

```

64.     WSACloseEvent(evObj);
65.     closesocket(hRecvSock);
66.     closesocket(hLisnSock);
67.     WSACleanup();
68.     return 0;
69. }
70.
71. void CALLBACK CompRoutine(
72.     DWORD dwError, DWORD szRecvBytes, LPWSAOVERLAPPED lpOverlapped, DWORD flags)
73. {
74.     if(dwError!=0)
75.     {
76.         ErrorHandling("CompRoutine error");
77.     }
78.     else
79.     {
80.         recvBytes=szRecvBytes;
81.         printf("Received message: %s \n", buf);
82.     }
83. }
84.
85. void ErrorHandling(char *message)
86. {
87.     // 与之前示例的ErrorHandling函数一致，故省略。
88. }

```

代码说明

- 第51行：调用WSARecv函数的同时向第七个参数传递Completion Routine函数的地址值。可在第71行找到Completion Routine函数的定义。需要注意，即使采用Completion Routine方式，也必须指定第六个参数WSAOVERLAPPED结构体变量的地址值。但不必为了hEvent成员的初始化而创建事件对象。
- 第54行：用于验证接收数据的操作是否为Pending状态。
- 第58行：为了使main线程进入alertable wait状态而调用的函数。为了调用此函数，第49行创建了多余的事件对象——Dummy Object。使用SleepEx函数即可避免Dummy Object的产生。
- 第59行：返回常量WAIT_IO_COMPLETION意味着I/O正常结束。
- 第71行：Completion Routine函数原型固定不变（返回值、参数及参数个数）。

下面给出结合OverlappedSend_win.c运行时的结果。由于在1台计算机中运行，且交换的数据量不大，所以I/O不会进入Pending状态。

❖ 运行结果：OverlappedSend_win.c

Send data size: 21

❖ 运行结果：CmplRoutinesRecv_win.c

Received message: Network is Computer!
Overlapped I/O Completed

下面给出传入WSARecv函数的最后一个参数的Completion Routine函数原型。

```
void CALLBACK CompletionROUTINE(
    DWORD dwError, DWORD cbTransferred, LPWSAOVERLAPPED lpOverlapped,
    DWORD dwFlags);
```

其中第一个参数中写入错误信息(正常结束时写入0),第二个参数中写入实际收发的字节数。第三个参数中写入WSASend、WSARecv函数的参数lpOverlapped, dwFlags中写入调用I/O函数时传入的特性信息或0。另外,返回值类型void后插入的CALLBACK关键字与main函数中声明的关键字WINAPI相同,都是用于声明函数的调用规范,所以定义Completion Routine函数时必须添加。

本章介绍了不少内容,这些都是为了理解第23章的IOCP而讲解的。可以说,本章是学习第23章的必要条件,请各位务必掌握本章内容。

提示

重叠 I/O 的讲解尚未结束

第23章将利用本章介绍的重叠 I/O 编写回声服务器端,并讲解重叠 I/O 的改进方案——IOCP。

22.3 习题

- (1) 异步通知I/O模型与重叠I/O模型在异步处理方面有哪些区别?
- (2) 请分析非阻塞I/O、异步I/O、重叠I/O之间的关系。
- (3) 阅读如下代码,请指出问题并给出解决方案。

```
while(1) {
    hRecvSock = accept(hLisnSock, (SOCKADDR * ) & recvAdr, &recvAdrSz);
    evObj = WSACreateEvent();
    memset(&overlapped, 0, sizeof(overlapped));
    overlapped.hEvent = evObj;
    dataBuf.len = BUF_SIZE;
    dataBuf.buf = buf;
    WSARecv(hRecvSock, &dataBuf, 1, &recvBytes, &flags, &overlapped, NULL);
}
```

虽然并不完整,但足以通过这部分代码片段发现结构上存在的问题。

- (4) 请从源代码角度说明调用WSASend函数后如何验证I/O是否进入Pending状态。
- (5) 线程的“alertable wait状态”的含义是什么?说出能使线程进入这种状态的2个函数。

done!

23

本章将介绍最后一个 Windows 扩展 I/O 模型，它是性能最好的 Windows 平台 I/O 模型，同时也是最难学习的模型。但有了第 22 章内容的铺垫，相信各位也能轻松驾驭。

23.1 通过重叠 I/O 理解 IOCP

本章的 IOCP（Input Output Completion Port，输入输出完成端口）服务器端模型是很多 Windows 程序员关注的焦点。各位若急于求成而跳过了第 21 章的内容，建议大家最好回顾一下。因为第 21 章和第 22 章介绍了本章的背景知识，而且，关于 IOCP 的内容实际上是从第 22 章开始的。

+ 热门话题：epoll 和 IOCP 的性能比较

为了突破 select 等传统 I/O 模型的极限，每种操作系统（内核级别）都会提供特有的 I/O 模型以提高性能。其中最具代表性的有 Linux 的 epoll、BSD 的 kqueue 及本章的 Windows 的 IOCP。它们都在操作系统级别提供支持并完成功能，但此处有一个持续性的争论热点：

“epoll 快还是 IOCP 快？”

对此的争论仍然可以在 www.yahoo.com 上看到，有时甚至会走入极端。因为服务器端的响应时间和并发服务数是衡量服务器端好坏的重要因素，所以存在这种争论也可以理解。但对于像我这种普通人来讲，这 2 种模型已经非常优秀了，因此几乎不会发生如下这种情况。至少我和我周围的开发人员之间未曾有过这类讨论。

✓ “基于 epoll 实现的服务器端在并发数量上有缺陷，但换成 IOCP 后没有任何问题！”

“IOCP 的响应时间太慢是个问题，换成 epoll 后就解决了！”

另外，在硬件性能和分配带宽充足的情况下，如果响应时间和并发数量出了问题，我会首先怀疑如下两点，修正后通常会解决大部分问题。

- ❑ 低效的I/O结构或低效的CPU使用
- ❑ 数据库设计和查询语句 (Query) 的结构

如果有人问为何IOCP相对更优 (通常是网上的言论), 我会回答: “不太清楚。” 虽然IOCP拥有其他I/O模型不具备的优点, 但这并非左右服务器端性能的绝对因素, 而且不可能在任何情况下都体现这种优点。它们之间的差异主要在于操作系统内部的工作机制。当然, 最终还是要靠各位自行判断。我只是想说, 周围也有不少开发人员与我观点一致。

+ 实现非阻塞模式的套接字

第22章中只介绍了执行重叠I/O的Sender和Receiver, 但还未利用该模型实现过服务器端。因此, 我们先利用重叠I/O模型实现回声服务器端。首先介绍创建非阻塞模式套接字的方法。我们曾在第17章创建过非阻塞模式的套接字, 与之类似, 在Windows中通过如下函数调用将套接字属性改为非阻塞模式。

```
SOCKET hLisnSock,
int mode = 1;
. . . . .
hListSock = WSASocket(PF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
ioctlsocket(hLisnSock, FIONBIO, &mode); //for non-blocking socket
. . . . .
```

上述代码中调用的ioctlsocket函数负责控制套接字I/O方式, 其调用具有如下含义:

“将hLisnSock句柄引用的套接字I/O模式 (FIONBIO) 改为变量mode中指定的形式。”

也就是说, FIONBIO是用于更改套接字I/O模式的选项, 该函数的第三个参数中传入的变量中若存有0, 则说明套接字是阻塞模式的; 如果存有非0值, 则说明已将套接字模式改为非阻塞模式。改为非阻塞模式后, 除了以非阻塞模式进行I/O外, 还具有如下特点。

- ❑ 如果在没有客户端连接请求的状态下调用accept函数, 将直接返回INVALID_SOCKET。调用WSAGetLastError函数时返回WSAEWOULDBLOCK。
- ❑ 调用accept函数时创建的套接字同样具有非阻塞属性。

因此, 针对非阻塞套接字调用accept函数并返回INVALID_SOCKET时, 应该通过WSAGetLastError函数确认返回INVALID_SOCKET的理由, 再进行适当处理。

+ 以纯重叠 I/O 方式实现回声服务器端

要想实现基于重叠I/O的服务器端, 必须具备非阻塞套接字, 所以先介绍了其创建方法。实际上, 因为有IOCP模型, 所以很少有人只用重叠I/O实现服务器端。但我认为: “为了正确理解

IOCP, 应当尝试用纯重叠I/O方式实现服务器端。”

即使坚持不用IOCP, 也应具备仅用重叠I/O方式实现类似IOCP的服务器端的能力。这样就可以在其他操作系统平台实现类似IOCP方式的服务器端, 而且不会因IOCP的限制而忽略服务器端功能的实现。

下面用纯重叠I/O模型实现回声服务器端, 希望各位亲自动手一试。接下来介绍示例, 由于代码量较大, 我们分3个部分学习。

❖ CmplRouEchoServ_win.c: main函数之前

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <winsock2.h>
4.
5. #define BUF_SIZE 1024
6. void CALLBACK ReadCompRoutine(DWORD, DWORD, LPWSAOVERLAPPED, DWORD);
7. void CALLBACK WriteCompRoutine(DWORD, DWORD, LPWSAOVERLAPPED, DWORD);
8. void ErrorHandling(char *message);
9.
10. typedef struct
11. {
12.     SOCKET hClnSock;
13.     char buf[BUF_SIZE];
14.     WSABUF wsaBuf;
15. } PER_IO_DATA, *LPPER_IO_DATA;
```

代码说明

- 第10行: 请注意观察此处的结构体。该结构体中包含套接字句柄、缓冲及缓冲相关信息。

该结构体中的信息足够进行数据交换, 下列代码将介绍该结构体的填充及使用方法。

❖ CmplRouEchoServ_win.c: main函数

```
1. int main(int argc, char* argv[])
2. {
3.     WSADATA wsaData;
4.     SOCKET hLisnSock, hRecvSock;
5.     SOCKADDR_IN lisnAdr, recvAdr;
6.     LPWSAOVERLAPPED lpOvLp;
7.     DWORD recvBytes;
8.     LPPER_IO_DATA hbInfo;
9.     int mode=1, recvAdrSz, flagInfo=0;
10.
11.     if(argc!=2) {
12.         printf("Usage: %s <port>\n", argv[0]);
13.         exit(1);
14.     }
```

```

15.
16.     if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
17.         ✓ ErrorHandling("WSAStartup() error!");
18.
19.     hLisnSock=WSASocket(PF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
20.     ✓ ioctlsocket(hLisnSock, FIONBIO, &mode); // for non-blocking mode socket
21.
22.     memset(&lisnAdr, 0, sizeof(lisnAdr));
23.     lisnAdr.sin_family=AF_INET;
24.     lisnAdr.sin_addr.s_addr=htonl(INADDR_ANY);
25.     lisnAdr.sin_port=htons(atoi(argv[1]));
26.
27.     if(bind(hLisnSock, (SOCKADDR*) &lisnAdr, sizeof(lisnAdr))==SOCKET_ERROR)
28.         ErrorHandling("bind() error");
29.     if(listen(hLisnSock, 5)==SOCKET_ERROR)
30.         ErrorHandling("listen() error");
31.
32.     recvAdrSz=sizeof(recvAdr);
33.     while(1)
34.     {
35.         SleepEx(100, TRUE); // for alertable wait state
36.         hRecvSock=accept(hLisnSock, (SOCKADDR*)&recvAdr,&recvAdrSz);
37.         if(hRecvSock==INVALID_SOCKET)
38.         {
39.             if(WSAGetLastError()==WSAEWOULDBLOCK)
40.                 continue;
41.             else
42.                 ErrorHandling("accept() error");
43.         }
44.         puts("Client connected.....");
45.
46.         lpOvLp=(LPWSAOVERLAPPED)malloc(sizeof(WSAOVERLAPPED));
47.         memset(lpOvLp, 0, sizeof(WSAOVERLAPPED));
48.
49.         hbInfo=(LPPER_IO_DATA)malloc(sizeof(PER_IO_DATA));
50.         hbInfo->hClnSock=(DWORD)hRecvSock;
51.         (hbInfo->wsaBuf).buf=hbInfo->buf;
52.         (hbInfo->wsaBuf).len=BUF_SIZE;
53.
54.         ✓ lpOvLp->hEvent=(HANDLE)hbInfo;
55.         WSARecv(hRecvSock, &(hbInfo->wsaBuf),
56.             1, &recvBytes, &flagInfo, lpOvLp, ReadCompRoutine);
57.     }
58.     closesocket(hRecvSock);
59.     closesocket(hLisnSock);
60.     WSACleanup();
61.     return 0;
62. }

```

代码说明

- 第19、20行：第20行将第19行创建的套接字改为非阻塞模式。套接字创建后默认为阻塞模式，故需要转换。
- 第33、36行：第33行的while循环中调用accept函数。套接字是非阻塞模式的，因此特别需要注意返回INVALID_SOCKET时的处理过程。
- 第46、47行：申请重叠I/O中需要使用的结构体变量的内存空间并初始化。之所以在循环内部申请WSAOVERLAPPED结构体空间，是因为每个客户端都需要独立的WSAOVERLAPPED结构体变量。
- 第49~52行：动态分配PER_IO_DATA结构体内存空间，并写入第36行创建的套接字句柄。该套接字在I/O过程中将使用第51行和第52行初始化的缓冲。
- 第54行：WSAOVERLAPPED结构体变量的hEvent成员中将写入第49行分配过空间的变量地址值。基于Completion Routine函数的重叠I/O中不需要事件对象，因此，hEvent中可以写入其他信息。
- 第55行：调用WSARecv函数时将ReadCompRoutine函数指定为Completion Routine。其中第六个参数WSAOVERLAPPED结构体变量地址值将传递到Completion Routine的第三个参数。因此，Completion Routine函数内可以访问完成I/O的套接字句柄和缓冲。另外，为了运行Completion Routine函数，第35行循环调用SleepEx函数。

最后介绍2个Completion Routine函数。实际的回声服务是通过这2个函数完成的，希望各位仔细观察提供服务的过程。

❖ CmplRouEchoServ_win.c: 2个Completion Routine函数和错误处理函数

```

1. void CALLBACK ReadCompRoutine(
2.     DWORD dwError, DWORD szRecvBytes, LPWSAOVERLAPPED lpOverlapped, DWORD flags)
3. {
4.     LPPER_IO_DATA hbInfo=(LPPER_IO_DATA)(lpOverlapped->hEvent);
5.     SOCKET hSock=hbInfo->hClnSocket;
6.     LPWSABUF bufInfo=&(hbInfo->wsaBuf);
7.     DWORD sentBytes;
8.
9.     if(szRecvBytes==0)
10.    {
11.        closesocket(hSock);
12.        free(lpOverlapped->hEvent); free(lpOverlapped);
13.        puts("Client disconnected.....");
14.    }
15.    else // echo!
16.    {
17.        bufInfo->len=szRecvBytes;
18.        WSASend(hSock, bufInfo, 1, &sentBytes, 0, lpOverlapped, WriteCompRoutine);
19.    }
20. }
21.
22. void CALLBACK WriteCompRoutine(
23.     DWORD dwError, DWORD szSendBytes, LPWSAOVERLAPPED lpOverlapped, DWORD flags)
24. {

```

```

25.  LPPER_IO_DATA hbInfo=(LPPER_IO_DATA)(lpOverlapped->hEvent);
26.  SOCKET hSock=hbInfo->hClnSock;
27.  LPWSABUF bufInfo=&(hbInfo->wsaBuf);
28.  DWORD recvBytes;
29.  int flagInfo=0;
30.  WSARecv(hSock, bufInfo,
31.         1, &recvBytes, &flagInfo, lpOverlapped, ReadCompRoutine);
32. }
33.
34. void ErrorHandler(char *message)
35. {
36.     fputs(message, stderr);
37.     fputc('\n', stderr);
38.     exit(1);
39. }

```

代码说明

- 第1行：该函数的调用意味着已完成数据输入，此时需要将接收的数据发送给回声客户端。
- 第4~6行：提取完成输入的套接字句柄和缓冲信息，因为WSAOVERLAPPED结构体变量的hEvent成员中保存了PER_IO_DATA结构体变量地址值。
- 第9行：变量szRecvBytes的值为0就意味着收到了EOF，因此需要进行相应处理。
- 第17、18行：将WriteCompRoutine函数指定为Completion Routine，同时调用WSASend函数。程序通过该语句向客户端发送回声消息。
- 第22、30行：发送回声消息后调用该函数。但需要再次接收数据，所以执行第30行的函数调用。

上述示例的工作原理整理如下。

- ❑ 有新的客户端连接时调用WSARecv函数，并以非阻塞模式接收数据，接收完成后调用ReadCompRoutine函数。
- ❑ 调用ReadCompRoutine函数后调用WSASend函数，并以非阻塞模式发送数据，发送完成后调用WriteCompRoutine函数。
- ❑ 此时调用的WriteCompRoutine函数将再次调用WSARecv函数，并以非阻塞模式等待接收数据。

通过交替调用ReadCompRoutine函数和WriteCompRoutine函数，反复执行数据的接收和发送操作。另外，每次增加1个客户端都会定义PER_IO_DATA结构体，以便将新创建的套接字句柄和缓冲信息传递给ReadCompRoutine函数和WriteCompRoutine函数。同时将该结构体地址值写入WSAOVERLAPPED结构体成员hEvent，并传递给Completion Routine函数。这非常重要，可概括如下：

“使用WSAOVERLAPPED结构体成员hEvent向完成I/O时自动调用的Completion Routine函数内部传递客户端信息（套接字和缓冲）。”

接下来需要验证运行结果，先要编写回声客户端，因为使用第4章的回声客户端会无法得到预想的结果。

+ 重新实现客户端

其实第4章实现并使用至今的回声客户端存在一些问题，关于这些问题及解决方案已在第5章进行了充分讲解。虽然在目前为止的各种模型的服务器端中使用稍有缺陷的回声客户端也不会引起太大问题，但本章的回声服务器端则不同。因此，需要按照第5章的提示解决客户端存在的问题，并结合改进后的客户端运行本章服务器端。之前已介绍过解决方法，故只给出代码。

❖ StableEchoCInt_win.c

```

1. #include <"头声明与第23章的示例CmplRoutinesRecv_win.c一致.">
2. #define BUF_SIZE 1024
3. void ErrorHandling(char *message);
4.
5. int main(int argc, char *argv[])
6. {
7.     WSADATA wsaData;
8.     SOCKET hSocket;
9.     SOCKADDR_IN servAdr;
10.    char message[BUF_SIZE];
11.    int strLen, readLen;
12.
13.    if(argc!=3) {
14.        printf("Usage: %s <IP> <port>\n", argv[0]);
15.        exit(1);
16.    }
17.    if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
18.        ErrorHandling("WSAStartup() error!");
19.
20.    hSocket=socket(PF_INET, SOCK_STREAM, 0);
21.    if(hSocket==INVALID_SOCKET)
22.        ErrorHandling("socket() error");
23.
24.    memset(&servAdr, 0, sizeof(servAdr));
25.    servAdr.sin_family=AF_INET;
26.    servAdr.sin_addr.s_addr=inet_addr(argv[1]);
27.    servAdr.sin_port=htons(atoi(argv[2]));
28.
29.    if(connect(hSocket, (SOCKADDR*)&servAdr, sizeof(servAdr))==SOCKET_ERROR)
30.        ErrorHandling("connect() error!");
31.    else
32.        puts("Connected.....");
33.
34.    while(1)
35.    {
36.        fputs("Input message(Q to quit): ", stdout);
37.        fgets(message, BUF_SIZE, stdin);
38.        if(!strcmp(message, "q\n") || !strcmp(message, "Q\n"))
39.            break;
40.
41.        strLen=strlen(message);

```

```

42.     send(hSocket, message, strLen, 0);
43.     readLen=0;
44.     while(1)
45.     {
46.         readLen+=recv(hSocket, &message[readLen], BUF_SIZE-1, 0);
47.         if(readLen>=strLen)
48.             break;
49.     }
50.     message[strLen]=0;
51.     printf("Message from server: %s", message);
52. }
53.
54. closesocket(hSocket);
55. WSACleanup();
56. return 0;
57. }
58.
59. void ErrorHandler(char *message)
60. {
61.     // 与其他Windows相关示例一致，故省略。
62. }

```

上述代码第44行的循环语句考虑到TCP的传输特性而重复调用了recv函数，直至接收完所有数据。将上述客户端结合之前的回声服务器端运行可以得到正确的运行结果，具体结果与一般的回声服务器端/客户端没有区别，故省略。

+ 从重叠 I/O 模型到 IOCP 模型

下面分析重叠I/O模型回声服务器端的缺点。

“重复调用非阻塞模式的accept函数和以进入alertable wait状态为目的的SleepEx函数将影响性能!”

如果正确理解了之前的示例，应该不难发现这一点。既不能为了处理连接请求而只调用accept函数，也不能为了Completion Routine而只调用SleepEx函数，因此轮流调用了非阻塞模式的accept函数和SleepEx函数（设置较短的超时时间）。这恰恰是影响性能的代码结构。

我也不知该如何弥补这一缺点，这属于重叠I/O结构固有的缺陷，但可以考虑如下方法：

“让main线程（在main函数内部）调用accept函数，再单独创建1个线程负责客户端I/O。”

其实这就是IOCP中采用的服务器端模型。换言之，IOCP将创建专用的I/O线程，该线程负责与所有客户端进行I/O。

知识补给站

负责全部I/O

有些读者会误认为IOCP就是创建专职I/O线程的一种模型。IOCP为了负责全部I/O工作，的确需要创建至少1个线程。但“在服务器端负责全部I/O”实质上相当于负责服务器端的全部工作，因此，它并不是仅负责I/O工作，而是至少创建1个线程并使其负责全部I/O的前后处理。尝试理解IOCP时不要把焦点集中于线程，而是注意观察如下2点。

❑ I/O是否以非阻塞模式工作？

❑ 如何确定非阻塞模式的I/O是否完成？

之前讲过的所有I/O模型及本章的IOCP模型都可以按上述标准进行区分。各位会发现，我讲解IOCP时也不会刻意强调创建线程的事实。

23.2 分阶段实现 IOCP 程序

本节我们编写最后一种服务器模型IOCP，比阅读代码更重要的是理解IOCP本身。

+ 创建“完成端口”

IOCP中已完成的I/O信息将注册到完成端口对象（Completion Port，简称CP对象），但这个过程并非单纯的注册，首先需要经过如下请求过程：

“该套接字的I/O完成时，请把状态信息注册到指定CP对象。”

该过程称为“套接字和CP对象之间的连接请求”。因此，为了实现基于IOCP模型的服务器端，需要做如下2项工作。

❑ 创建完成端口对象。

❑ 建立完成端口对象和套接字之间的联系。

此时的套接字必须被赋予重叠属性。上述2项工作可以通过1个函数完成，但为了创建CP对象，先介绍如下函数。

```
#include <windows.h>
```

```
HANDLE CreateIoCompletionPort(
    HANDLE FileHandle, HANDLE ExistingCompletionPort, ULONG_PTR CompletionKey,
    DWORD NumberOfConcurrentThreads);
```

➔ 成功时返回 CP 对象句柄，失败时返回 NULL。

● FileHandle	创建CP对象时传递INVALID_HANDLE_VALUE。
● ExistingCompletionPort	创建CP对象时传递NULL。
● CompletionKey	创建CP对象时传递0。
● NumberOfConcurrentThreads	分配给CP对象的用于处理I/O的线程数。例如，该参数为2时，说明分配给CP对象的可以同时运行的线程数最多为2个；如果该参数为0，系统中CPU个数就是可同时运行的最大线程数。

以创建CP对象为目的调用上述函数时，只有最后一个参数才真正具有含义。可以用如下代码段将分配给CP对象的用于处理I/O的线程数指定为2。

```
HANDLE hCpObject;
. . . . .
hCpObject = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 2);
```

+ 连接完成端口对象和套接字

既然有了CP对象，接下来就要将该对象连接到套接字，只有这样才能使已完成的套接字I/O信息注册到CP对象。下面以建立连接为目的再次介绍CreateIoCompletionPort函数。

```
#include <windows.h>
```

```
HANDLE CreateIoCompletionPort(
    HANDLE FileHandle, HANDLE ExistingCompletionPort, ULONG_PTR CompletionKey,
    DWORD NumberOfConcurrentThreads);
```

→ 成功时返回 CP 对象句柄，失败时返回 NULL。

● FileHandle	要连接到CP对象的套接字句柄。
● ExistingCompletionPort	要连接套接字的CP对象句柄。
● CompletionKey	传递已完成I/O相关信息，关于该参数将在稍后介绍的GetQueuedCompletionStatus函数中共同讨论。
● NumberOfConcurrentThreads	无论传递何值，只要该函数的第二个参数非NULL就会忽略。

上述函数的第二种功能就是将FileHandle句柄指向的套接字和ExistingCompletionPort指向的CP对象相连。该函数的调用方式如下。

```
HANDLE hCpObject;
SOCKET hSock;
. . . . .
CreateIoCompletionPort((HANDLE)hSock, hCpObject, (DWORD)ioInfo, 0);
```

调用 `CreateIoCompletionPort` 函数后，只要针对 `hSock` 的 I/O 完成，相关信息就将注册到 `hCpObject` 指向的 CP 对象。

+ 确认完成端口已完成的 I/O 和线程的 I/O 处理

我们已经掌握了 CP 对象的创建及其与套接字建立连接的方法，接下来就要学习如何确认 CP 中注册的已完成的 I/O。完成该功能的函数如下。

```
#include <windows.h>
```

```
BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort, LPDWORD lpNumberOfBytes, PULONG_PTR lpCompletionKey,
    LPOVERLAPPED * lpOverlapped, DWORD dwMilliseconds);
```

→ 成功时返回 TRUE，失败时返回 FALSE。

- CompletionPort 注册有已完成 I/O 信息的 CP 对象句柄。
- lpNumberOfBytes 用于保存 I/O 过程中传输的数据大小的变量地址值。
- lpCompletionKey 用于保存 `CreateIoCompletionPort` 函数的第三个参数值的变量地址值。
- lpOverlapped 用于保存调用 `WSASend`、`WSARecv` 函数时传递的 `OVERLAPPED` 结构体地址的变量地址值。
- dwMilliseconds 超时信息，超过该指定时间后将返回 FALSE 并跳出函数。传递 INFINITE 时，程序将阻塞，直到已完成 I/O 信息写入 CP 对象。

虽然只介绍了 2 个 IOCP 相关函数，但依然有些复杂，特别是上述函数的第三个和第四个参数更是如此。其实这 2 个参数主要是为了获取需要的信息而设置的，下面介绍这 2 种信息的含义。

“通过 `GetQueuedCompletionStatus` 函数的第三个参数得到的是以连接套接字和 CP 对象为目的而调用的 `CreateCompletionPort` 函数的第三个参数值。”

“通过 `GetQueueCompletionStatus` 函数的第四个参数得到的是调用 `WSASend`、`WSARecv` 函数时传入的 `WSAOVERLAPPED` 结构体变量地址值。”

各位需要通过示例理解这 2 个参数的使用方法。接下来讨论其调用主体，究竟由谁（何时）调用上述函数比较合理呢？如各位所料，应该由处理 IOCP 中已完成 I/O 的线程调用。

“那 I/O 如何分配给线程呢？”

如前所述，IOCP 中将创建全职 I/O 线程，由该线程针对所有客户端进行 I/O。而且 `CreateIoCompletionPort` 函数中也有参数用于指定分配给 CP 对象的最大线程数，所以各位或许会有如下疑问：

“是否自动创建线程并处理I/O?”

当然不是！应该由程序员自行创建调用WSASend、WSARecv等I/O函数的线程，只是该线程为了确认I/O的完成会调用GetQueuedCompletionStatus函数。虽然任何线程都能调用GetQueuedCompletionStatus函数，但实际得到I/O完成信息的线程数不会超过调用CreateIoCompletionStatus函数时指定的最大线程数。相信大家也理解了分配给CP对象的线程具有的含义。

以上就是IOCP服务器端实现时需要的全部函数及其理论说明，下面通过源代码理解程序的整体结构。

知识补给站

分配给完成端口的合理的线程个数

分配给CP对象的合理的线程数应该与CPU个数相同，这是MSDN中的说明。但现在主流的CPU核数（具有运算功能的内核）大部分都是2个或更多。核数和CPU在概念上有差异，但如果是双核则可以同时运行2个线程，因此，分配给CP对象的合理的线程数可以理解为CPU中的核数。一般人不会刻意区分核与CPU，而是把双核视为2个CPU。在多核技术相当成熟的背景下，这种看法不失为一种合理的选择。若想准确掌握发挥最佳性能的线程数，只能通过大量实验得到。

+ 实现基于IOCP的回声服务器端

虽然介绍了IOCP相关的理论知识，但离开示例很难真正掌握IOCP的使用方法。因此，我将介绍便于理解和运用的（极为普通的）基于IOCP的回声服务器端。首先给出IOCP回声服务器端的main函数之前的部分。

❖ IOCP EchoServ_win.c: main函数之前

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <process.h>
4. #include <winsock2.h>
5. #include <windows.h>
6.
7. #define BUF_SIZE 100
8. #define READ 3
9. #define WRITE 5
10.
11. typedef struct // socket info
12. {
13.     SOCKET hClnSock;
14.     SOCKADDR_IN clnAdr;
15. } PER_HANDLE_DATA, *LPPER_HANDLE_DATA;
```

```

16.
17. typedef struct      // buffer info
18. {
19.     OVERLAPPED overlapped;
20.     WSABUF wsaBuf;
21.     char buffer[BUF_SIZE];
22.     int rwMode;      // READ or WRITE
23. } PER_IO_DATA, *LPPER_IO_DATA;
24.
25. DWORD WINAPI EchoThreadMain(LPVOID CompletionPortIO);
26. void ErrorHandling(char *message);

```

代码说明

- 第11行：保存与客户端相连套接字的结构体。请各位注意观察该结构体何时被分配空间、如何被传递、如何被使用。
- 第17行：将I/O中使用的缓冲和重叠I/O中需要的OVERLAPPED结构体变量封装到同一结构体中进行定义。该结构体变量同样属于重点观察对象。

观察第17行结构体的使用方法前，应该明确如下事实：

“结构体变量地址值与结构体第一个成员的地址值相同。”

这也意味着下列代码段将输出EQUAL。

```

PER_IO_DATA ioData;
if(&ioData == &(ioData.overlapped))
    puts("EQUAL");
else
    puts("NOT EQUAL");

```

下列示例正是利用这种特性编写的，务必牢记。接下来介绍main函数部分。

❖ IOCPEchoServ_win.c: main函数

```

1. int main(int argc, char* argv[])
2. {
3.     WSADATA wsaData;
4.     HANDLE hComPort;
5.     SYSTEM_INFO sysInfo;
6.     LPPER_IO_DATA ioInfo;
7.     LPPER_HANDLE_DATA handleInfo;
8.
9.     SOCKET hServSock;
10.    SOCKADDR_IN servAdr;
11.    int recvBytes, i, flags=0;
12.    if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
13.        ErrorHandling("WSAStartup() error!");
14.

```

```

15. hComPort=CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
16. GetSystemInfo(&sysInfo);
17. for(i=0; i<sysInfo.dwNumberOfProcessors; i++)
18.     _beginthreadex(NULL, 0, EchoThreadMain, (LPVOID)hComPort, 0, NULL);
19.
20. hServSock=WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
21. memset(&servAdr, 0, sizeof(servAdr));
22. servAdr.sin_family=AF_INET;
23. servAdr.sin_addr.s_addr=htonl(INADDR_ANY);
24. servAdr.sin_port=htons(atoi(argv[1]));
25.
26. bind(hServSock, (SOCKADDR*)&servAdr, sizeof(servAdr));
27. listen(hServSock, 5);
28.
29. while(1)
30. {
31.     SOCKET hClntSock;
32.     SOCKADDR_IN clntAdr;
33.     int addrLen=sizeof(clntAdr);
34.
35.     hClntSock=accept(hServSock, (SOCKADDR*)&clntAdr, &addrLen);
36.     handleInfo=(LPPER_HANDLE_DATA)malloc(sizeof(PER_HANDLE_DATA));
37.     handleInfo->hClntSock=hClntSock;
38.     memcpy(&(handleInfo->clntAdr), &clntAdr, addrLen);
39.
40.     CreateIoCompletionPort((HANDLE)hClntSock, hComPort, (DWORD)handleInfo, 0);
41.
42.     ioInfo=(LPPER_IO_DATA)malloc(sizeof(PER_IO_DATA));
43.     memset(&(ioInfo->overlapped), 0, sizeof(OVERLAPPED));
44.     ioInfo->wsaBuf.len=BUF_SIZE;
45.     ioInfo->wsaBuf.buf=ioInfo->buffer;
46.     ioInfo->rwMode=READ;
47.     WSAREcv(handleInfo->hClntSock, &(ioInfo->wsaBuf),
48.             1, &recvBytes, &flags, &(ioInfo->overlapped), NULL);
49. }
50. return 0;
51. }

```

代码说明

- 第15行：创建CP对象。最后一个参数可以用非0值，所以可以向CP对象分配相当于核数（CPU数）的线程。
- 第16行：调用GetSystemInfo函数以获取当前系统信息。
- 第17、18行：成员变量dwNumberOfProcessors中将写入CPU个数（双核CPU时写入2）。通过这2行创建了与CPU个数相当的线程。另外，创建线程时传递第15行创建的CP对象句柄。线程将通过该句柄访问CP对象。换言之，CP对象将通过该句柄分配到线程。
- 第36~38行：动态分配PER_HANDLE_DATA结构体，并写入客户端相连套接字和客户端地址信息。

- 第40行：连接第15行创建的CP对象和第35行创建的套接字。针对该套接字的重叠I/O完成时，已完成信息将写入连接的CP对象，这会引发GetQueue...函数的返回。请注意观察第三个参数的值。该值是第36~38行中声明并初始化的结构体变量地址值，它同样是在GetQueued...函数返回时得到的。 ✓
- 第42行：动态分配PER_IO_DATA结构体变量空间。相当于同时准备了WSARecv函数中需要的OVERLAPPED结构体变量、WSABUF结构体变量及缓冲。 ✓
- 第46行：IOCP本身不会帮我们区分输入完成和输出完成的状态。无论输入还是输出，只通知完成I/O的状态，因此需要通过额外的变量区分这2种I/O。PER_IO_DATA结构体中的rwMode就用于完成该功能。
- 第47行：WSARecv函数的第七个参数为OVERLAPPED结构体变量地址值，该值可以在GetQueue...函数返回时得到。但结构体变量地址值与第一个成员的地址值相同，也就相当于传入了PER_IO_DATA结构体变量地址值。 ✓

各位有必要整理并思考上述代码说明中提到的“GetQueued...函数返回时得到的2个值”。最后给出线程main函数，这部分代码需要结合之前的main函数进行分析。

❖ IOCPEchoServ_win.c: 线程的main函数和错误处理函数

```

1.  DWORD WINAPI EchoThreadMain(LPVOID pComPort)
2.  {
3.      HANDLE hComPort=(HANDLE)pComPort;
4.      SOCKET sock;
5.      DWORD bytesTrans;
6.      LPPER_HANDLE_DATA handleInfo;
7.      LPPER_IO_DATA ioInfo;
8.      DWORD flags=0;
9.
10.     while(1)
11.     {
12.         GetQueuedCompletionStatus(hComPort, &bytesTrans,
13.             (LPDWORD)&handleInfo, (LPOVERLAPPED*)&ioInfo, INFINITE);
14.         sock=handleInfo->hClnSock;
15.
16.         if(ioInfo->rwMode==READ)
17.         {
18.             puts("message received!");
19.             if(bytesTrans==0) //传输EOF时
20.             {
21.                 closesocket(sock);
22.                 free(handleInfo); free(ioInfo);
23.                 continue;
24.             }
25.
26.             memset(&(ioInfo->overlapped), 0, sizeof(OVERLAPPED));
27.             ioInfo->wsaBuf.len=bytesTrans;
28.             ioInfo->rwMode=WRITE;
29.             WSARecv(sock, &(ioInfo->wsaBuf),
30.                 1, NULL, 0, &(ioInfo->overlapped), NULL);
31.

```

```

32.         ioInfo=(LPPER_IO_DATA)malloc(sizeof(PER_IO_DATA));
33.         memset(&(ioInfo->overlapped), 0, sizeof(OVERLAPPED));
34.         ioInfo->wsaBuf.len=BUF_SIZE;
35.         ioInfo->wsaBuf.buf=ioInfo->buffer;
36.         ioInfo->rwMode=READ;
37.         WSARecv(sock, &(ioInfo->wsaBuf),
38.                 1, NULL, &flags, &(ioInfo->overlapped), NULL);
39.     }
40.     else
41.     {
42.         puts("message sent!");
43.         free(ioInfo);
44.     }
45. }
46. return 0;
47. }
48.
49. void ErrorHandler(char *message)
50. {
51.     fputs(message, stderr);
52.     fputc('\n', stderr);
53.     exit(1);
54. }

```

代码说明

- 第1行：由线程运行EchoThreadMain函数。从该函数的第12行可以看到，调用了GetQueued...函数。调用GetQueued...函数的线程就是分配给CP对象的线程。
- 第12行：GetQueued...函数在I/O完成且已注册相关信息时返回（因为最后一个参数为INFINITE）。另外，返回时可以通过第三个和第四个参数得到之前提过的2个信息（若不太清楚，可以回顾main函数的代码说明）。
- 第16行：指针ioInfo中保存的既是OVERLAPPED结构体变量地址值，也是PER_IO_DATA结构体变量地址值。因此，可以通过检查rwMode成员中的值判断是输入完成还是输出完成。
- 第26~30行：将服务器端收到的消息发送给客户端。
- 第32~38行：再次发送消息后接收客户端消息。
- 第40行：完成的I/O为输出时执行的else区域。

关于IOCP的讲解到此结束。该示例可以结合StableEchoClnt_win.c运行，运行结果与普通回声服务器端/客户端没有区别，故省略。

+ IOCP 性能更优的原因

盲目地认为“因为是IOCP，所以性能更好”的想法并不可取。之前已介绍了Linux和Windows下多种服务器端模型，各位应该可以分析出它们在性能上的优势。将其在代码级别与select进行对比，可以发现如下特点。

- 因为是非阻塞模式的I/O，所以不会由I/O引发延迟。

- ☐ 查找已完成I/O时无需添加循环。
- ☐ 无需将作为I/O对象的套接字句柄保存到数组进行管理。
- ☐ 可以调整处理I/O的线程数，所以可在实验数据的基础上选用合适的线程数。

仅凭这些特点也能判断IOCP属于高性能模型，IOCP是Windows特有的功能，所以很大程度上要归功于操作系统。无需怀疑它提供的性能，我认为IOCP和Linux的epoll都是非常优秀的服务器端模型。

23.3 习题

- (1) 完成端口对象将分配多个线程用于处理I/O。如何创建这些线程？如何分配？请从源代码级别进行说明。
- (2) CreateIoCompletionPort函数与其他函数不同，提供2种功能。请问是哪2种？
- (3) 完成端口对象和套接字之间的连接意味着什么？如何连接？
- (4) 下列关于IOCP的说法错误的是？
 - a. 以最少的线程处理多数I/O的结构，因此可以减少上下文切换引起的性能低下。
 - b. 执行I/O的过程中，服务器端无需等待I/O完成，可以执行其他任务，故能提高CPU效率。
 - c. I/O完成时会自动调用相关Completion Routine函数，因此没必要调用特定函数以等待I/O完成。
 - d. 除Windows外，其他操作系统同样支持IOCP，所以这种模型具有良好的移植性。
- (5) 判断下列关于IOCP中选择合理线程数的方法是否合适。
 - ☐ 通常会选择与CPU数同样数量的线程。()
 - ☐ 最好在条件允许的范围内通过实验决定线程数。()
 - ☐ 分配的线程数越多越好。例如，1个线程就足够的情况下应该多分配几个，比如创建3个线程分配给IOCP。()
- (6) 利用本章的IOCP模型实现聊天服务器端，该聊天服务器端应当结合第20章的聊天客户端chat_clnt_win.c正常运行。编写程序时不必刻意套用本章IOCP示例中的框架，那样反而会加大实现难度。

Along!

alone!

Part 04

结束网络编程

制作HTTP服务器端

各位已学习了大量编程知识，即便如此，大家脑海中浮现的大部分还是回声服务器端/客户端，最多也就是聊天程序。但掌握了这些基础程序的编写方法，就相当于具备了开发应用层网络程序的基本能力。我们已经能够编写服务器端和客户端进行数据交换，接下来应该学习应用程序的编写方法。

24.1 HTTP 概要

本章以编写真实应用程序为目标，在所学理论知识的基础上，编写HTTP（Hypertext Transfer Protocol，超文本传输协议）服务器端，即Web服务器端。

+ 理解 Web 服务器端

互联网的普及使Web服务器端为大众熟知。下面是我对Web服务器端的定义：

“基于HTTP协议，将网页对应文件传输给客户端的服务器端。”

HTTP是Hypertext Transfer Protocol的缩写，Hypertext（超文本）是可以根据客户端请求而跳转的结构化信息。例如，各位通过浏览器访问图灵社区的主页时，首页文件将传输到浏览器并展现给大家，此时各位可以点击鼠标跳转到任意页面。这种可跳转的文本（Text）称为超文本。

HTTP协议又是什么呢？HTTP是以超文本传输为目的而设计的应用层协议，这种协议同样属于基于TCP/IP实现的协议，因此，我们也可以直接实现HTTP。从结果上看，实现该协议相当于实现Web服务器端。另外，浏览器也属于基于套接字的客户端，因为连接到任意Web服务器端时，浏览器内部也会创建套接字。只不过浏览器多了一项功能，它将服务器端传输的HTML格式的超文本解析为可读性较强的视图。总之，Web服务器端是以HTTP协议为基础传输超文本的服务器端。



HTTP 是专用名词

HTTP 中的 P 是 Protocol (协议) 的缩写, 所以 “HTTP 协议” 这种表示其实重复表达了 “协议” 一词。但 HTTP 属于专用名词, 因此通常直接表述为 “HTTP 协议”。

+ HTTP

下面详细讨论HTTP协议。虽然它相对简单, 但要完全驾驭也并非易事。接下来只介绍编写Web服务器端时的必要内容。

无状态的Stateless协议

为了在网络环境下同时向大量客户端提供服务, HTTP协议的请求及响应方式设计如图24-1所示。

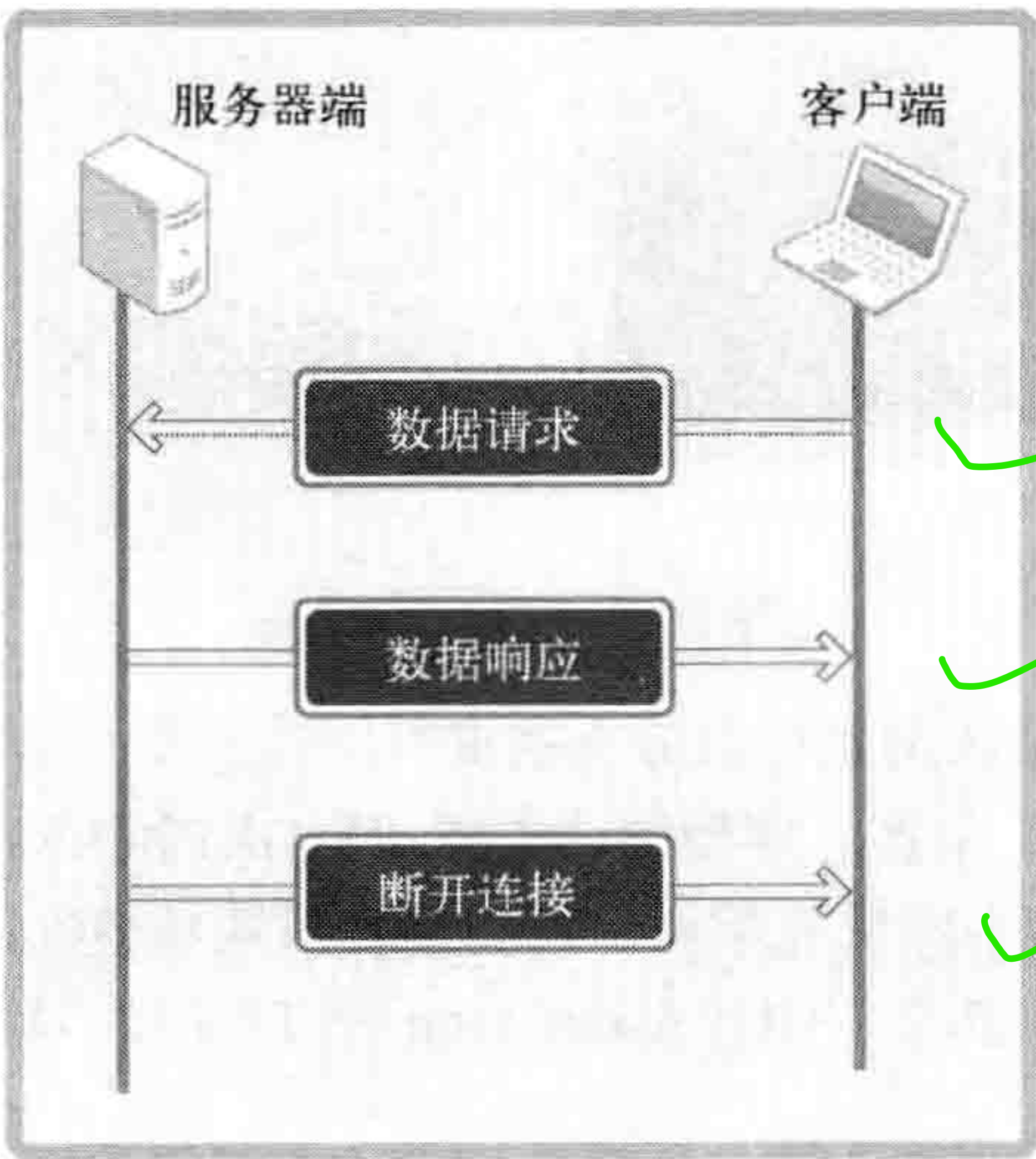


图24-1 HTTP请求/响应过程

从图24-1中可以看到, 服务器端响应客户端请求后立即断开连接。换言之, 服务器端不会维持客户端状态。即使同一客户端再次发送请求, 服务器端也无法辨认出是原先那个, 而会以相同方式处理新请求。因此, HTTP又称 “无状态的Stateless协议”。

提示

Cookie & Session

为了弥补 HTTP 无法保持连接的缺点，Web 编程中通常会使用 Cookie 和 Session 技术。相信各位都接触过购物网站的购物车功能，即使关闭浏览器也不会丢失购物车内的信息（甚至不用登录）。这种保持状态的功能都是通过 Cookie 和 Session 技术实现的。

请求消息（Request Message）的结构

下面介绍客户端向服务器端发送的请求消息的结构。Web 服务器端需要解析并响应客户端请求，客户端和服务端之间的数据请求方式标准如图24-2所示。

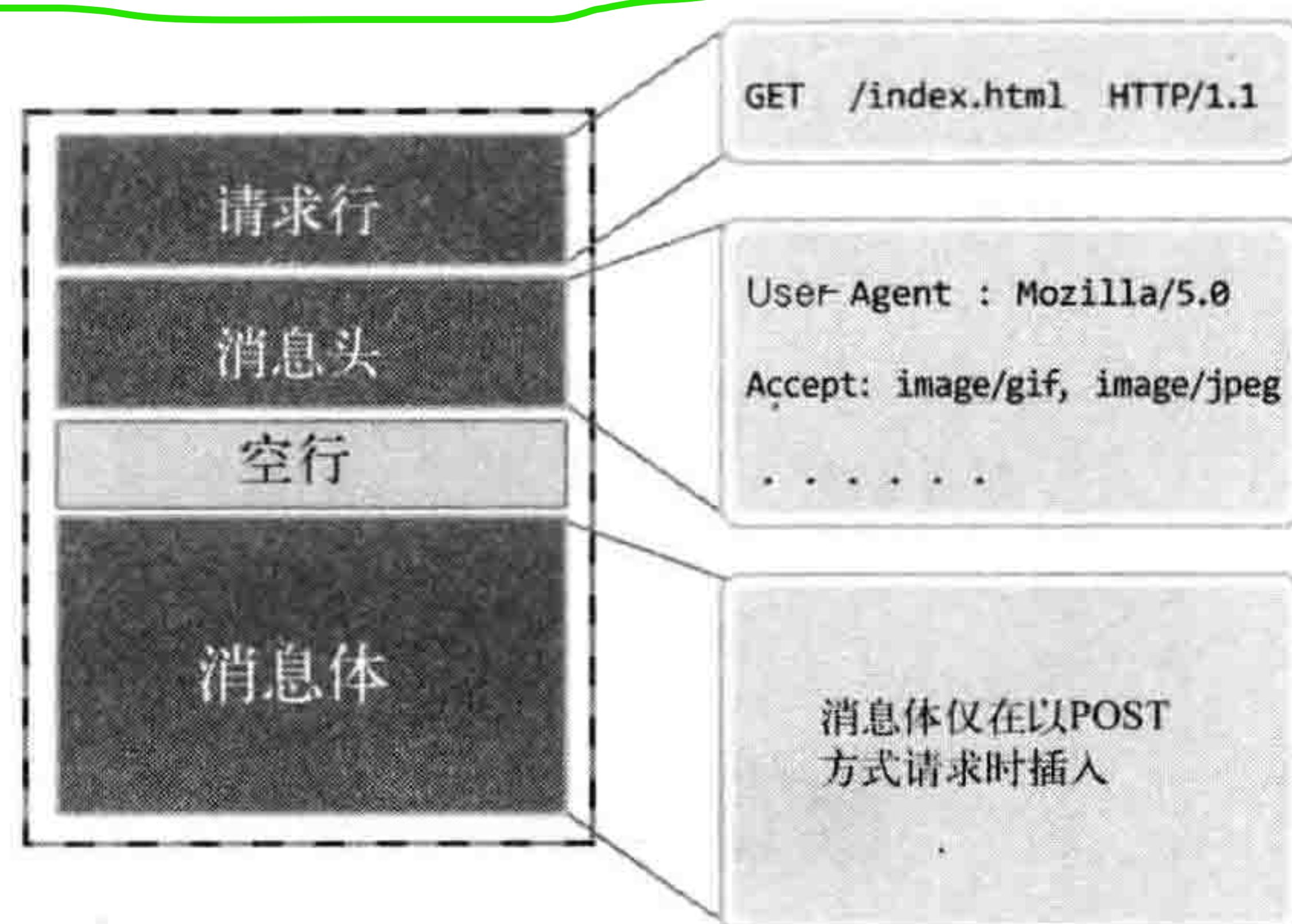


图24-2 HTTP请求头

从图24-2中可以看到，请求消息可以分为请求行、消息头、消息体等3个部分。其中，请求行含有请求方式（请求目的）信息。典型的请求方式有GET和POST，GET主要用于请求数据，POST主要用于传输数据。为了降低复杂度，我们实现只能响应GET请求的Web服务器端。下面解释图24-2中的请求行信息。其中“GET /index.html HTTP/1.1”具有如下含义：

“请求（GET）index.html文件，希望以1.1版本的HTTP协议进行通信。”

请求行只能通过1行（line）发送，因此，服务器端很容易从HTTP请求中提取第一行，并分析请求行中的信息。

请求行下面的消息头中包含发送请求的（将要接收响应信息的）浏览器信息、用户认证信息等关于HTTP消息的附加信息。最后的消息体中装有客户端向服务器端传输的数据，为了装入数据，需要以POST方式发送请求。但我们的目标是实现GET方式的服务器端，所以可以忽略这部分内容。另外，消息体和消息头之间以空行分开，因此不会发生边界问题。

响应消息（Response Message）的结构

下面介绍Web服务器端向客户端传递的响应信息的结构。从图24-3中可以看到，该响应消息由状态行、头信息、消息体等3个部分构成。状态行中含有关于请求的状态信息，这是其与请求消息相比最为显著的区别。

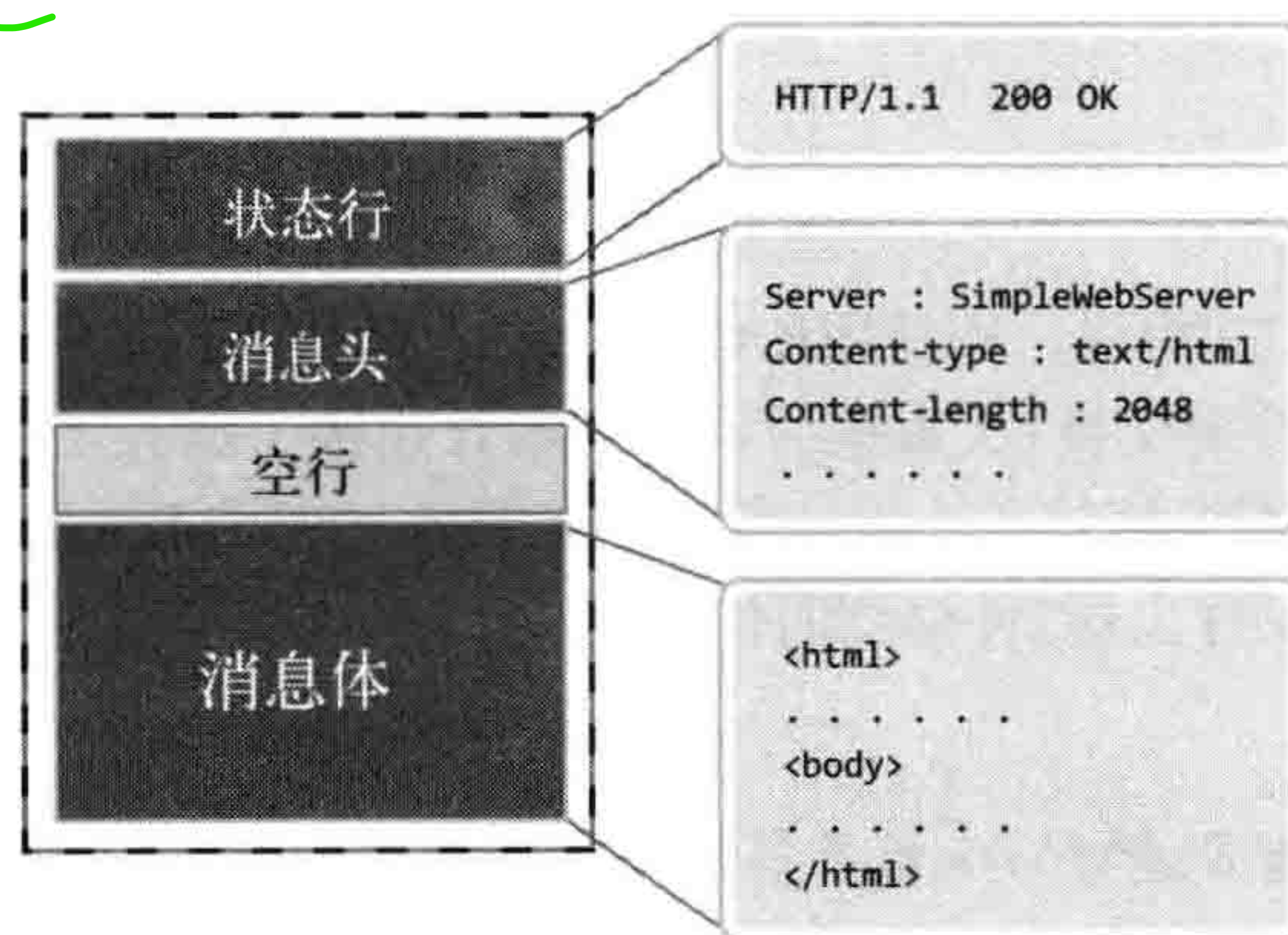


图24-3 HTTP响应头

从图24-3中可以看到，第一个字符串状态行中含有关于客户端请求的处理结果。例如，客户端请求index.html文件时，表示index.html文件是否存在、服务器端是否发生问题而无法响应等不同情况的信息将写入状态行。图24-3中的“HTTP/1.1 200 OK”具有如下含义：

“我想用HTTP1.1版本进行响应，你的请求已正确处理（200 OK）。”

表示“客户端请求的执行结果”的数字称为状态码，典型的有以下几种。

- ❑ 200 OK：成功处理了请求！
- ❑ 404 Not Found：请求的文件不存在！
- ❑ 400 Bad Request：请求方式错误，请检查！

消息头中含有传输的数据类型和长度等信息。图24-3中的消息头含有如下信息：

“服务器端名为SimpleWebServer，传输的数据类型为text/html（html格式的文本数据）。数据长度不超过2048字节。”

最后插入1个空行后，通过消息体发送客户端请求的文件数据。以上就是实现Web服务器端过程中必要的HTTP协议。要编写完整的Web服务器端还需要更多HTTP协议相关知识，而对于我们的目标而言，这些内容已经足够了。

24.2 实现简单的 Web 服务器端

现在开始在HTTP协议的基础上编写Web服务器端。先给出Windows平台下的示例，再给出Linux下的示例。前面介绍了HTTP协议的相关背景知识，有了这些基础就不难分析源代码。因此，除了一些简单的注释外，不再另行说明代码。

+ 实现基于 Windows 的多线程 Web 服务器端

Web服务器端采用HTTP协议，即使用IOCP或epoll模型也不会大幅提升性能（当然并不是完全没有）。客户端和服务端交换1次数据后将立即断开连接，没有足够时间发挥IOCP或epoll的优势。在服务器端和客户端保持较长连接的前提下频繁发送大小不一的消息时（最典型的就是网游服务器端），才能真正发挥出这2种模型的优势。

提示

能否通过 Web 服务器端比较 IOCP 和 epoll 的性能

利用 IOCP 和 epoll 实现 Web 服务器端”本身没有问题，现实中有时也会通过 IOCP 或 epoll 实现类似的服务器端，但无法通过这种服务器端完全体会到 IOCP 和 epoll 的优点。我从事过软件性能评估相关工作，当时，客户公司的工程师向我展示过 Web 服务器端中 IOCP 和 epoll 的性能分析结果，我也耐心解释了为何分析结果没有太大意义。

因此，我通过多线程模型实现Web服务器端。也就是说，客户端每次请求时，都创建1个新线程响应客户端请求。

❖ webserv_win.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock2.h>
#include <process.h>

#define BUF_SIZE 2048
#define BUF_SMALL 100

unsigned WINAPI RequestHandler(void* arg);
char* ContentType(char* file);
void SendData(SOCKET sock, char* ct, char* fileName);
void SendErrorMsg(SOCKET sock);
void ErrorHandling(char *message);

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

```

{
    WSADATA wsaData;
    SOCKET hServSock, hClntSock;
    SOCKADDR_IN servAdr, clntAdr;

    HANDLE hThread;
    DWORD dwThreadID;
    int clntAdrSize;

    if(argc!=2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }

    if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
        ErrorHandling("WSAStartup() error!");

    hServSock=socket(PF_INET, SOCK_STREAM, 0);
    memset(&servAdr, 0, sizeof(servAdr));
    servAdr.sin_family=AF_INET;
    servAdr.sin_addr.s_addr=htonl(INADDR_ANY);
    servAdr.sin_port=htons(atoi(argv[1]));

    if(bind(hServSock, (SOCKADDR*)&servAdr, sizeof(servAdr))==SOCKET_ERROR)
        ErrorHandling("bind() error");
    if(listen(hServSock, 5)==SOCKET_ERROR)
        ErrorHandling("listen() error");

    /*请求及响应*/
    while(1)
    {
        clntAdrSize=sizeof(clntAdr);
        hClntSock=accept(hServSock, (SOCKADDR*)&clntAdr, &clntAdrSize);
        printf("Connection Request : %s:%d\n",
            inet_ntoa(clntAdr.sin_addr), ntohs(clntAdr.sin_port));
        hThread=(HANDLE)_beginthreadex(
            NULL, 0, RequestHandler, (void*)hClntSock, 0, (unsigned*)&dwThreadID);
    }
    closesocket(hServSock);
    WSACleanup();
    return 0;
}

unsigned WINAPI RequestHandler(void *arg)
{
    SOCKET hClntSock=(SOCKET)arg;
    char buf[BUF_SIZE];
    char method[BUF_SMALL];
    char ct[BUF_SMALL];
    char fileName[BUF_SMALL];

    recv(hClntSock, buf, BUF_SIZE, 0);
    if(strstr(buf, "HTTP/")==NULL) // 查看是否为HTTP提出的请求
    {

```

```

        SendErrorMsg(hClnSock);
        closesocket(hClnSock);
        return 1;
    }

    strcpy(method, strtok(buf, " /"));
    if(strcmp(method, "GET")) // 查看是否为GET方式的请求
        SendErrorMsg(hClnSock);

    strcpy(fileName, strtok(NULL, " /")); // 查看请求文件名
    strcpy(ct, ContentType(fileName)); // 查看Content-type
    SendData(hClnSock, ct, fileName); // 响应
    return 0;
}

void SendData(SOCKET sock, char* ct, char* fileName)
{
    char protocol[]="HTTP/1.0 200 OK\r\n";
    char servName[]="Server:simple web server\r\n";
    char cntLen[]="Content-length:2048\r\n";
    char cntType[BUF_SMALL];
    char buf[BUF_SIZE];
    FILE* sendFile;

    sprintf(cntType, "Content-type:%s\r\n\r\n", ct);
    if((sendFile=fopen(fileName, "r"))==NULL)
    {
        SendErrorMsg(sock);
        return;
    }

    /*传输头信息*/
    send(sock, protocol, strlen(protocol), 0);
    send(sock, servName, strlen(servName), 0);
    send(sock, cntLen, strlen(cntLen), 0);
    send(sock, cntType, strlen(cntType), 0);

    /*传输请求数据*/
    while(fgets(buf, BUF_SIZE, sendFile)!=NULL)
        send(sock, buf, strlen(buf), 0);

    closesocket(sock); // 由HTTP协议响应后断开
}

void SendErrorMsg(SOCKET sock) // 发生错误时传递消息
{
    char protocol[]="HTTP/1.0 400 Bad Request\r\n";
    char servName[]="Server:simple web server\r\n";
    char cntLen[]="Content-length:2048\r\n";
    char cntType[]="Content-type:text/html\r\n\r\n";
    char content[]="<html><head><title>NETWORK</title></head>"
        "<body><font size=+5><br> 发生错误! 查看请求文件名和请求方式! "
        "</font></body></html>";

```

```

send(sock, protocol, strlen(protocol), 0);
send(sock, servName, strlen(servName), 0);
send(sock, cntLen, strlen(cntLen), 0);
send(sock, cntType, strlen(cntType), 0);
send(sock, content, strlen(content), 0);
closesocket(sock);
}

char* ContentType(char* file) // 区分Content-type
{
    char extension[BUF_SMALL];
    char fileName[BUF_SMALL];
    strcpy(fileName, file);
    strtok(fileName, ".");
    strcpy(extension, strtok(NULL, "."));
    if(!strcmp(extension, "html")||!strcmp(extension, "htm"))
        return "text/html";
    else
        return "text/plain";
}

void ErrorHandler(char* message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}

```

图24-4给出上述示例的运行结果。首先启动该服务器端，再启动Web浏览器进行连接。



图24-4 运行结果1

从图24-4的运行结果可以看出，地址栏中输入了如下地址：

`http://localhost:9190/index.html`

该请求相当于连接到IP为127.0.0.1、端口为9190的套接字，并请求获取index.html文件。可以用回送地址127.0.0.1代替“localhost”。通过图24-5观察客户端，可以看出未找到请求的文件。



图24-5 运行结果2

可以在页面上半部分找到状态码400，因为地址栏中输入的index2.html文件不存在。

+ 实现基于 Linux 的多线程 Web 服务器端

Linux下的Web服务器端与上述示例不同，将使用标准I/O函数。在此列出的目的主要是为了让各位多复习各种知识点，没有任何特殊含义。

❖ webserv_linux.c

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

#define BUF_SIZE 1024
#define SMALL_BUF 100

void* request_handler(void* arg);
void send_data(FILE* fp, char* ct, char* file_name);
char* content_type(char* file);
void send_error(FILE* fp);
void error_handling(char* message);

int main(int argc, char *argv[])
{
```

```

int serv_sock, clnt_sock;
struct sockaddr_in serv_adr, clnt_adr;
int clnt_adr_size;
char buf[BUF_SIZE];
pthread_t t_id;
if(argc!=2) {
    printf("Usage : %s <port>\n", argv[0]);
    exit(1);
}

serv_sock=socket(PF_INET, SOCK_STREAM, 0);
memset(&serv_adr, 0, sizeof(serv_adr));
serv_adr.sin_family=AF_INET;
serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
serv_adr.sin_port = htons(atoi(argv[1]));
if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
    error_handling("bind() error");
if(listen(serv_sock, 20)==-1)
    error_handling("listen() error");

while(1)
{
    clnt_adr_size=sizeof(clnt_adr);
    clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_size);
    printf("Connection Request : %s:%d\n",
        inet_ntoa(clnt_adr.sin_addr), ntohs(clnt_adr.sin_port));
    pthread_create(&t_id, NULL, request_handler, &clnt_sock);
    pthread_detach(t_id);
}
close(serv_sock);
return 0;
}

void* request_handler(void *arg)
{
    int clnt_sock=((int*)arg);
    char req_line[SMALL_BUF];
    FILE* clnt_read;
    FILE* clnt_write;

    char method[10];
    char ct[15];
    char file_name[30];

    clnt_read=fdopen(clnt_sock, "r");
    clnt_write=fdopen(dup(clnt_sock), "w");
    fgets(req_line, SMALL_BUF, clnt_read);
    if(strstr(req_line, "HTTP/")==NULL)
    {
        send_error(clnt_write);
        fclose(clnt_read);
        fclose(clnt_write);
        return;
    }
}

```

```
strcpy(method, strtok(req_line, " /"));
strcpy(file_name, strtok(NULL, " /"));
strcpy(ct, content_type(file_name));
if(strcmp(method, "GET")!=0)
{
    send_error(clnt_write);
    fclose(clnt_read);
    fclose(clnt_write);
    return;
}

fclose(clnt_read);
send_data(clnt_write, ct, file_name);
}

void send_data(FILE* fp, char* ct, char* file_name)
{
    char protocol[]="HTTP/1.0 200 OK\r\n";
    char server[]="Server:Linux Web Server \r\n";
    char cnt_len[]="Content-length:2048\r\n";
    char cnt_type[SMALL_BUF];
    char buf[BUF_SIZE];
    FILE* send_file;

    sprintf(cnt_type, "Content-type:%s\r\n\r\n", ct);
    send_file=fopen(file_name, "r");
    if(send_file==NULL)
    {
        send_error(fp);
        return;
    }

    /*传输头信息*/
    fputs(protocol, fp);
    fputs(server, fp);
    fputs(cnt_len, fp);
    fputs(cnt_type, fp);

    /*传输请求数据*/
    while(fgets(buf, BUF_SIZE, send_file)!=NULL)
    {
        fputs(buf, fp);
        fflush(fp);
    }
    fflush(fp);
    fclose(fp);
}

char* content_type(char* file)
{
    char extension[SMALL_BUF];
    char file_name[SMALL_BUF];
    strcpy(file_name, file);
```

```

    strtok(file_name, ".");
    strcpy(extension, strtok(NULL, "."));

    if(!strcmp(extension, "html")||!strcmp(extension, "htm"))
        return "text/html";
    else
        return "text/plain";
}

void send_error(FILE* fp)
{
    char protocol[]="HTTP/1.0 400 Bad Request\r\n";
    char server[]="Server:Linux Web Server \r\n";
    char cnt_len[]="Content-length:2048\r\n";
    char cnt_type[]="Content-type:text/html\r\n\r\n";
    char content[]="<html><head><title>NETWORK</title></head>"
        "<body><font size=+5><br>发生错误! 查看请求文件名和请求方式! "
        "</font></body></html>";

    fputs(protocol, fp);
    fputs(server, fp);
    fputs(cnt_len, fp);
    fputs(cnt_type, fp);
    fflush(fp);
}

void error_handling(char* message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}

```

关于TCP/IP套接字编程的讲解到此结束。第25章将简单介绍网络编程进阶部分需要学习的内容。

24.3 习题

(1) 下列关于Web服务器端和Web浏览器的说法错误的是?

- Web浏览器并不是通过自身创建的套接字连接服务器端的客户端。
- Web服务器通过TCP套接字提供服务，因为它将保持较长的客户端连接并交换数据。
- 超文本与普通文本的最大区别是其具有可跳转的特性。
- Web服务器端可视为向浏览器提供请求文件的文件传输服务器端。
- 除Web浏览器外，其他客户端都无法访问Web服务器端。

(2) 下列关于HTTP协议的描述错误的是?

- HTTP协议是无状态的Stateless协议，不仅可以通过TCP实现，还可通过UDP实现。

- b. HTTP 协议是无状态的 Stateless 协议，因为其在 1 次请求和响应过程完成后立即断开连接。因此，如果同一服务器端和客户端需要 3 次请求及响应，则意味着要经过 3 次套接字创建过程。
 - c. 服务器端向客户端传递的状态码中含有请求处理结果信息。
 - d. HTTP 协议是基于因特网的协议，因此，为了同时向大量客户端提供服务，HTTP 被设计为 Stateless 协议。
- (3) IOCP 和 epoll 是可以保证高性能的典型服务器端模型，但如果在基于 HTTP 协议的 Web 服务器端使用这些模型，则无法保证一定能得到高性能。请说明原因。

done!

done!

相信各位已经对套接字编程和系统编程有了一定认识。入门的过程感觉有些漫长，可一旦有了基础，接下来的学习并非难事。进入本章意味着我们已经走过了漫长的入门过程。

本章将介绍进阶学习的内容。我希望多介绍一些好的学习方法和经验教训，这也是编写本章的宗旨。

25.1 网络编程学习的其他内容

+ 成为网络程序员的前提是成为程序员

如果对某一领域感兴趣，自然会更关注该领域，并会产生从事该领域相关工作的想法。也有读者向我询问如何成为一名网络程序员，但按照这部分读者的标准，我也很难成为合格的网络程序员。因为我的兴趣并非完全集中于网络领域，而且现阶段也并不从事网络编程相关工作。但工作中也会遇到网络编程的部分，这种情况下也会编写网络程序。

网络编程曾被认为是一种专业领域（现阶段也有人这么认为），但大部分最新软件是基于网络环境开发的，因此，网络编程成为程序员的一种必备技能。

如果对网络编程感兴趣，就应该在兴趣驱使学习程序员必备的所有知识。因为在现代服务器端，收发数据的网络编程部分所占篇幅远远比不上其他部分。优秀的服务器端并不是由优秀的网络程序员实现的，而是由优秀的程序员编写的。

+ 编写这些程序时的参考书

编程经验的不足使我们更加依赖书籍。我接触陌生领域的项目时，也会收集国内外最新出版的图书。但我们无法通过书籍解决项目中遇到的问题，因为软件本身属于创造性工作的产物。编

写同样功能的软件时，每位程序员的实现方法各不相同，此处没有正确答案。本身没有正解的东西很难从书中找到需要的信息，因此，在具备了坚实的理论知识的基础上，应该加强编程训练。

假设各位要开发在线俄罗斯方块对战游戏，大家能在身边找到有经验的程序员。这时不应该提如下这种问题：

“我想实现在线俄罗斯方块对战游戏，该如何编写呢？应该读哪些书？”

我能理解各位的焦虑，但提出这种问题无法得到满意的答复（被提问的人也不知所措）。应该缩小问题范围，而且问题要具体：

“我想实现在线俄罗斯方块对战游戏。用什么方法传输对方的砖块信息比较合理呢？我想出了一种方法，不知是否合理。”

提问的范围越小、方式越具体越好。一个大问题可以分成若干个子问题，通过这样的提问可以思考和理解更多内容。

+ 后续学习内容

大部分学生对系统编程（System Programming）有着极深的恐惧感，特别是在没能理解其本质的情况下，这种感觉尤为深刻。但本书的绝大部分内容都与系统编程有关，也就是说，各位已经掌握了相当多的系统编程知识。

网络编程与操作系统有着密切的关系，IOCP和epoll就是典型的例子。因此，我们无法完全避免依赖于操作系统的代码。而编写依赖于操作系统的代码时，有必要深入学习系统编程。因此，如果各位想了解下一步应该学习哪些内容，我建议大家进一步学习Windows和Linux系统编程。学习过程中会加深对操作系统的理解，同时，系统编程知识可以应用到多种领域，可谓一举多得。操作系统是计算机专业的5大核心科目之一，学习系统编程必定有百利而无一害。

25.2 网络编程相关书籍介绍

最后推荐一些书目，各位可以将此视为我的主观建议。大家身边这种人越多越好，也希望各位多倾听其他前辈的意见。

+ 系统编程相关书籍

如前所述，最好多学习系统编程相关知识。下面介绍2本相关书籍。

✓ 《UNIX环境高级编程（第2版）》

这本书的作者是鼎鼎大名的W. Richard Stevens，由Addison Wesley出版。这是一本广为人知

的好书，如果您希望学习UNIX系列操作系统的系统编程，我向您强烈推荐。这本书的学习方法大致可分为2种：第一种方法是短时间内浏览一遍，通过这种快速阅读可以了解主要内容，并掌握各种系统级别（操作系统级别）的函数，浏览后可以参考需要的部分；第二种方法是从开始就把本书当作参考书，如果已经有了其他操作系统下的编程经验，可以通过这种方式提高学习效率。对于缺乏系统编程经验的读者，我还是推荐第一种方法。

✔ 《Windows核心编程》

本书的作者是Jeffrey Richter，由Microsoft Press出版。可以说本书与前一本书分别引领了Windows和UNIX领域的编程技术趋势。

+ 协议相关书籍

相信各位在学习网络编程时也有所体会，要想编好网络程序，需要深入理解TCP/IP协议。下面介绍与之相关的2本书。

✔ 《TCP/IP详解》（卷1~卷3）

本书的作者同样是W. Richard Stevens，也由Addison Wesley出版。毋庸置疑，《TCP/IP详解》系列极为有名，网络方向的研究生案头一般都会摆着这3本书（特别是卷1和卷2）。但就各位现在的水平而言，卷1已经足够了。卷1讲解的是TCP/IP协议，卷2和卷3主要介绍TCP/IP的代码实现和扩展的（Transactional）TCP协议。下面为初学者推荐另一本书。

✔ 《TCP/IP协议族》

本书的作者是Behrouz A. Forouzan，由McGraw Hill出版。其优点是语句简洁，并加入大量插图以帮助理解。这本书本身就浅显易懂，改版中又加入了最新的热点问题，不仅是学习TCP/IP协议的好书，而且涵盖了多种应用层协议。据我所知，不少大学都将此书用作教材。

所有学习到此结束，各位辛苦了！

感谢各位选择本书！

作者 尹圣雨敬上

索引

B

本地广播 237, 239
标准I/O函数 246, 252, 258
标准错误 9
标准输出 9, 222, 250
标准输入 9, 202, 224
并发服务器 155, 173, 192

C

CRITICAL_SECTION对象 329
传输 (Transport) 层 101

D

大端序 (Big Endian) 42
单线程模型 318
地址分配错误 (Binding Error) 146
地址再分配 149
端口号 3, 6, 9
断开连接 95, 100, 102
堆 (Heap) 285
多播 (Multicast) 230
多进程服务器端 155, 157, 194
多线程服务器端 284, 306, 339
多线程模型 317, 326, 394

E

二进制信号量 304, 334, 336

F

非阻塞I/O 370
非阻塞模式 278, 281, 357
分布式数据库系统, 130
分离I/O流 255
服务器端套接字 6, 8, 21
复制文件描述符 177, 216, 260

G

管道 (PIPE) 183
广播 (Broadcast) 236

H

函数调用相关规定 321
后台处理 (Background Processing) 161
互斥量对象 331, 332
滑动窗口 (Sliding Window) 92
缓冲 11, 13, 23
缓冲 (buffer) 28
回声服务器端 72, 76, 80
回声客户端 72, 74, 76

I

I/O复用 (Multiplexing) 194
I/O复用服务器端 196, 203, 207
I/O缓冲 91, 100, 140

J

基于事件对象的同步 336
基于信号量对象的同步 334
监听 (listening) 套接字 6
僵尸 (Zombie) 进程 159
进程 (Process) 156
进程ID 157, 161, 164
进程间双向通信 185
进程间通信 183, 188, 193

K

开放式系统 (Open System) 61
客户端套接字 6, 8, 69

L

连接请求等待队列 65, 68, 69
链路层 62, 81
临界区 (Critical Section) 292
流 (Stream) 119
流形成的状态 119
路由 (Routing) 212, 231
路由器 (Router) 37

M

manual-reset模式事件对象 348
默认DNS服务器 128, 139

N

non-signaled状态 322, 331, 338
内核模式同步 328, 331, 343

O

OSI 7 Layer (层) 61

Q

轻量级进程 285
请求消息 (Request Message) 392
全双工 (Full-duplex) 方式 93

S

事件对象 336, 347, 353
事件同步对象 336
受理连接请求 5, 6, 66
数据边界 (Boundary) 28, 35
数据区 285, 286

T

TCP/IP协议栈 59, 62, 81
TCP/UDP层 63
Time-wait状态 145, 150, 154
套接字句柄 18, 23, 32
套接字类型 26, 32, 142
条件触发 (Level Trigger) 273
条件触发的事件特性 274
同步I/O 344
同步消息 93
同时访问 183, 261, 292

W

完成端口对象 379, 387
网络编程 2, 4, 15
网络地址 (Internet Address) 36
网络地址 (网络ID) 37
网络接口卡 38, 62
网络流量 151, 230, 243
网络字节序 (Network Byte Order) 43
文件句柄 9, 10, 20
文件描述符 (File Descriptor) 9

X

系统函数 20, 130, 247

线程 72, 156, 193
线程ID 287, 289, 307
响应消息 (Response Message) 393
小端序 (Little Endian) 42
协议 (Protocol) 26
协议族 (Protocol Family) 27
信号处理 (Signal Handling) 165
信号量对象 304, 334
虚拟网络 236, 243

Y

异步 (Asynchronous) 344
异步I/O 345, 370

异步通知I/O模型 344, 350, 352
应用层 64, 85, 390
应用层协议 64, 85, 87
用户模式 (User mode) 327
用户模式同步 328, 343
域名 128, 138

Z

栈 (Stack) 285
知名端口 (Well-known PORT) 39
重叠I/O 228, 357, 370
字节序 41, 58

欢迎加入

图灵社区 ituring.com.cn

——最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

优惠提示：现在购买电子书，读者将获赠书款20%的社区银子，可用于兑换纸质样书。

——最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

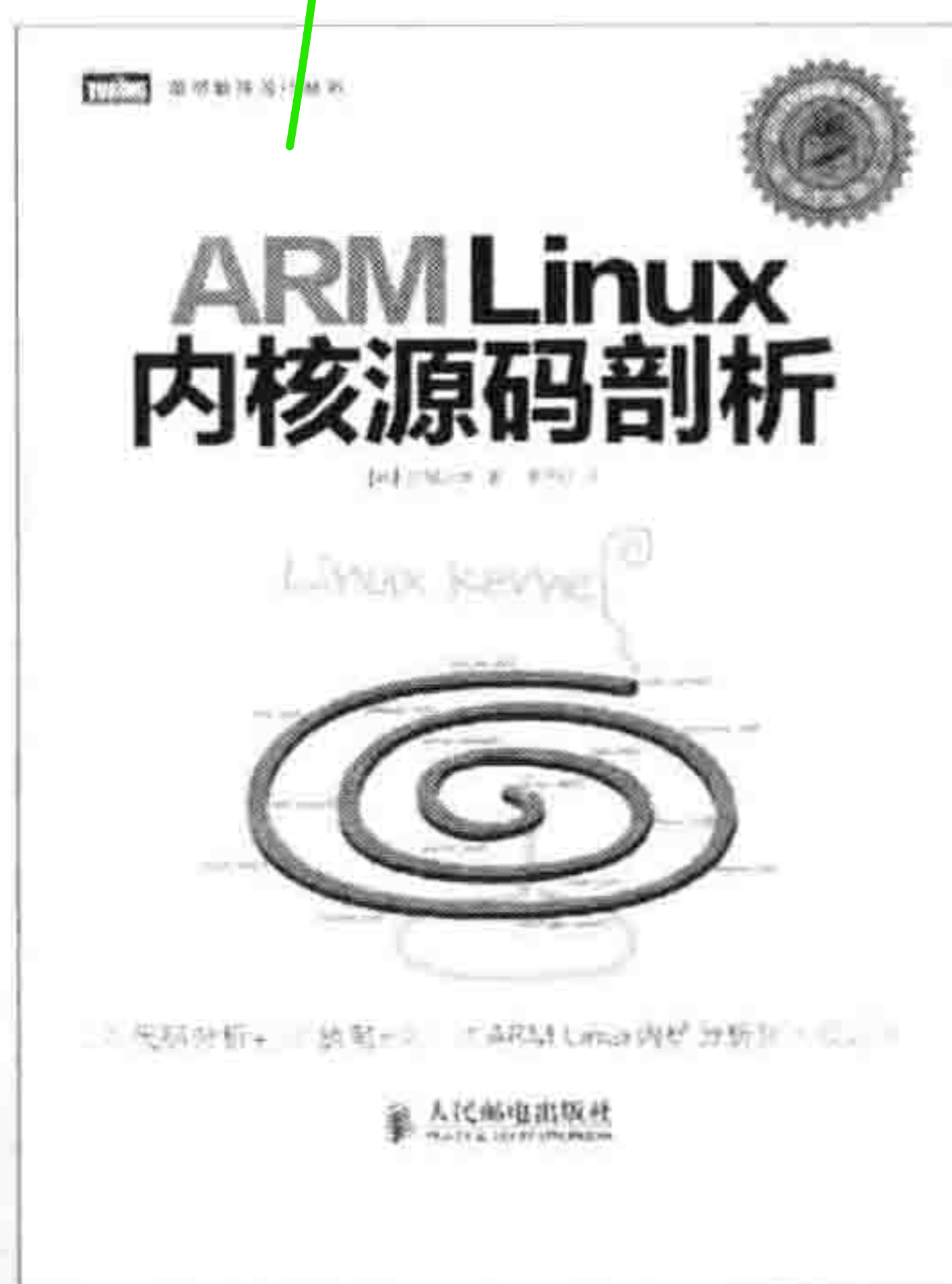
图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

——最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、乐译、评选等多种活动，赢取积分和银子，积累个人声望。

图灵最新重点图书



在本书是多位作者在 3 年 Liunx 内核分析经验和庞大资料基础上写成的，收录了其他同类书未曾讲解的内容并进行逐行分析，一扫当前市场中其他理论书带给读者的郁闷。书中详细的代码分析与大量插图能够使读者对 Linux 内核及 ARM 获得正确认识，自然而然习得如何有效分析定期发布的 Linux 内核。

本书适合想从 Linux 内核启动开始透彻分析全部启动过程的读者，因 Linux 代码量庞大而束手无策的人、想要了解 Linux 实际运行过程的人、渴求 OS 实操理论的人，本书必将成为他们不可或缺的参考书。

ARM Linux 内核源码剖析

书号: 978-7-115-35910-0

作者: [韩]尹锡训等著 崔范松译

定价: 99.00 元

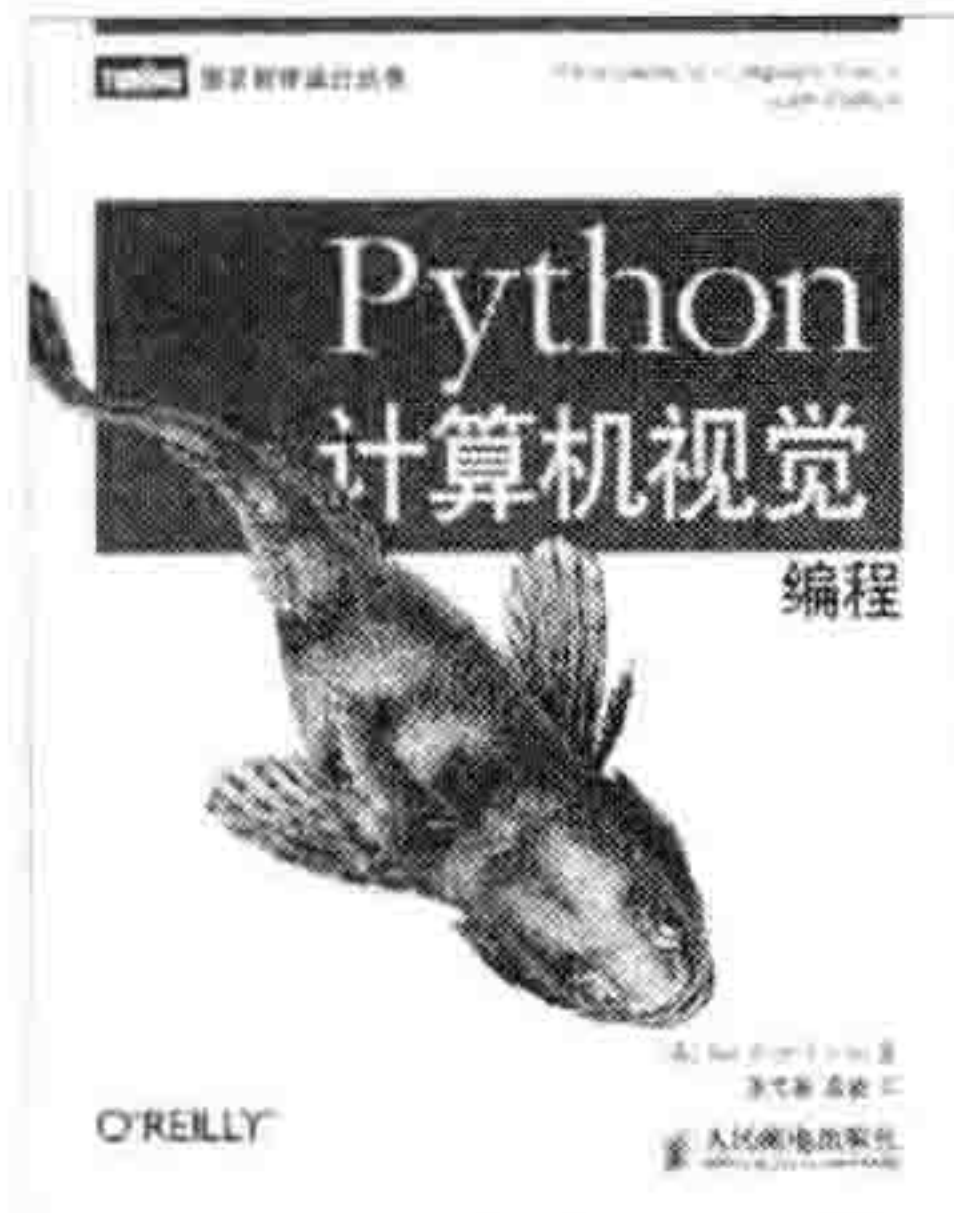


逆向工程核心原理

书号: 978-7-115-35018-3

作者: [韩]李承远著 武传海译

定价: 109.00 元



Python 计算机视觉编程

书号: 978-7-115-35232-3

作者: Jan Erik Solem

定价: 69.00 元



深入理解 Oracle 12c 数据库管理 (第2版)

书号: 978-7-115-35540-9

作者: Darl Kuhn

定价: 119.00 元



两周自制脚本语言

书号: 978-7-115-35564-5

作者: [日]千叶滋著 陈筱烟译

定价: 59.00 元



机器学习系统设计

书号: 978-7-115-35682-6

作者: Willi Richert, Luis Pedro Coelho

定价: 49.00 元



Android 编程实战

书号: 978-7-115-35733-5

作者: Erik Hellman

定价: 69.00 元