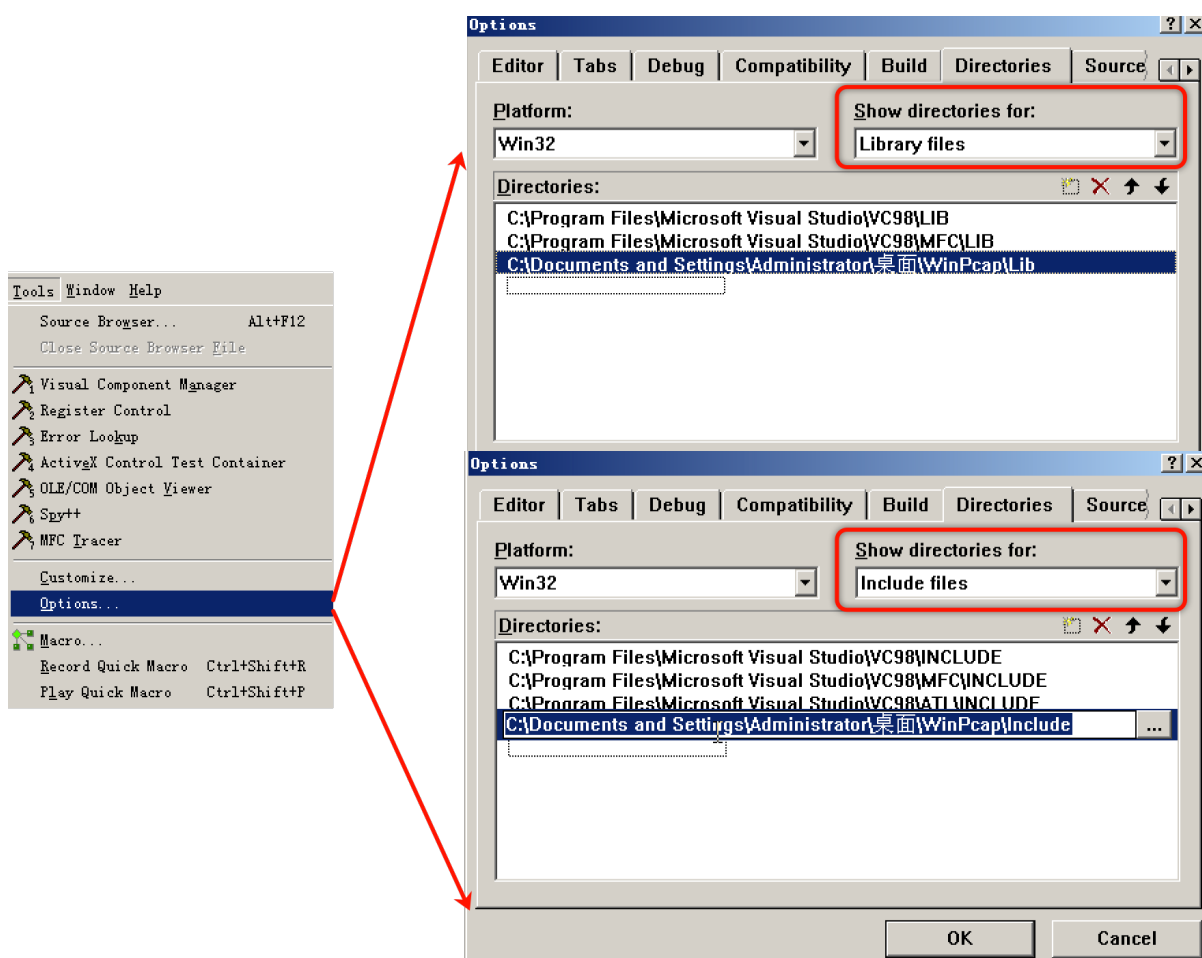
8.1.2 下载与配置

WinPcap库并不是Windows系统自带的，而是由外部开发者去维护的，我们可以从这个地址去下载：<https://www.winpcap.org/archive/4.0.1-WpdPack.zip>，如果你是基于VC6去使用的话，选择>=4.0.1的版本。

下载下来之后是一个压缩包，主要的就是Lib和Include这两个文件夹：



将这两个文件夹存放好，在VC6中打开Tools-Options，填入两个文件夹所在路径：



8.2 实际代码编写

接下来我们就要使用这个扩展库去遍历获取物理网卡的相关信息，首先我们需要包含头文件和库文件：

```

1  #include <WINSOCK2.H>
2  #include <pcap.h>
3  #pragma comment(lib, "ws2_32.lib")
4  #pragma comment(lib, "wpcap.lib")

```

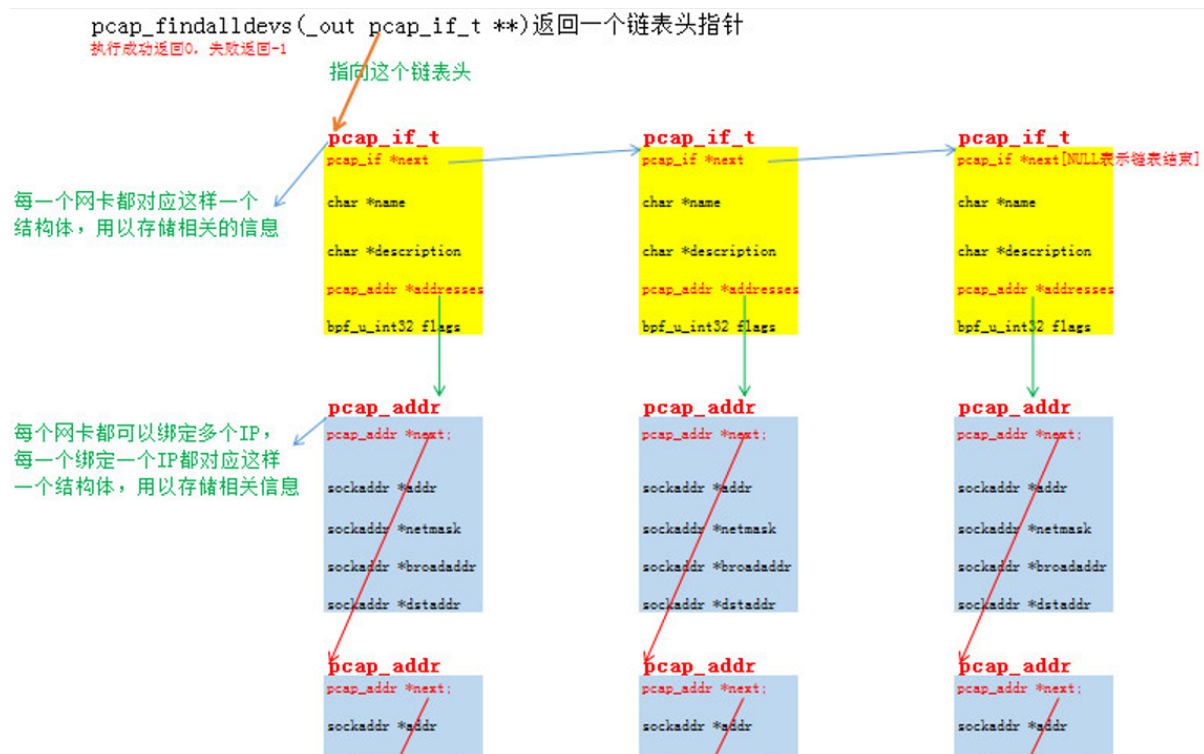
接着我们需要通过封装好的函数pcap_findalldevs去获取物理网卡信息，其语法如下（参考官方手册：https://www.winpcap.org/docs/docs_412/html/）：

```

1  int pcap_findalldevs (
2      pcap_if_t ** alldevsp, // 输出参数，这是一个二级指针，它实际上是一个链表
3      char * errbuf // 输出参数，执行失败的错误信息存放的缓冲区
4  ); // 执行成功返回0，失败返回-1

```

如下图所示就是这个结构体，我们可以看出，实际上他是有俩个成员都是链表，一个是每个网卡对应的next成员指针，一个是每个网卡对应的addresses成员指针的next成员：



那么我们可以根据这个结构去获取网卡的信息：

```

1  int main(int argc, char* argv[])
2  {
3      pcap_if_t* alldevsp = NULL;
4      pcap_if_t* tmpdevsp = NULL;
5      char errbuf[PCAP_ERRBUF_SIZE] = {0}; // 错误信息
6      int iRet = pcap_findalldevs(&alldevsp, errbuf); // 获取返回值
7      if (iRet == 0) { // 判断是否为0
8          tmpdevsp = alldevsp;
9          // 遍历网卡
10         do {
11             printf("Name: %s\nDesc: %s\n", tmpdevsp->name, tmpdevsp-
>description);
12             pcap_addr* tmpaddr = tmpdevsp->addresses;
13             // 遍历地址
14             do {
15                 sockaddr_in* ipaddr = (sockaddr_in*)tmpaddr->addr;
16                 printf("IP: %s\n", inet_ntoa(ipaddr->sin_addr));
17             } while (tmpaddr = tmpaddr->next);
18             printf("=====\n\n");
19         } while (tmpdevsp = tmpdevsp->next);
20     } else {
21         printf("pcap_findalldevs error: %s \n", errbuf);
22     }
23     pcap_freealldevs(alldevsp); // 释放资源
24     return 0;

```

25

}

9 原始数据包的获取与打印

原始数据包就是适配器（物理网卡、虚拟网卡都可以称为适配器）接收到，没有经过任何处理的包。

9.1 获取适配器上的数据包

获取适配器上的数据包，一共有如下几个步骤：

- 1 1. 遍历适配器，找到需要捕获的适配器(pcap_findalldevs函数)
- 2 2. 打开指定的适配器(pcap_open函数)
- 3 3. 关闭适配器，释放资源(pcap_freealldevs函数)
- 4 4. 设置回调函数开始捕获数据包(pcap_loop函数)
- 5 5. 编写回调函数进行数据的接收与处理

pcap_open函数语法：

```

1 pcap_t* pcap_open(
2     const char *source, // 要打开的适配器名字
3     int snaplen, // 捕获包的长度（TCP包的最大长度是1460字节，UDP包则是65535字节）
4     int flags, // 这里就是设置你的适配器模式，我们需要抓包的话就要设置成混合模式，写1
// 或者true都可以
5     int read_timeout, // 读取超时时间，这里可以直接写NULL，不管它
6     struct pcap_rmtauth *auth, // 认证，pcap_open
7     char *errbuf // 错误信息
8 );

```

pcap_loop函数语法：

```

1 int pcap_loop(
2     pcap_t *, // pcap_open返回的指针
3     int, // 填0
4     pcap_handler, // 回调函数
5     u_char * // 填NULL
6 );
7
8 int pcap_loop (
9     pcap_t * p, // pcap_open返回的指针
10    int cnt, // 填0
11    pcap_handler callback, // 回调函数
12    u_char * user // 用户自定义内容
13 );

```

回调函数原型：

```

1 typedef void(* pcap_handler)(u_char *user, const struct pcap_pkthdr
2     *pkt_header, const u_char *pkt_data)

```

```

3 void pcap_handler(
4     u_char *user, // 用户定义的参数, 包含了捕获会话的状态, 它对应于 pcap_dispatch
    和 pcap_loop 函数的user参数
5     const struct pcap_pkthdr *pkt_header, // 结构体指针, 捕获驱动与数据包相关联
    的头 (不是协议头)
6     const u_char *pkt_data // 指针, 指向数据包的数据, 包括协议头。
7 );

```

这里面有一个结构体, 我们需要看一下它的成员分别是什么:

```

1 struct pcap_pkthdr {
2     struct timeval ts; // 数据包捕获的时间戳
3     bpf_u_int32 caplen; // 捕获到包的长度
4     bpf_u_int32 len; // 这个包的长度
5 };

```

第一个成员是一个结构体timeval, 自行了解, 这里不再赘述, 第二与第三个成员, 理论应该是一样的大小, 但有可能捕获到数据包的长度与实际的长度有出入 (数据丢失)。

9.2 适配器的混合模式

通常, 适配器(物理网卡)有多种工作模式, 设置为混合模式之后, 可以接收所有流经当前网卡的数据包, 即使不是发给自己的。

9.3 实际代码编写

在实际编写代码之前, 我们是要在包含pcap.h头文件之前定一个宏:

```

1 #define HAVE_REMOTE
2 #include <WINSOCK2.H>
3 #include <pcap.h>
4 #pragma comment(lib, "ws2_32.lib")
5 #pragma comment(lib, "wpcap.lib")

```

这是因为我们即将要使用到pcap_open这个函数, 只有定义了这个宏, 才能包含定义了这个函数的头文件, 这个我们可以从pcap.h头文件中看到:

```

335 #ifdef HAVE_REMOTE
336 /* Includes most of the public stuff that is needed
   for the remote capture */
337 #include "remote-ext.h"
338 #endif /* HAVE_REMOTE */

```



```

414 pcap_t *pcap_open(const char *source, int snaplen,
   int flags, int read_timeout, struct pcap_rmtauth
   *auth, char *errbuf);

```

接着，我们就按照顺序逐步去编写代码即可：

```

1 void Mypcap_handler(u_char *user, const struct pcap_pkthdr *pkt_header,
2 const u_char *pkt_data);
3
4 int main(int argc, char* argv[])
5 {
6     pcap_if_t* alldevsp = NULL;
7     pcap_if_t* tmpdevsp = NULL;
8     char errbuf[PCAP_ERRBUF_SIZE] = {0}; // 错误信息
9     // 1. 遍历适配器
10    int iRet = pcap_findalldevs(&alldevsp, errbuf); // 获取返回值
11    if (iRet != 0) {
12        printf("pcap_findalldevs error: %s \n", errbuf);
13    }
14    tmpdevsp = alldevsp;
15    do
16    {
17        printf("Name: %s \nDesc: %s \n=====\n\n", tmpdevsp-
18        >name, tmpdevsp->description);
19        } while (tmpdevsp = tmpdevsp->next);
20
21    // 2. 打开指定的适配器
22    pcap_t* pcap = pcap_open("\\Device\\
23    \\NPF_{C7C05FAA-6043-4EB3-9059-329655AC6FB0}", 65535, 1, NULL, NULL,
24    errbuf);
25
26    if (!pcap) {
27        printf("pcap_open error: %s \n", errbuf);
28    }
29
30    // 3. 关闭适配器
31    pcap_freealldevs(alldevsp);
32
33    // 4. 捕获数据包

```

```

30     pcap_loop(pcap, 0, Mypcap_handler, NULL);
31     return 0;
32 }
33
34 void Mypcap_handler(u_char *user, const struct pcap_pkthdr *pkt_header,
35 const u_char *pkt_data) {
36     // 5. 编写回调函数
37     struct tm *ltime;
38     char timestr[16] = {0};
39     time_t local_tv_sec;
40
41     // 将时间戳转换成可识别的格式
42     local_tv_sec = pkt_header->ts.tv_sec;
43     ltime = localtime(&local_tv_sec);
44     strftime(timestr, sizeof timestr, "%H:%M:%S", ltime);
45     printf("Time: %s, Caplen: %d, Len: %d \n", timestr, pkt_header->caplen, pkt_header->len);
46
47     // 根据分析,我们发现数据包的前六个字节为目地的MAC地址,第七个字节到第12个字节则为源的MAC地址,所以在这里输出
48     u_char dstMac[6] = {0};
49     for(int i = 0; i < 6; i++) {
50         dstMac[i] = *(pkt_data+i+0);
51     }
52     printf("Dst Mac: %02x:%02x:%02x:%02x:%02x:%02x \n", dstMac[0], dstMac[1], dstMac[2], dstMac[3], dstMac[4], dstMac[5]);
53
54     u_char srcMac[6] = {0};
55     for(i = 0; i < 6; i++) {
56         srcMac[i] = *(pkt_data+i+6);
57     }
58     printf("Src Mac: %02x:%02x:%02x:%02x:%02x:%02x \n", srcMac[0], srcMac[1], srcMac[2], srcMac[3], srcMac[4], srcMac[5]);
59
60     // pkt_data 给的是存储数据的内存首地址,我们根据实际捕获的长度去遍历即可获取完整数据
61     printf("\nData:\n-----\n");
62     for(i = 0; i < pkt_header->caplen; i++) {
63         printf("%x", pkt_data[i]);
64     }
65     printf("\n-----\n");
66 }

```

The image shows a Visual Studio IDE with a C++ project named 'CaptrueAdapter'. The code in 'CaptrueAdapter.cpp' is as follows:

```

printf("Src Mac: ",
for(int i = 0; i < 6; i++) {
    // 02表示不足两位前面补0输出, 如果超过两位, 则以实际输出
    printf("%02x",
}

printf("\nSrc Mac:

for(i = 6; i < 12;
    printf("%02x",
}

// pkt_data 给的是存
printf("\nData:\n--
for(i = 0; i < pkt
    printf("%x", pk
}
printf("\n-----

```

The output window shows the following information:

```

Name: \Device\NPF_{C7C05FAA-6043-4EB3-9059-329655AC6FB0}
Desc: VMware Accelerated AMD PCNet Adapter (Microsoft's Packet Scheduler)
=====
Name: \Device\NPF_{23F0D81D-A2BA-4CE8-9BEB-7678174552E8}
Desc: Microsoft
=====
Time: 21:48:33, Caplen: 73, Len: 73
Dst Mac: 3e22fb3c7064
Src Mac: 000c29c03f57
Data:
-----
3e22fb3c70640c29c03f578045003baf63008011d326ac10b05ac10b0

```

The Windows command prompt shows the following network configuration:

```

Physical Address. . . . . : 00-0C-29-C0-3F-57
Dhcp Enabled. . . . . : Yes
Autoconfiguration Enabled . . . . : Yes
IP Address. . . . . : 172.16.176.5
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 172.16.176.1

```


10 MAC帧结构与MAC包的解析

10.1 MAC帧结构

在上一章中，我们用代码分别输出了原始数据包中的目标主机的MAC地址和源主机的MAC地址，这分别占6字节，还有2字节表示类型，加起来就是14字节，我们称之为MAC帧结构：

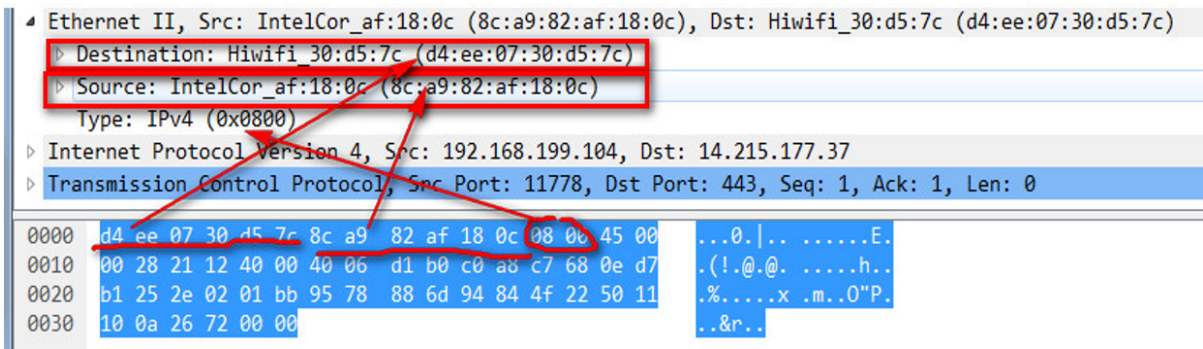


MAC层是数据链路层的两个子层之一，如上图中的这2个字节则表示当前数据包属于哪一种网络层协议的类型（例如：IP、ARP）。

我们可以通过获取适配器上的原始数据包，在最开始的位置找到MAC帧结构，这是因为我们可以直接从网络模型看出网络协议的层级，这种数据包应该是按照层级一层一层的追加的，所以MAC帧结构应该在最开始的位置。

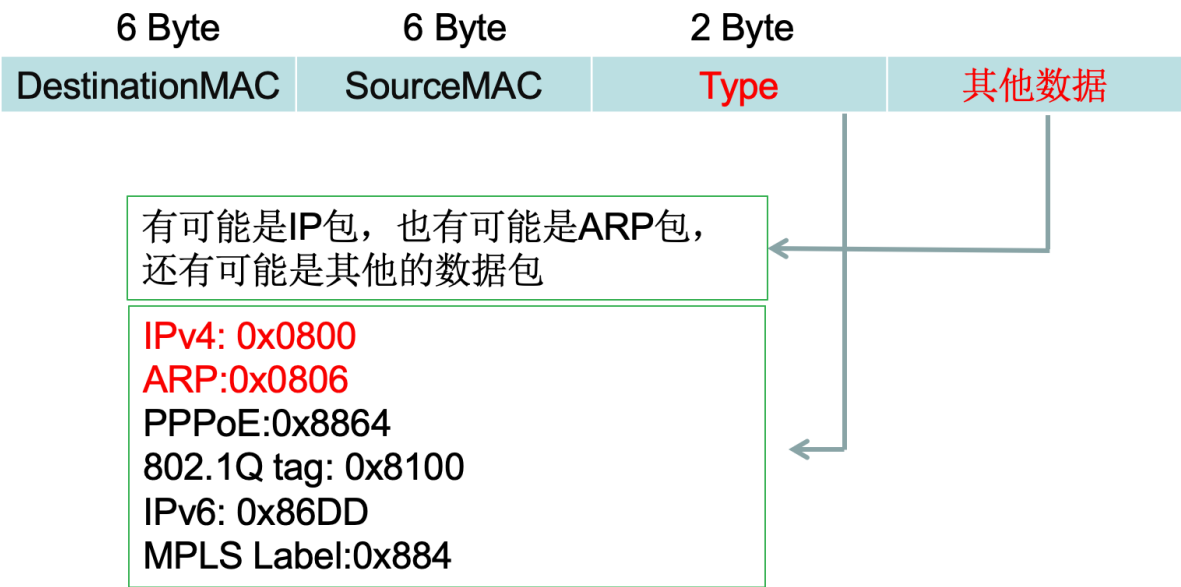
10.2 Wireshark抓包解析

我们可以通过Wireshark去抓包，随便选择一个包展开就可以看到这样的一个结构：



11 IP帧结构与IP包的解析

接下来我们需要定位IP帧结构，而我们想定位这个结构的话，按照网络协议的层级应该先从MAC帧说起：



如上图所示，我们的IP帧就在其他数据中，我们根据MAC帧的最后2字节推断出其他数据中的帧对应什么帧。

11.1 IP帧结构

如下图就是IP帧结构，图中多次出现的位，表示的是BIT位：

4位版本	4位首部长度	8位服务类型 (TOS)	16位总长度字节数	
16位标识			3位标志	13位段偏移
8位生存时间 (TTL)		8位协议	16位首部校验和	
32位源IP地址				
32位目的IP地址				
选项 (如果有)				
数据...				

其分别表示如下：

1. **版本**：版本占了4位，用来表示该协议采用的是那一个版本的IP，相同版本的IP才能进行通信，一般此处的值为4，表示IPv4；
2. **首部头长度**：该字段用四位表示，表示整个IP报文的头的长度，这里的长度表示有多少个单位，1个单位是4字节，如长度为1则表示该IP报文的头大小为4字节；它的范围即二进制数0000-1111（十进制数0-15），其中一个最小长度为0字节，最大长度为60字节，一般来说此处的值为0101，表示头长度为20字节；
3. **服务类型 (TOS)**：该字段用8位表示，该字段一般情况下不使用；
4. **总长度字节数**：该字段表示整个IP报文的长度，这里的长度也表示有多少个单位，1个单位是1字节，能表示的最大字节为 $2^{16}-1=65535$ 字节，不过由于链路层的MTU限制，超过1480字节后就会被分段（以太网MTU为1500的情况下，除去20字节的包头）；
5. **标识**：该字段是IP软件实现的时候自动产生的，该字段的目的是为了接受方的按序接受而设置的，而是在IP分段以后，用来标识同一段分段的，方便IP分段的重组；
6. **标志**：该字段是与IP分段有关的，其中有三位，但只有两位是有效的，分别为MF、DF、MF；MF表示后面是否还有分段，为1时，表示后面还有分段；DF表示是否能分段，为0表示可以分段；
7. **段偏移**：该字段是与IP分段后，相应的IP段在总的IP段的位置；
8. **生存时间 (TTL)**：该字段表示生存周期，该值占8位，IP分段每经过一个路由器该值减一，它的出现是为了防止路由环路，浪费带宽的问题，Window系统默认为128；

9. **协议**：该值表示上层的协议，占8位，其中1表示ICMP、2表示IGMP、6表示TCP、17表示UDP、89表示OSPF；
10. **首部校验和**：该值是对整个数据包的包头进行的校验，占16位；
11. **源地址和目的地址**：表示发送IP段的源和目的IP，分别占32位；
12. **可选的部分**：一些特殊的要求会加在这个部分，一般情况下是不会有这个字段的；
13. 数据。

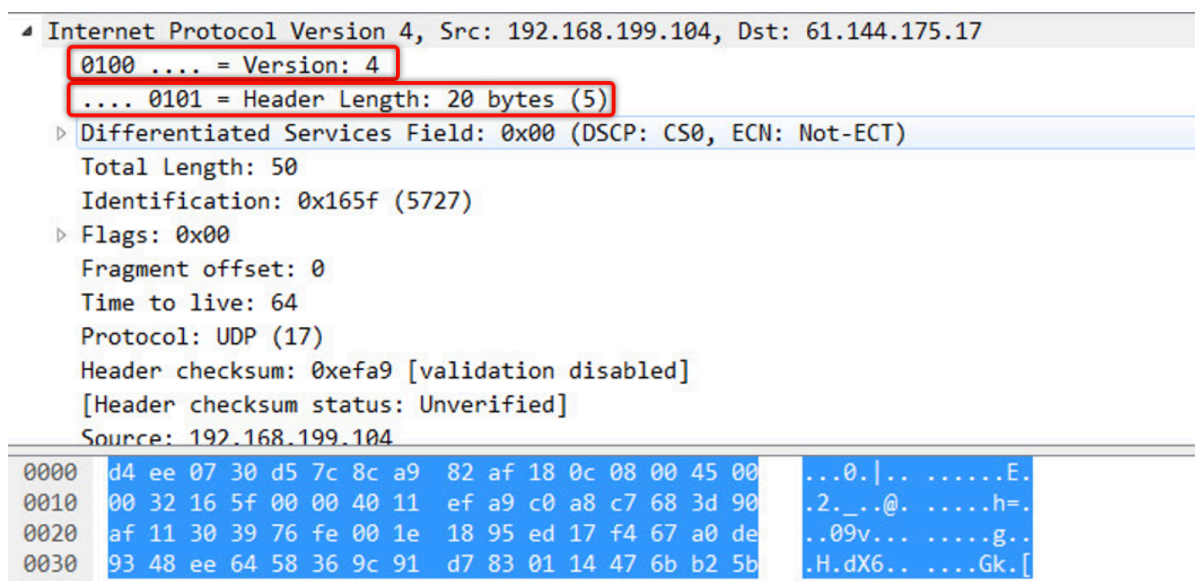
大概了解一下即可，我们主要关注几个重要字段：

1. 版本
2. 首部长度
3. 总长度
4. 协议
5. 源IP
6. 目的IP

注意：网络传输的字节序和你主机本身的字节序是不一样的，所以在转换的时候需要注意一下。

11.2 Wireshark抓包解析

我们可以通过Wireshark去抓包，随便选择一个包展开就可以看到这样的一个结构：



同样，我们可以根据对应的位，编写程序去解析。

11.3 实际代码编写

实际编写代码如下，在编写代码的过程中不要忘记偏移量+14再获取IP报文的信息：

```

1 void Mypcap_handler(u_char *user, const struct pcap_pkthdr *pkt_header,
2   const u_char *pkt_data);
3
4 int main(int argc, char* argv[])
5 {

```

```

5     pcap_if_t* alldevsp = NULL;
6     pcap_if_t* tmpdevsp = NULL;
7     char errbuf[PCAP_ERRBUF_SIZE] = {0}; // 错误信息
8     // 1. 遍历适配器
9     int iRet = pcap_findalldevs(&alldevsp, errbuf); // 获取返回值
10    if (iRet != 0) {
11        printf("pcap_findalldevs error: %s \n", errbuf);
12    }
13    tmpdevsp = alldevsp;
14    do
15    {
16        printf("Name: %s \nDesc: %s \n===== \n\n", tmpdevsp-
>name, tmpdevsp->description);
17        while (tmpdevsp = tmpdevsp->next);
18
19        // 2. 打开指定的适配器
20        pcap_t* pcap = pcap_open("\\Device\
\NPF_{C7C05FAA-6043-4EB3-9059-329655AC6FB0}", 65535, 1, NULL, NULL,
errbuf);
21
22        if (!pcap) {
23            printf("pcap_open error: %s \n", errbuf);
24        }
25
26        // 3. 关闭适配器
27        pcap_freealldevs(alldevsp);
28
29        // 4. 捕获数据包
30        pcap_loop(pcap, 0, Mypcap_handler, NULL);
31        return 0;
32    }
33
34    char* MyStrCpy(char* oStr, int index, int number) {
35        char* tmpStr = oStr;
36        char* resStr = (char*)malloc(number);
37        for (int i = 0; i < number; i++) {
38            *(resStr+i) = *(tmpStr+(index+i));
39        }
40        return resStr;
41    }
42
43    void Mypcap_handler(u_char *user, const struct pcap_pkthdr *pkt_header,
const u_char *pkt_data) {
44        // 5. 编写回调函数
45        struct tm *ltime;
46        char timestr[16] = {0};
47        time_t local_tv_sec;
48
49        // 将时间戳转换成可识别的格式
50        local_tv_sec = pkt_header->ts.tv_sec;
51        ltime = localtime(&local_tv_sec);
52        strftime(timestr, sizeof timestr, "%H:%M:%S", ltime);

```

```

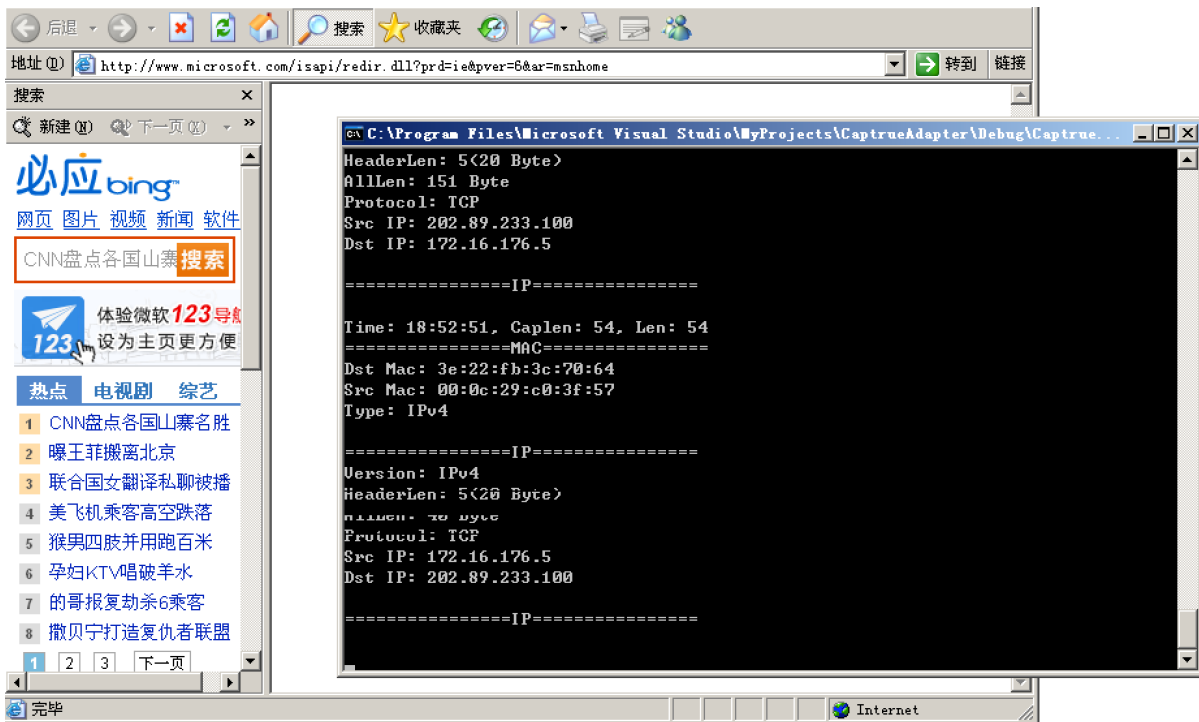
53     printf("Time: %s, Caplen: %d, Len: %d \n", timestr, pkt_header->caplen, pkt_header->len);
54
55     printf("====MAC=====\n");
56
57     u_char dstMac[6] = {0};
58     for(int i = 0; i < 6; i++) {
59         dstMac[i] = *(pkt_data+i+0);
60     }
61     printf("Dst Mac: %02x:%02x:%02x:%02x:%02x:%02x \n", dstMac[0], dstMac[1], dstMac[2], dstMac[3], dstMac[4], dstMac[5]);
62
63
64     u_char srcMac[6] = {0};
65     for(i = 0; i < 6; i++) {
66         srcMac[i] = *(pkt_data+i+6);
67     }
68     printf("Src Mac: %02x:%02x:%02x:%02x:%02x:%02x \n", srcMac[0], srcMac[1], srcMac[2], srcMac[3], srcMac[4], srcMac[5]);
69
70     u_char type[2] = {0};
71
72     for(i = 0; i < 2; i++) {
73         type[i] = *(pkt_data+i+12);
74     }
75
76     printf("Type: ");
77     if (type[0] == 0x08) {
78         if (type[1] == 0x00) {
79             printf("IPv4\n");
80             printf("\n====IP=====\n");
81             u_char versionAndHeaderLen = *(pkt_data+14);
82             u_char version = versionAndHeaderLen >> 4; // 版本
83             u_char headerLen = versionAndHeaderLen << 4;
84             headerLen = headerLen >> 4; // 首部长度
85             u_char realHeaderLen = headerLen * 4;
86
87             if (version == 0x04) {
88                 printf("Version: IPv4 \n");
89             }
90             printf("HeaderLen: %d(%d Byte) \n", headerLen, realHeaderLen);
91
92             u_char allLen[2] = {0};
93             for (i = 0; i < 2; i++) {
94                 allLen[i] = *(pkt_data+i+14+2);
95             }
96             // 手动字节序的转换
97             u_short resLen = 0;
98             u_char* tmpLen = (u_char*)&resLen;
99             tmpLen[0] = allLen[1];
100            tmpLen[1] = allLen[0];
101            // 基于函数转换
102            // u_short resLen = ntohs(*((u_short*)allLen));

```

```

103         printf("AllLen: %d Byte \n", resLen);
104
105         u_char protocol = *(pkt_data+14+9);
106         switch (protocol) {
107             case 1:
108                 printf("Protocol: ICMP \n");
109                 break;
110             case 2:
111                 printf("Protocol: IGMP \n");
112                 break;
113             case 6:
114                 printf("Protocol: TCP \n");
115                 break;
116             case 17:
117                 printf("Protocol: UDP \n");
118                 break;
119             case 89:
120                 printf("Protocol: OSPF \n");
121                 break;
122         }
123
124         u_char srcIP[4] = {0};
125         for (i = 0; i < 4; i++) {
126             srcIP[i] = *(pkt_data+i+14+12);
127         }
128         printf("Src IP: %d.%d.%d.%d\n", srcIP[0], srcIP[1], srcIP[2],
srcIP[3]);
129
130         u_char dstIP[4] = {0};
131         for (i = 0; i < 4; i++) {
132             dstIP[i] = *(pkt_data+i+14+16);
133         }
134         printf("Dst IP: %d.%d.%d.%d \n", dstIP[0], dstIP[1], dstIP[2],
dstIP[3]);
135
136         printf("\n=====IP=====\\n\\n");
137         } else if (type[1] == 0x06) {
138             printf("ARP\\n");
139         }
140     }

```



12 TCP帧结构与TCP包的解析

我们去了解IP包的时候，是从MAC包出发的，那么同样我们了解TCP包也需要去从IP包出发，首先我们知道整个IP包包含的数据中就有TCP/UDP包，想知道这个数据是属于什么协议，就要通过IP帧结构的协议字段，要知道TCP/UDP包数据的大小，就要通过总长度减去首部长度获取。

4位版本	4位首部长度	8位服务类型 (TOS)	16位总长度字节数	
16位标识			3位标志	13位段偏移
8位生存时间 (TTL)	8位协议		16位首部校验和	
32位源IP地址				
32位目的IP地址				
选项 (如果有)				
数据...				

12.1 TCP帧结构

TCP帧结构如下图所示：

0	15	16	31						
源端口 (source port)		目的端口 (destination port)							
序列号(sequence number)									
确认号 (acknowledgement number)									
偏移 (data offset)	保留 (reserved)	URG	ACK	PSH	RST	SYN	FIN	窗口 (windows)	
校验和 (checksum)				紧急指针 (urgent pointer)					
选项 (option)						填充 (Padding)			
数据 (data)									

这里出现了很多，我们没有见过的词，我们来了解一下几个常用的：

1. 源端口：发送方应用程序对应的端口；
2. 目的端口：接收方应用程序对应的端口；
3. 序列号：这个我们已经在之前了解过了，就是我们所说的seq；
4. 确认号：这个我们也了解过了，就是我们所说的ack；
5. 偏移：这个就类似于我们IP帧结构中的首部长度，1个单位4字节；
6. 窗口：接收窗口的大小，表示接收端希望接受的字节数。

对抓包来说比较重要的字段如下：

1. 源端口
2. 目的端口
3. 偏移

12.2 Wireshark抓包分析

我们可以通过Wireshark去抓包，随便选择一个包展开就可以看到这样的一个结构：

Transmission Control Protocol, Src Port: 10270, Dst Port: 443, Seq: 2, Ack: 1, Len: 0			
Source Port:	10270		
Destination Port:	443		
[Stream index:	706]		
[TCP Segment Len:	0]		
Sequence number:	2	(relative sequence number)	
Acknowledgment number:	1	(relative ack number)	
Header Length:	20 bytes		
Flags:	0x014 (RST, ACK)		
Window size value:	0		
0000	d4 ee 07 30 d5 7c 8c a9 82 af 18 0c 08 00 45 00	...0.	E.
0010	00 28 57 ea 40 00 40 06 9d d8 c0 a8 c7 68 0e d7	.(T.@.@.	h..
0020	b1 25 28 1e 01 bb 20 7a 2b 5e d0 b6 b8 d5 50 14	..%(... z +^....	P.
0030	00 00 68 35 00 00	..h...	

12.3 实际代码编写

```

1  int main(int argc, char* argv[])
2  {
3      pcap_if_t* alldevsp = NULL;
4      pcap_if_t* tmpdevsp = NULL;
5      char errbuf[PCAP_ERRBUF_SIZE] = {0}; // 错误信息
6      // 1. 遍历适配器
7      int iRet = pcap_findalldevs(&alldevsp, errbuf); // 获取返回值
8      if (iRet != 0) {
9          printf("pcap_findalldevs error: %s \n", errbuf);
10     }
11     tmpdevsp = alldevsp;
12     do
13     {
14         printf("Name: %s \nDesc: %s \n===== \n\n", tmpdevsp->name, tmpdevsp->description);
15         } while (tmpdevsp = tmpdevsp->next);
16
17     // 2. 打开指定的适配器
18     pcap_t* pcap = pcap_open("\\Device\\NPF_{C7C05FAA-6043-4EB3-9059-329655AC6FB0}", 65535, 1, NULL, NULL, errbuf);
19
20     if (!pcap) {
21         printf("pcap_open error: %s \n", errbuf);
22     }
23
24     // 3. 关闭适配器
25     pcap_freealldevs(alldevsp);
26
27     // 4. 捕获数据包
28     pcap_loop(pcap, 0, Mypcap_handler, NULL);
29     return 0;

```

```

30 }
31
32 char* MyStrCpy(char* oStr, int index, int number) {
33     char* tmpStr = oStr;
34     char* resStr = (char*)malloc(number);
35     for (int i = 0; i < number; i++) {
36         *(resStr+i) = *(tmpStr+(index+i));
37     }
38     return resStr;
39 }
40
41 void Mypcap_handler(u_char *user, const struct pcap_pkthdr *pkt_header,
42 const u_char *pkt_data) {
43     // 5. 编写回调函数
44     struct tm *ltime;
45     char timestr[16] = {0};
46     time_t local_tv_sec;
47
48     // 将时间戳转换成可识别的格式
49     local_tv_sec = pkt_header->ts.tv_sec;
50     ltime = localtime(&local_tv_sec);
51     strftime(timestr, sizeof timestr, "%H:%M:%S", ltime);
52     printf("Time: %s, Caplen: %d, Len: %d \n", timestr, pkt_header->caplen, pkt_header->len);
53
54     printf("====MAC=====\n");
55
56     u_char dstMac[6] = {0};
57     for(int i = 0; i < 6; i++) {
58         dstMac[i] = *(pkt_data+i+0);
59     }
60     printf("Dst Mac: %02x:%02x:%02x:%02x:%02x:%02x \n", dstMac[0], dstMac[1], dstMac[2], dstMac[3], dstMac[4], dstMac[5]);
61
62     u_char srcMac[6] = {0};
63     for(i = 0; i < 6; i++) {
64         srcMac[i] = *(pkt_data+i+6);
65     }
66     printf("Src Mac: %02x:%02x:%02x:%02x:%02x:%02x \n", srcMac[0], srcMac[1], srcMac[2], srcMac[3], srcMac[4], srcMac[5]);
67
68     u_char type[2] = {0};
69
70     for(i = 0; i < 2; i++) {
71         type[i] = *(pkt_data+i+12);
72     }
73
74     printf("Type: ");
75     if (type[0] == 0x08) {
76         if (type[1] == 0x00) {
77             printf("IPv4\n");
78             printf("\n====IP=====\n");

```

```

79     u_char versionAndHeaderLen = *(pkt_data+14);
80     u_char version = versionAndHeaderLen >> 4; // 版本
81     u_char headerLen = versionAndHeaderLen << 4;
82     headerLen = headerLen >> 4; // 首部长度
83     u_char realHeaderLen = headerLen * 4;
84
85     if (version == 0x04) {
86         printf("Version: IPv4 \n");
87     }
88     printf("HeaderLen: %d(%d Byte) \n", headerLen, realHeaderLen);
89
90
91     u_char allLen[2] = {0};
92     for (i = 0; i < 2; i++) {
93         allLen[i] = *(pkt_data+i+14+2);
94     }
95     u_short resLen = ntohs(*(u_short*)allLen);
96
97     printf("AllLen: %d Byte \n", resLen);
98
99     u_char protocol = *(pkt_data+14+9);
100    switch (protocol) {
101    case 1:
102        printf("Protocol: ICMP \n");
103        break;
104    case 2:
105        printf("Protocol: IGMP \n");
106        break;
107    case 6:
108        printf("Protocol: TCP \n");
109        break;
110    case 17:
111        printf("Protocol: UDP \n");
112        break;
113    case 89:
114        printf("Protocol: OSPF \n");
115        break;
116    }
117
118    u_char srcIP[4] = {0};
119    for (i = 0; i < 4; i++) {
120        srcIP[i] = *(pkt_data+i+14+12);
121    }
122    printf("Src IP: %d.%d.%d.%d\n", srcIP[0], srcIP[1], srcIP[2],
srcIP[3]);
123
124    u_char dstIP[4] = {0};
125    for (i = 0; i < 4; i++) {
126        dstIP[i] = *(pkt_data+i+14+16);
127    }
128    printf("Dst IP: %d.%d.%d.%d \n", dstIP[0], dstIP[1], dstIP[2],
dstIP[3]);
129    printf("\n=====IP=====\\n\\n");

```

```

130     if (protocol == 6) {
131         printf("=====TCP=====\n");
132         u_char srcPort[2] = {0};
133         for (int x = 0; x < 2; x++) {
134             srcPort[x] = *(pkt_data+14+realHeaderLen+x);
135         }
136         u_short resSrcPort = ntohs(*(u_short*)srcPort);
137         printf("Src Port: %d \n", resSrcPort);
138         u_char dstPort[2] = {0};
139         for (x = 0; x < 2; x++) {
140             dstPort[x] = *(pkt_data+14+realHeaderLen+x);
141         }
142         u_short resDstPort = ntohs(*(u_short*)dstPort);
143         printf("Dst Port: %d \n", resDstPort);
144         u_char offsetAndReserved = *(pkt_data+14+realHeaderLen+12)
;
145         u_char offset = offsetAndReserved >> 4;
146         printf("Offset: %d\n", offset);
147         printf("=====TCP=====\n");
148     }
149
150
151
152     } else if (type[1] == 0x06) {
153         printf("ARP\n");
154     }
155 }
156
157 }

```

The screenshot shows two windows. The left window displays the output of a packet capture analysis, and the right window shows the 'Active Connections' list from the Windows command prompt.

Left Window Output:

```

Dst Port: 80
Offset: 5
=====TCP=====
Time: 22:07:07, Caplen: 54, Len: 54
=====MAC=====
Dst Mac: 3e:22:fb:3c:70:64
Src Mac: 00:0c:29:c0:3f:57
Type: IPv4
=====IP=====
Version: IPv4
HeaderLen: 5(20 Byte)
AllLen: 40 Byte
Protocol: TCP
Src IP: 172.16.176.5
Dst IP: 188.181.49.12
=====TCP=====
Src Port: 1115
Dst Port: 443
Offset: 5
=====TCP=====

```

Right Window (Active Connections):

Proto	Local Address	Foreign Address	State	PID
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING	972
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING	4
TCP	127.0.0.1:1029	0.0.0.0:0	LISTENING	1652
TCP	172.16.176.5:139	0.0.0.0:0	LISTENING	4
TCP	172.16.176.5:1116	202.89.233.100:80	ESTABLISHED	1576
TCP	172.16.176.5:1117	122.246.3.126:80	ESTABLISHED	1576
TCP	172.16.176.5:1118	122.225.34.203:80	ESTABLISHED	1576
TCP	172.16.176.5:1119	202.89.233.100:80	ESTABLISHED	1576
UDP	0.0.0.0:445	***		4
UDP	0.0.0.0:500	***		740
UDP	0.0.0.0:1025	***		1224
UDP	0.0.0.0:4500	***		740
UDP	0.0.0.0:8099	***		3676
UDP	127.0.0.1:123	***		1116
UDP	127.0.0.1:1037	***		1576
UDP	127.0.0.1:1900	***		1296
UDP	172.16.176.5:123	***		1116
UDP	172.16.176.5:137	***		4
UDP	172.16.176.5:138	***		4
UDP	172.16.176.5:1900	***		1296

C:\Documents and Settings\Administrator>

13 模仿360/QQ管家获取网速

我们都用过360或者QQ管家，它一般会有个悬浮的窗口，然后实时的显示当前的上下行网速，我们就通过学习的知识来实现这个功能。

13.1 网速的定义

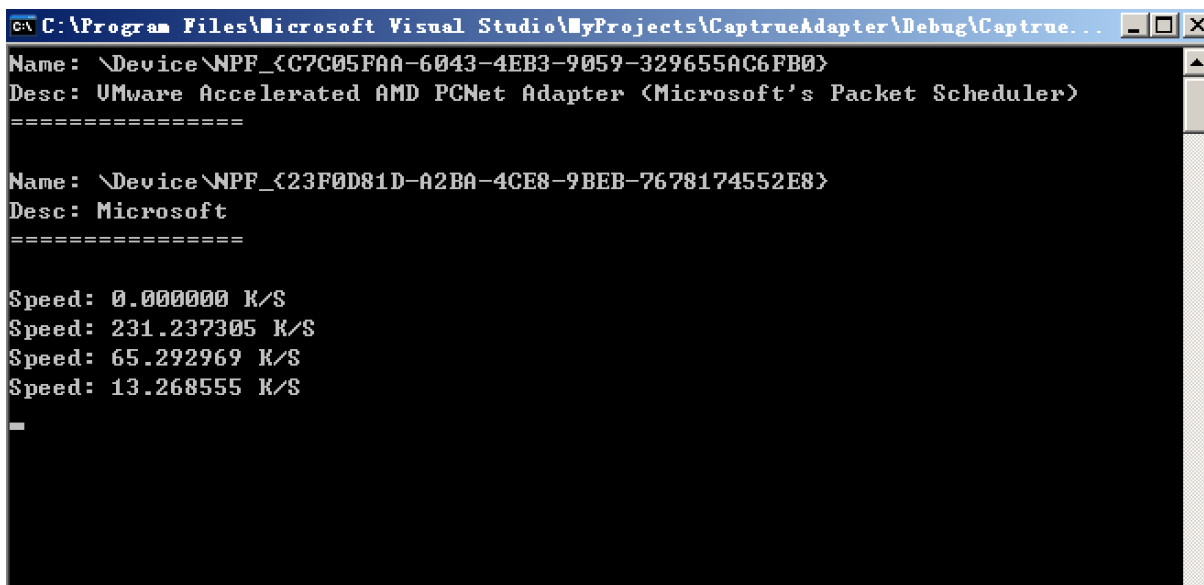
一段时间内下载数据的大小除以持续时长就是网速： $speed = len / time$

那么如上公式中的下载数据的大小和持续时长如何获取呢？那就是通过之前所学习的回调函数的pcap_pkthdr结构体来获取。

13.2 实际代码编写

```

1  void Mypcap_handler(u_char *user, const struct pcap_pkthdr *pkt_header,
2  const u_char *pkt_data) {
3      // 5. 编写回调函数
4      // 初始化开始时间和结束时间
5      // 这里使用static关键词是为了避免重复初始化
6      static timeval tBegin, tEnd = {0};
7      // 初始化数据长度
8      static DWORD dataLen = 0;
9      // 结束时间为抓包时间
10     tEnd = pkt_header->ts;
11     // 数据长度递增
12     dataLen += pkt_header->caplen;
13     // 计算得出开始时间减去结束时间
14     DWORD fTime = tEnd.tv_sec - tBegin.tv_sec + (tEnd.tv_usec -
15     tBegin.tv_usec) / 1000000;
16     // 当时间大于1秒时，计算网速
17     if (fTime > 1){
18         float speed = dataLen / fTime;
19         printf("Speed: %f K/S \n", speed/1024);
20         // 将开始时间移到结束时间，提供下一次计算
21         tBegin = tEnd;
22         // 初始化数据
23         dataLen = 0;
24     }
25 }
```



```
C:\Program Files\Microsoft Visual Studio\MyProjects\CaptrueAdapter\Debug\Captrue...
Name: \Device\NPF_{C7C05FAA-6043-4EB3-9059-329655AC6FB0}
Desc: VMware Accelerated AMD PCNet Adapter (Microsoft's Packet Scheduler)
=====

Name: \Device\NPF_{23F0D81D-A2BA-4CE8-9BEB-7678174552E8}
Desc: Microsoft
=====

Speed: 0.000000 K/S
Speed: 231.237305 K/S
Speed: 65.292969 K/S
Speed: 13.268555 K/S
-
```


14 RSA加密的基本原理

14.1 为什么需要加密

网络上传输的数据很容易被抓包，如果不加密，网络数据就很容易窃取，诸如用户名、密码这些敏感的信息一旦丢失，将会造成巨大的损失。

14.2 常用的加密方式

1. **对称加密**：加密方和解密方使用同一个密钥；**优点**：加密解密过程简单，高效；**缺点**：有一方泄密了，则整个加密就失去了意义。
2. **非对称加密**：加密方和解密方使用不同的密钥；**优点**：解密的密钥无法用加密的密钥来解密，即使加密方暴露出了密钥也没事，因为这个密钥只能加密，而无法解密，所以就提高了安全性；**缺点**：效率比较低，过程比较繁琐。

RSA就属于非对称加密，也就是我们本章节所学的。

14.3 辅助概念

1. **质数**：是指在大于1的自然数中，除了1和它本身以外不再有其他因数的自然数。
2. **互为质数**：一般指互质数，互质数为数学中的一种概念，即两个或多个整数的公因数只有1的非零自然数。公因数只有1的两个非零自然数，叫做互质数。

14.4 RSA加密密钥的获取

RSA加密密钥获取步骤如下所示：

1. 随机选取两个数 p 、 q （满足互质数条件）；
2. 按照公式获取公开模数： $n = p * q$ ，它的二进制位就是密钥长度；
3. 按照公式获取： $g = f(p, q) = (p-1) * (q-1)$ ；
4. 在1和 g 之间任意一个随机整数 e （公开指数，满足 $1 < e < g$ ）；
5. 由 $e * d \bmod g = 1$ 关系式推导出来 d （私有指数）
6. 公开密钥 = (e, n) 用来加密
7. 私有密钥 = (d, n) 用来解密

14.5 RSA加密解密算法

设 M 为需要加密的明文数据，加密算法为： **$\text{Encrypt_Message} = M^e \bmod n$**

设 D 为需要解密的密文数据，解密算法为： **$\text{Decrypt_Message} = D^d \bmod n$**

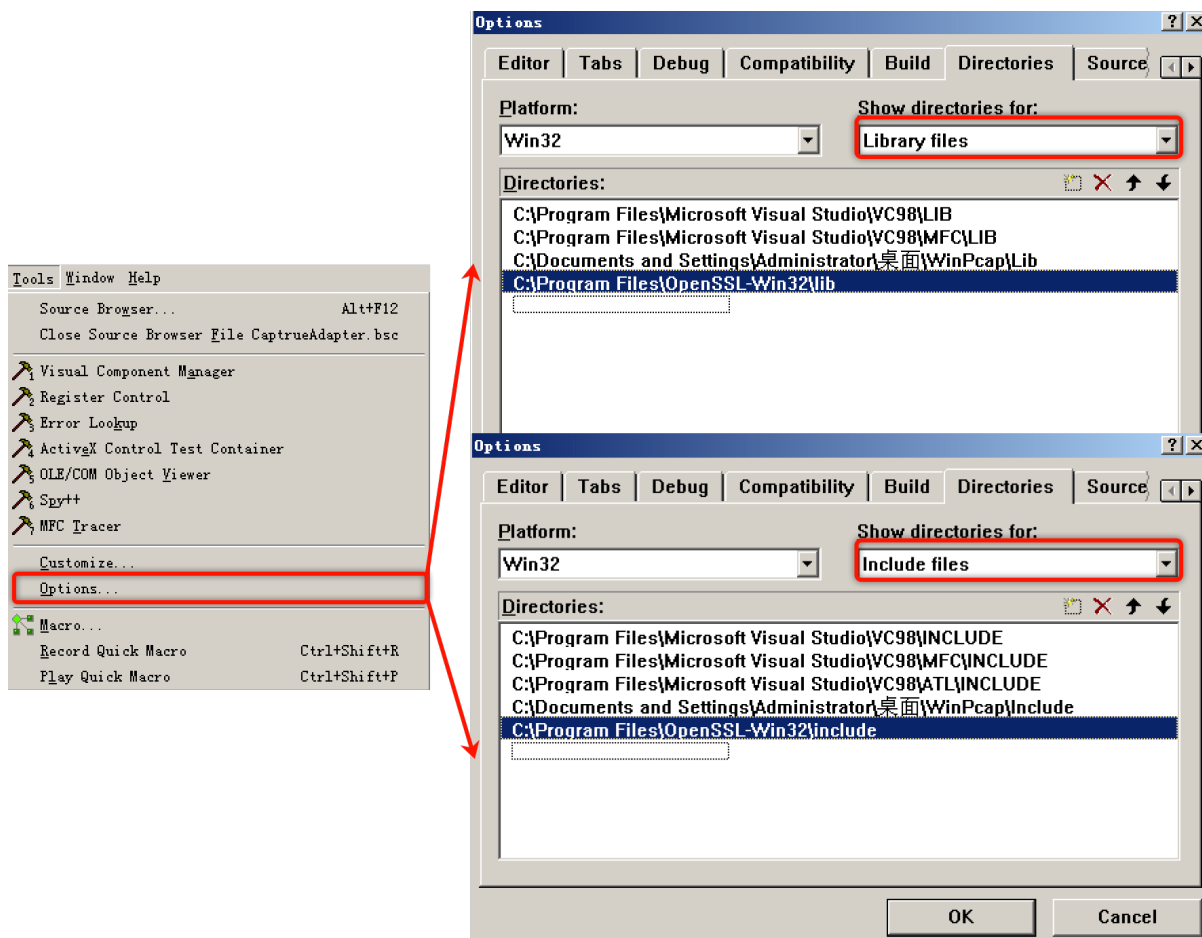
过程可以借助RSA-Tool和Big Integer Calculator工具。

15 RSA库的使用

我们需要借助RSA库在代码上实现加密与解密，这个库是OPENSSL。

15.1 安装与配置

如果你去官方下载的话是下载到的源码，需要自己去编译打包，和WinPcap库一样，将对应路径填写到VC6的配置中去：



15.2 RSA库之RSA结构体及API

RSA结构体如下：

```

1 struct RSA
2 {
3     BIGNUM *n;           // public modulus
4     BIGNUM *e;           // public exponent
5     BIGNUM *d;           // private exponent
6     BIGNUM *p;           // secret prime factor
7     BIGNUM *q;           // secret prime factor

```

```

8     BIGNUM *dmp1;           // d mod (p-1)
9     BIGNUM *dmq1;           // d mod (q-1)
10    BIGNUM *iqmp;            // q-1 mod p
11    // ...
12 };

```

其对应的API：

```

1    // 初始化一个RSA结构
2    RSA * RSA_new(void);
3
4    // 释放一个RSA结构
5    void RSA_free(RSA *rsa);

```

15.3 RSA库之BIGNUM及API

BIGNUM结构体如下：

```

1    typedef struct bignum_st BIGNUM;
2
3    struct bignum_st
4    {
5        BN_ULONG *d;           /* Pointer to an array of 'BN_BITS2' bit
6        chunks. */
7        int top;                /* Index of last used d +1. */
8        /* The next are internal book keeping for bn_expand. */
9        int dmax;               /* Size of the d array. */
10       int neg;                 /* one if the number is negative */
11       int flags;
12 };

```

其对应的API：

```

1    // 新生成一个BIGNUM结构
2    BIGNUM *BN_new(void);
3
4    // 释放一个BIGNUM结构
5    void BN_free(BIGNUM *a);
6
7    // 将16进制字符串转成大数
8    int BN_hex2bn(BIGNUM **a, const char *str);

```

15.4 RSA库之加密解密函数

15.4.1 公钥加密函数

```
1  int RSA_public_encrypt(  
2      int flen, // 要加密信息长度  
3      unsigned char *from, // 要加密信息  
4      unsigned char *to, // 输出参数, 加密后的信息  
5      RSA *rsa,  
6      int padding // 采取的加密方案, 分为: RSA_PKCS1_PADDING、  
7      RSA_PKCS1_OAEP_PADDING、RSA_SSLV23_PADDING、RSA_NO_PADDING  
    );
```

15.4.2 私钥解密函数

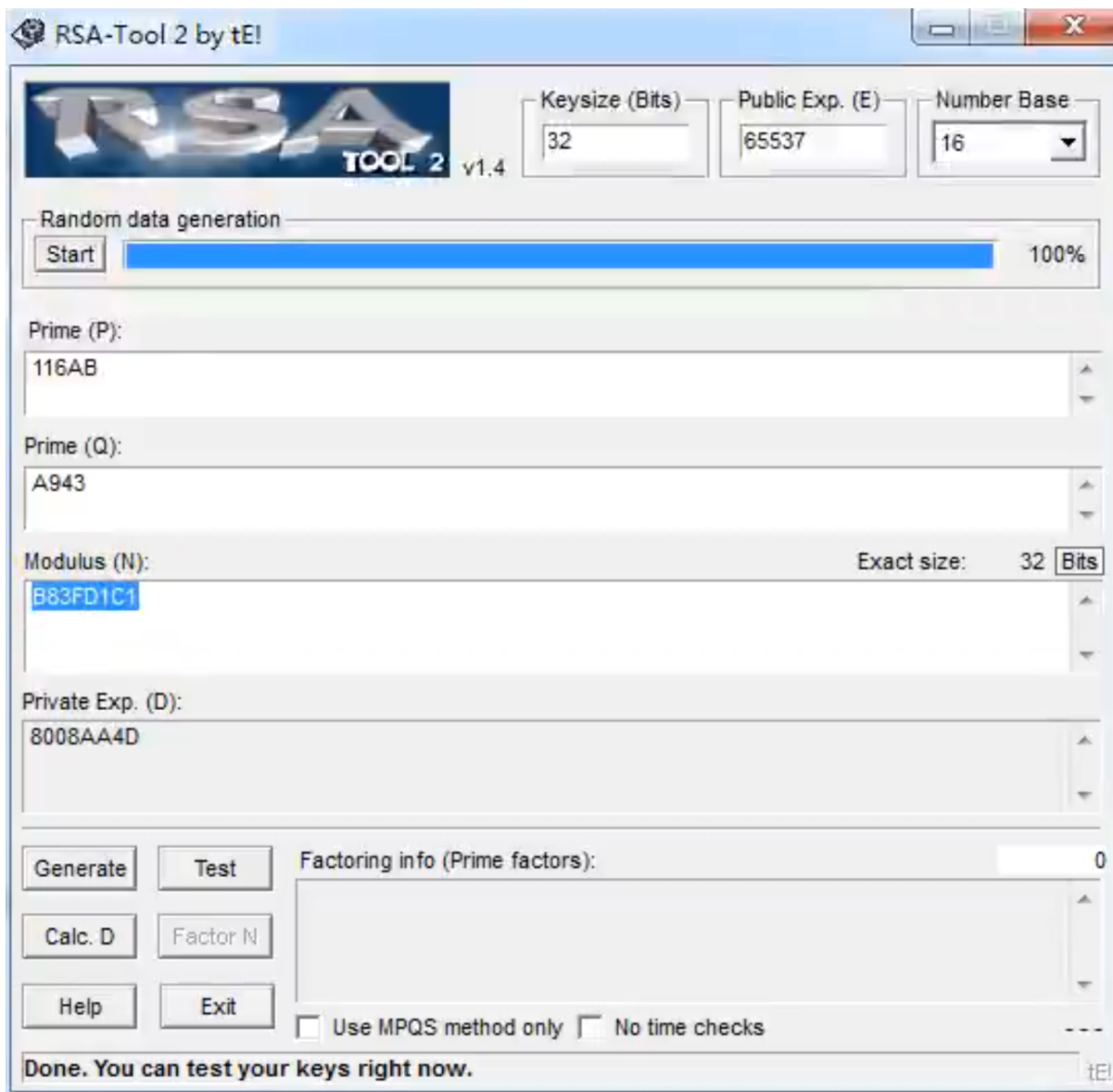
```
1  int RSA_private_decrypt(  
2      int flen, // 要解密的信息长度  
3      unsigned char *from, // 要解密的信息  
4      unsigned char *to, // 输出参数, 解密后的信息  
5      RSA *rsa,  
6      int padding // 采取的解密方案, 分为: RSA_PKCS1_PADDING、  
7      RSA_PKCS1_OAEP_PADDING、RSA_SSLV23_PADDING、RSA_NO_PADDING  
    );
```

15.5 实际代码编写

首先我们需要包含一个头文件和两个库文件：

```
1  #include <openssl/rsa.h>  
2  #pragma comment(lib, "libeay32.lib")  
3  #pragma comment(lib, "ssleay32.lib")
```

然后就是定义E、D、N，这个我们根据RSA-Tool生成的填写进去即可：



```

1  #define E "10001"
2  #define D "8008AA4D"
3  #define N "B83FD1C1"

```

接着就是写自己的加密函数，解密函数同理可得：

```

1  void MyRSAEncrypt(const unsigned char* pOrgData, int dataLen, unsigned
2  char* pEncryptBuf) {
3      // 初始化一个RSA结构
4      RSA *pRsa = RSA_new();
5      // 初始化BIGNUM结构
6      BIGNUM* pbn_e = BN_new();
7      BIGNUM* pbn_n = BN_new();

```

```
7 // 将16进制字符串转成大数
8 BN_hex2bn(&pbn_e, E);
9 BN_hex2bn(&pbn_n, N);
10 // RSA结构成员赋值
11 pRsa->e = pbn_e;
12 pRsa->n = pbn_n;
13
14 // 公钥加密函数
15 int iRet = RSA_public_encrypt(dataLen, pOrgData, pEncryptBuf, pRsa,
RSA_NO_PADDING);
16 if (iRet == -1) {
17     printf("RSA_public_encrypt Error!\n");
18 }
19
20 // 释放BIGNUM结构
21 BN_free(pbn_e);
22 BN_free(pbn_n);
23 // 释放RSA结构
24 RSA_free(pRsa);
25 }
```