

# 《TCP/IP网络编程》学习笔记

此仓库是我的《TCP/IP网络编程》学习笔记及具体代码实现，代码部分请参考本仓库对应章节文件夹下的代码。如果本笔记的内容对你有用，请点击一个 [star](#)，转载请注明出处，谢谢。

我的环境是：Ubuntu18.04 LTS

编译器版本：`g++ (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0` 和 `gcc (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0`

所以本笔记中只学习有关于 Linux 的部分。

本项目在 GitHub 地址为：[TCP-IP-NetworkNote](#)

如果在阅读本笔记的过程中发现错别字，及 bug，请向本项目提交 [PR](#)。

笔记的 PDF 版本可以在本项目 [releases](#) 中找到及下载。

## 第 1 章：理解网络编程和套接字

本章代码，在[TCP-IP-NetworkNote](#)中可以找到，直接点连接可能进不去。

### 1.1 理解网络编程和套接字

#### 1.1.1 构建打电话套接字

以电话机打电话的方式来理解套接字。

调用 **socket** 函数（安装电话机）时进行的对话：

问：接电话需要准备什么？

答：当然是电话机。

有了电话机才能安装电话，于是就要准备一个电话机，下面函数相当于电话机的套接字。

```
1 #include <sys/socket.h>
2 int socket(int domain, int type, int protocol);
3 //成功时返回文件描述符, 失败时返回-1
```

调用 **bind** 函数（分配电话号码）时进行的对话：

问：请问我的电话号码是多少

答：我的电话号码是123-1234

套接字同样如此。就想给电话机分配电话号码一样，利用以下函数给创建好的套接字分配地址信息（IP地址和端口号）：

```
1 #include <sys/socket.h>
2 int bind(int sockfd, struct sockaddr *myaddr, socklen_t addrlen);
3 //成功时返回0, 失败时返回-1
```

调用 `bind` 函数给套接字分配地址之后，就基本完成了所有的准备工作。接下来是需要连接电话线并等待来电。

#### 调用 `listen` 函数（连接电话线）时进行的对话：

问：已架设完电话机后是否只需链接电话线？

答：对，只需要连接就能接听电话。

一连接电话线，电话机就可以转换为可接听状态，这时其他人可以拨打电话请求连接到该机。同样，需要把套接字转化成可接受连接状态。

```
1 #include <sys/socket.h>
2 int listen(int sockfd, int backlog);
3 //成功时返回0, 失败时返回-1
```

连接好电话线以后，如果有人拨打电话就响铃，拿起话筒才能接听电话。

#### 调用 `accept` 函数（拿起话筒）时进行的对话：

问：电话铃响了，我该怎么办？

答：接听啊。

```
1 #include <sys/socket.h>
2 int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
3 //成功时返回文件描述符, 失败时返回-1
```

网络编程中和接受连接请求的套接字创建过程可整理如下：

1. 第一步：调用 `socket` 函数创建套接字。
2. 第二步：调用 `bind` 函数分配IP地址和端口号。
3. 第三步：调用 `listen` 函数转换为可接受请求状态。
4. 第四步：调用 `accept` 函数受理套接字请求。

### 1.1.2 编写 `Hello World` 套接字程序

#### 服务端：

服务器端（server）是能够受理连接请求的程序。下面构建服务端以验证之前提到的函数调用过程，该服务器端收到连接请求后向请求者返回 `Hello World!` 答复。除各种函数的调用顺序外，我们还未涉及任何实际编程。因此，阅读代码时请重点关注套接字相关的函数调用过程，不必理解全过程。

服务器端代码请参见：[hello\\_server.c](#)

#### 客户端：

客户端程序只有调用 `socket` 函数创建套接字 和 调用 `connect` 函数向服务端发送连接请求 这两个步骤，下面给出客户端，需要查看以下两方面的内容：

1. 调用 `socket` 函数 和 `connect` 函数
2. 与服务端共同运行以收发字符串数据

客户端代码请参见：[hello\\_client.c](#)

#### 编译：

分别对客户端和服务端程序进行编译：

```
1 | gcc hello_server.c -o hserver
2 | gcc hello_client.c -o hclient
```

运行：

```
1 | ./hserver 9190
2 | ./hclient 127.0.0.1 9190
```

运行的时候，首先再 9190 端口启动服务，然后 heserver 就会一直等待客户端进行响应，当客户端监听位于本地的 IP 为 127.0.0.1 的地址的9190端口时，客户端就会收到服务端的回应，输出 Hello World!

## 1.2 基于 Linux 的文件操作

讨论套接字的过程中突然谈及文件也许有些奇怪。但是对于 Linux 而言，socket 操作与文件操作没有区别，因而有必要详细了解文件。在 Linux 世界里，socket 也被认为是文件的一种，因此在网络数据传输过程中自然可以使用 I/O 的相关函数。Windows 与 Linux 不同，是要区分 socket 和文件的。因此在 Windows 中需要调用特殊的的数据传输相关函数。

### 1.2.1 底层访问和文件描述符

分配给标准输入输出及标准错误的文件描述符。

文件描述符	对象
0	标准输入：Standard Input
1	标准输出：Standard Output
2	标准错误：Standard Error

文件和套接字一般经过创建过程才会被分配文件描述符。

文件描述符也被称为「文件句柄」，但是「句柄」主要是 Windows 中的术语。因此，在本书中如果设计 Windows 平台将使用「句柄」，如果是 Linux 将使用「描述符」。

### 1.2.2 打开文件：

```
1 | #include <sys/types.h>
2 | #include <sys/stat.h>
3 | #include <fcntl.h>
4 | int open(const char *path, int flag);
5 |
6 | /*
7 | 成功时返回文件描述符，失败时返回-1
8 | path : 文件名的字符串地址
9 | flag : 文件打开模式信息
  | */
```

文件打开模式如下表：

打开模式	含义
O_CREAT	必要时创建文件
O_TRUNC	删除全部现有数据
O_APPEND	维持现有数据，保存到其后面
O_RDONLY	只读打开
O_WRONLY	只写打开
O_RDWR	读写打开

### 1.2.3 关闭文件：

```

1 #include <unistd.h>
2 int close(int fd);
3 /*
4 成功时返回 0 , 失败时返回 -1
5 fd : 需要关闭的文件或套接字的文件描述符
6 */

```

若调用此函数同时传递文件描述符参数，则关闭（终止）响应文件。另外需要注意的是，此函数不仅可以关闭文件，还可以关闭套接字。再次证明了「Linux 操作系统不区分文件与套接字」的特点。

### 1.2.4 将数据写入文件：

```

1 #include <unistd.h>
2 ssize_t write(int fd, const void *buf, size_t nbytes);
3 /*
4 成功时返回写入的字节数 , 失败时返回 -1
5 fd : 显示数据传输对象的文件描述符
6 buf : 保存要传输数据的缓冲值地址
7 nbytes : 要传输数据的字节数
8 */

```

在此函数的定义中，size\_t 是通过 typedef 声明的 unsigned int 类型。对 ssize\_t 来说，ssize\_t 前面多加的 s 代表 signed，即 ssize\_t 是通过 typedef 声明的 signed int 类型。

创建新文件并保存数据：

代码见：[low\\_open.c](#)

编译运行：

```

1 gcc low_open.c -o lopen
2 ./lopen

```

然后会生成一个 `data.txt` 的文件，里面有 `Let's go!`

## 1.2.5 读取文件中的数据:

与之前的 `write()` 函数相对应，`read()` 用来输入（接收）数据。

```
1 #include <unistd.h>
2 ssize_t read(int fd, void *buf, size_t nbytes);
3 /*
4 成功时返回接收的字节数（但遇到文件结尾则返回 0），失败时返回 -1
5 fd : 显示数据接收对象的文件描述符
6 buf : 要保存接收的数据的缓冲地址值。
7 nbytes : 要接收数据的最大字节数
8 */
```

下面示例通过 `read()` 函数读取 `data.txt` 中保存的数据。

代码见：[low\\_read.c](#)

编译运行：

```
1 gcc low_read.c -o lread
2 ./lread
```

在上一步的 `data.txt` 文件与没有删的情况下，会输出：

```
1 file descriptor: 3
2 file data: Let's go!
```

关于文件描述符的 I/O 操作到此结束，要明白，这些内容同样适合于套接字。

## 1.2.6 文件描述符与套接字

下面将同时创建文件和套接字，并用整数型态比较返回的文件描述符的值。

代码见：[fd\\_seril.c](#)

编译运行：

```
1 gcc fd_seril.c -o fds
2 ./fds
```

输出结果：

```
1 file descriptor 1: 3
2 file descriptor 2: 15
3 file descriptor 3: 16
```

## 1.3 基于 Windows 平台的实现

暂略

## 1.4 基于 Windows 的套接字相关函数及示例

暂略

## 1.5 习题

!以下部分的答案，仅代表我个人观点，可能不是正确答案

1. 套接字在网络编程中的作用是什么？为何称它为套接字？

答：操作系统会提供「套接字」（socket）的部件，套接字是网络数据传输用的软件设备。因此，「网络编程」也叫「套接字编程」。「套接字」就是用来连接网络的工具。

2. 在服务器端创建套接字以后，会依次调用 `listen` 函数和 `accept` 函数。请比较二者作用。

答：调用 `listen` 函数将套接字转换成可受连接状态（监听），调用 `accept` 函数受理连接请求。如果在没有连接请求的情况下调用该函数，则不会返回，直到有连接请求为止。

3. Linux 中，对套接字数据进行 I/O 时可以直接使用文件 I/O 相关函数；而在 Windows 中则不可以。原因为何？

答：暂略。

4. 创建套接字后一般会给他分配地址，为什么？为了完成地址分配需要调用哪个函数？

答：套接字被创建之后，只有为其分配了IP地址和端口号后，客户端才能够通过IP地址及端口号与服务器端建立连接，需要调用 `bind` 函数来完成地址分配。

5. Linux 中的文件描述符与 Windows 的句柄实际上非常类似。请以套接字为对象说明它们的含义。

答：暂略。

6. 底层 I/O 函数与 ANSI 标准定义的文件 I/O 函数有何区别？

答：文件 I/O 又称为低级磁盘 I/O，遵循 POSIX 相关标准。任何兼容 POSIX 标准的操作系统上都支持文件 I/O。标准 I/O 被称为高级磁盘 I/O，遵循 ANSI C 相关标准。只要开发环境中有关于标准 I/O 库，标准 I/O 就可以使用。（Linux 中使用的是 GLIBC，它是标准 C 库的超集。不仅包含 ANSI C 中定义的函数，还包括 POSIX 标准中定义的函数。因此，Linux 下既可以使用标准 I/O，也可以使用文件 I/O）。

7. 参考本书给出的示例 `low_open.c` 和 `low_read.c`，分别利用底层文件 I/O 和 ANSI 标准 I/O 编写文件复制程序。可任意指定复制程序的使用方法。

答：暂略。

## 第 2 章 套接字类型与协议设置

本章代码，在[TCP-IP-NetworkNote](#)中可以找到，直接点连接可能进不去。

本章仅需了解创建套接字时调用的 `socket` 函数。

### 2.1 套接字协议及数据传输特性

#### 2.1.1 创建套接字

```

1 #include <sys/socket.h>
2 int socket(int domain, int type, int protocol);
3 /*
4 成功时返回文件描述符，失败时返回-1
5 domain: 套接字中使用的协议族 (Protocol Family)
6 type: 套接字数据传输的类型信息
7 protocol: 计算机间通信中使用的协议信息
8 */

```

## 2.1.2 协议族 (Protocol Family)

通过 `socket` 函数的第一个参数传递套接字中使用的协议分类信息。此协议分类信息称为协议族，可分成如下几类：

头文件 `sys/socket.h` 中声明的协议族

名称	协议族
PF_INET	IPV4 互联网协议族
PF_INET6	IPV6 互联网协议族
PF_LOCAL	本地通信 Unix 协议族
PF_PACKET	底层套接字的协议族
PF_IPX	IPX Novel 协议族

本书着重讲 `PF_INET` 对应的 `IPV4` 互联网协议族。其他协议并不常用，或并未普及。另外，套接字中采用的最终的协议信息是通过 `socket` 函数的第三个参数传递的。在指定的协议族范围内通过第一个参数决定第三个参数。

## 2.1.3 套接字类型 (Type)

套接字类型指的是套接字的数据传输方式，是通过 `socket` 函数的第二个参数进行传递，只有这样才能决定创建的套接字的数据传输方式。已经通过第一个参数传递了协议族信息，为什么还要决定数据传输方式？问题就在于，决定了协议族并不能同时决定数据传输方式。换言之，`socket` 函数的第一个参数 `PF_INET` 协议族中也存在多种数据传输方式。

### 2.1.4 套接字类型1：面向连接的套接字 (SOCK\_STREAM)

如果 `socket` 函数的第二个参数传递 `SOCK_STREAM`，将创建面向连接的套接字。

传输方式特征整理如下：

- 传输过程中数据不会消失
- 按序传输数据
- 传输的数据不存在数据边界 (Boundary)

这种情形适用于之前说过的 `write` 和 `read` 函数

传输数据的计算机通过调用3次 `write` 函数传递了 100 字节的数据，但是接受数据的计算机仅仅通过调用 1 次 `read` 函数调用就接受了全部 100 个字节。

收发数据的套接字内部有缓冲（buffer），简言之就是字节数组。只要不超过数组容量，那么数据填满缓冲后过1次read函数的调用就可以读取全部，也有可能调用多次来完成读取。

#### 套接字缓冲已满是否意味着数据丢失？

答：缓冲并不总是满的。如果读取速度比数据传入过来的速度慢，则缓冲可能被填满，但是这时也不会丢失数据，因为传输套接字此时会停止数据传输，所以面向连接的套接字不会发生数据丢失。

套接字联机必须一一对应。面向连接的套接字可总结为：

可靠地、按序传递的、基于字节的面向连接的数据传输方式的套接字。

### 2.1.5 面向消息的套接字（SOCK\_DGRAM）

如果socket函数的第二个参数传递SOCK\_DGRAM，则将创建面向消息的套接字。面向消息的套接字可以比喻成高速移动的摩托车队。特点如下：

- 强调快速传输而非传输有序
- 传输的数据可能丢失也可能损毁
- 传输的数据有边界
- 限制每次传输数据的大小

面向消息的套接字比面向连接的套接字更具哟传输速度，但可能丢失。特点可总结为：

不可靠的、不按序传递的、以数据的高速传输为目的套接字。

### 2.1.6 协议的最终选择

socket函数的第三个参数决定最终采用的协议。前面已经通过前两个参数传递了协议族信息和套接字数据传输方式，这些信息还不够吗？为什么要传输第三个参数呢？

可以应对同一协议族中存在的多个数据传输方式相同的协议，所以数据传输方式相同，但是协议不同，需要用第三个参数指定具体的协议信息。

本书用的是Ipv4的协议族，和面向连接的数据传输，满足这两个条件的协议只有 IPPROTO\_TCP，因此可以如下调用socket函数创建套接字，这种套接字称为TCP套接字。

```
1 int tcp_socket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

SOCK\_DGRAM指的是面向消息的数据传输方式，满足上述条件的协议只有 IPPROTO\_UDP。这种套接字称为UDP套接字：

```
1 int udp_socket = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

### 2.1.7 面向连接的套接字：TCP套接字示例

需要对第一章的代码做出修改，修改好的代码如下：

- [tcp\\_client.c](#)
- [tcp\\_server.c](#)

编译：

```
1 | gcc tcp_client.c -o hclient  
2 | gcc tcp_server.c -o hserver
```

运行：

```
1 | ./hserver 9190  
2 | ./hclient 127.0.0.1 9190
```

结果：

```
1 | Message from server : Hello World!  
2 | Function read call count: 13
```

从运行结果可以看出服务端发送了13字节的数据，客户端调用13次read函数进行读取。

## 2.2 Windows 平台下的实现及验证

暂略

## 2.3 习题

1. 什么是协议？在收发数据中定义协议有何意义？

答：协议是对话中使用的通信规则，简言之，协议就是为了完成数据交换而定好的约定。在收发数据中定义协议，能够让计算机之间进行正确无误的对话，以此来交换数据。

2. 面向连接的套接字 TCP 套接字传输特性有 3 点，请分别说明。

答：①传输过程中数据不会消失②按序传输数据③传输的数据不存在数据边界（Boundary）

3. 下面那些是面向消息的套接字的特性？

- 传输数据可能丢失
- 没有数据边界（Boundary）
- 以快速传递为目标
- 不限制每次传输数据大小
- 与面向连接的套接字不同，不存在连接概念

4. 下列数据适合用哪类套接字进行传输？

- 演唱会现场直播的多媒体数据（UDP）
- 某人压缩过的文本文件（TCP）
- 网上银行用户与银行之间的数据传递（TCP）

5. 何种类型的套接字不存在数据边界？这类套接字接收数据时应该注意什么？

答：TCP 不存在数据边界。在接收数据时，需要保证在接收套接字的缓冲区填充满之时就从buffer里读取数据。也就是，在接收套接字内部，写入buffer的速度要小于读出buffer的速度。

## 第 3 章 地址族与数据序列

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

把套接字比喻成电话，那么目前只安装了电话机，本章讲解给电话机分配号码的方法，即给套接字分配 IP 地址和端口号。

## 3.1 分配给套接字的 IP 地址与端口号

IP 是 Internet Protocol（网络协议）的简写，是为手法网络数据而分配给计算机的值。端口号并非赋予计算机的值，而是为了区分程序中创建的套接字而分配给套接字的端口号。

### 3.1.1 网络地址（Internet Address）

为使计算机连接到网络并收发数据，必须为其分配 IP 地址。IP 地址分为两类。

- IPv4 (Internet Protocol version 4) 4 字节地址族
- IPv6 (Internet Protocol version 6) 6 字节地址族

两者之间的主要差别是 IP 地址所用的字节数，目前通用的是 IPv4，IPv6 的普及还需要时间。

IPv4 标准的 4 字节 IP 地址分为网络地址和主机（指计算机）地址，且分为 A、B、C、D、E 等类型。



图3-1 IPv4地址族

数据传输过程：

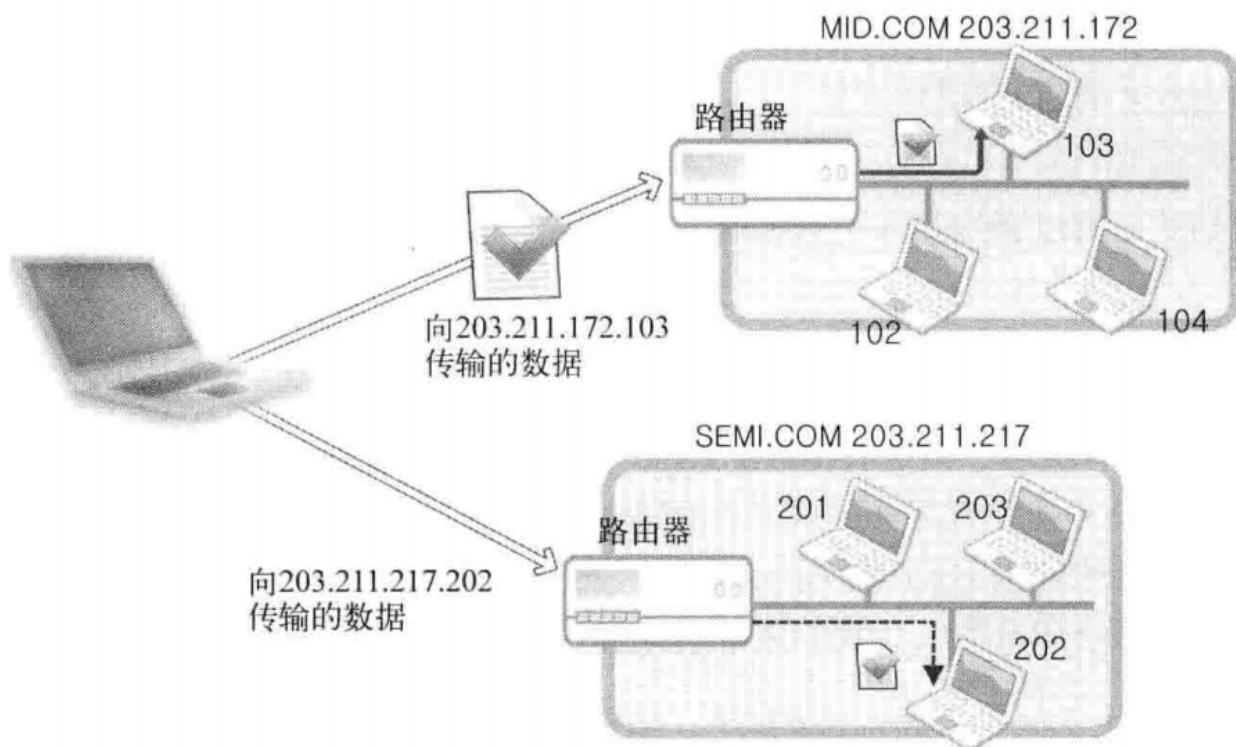


图3-2 基于IP地址的数据传输过程

某主机向 203.211.172.103 和 203.211.217.202 传递数据，其中 203.211.172 和 203.211.217 为该网络的网络地址，所以「向相应网络传输数据」实际上是向构成网络的路由器或者交换机传输数据，然后又路由器或者交换机根据数据中的主机地址向目标主机传递数据。

### 3.1.2 网络地址分类与主机地址边界

只需通过IP地址的第一个字节即可判断网络地址占用的总字节数，因为我们根据IP地址的边界区分网络地址，如下所示：

- A类地址的首字节范围为：0~127
- B类地址的首字节范围为：128~191
- C类地址的首字节范围为：192~223

还有如下这种表示方式：

- A类地址的首位以0开始
- B类地址的前2位以10开始
- C类地址的前3位以110开始

因此套接字手法数据时，数据传到网络后即可轻松找到主机。

### 3.1.3 用于区分套接字的端口号

IP地址用于区分计算机，只要有IP地址就能向目标主机传输数据，但是只有这些还不够，我们需要把信息传输给具体的应用程序。

所以计算机一般有 NIC（网络接口卡）数据传输设备。通过 NIC 接受的数据内有端口号，操作系统参考端口号把信息传给相应的应用程序。

端口号由 16 位构成，可分配的端口号范围是 0~65535。但是 0~1023 是知名端口，一般分配给特定的应用程序，所以应当分配给此范围之外的值。

虽然端口号不能重复，但是 TCP 套接字和 UDP 套接字不会共用端口号，所以允许重复。如果某 TCP 套接字使用了 9190 端口号，其他 TCP 套接字就无法使用该端口号，但是 UDP 套接字可以使用。

总之，数据传输目标地址同时包含IP地址和端口号，只有这样，数据才会被传输到最终的目的应用程序。

## 3.2 地址信息的表示

应用程序中使用的IP地址和端口号以结构体的形式给出了定义。本节围绕结构体讨论目标地址的表示方法。

### 3.2.1 表示 IPV4 地址的结构体

结构体的定义如下

```
1 struct sockaddr_in
2 {
3     sa_family_t sin_family; //地址族 (Address Family)
4     uint16_t sin_port;      //16 位 TCP/UDP 端口号
5     struct in_addr sin_addr; //32位 IP 地址
6     char sin_zero[8];       //不使用
7 }
```

该结构体中提到的另一个结构体 in\_addr 定义如下，它用来存放 32 位IP地址

```
1 struct in_addr
2 {
3     in_addr_t s_addr; //32位IPV4地址
4 }
```

关于以上两个结构体的一些数据类型。

数据类型名称	数据类型说明	声明的头文件
int 8_t	signed 8-bit int	sys/types.h
uint8_t	unsigned 8-bit int (unsigned char)	sys/types.h
int16_t	signed 16-bit int	sys/types.h
uint16_t	unsigned 16-bit int (unsigned short)	sys/types.h
int32_t	signed 32-bit int	sys/types.h
uint32_t	unsigned 32-bit int (unsigned long)	sys/types.h
sa_family_t	地址族 (address family)	sys/socket.h
socklen_t	长度 (length of struct)	sys/socket.h
in_addr_t	IP地址, 声明为 uint_32_t	netinet/in.h
in_port_t	端口号, 声明为 uint_16_t	netinet/in.h

为什么要额外定义这些数据类型呢? 这是考虑扩展性的结果

### 3.2.2 结构体 `sockaddr_in` 的成员分析

- 成员 `sin_family`

每种协议适用的地址族不同, 比如, IPV4 使用 4 字节的地址族, IPV6 使用 16 字节的地址族。

地址族

地址族 (Address Family)	含义
AF_INET	IPV4用的地址族
AF_INET6	IPV6用的地址族
AF_LOCAL	本地通信中采用的 Unix 协议的地址族

AF\_LOACL 只是为了说明具有多种地址族而添加的。

- 成员 `sin_port`

该成员保存 16 位端口号, 重点在于, 它以网络字节序保存。

- 成员 `sin_addr`

该成员保存 32 位IP地址信息, 且也以网络字节序保存

- 成员 `sin_zero`

无特殊含义。只是为结构体 `sockaddr_in` 结构体变量地址值将以如下方式传递给 `bind` 函数。

在之前的代码中

```
1 if (bind(serv_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == -1)
2     error_handling("bind() error");
```

此处 bind 第二个参数期望得到的是 sockaddr 结构体变量的地址值，包括地址族、端口号、IP 地址等。

```
1 struct sockaddr
2 {
3     sa_family_t sin_family; //地址族
4     char sa_data[14];       //地址信息
5 }
```

此结构体 sa\_data 保存的地址信息中需要包含 IP 地址和端口号，剩余部分应该填充 0，但是这样对于包含地址的信息非常麻烦，所以出现了 sockaddr\_in 结构体，然后强制转换成 sockaddr 类型，则生成符合 bind 条件的参数。

### 3.3 网络字节序与地址变换

不同的 CPU 中，4 字节整数值 1 在内存空间保存方式是不同的。

有些 CPU 这样保存：

```
1 00000000 00000000 00000000 00000001
```

有些 CPU 这样保存：

```
1 00000001 00000000 00000000 00000000
```

两种一种是顺序保存，一种是倒序保存。

#### 3.3.1 字节序 (Order) 与网络字节序

CPU 保存数据的方式有两种，这意味着 CPU 解析数据的方式也有 2 种：

- 大端序 (Big Endian)：高位字节存放到低位地址
- 小端序 (Little Endian)：高位字节存放到高位地址

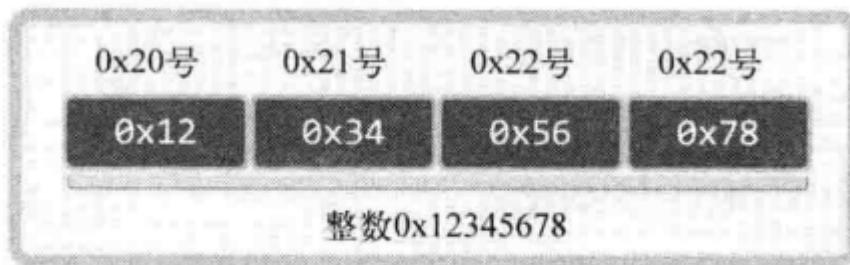


图3-4 大端序字节表示

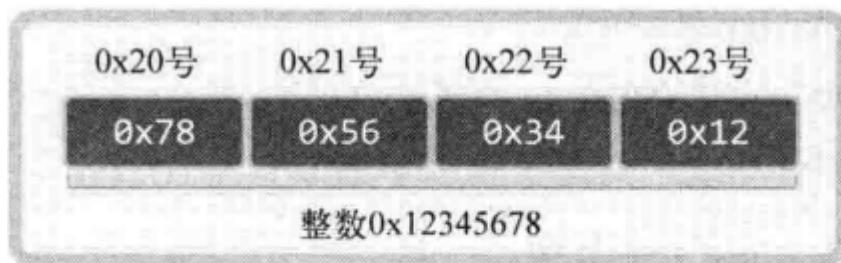


图3-5 小端序字节表示

两台字节序不同的计算机在数据传递的过程中可能出现的问题：

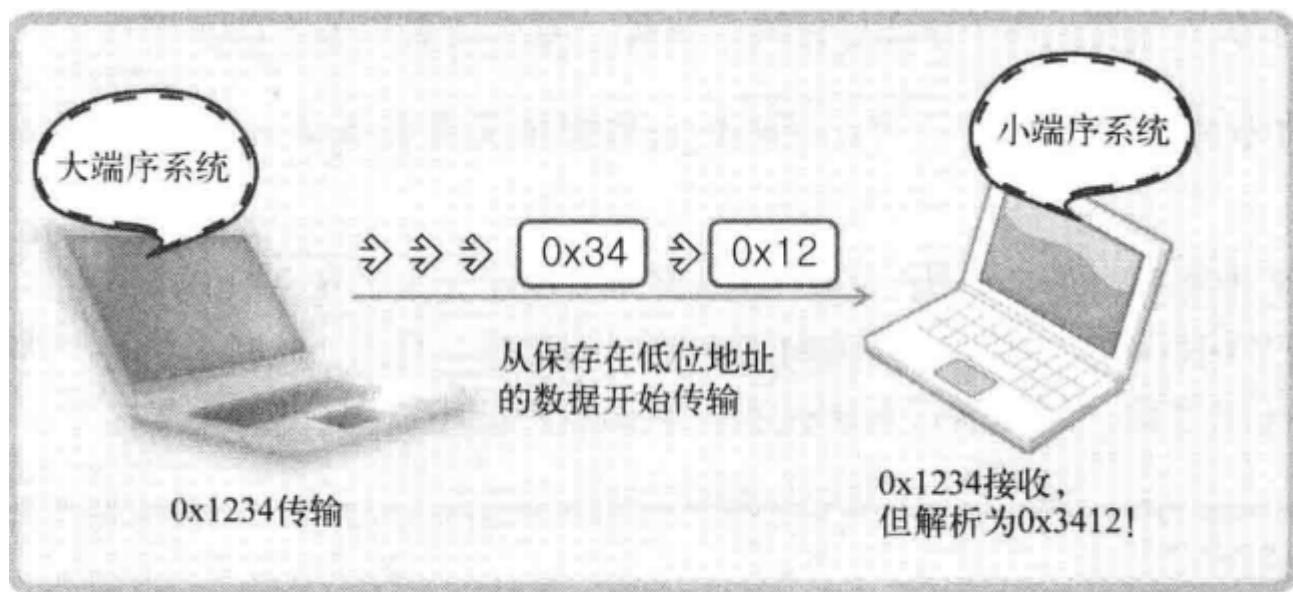


图3-6 字节序问题

因为这种原因，所以在通过网络传输数据时必须约定统一的方式，这种约定被称为网络字节序，非常简单，统一为大端序。即，先把数据数组转化成大端序格式再进行网络传输。

### 3.3.2 字节序转换

帮助转换字节序的函数：

```

1 unsigned short htons(unsigned short);
2 unsigned short ntohs(unsigned short);
3 unsigned long htonl(unsigned long);
4 unsigned long ntohl(unsigned long);

```

通过函数名称掌握其功能，只需要了解：

- htons 的 h 代表主机 (host) 字节序。
- htons 的 n 代表网络 (network) 字节序。
- s 代表 short
- l 代表 long

下面的代码是示例，说明以上函数调用过程：

#### [endian\\_conv.c](#)

```
1 #include <stdio.h>
2 #include <arpa/inet.h>
3 int main(int argc, char *argv[])
4 {
5     unsigned short host_port = 0x1234;
6     unsigned short net_port;
7     unsigned long host_addr = 0x12345678;
8     unsigned long net_addr;
9
10    net_port = htons(host_port); //转换为网络字节序
11    net_addr = htonl(host_addr);
12
13    printf("Host ordered port: %#x \n", host_port);
14    printf("Network ordered port: %#x \n", net_port);
15    printf("Host ordered address: %#lx \n", host_addr);
16    printf("Network ordered address: %#lx \n", net_addr);
17
18    return 0;
19 }
```

编译运行：

```
1 gcc endian_conv.c -o conv
2 ./conv
```

结果：

```
1 Host ordered port: 0x1234
2 Network ordered port: 0x3412
3 Host ordered address: 0x12345678
4 Network ordered address: 0x78563412
```

这是在小端 CPU 的运行结果。大部分人会得到相同的结果，因为 Intel 和 AMD 的 CPU 都是小端序为标准。

## 3.4 网络地址的初始化与分配

### 3.4.1 将字符串信息转换为网络字节序的整数型

sockaddr\_in 中需要的是 32 位整数型，但是我们只熟悉点分十进制表示法，那么改如何把类似于 201.211.214.36 转换为 4 字节的整数类型数据呢？幸运的是，有一个函数可以帮助我们完成它。

```
1 #include <arpa/inet.h>
2 in_addr_t inet_addr(const char *string);
```

具体示例：

### [inet\\_addr.c](#)

```
1 #include <stdio.h>
2 #include <arpa/inet.h>
3 int main(int argc, char *argv[])
4 {
5     char *addr1 = "1.2.3.4";
6     char *addr2 = "1.2.3.256";
7
8     unsigned long conv_addr = inet_addr(addr1);
9     if (conv_addr == INADDR_NONE)
10         printf("Error occurred! \n");
11     else
12         printf("Network ordered integer addr: %#lx \n", conv_addr);
13
14     conv_addr = inet_addr(addr2);
15     if (conv_addr == INADDR_NONE)
16         printf("Error occurred! \n");
17     else
18         printf("Network ordered integer addr: %#lx \n", conv_addr);
19     return 0;
20 }
```

编译运行：

```
1 gcc inet_addr.c -o addr
2 ./addr
```

输出：

```
1 Network ordered integer addr: 0x4030201
2 Error occurred!
```

1 个字节能表示的最大整数是 255，所以代码中 addr2 是错误的 IP 地址。从运行结果看，inet\_addr 不仅可以转换地址，还可以检测有效性。

inet\_aton 函数与 inet\_addr 函数在功能上完全相同，也是将字符串形式的 IP 地址转换成整数型的 IP 地址。只不过该函数用了 in\_addr 结构体，且使用频率更高。

```
1 #include <arpa/inet.h>
2 int inet_aton(const char *string, struct in_addr *addr);
3 /*
4 成功时返回 1 , 失败时返回 0
5 string: 含有需要转换的IP地址信息的字符串地址值
6 addr: 将保存转换结果的 in_addr 结构体变量的地址值
7 */
```

函数调用示例：

### [inet\\_aton.c](#)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <arpa/inet.h>
4 void error_handling(char *message);
5
6 int main(int argc, char *argv[])
7 {
8     char *addr = "127.232.124.79";
9     struct sockaddr_in addr_inet;
10
11     if (!inet_aton(addr, &addr_inet.sin_addr))
12         error_handling("Conversion error");
13     else
14         printf("Network ordered integer addr: %#x \n", addr_inet.sin_addr.s_addr);
15     return 0;
16 }
17
18 void error_handling(char *message)
19 {
20     fputs(message, stderr);
21     fputc('\n', stderr);
22     exit(1);
23 }
```

编译运行：

```
1 gcc inet_aton.c -o aton
2 ./aton
```

运行结果：

```
1 Network ordered integer addr: 0x4f7ce87f
```

可以看出，已经成功的把转换后的地址放进了 addr\_inet.sin\_addr.s\_addr 中。

还有一个函数，与 inet\_aton() 正好相反，它可以把网络字节序整数型IP地址转换成我们熟悉的字符串形式，函数原型如下：

```
1 #include <arpa/inet.h>
2 char *inet_ntoa(struct in_addr adr);
```

该函数将通过参数传入的整数型IP地址转换为字符串格式并返回。但要小心，返回值为 `char` 指针，返回字符串地址意味着字符串已经保存在内存空间，但是该函数未向程序员要求分配内存，而是再内部申请了内存保存了字符串。也就是说调用了该函数候要立即把信息复制到其他内存空间。因此，若再次调用 `inet_ntoa` 函数，则有可能覆盖之前保存的字符串信息。总之，再次调用 `inet_ntoa` 函数前返回的字符串地址是有效的。若需要长期保存，则应该将字符串复制到其他内存空间。

示例：

### [inet\\_ntoa.c](#)

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <arpa/inet.h>
4
5 int main(int argc, char *argv[])
6 {
7     struct sockaddr_in addr1, addr2;
8     char *str_ptr;
9     char str_arr[20];
10
11     addr1.sin_addr.s_addr = htonl(0x1020304);
12     addr2.sin_addr.s_addr = htonl(0x1010101);
13     //把addr1中的结构体信息转换为字符串的IP地址形式
14     str_ptr = inet_ntoa(addr1.sin_addr);
15     strcpy(str_arr, str_ptr);
16     printf("Dotted-Decimal notation1: %s \n", str_ptr);
17
18     inet_ntoa(addr2.sin_addr);
19     printf("Dotted-Decimal notation2: %s \n", str_ptr);
20     printf("Dotted-Decimal notation3: %s \n", str_arr);
21     return 0;
22 }
```

编译运行：

```
1 gcc inet_ntoa.c -o ntoa
2 ./ntoa
```

输出：

```
1 Dotted-Decimal notation1: 1.2.3.4
2 Dotted-Decimal notation2: 1.1.1.1
3 Dotted-Decimal notation3: 1.2.3.4
```

## 3.4.2 网络地址初始化

结合前面的内容，介绍套接字创建过程中，常见的网络信息初始化方法：

```
1 struct sockaddr_in addr;
2 char *serv_ip = "211.217.168.13";           //声明IP地址族
3 char *serv_port = "9190";                   //声明端口号字符串
4 memset(&addr, 0, sizeof(addr));           //结构体变量 addr 的所有成员初始化为0
5 addr.sin_family = AF_INET;                 //制定地址族
6 addr.sin_addr.s_addr = inet_addr(serv_ip); //基于字符串的IP地址初始化
7 addr.sin_port = htons(atoi(serv_port));    //基于字符串的IP地址端口号初始化
```

## 3.5 基于 Windows 的实现

略

## 3.6 习题

答案仅代表本人个人观点，不一定正确

### 1. IP地址族 IPV4 与 IPV6 有什么区别？在何种背景下诞生了 IPV6？

答：主要差别是IP地址所用的字节数，目前通用的是IPV4，目前IPV4的资源已耗尽，所以诞生了IPV6，它具有更大的地址空间。

### 2. 通过 IPV4 网络 ID、主机 ID 及路由器的关系说明公司局域网的计算机传输数据的过程

答：网络ID是为了区分网络而设置的一部分IP地址，假设向 `www.baidu.com` 公司传输数据，该公司内部构建了局域网。因为首先要向 `baidu.com` 传输数据，也就是说并非一开始就浏览所有四字节IP地址，首先找到网络地址，进而由 `baidu.com` （构成网络的路由器）接收到数据后，传输到主机地址。比如向 203.211.712.103 传输数据，那就先找到 203.211.172 然后由这个网络的网关找主机号为 172 的机器传输数据。

### 3. 套接字地址分为IP地址和端口号，为什么需要IP地址和端口号？或者说，通过IP地址可以区分哪些对象？通过端口号可以区分哪些对象？

答：有了IP地址和端口号，才能把数据准确的传送到某个应用程序中。通过IP地址可以区分具体的主机，通过端口号可以区分主机上的应用程序。

### 4. 请说明IP地址的分类方法，并据此说出下面这些IP的分类。

- 214.121.212.102 (C类)
- 120.101.122.89 (A类)
- 129.78.102.211 (B类)

分类方法：A类地址的首字节范围为：0~127、B类地址的首字节范围为：128~191、C类地址的首字节范围为：192~223

### 5. 计算机通过路由器和交换机连接到互联网，请说出路由器和交换机的作用。

答：路由器和交换机完成外网和本网主机之间的数据交换。

### 6. 什么是知名端口？其范围是多少？知名端口中具有代表性的 HTTP 和 FTP 的端口号各是多少？

答：知名端口是要把该端口分配给特定的应用程序，范围是 0~1023，HTTP 的端口号是 80，FTP 的端口号是 20 和 21

### 7. 向套接字分配地址的 bind 函数原型如下：

```
1 | int bind(int sockfd, struct sockaddr *myaddr, socklen_t addrlen);
```

而调用时则用：

```
1 | bind(serv_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr))
```

此处 **serv\_addr** 为 **sockaddr\_in** 结构体变量。与函数原型不同，传入的是 **sockaddr\_in** 结构体变量，请说明原因。

答：因为对于详细的地址信息使用 **sockaddr** 类型传递特别麻烦，进而有了 **sockaddr\_in** 类型，其中基本与前面的类型保持一致，还有 **sa\_data[4]** 来保存地址信息，剩余全部填 0，所以强制转换后，不影响程序运行。

8. 请解释大端序，小端序、网络字节序，并说明为何需要网络字节序。

答：CPU 向内存保存数据有两种方式，大端序是高位字节存放低位地址，小端序是高位字节存放高位地址，网络字节序是为了方便传输的信息统一性，统一成了大端序。

9. 大端序计算机希望把 4 字节整数型 12 传递到小端序计算机。请说出数据传输过程中发生的字节序变换过程。

答：0x12->0x21

10. 怎样表示回送地址？其含义是什么？如果向会送地址处传输数据将会发生什么情况？

答：127.0.0.1 表示回送地址，指的是计算机自身的IP地址，无论什么程序，一旦使用回送地址发送数据，协议软件立即返回，不进行任何网络传输。

## 第 4 章 基于 TCP 的服务端/客户端（1）

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

### 4.1 理解 TCP 和 UDP

根据数据传输方式的不同，基于网络协议的套接字一般分为 TCP 套接字和 UDP 套接字。因为 TCP 套接字是面向连接的，因此又被称为基于流（stream）的套接字。

TCP 是 Transmission Control Protocol（传输控制协议）的简写，意为「对数据传输过程的控制」。因此，学习控制方法及范围有助于正确理解 TCP 套接字。

#### 4.1.1 TCP/IP 协议栈

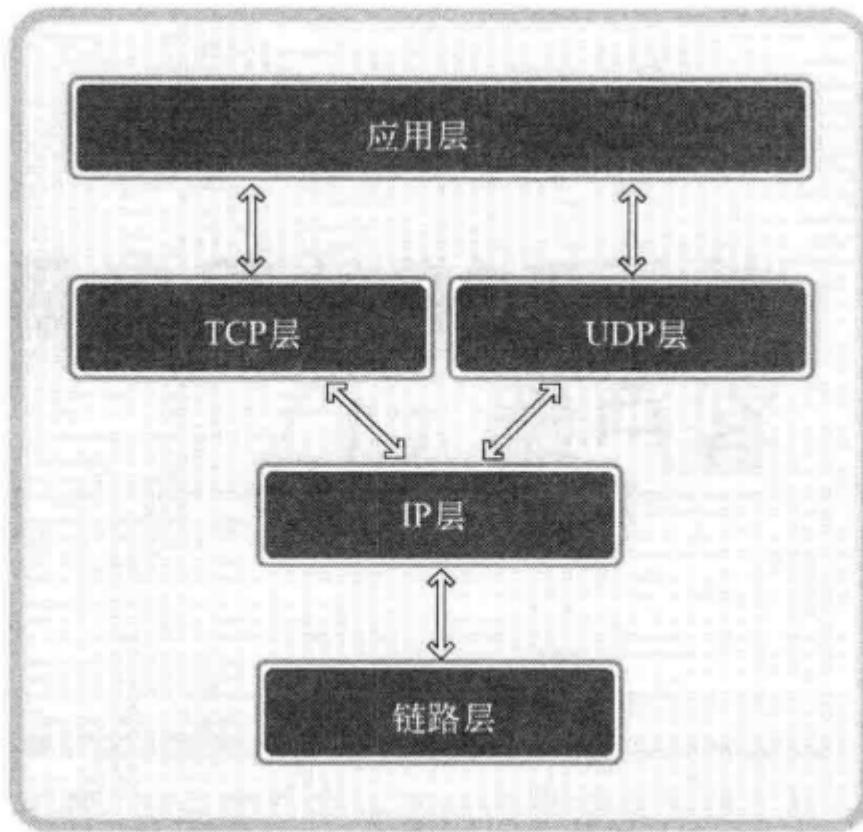


图4-1 TCP/IP协议栈

TCP/IP 协议栈共分为 4 层，可以理解为数据收发分成了 4 个层次化过程，通过层次化的方式来解决问题

### 4.1.2 链路层

链路层是物理链接领域标准化的结果，也是最基本的领域，专门定义 LAN、WAN、MAN 等网络标准。若两台主机通过网络进行数据交换，则需要物理连接，链路层就负责这些标准。

### 4.1.3 IP 层

转备好物理连接后就要传输数据。为了在复杂网络中传输数据，首先要考虑路径的选择。向目标传输数据需要经过哪条路径？解决此问题的就是IP层，该层使用的协议就是IP。

IP 是面向消息的、不可靠的协议。每次传输数据时会帮我们选择路径，但并不一致。如果传输过程中发生错误，则选择其他路径，但是如果发生数据丢失或错误，则无法解决。换言之，IP协议无法应对数据错误。

### 4.1.4 TCP/UDP 层

IP 层解决数据传输中的路径选择问题，秩序照此路径传输数据即可。TCP 和 UDP 层以 IP 层提供的路径信息为基础完成实际的数据传输，故该层又称为传输层。UDP 比 TCP 简单，现在我们只解释 TCP。TCP 可以保证数据的可靠传输，但是它发送数据时以 IP 层为基础（这也是协议栈层次化的原因）

IP 层只关注一个数据包（数据传输基本单位）的传输过程。因此，即使传输多个数据包，每个数据包也是由 IP 层实际传输的，也就是说传输顺序及传输本身是不可靠的。若只利用 IP 层传输数据，则可能导致后传输的数据包 B 比先传输的数据包 A 提早到达。另外，传输的数据包 A、B、C 中可能只收到 A 和 C，甚至收到的 C 可能已经损毁。反之，若添加 TCP 协议则按照如下对话方式进行数据交换。

主机A：正确接受第二个数据包

主机B：恩，知道了

主机A：正确收到第三个数据包

主机B：可我已经发送第四个数据包了啊！哦，您没收到吧，我给你重新发。

这就是 TCP 的作用。如果交换数据的过程中可以确认对方已经收到数据，并重传丢失的数据，那么即便IP层不保证数据传输，这类通信也是可靠的。

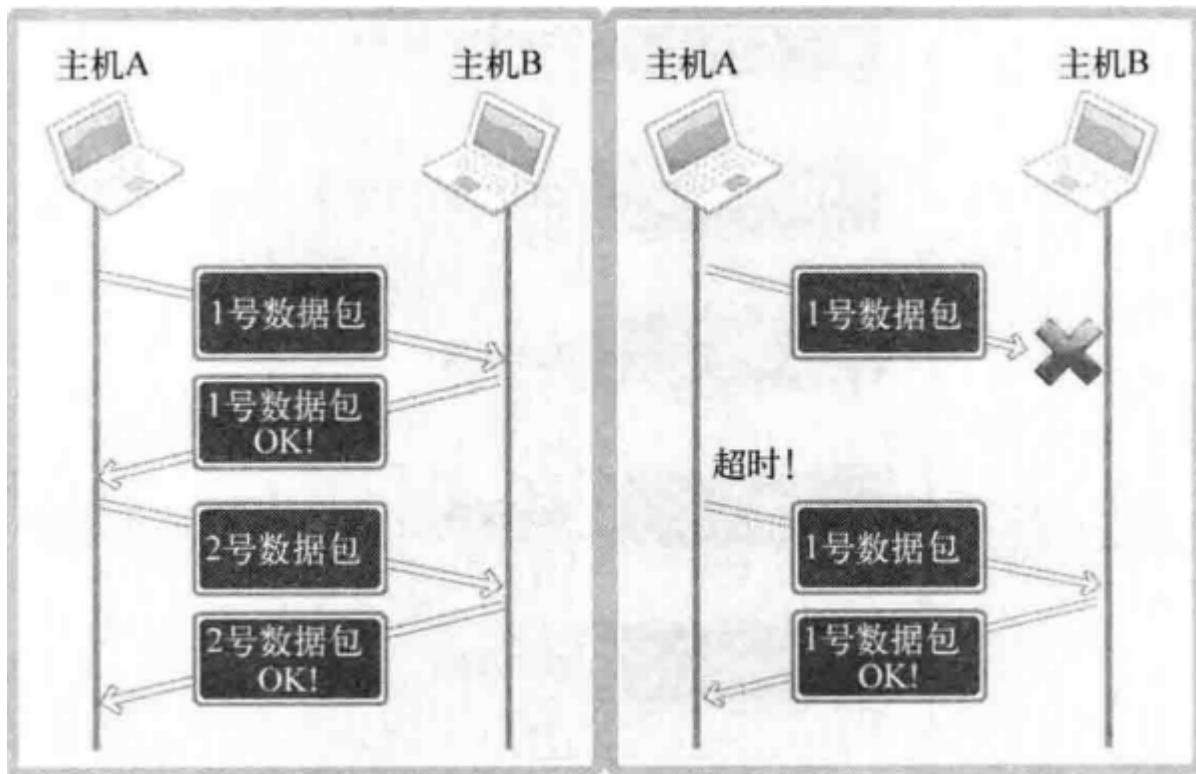


图4-5 传输控制协议

#### 4.1.5 应用层

上述内容是套接字通信过程中自动处理的。选择数据传输路径、数据确认过程都被隐藏到套接字内部。向程序员提供的工具就是套接字，只需要利用套接字编出程序即可。编写软件的过程中，需要根据程序的特点来决定服务器和客户端之间的数据传输规则，这便是应用层协议。

### 4.2 实现基于 TCP 的服务器/客户端

#### 4.2.1 TCP 服务端的默认函数的调用程序

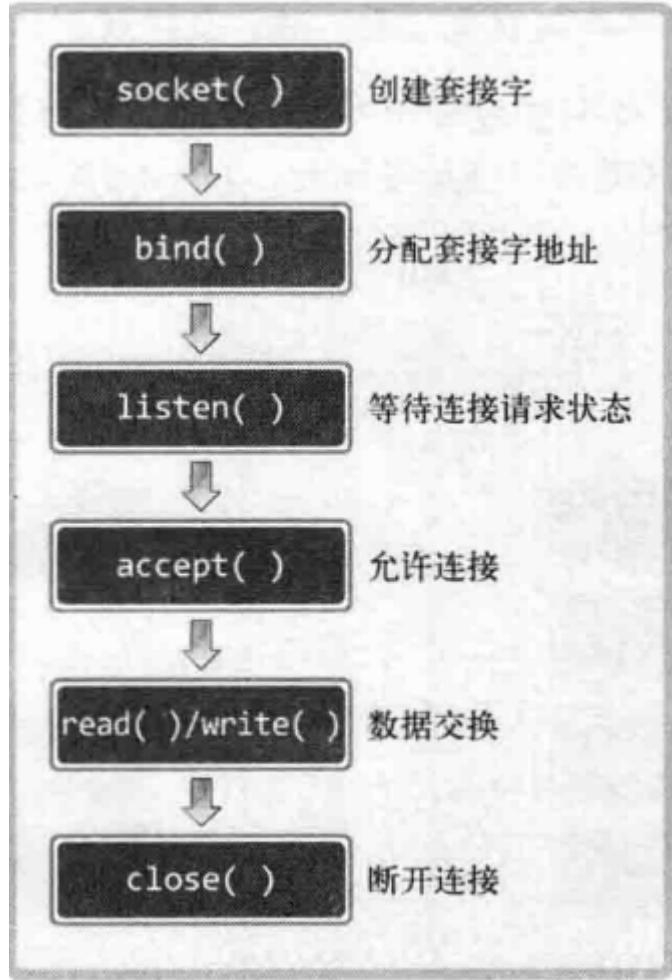


图4-6 TCP服务器端函数调用顺序

调用 `socket` 函数创建套接字，声明并初始化地址信息的结构体变量，调用 `bind` 函数向套接字分配地址。

#### 4.2.2 进入等待连接请求状态

已经调用了 `bind` 函数给套接字分配地址，接下来就是要通过调用 `listen` 函数进入等待连接请求状态。只有调用了 `listen` 函数，客户端才能进入可发出连接请求的状态。换言之，这时客户端才能调用 `connect` 函数。

```

1 #include <sys/socket.h>
2 int listen(int sockfd, int backlog);
3 //成功时返回0, 失败时返回-1
4 //sock: 希望进入等待连接请求状态的套接字文件描述符, 传递的描述符套接字参数称为服务端套接字
5 //backlog: 连接请求等待队列的长度, 若为5, 则队列长度为5, 表示最多使5个连接请求进入队列

```

#### 4.2.3 受理客户端连接请求

调用 `listen` 函数后，则应该按序受理。受理请求意味着可接受数据的状态。进入这种状态所需的部件是套接字，但是此时使用的不是服务端套接字，此时需要另一个套接字，但是没必要亲自创建，下面的函数将自动创建套接字。

```

1 #include <sys/socket.h>
2 int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
3 /*
4 成功时返回文件描述符，失败时返回-1
5 sock: 服务端套接字的文件描述符
6 addr: 保存发起连接请求的客户端地址信息的变量地址值
7 addrlen: 的第二个参数addr结构体的长度，但是存放有长度的变量地址。
8 */

```

accept 函数受理连接请求队列中待处理的客户端连接请求。函数调用成功后，accept 内部将产生用于数据 I/O 的套接字，并返回其文件描述符。需要强调的是套接字是自动创建的，并自动与发起连接请求的客户端建立连接。

#### 4.2.4 回顾 Hello World 服务端

- 代码：[hello\\_server.c](#)

重新整理一下代码的思路

- 服务端实现过程中首先要创建套接字，此时的套接字并非是真正的服务端套接字
- 为了完成套接字地址的分配，初始化结构体变量并调用 bind 函数。
- 调用 listen 函数进入等待连接请求状态。连接请求状态队列的长度设置为5.此时的套接字才是服务端套接字。
- 调用 accept 函数从队头取 1 个连接请求与客户端建立连接，并返回创建的套接字文件描述符。另外，调用 accept 函数时若等待队列为空，则 accept 函数不会返回，直到队列中出现新的客户端连接。
- 调用 write 函数向客户端传送数据，调用 close 关闭连接

#### 4.2.5 TCP 客户端的默认函数调用顺序

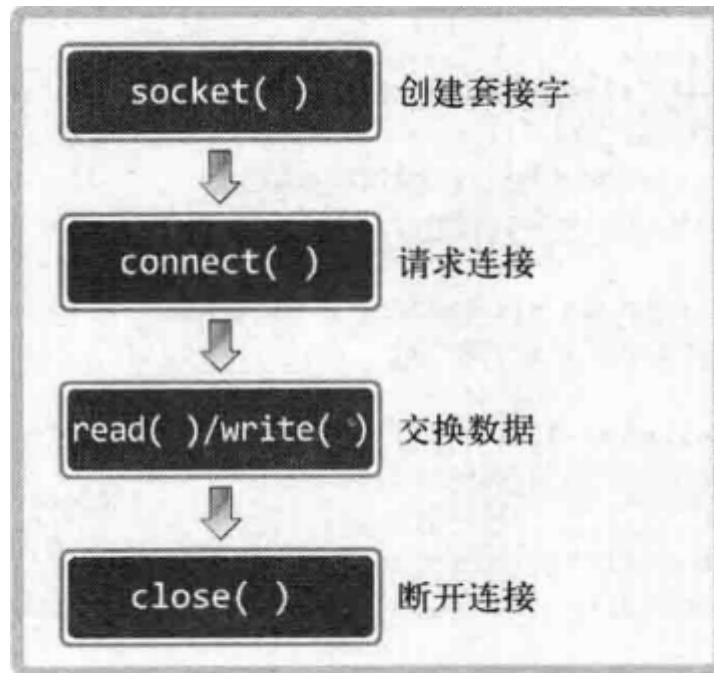


图4-9 TCP客户端函数调用顺序

与服务端相比，区别就在于「请求连接」，他是创建客户端套接字后向服务端发起的连接请求。服务端调用 listen 函数后创建连接请求等待队列，之后客户端即可请求连接。

```
1 #include <sys/socket.h>
2 int connect(int sock, struct sockaddr *servaddr, socklen_t addrlen);
3 /*
4 成功时返回0, 失败返回-1
5 sock:客户端套接字文件描述符
6 servaddr: 保存目标服务器端地址信息的变量地址值
7 addrlen: 以字节为单位传递给第二个结构体参数 servaddr 的变量地址长度
8 */
```

客户端调用 `connect` 函数后，发生以下函数之一才会返回（完成函数调用）：

- 服务端接受连接请求
- 发生断网等一场状况而中断连接请求

注意：接受连接不代表服务端调用 `accept` 函数，其实只是服务器端把连接请求信息记录到等待队列。因此 `connect` 函数返回后并不应该立即进行数据交换。

#### 4.2.6 回顾 Hello World 客户端

- 代码：[hello\\_client.c](#)

重新理解这个程序：

1. 创建准备连接服务器的套接字，此时创建的是 TCP 套接字
2. 结构体变量 `serv_addr` 中初始化IP和端口信息。初始化值为目标服务器端套接字的IP和端口信息。
3. 调用 `connect` 函数向服务端发起连接请求
4. 完成连接后，接收服务端传输的数据
5. 接收数据后调用 `close` 函数关闭套接字，结束与服务器端的连接。

#### 4.2.7 基于 TCP 的服务端/客户端函数调用关系

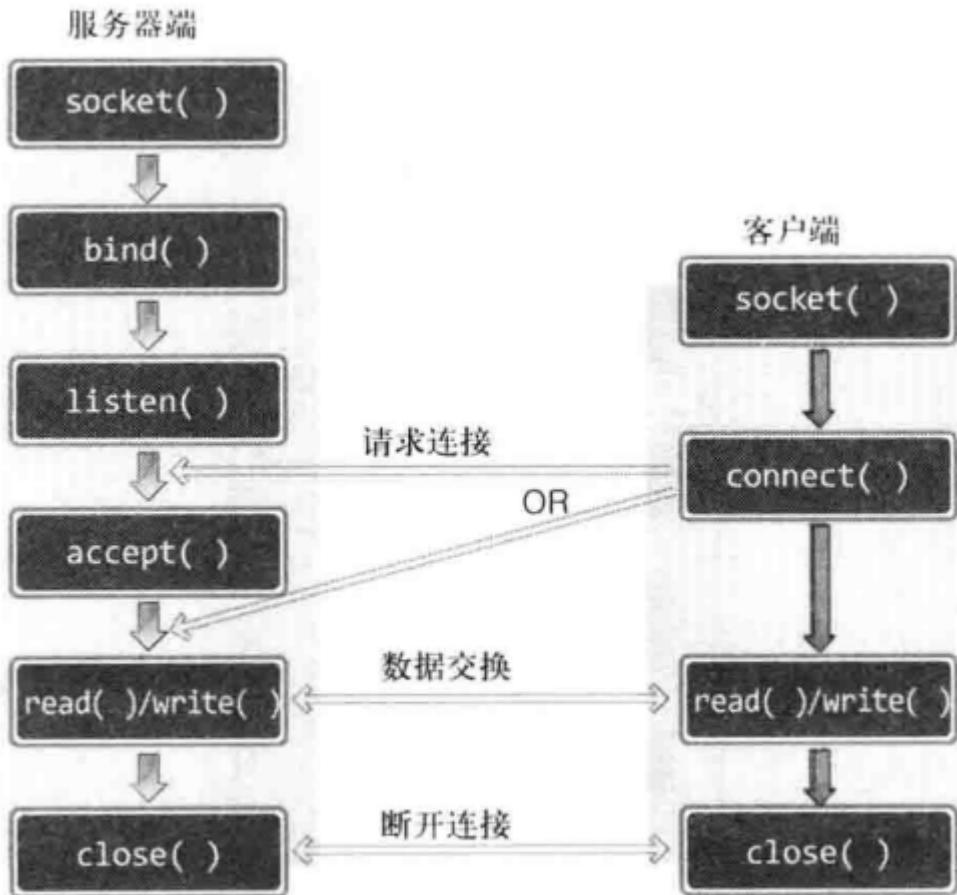


图4-10 函数调用关系

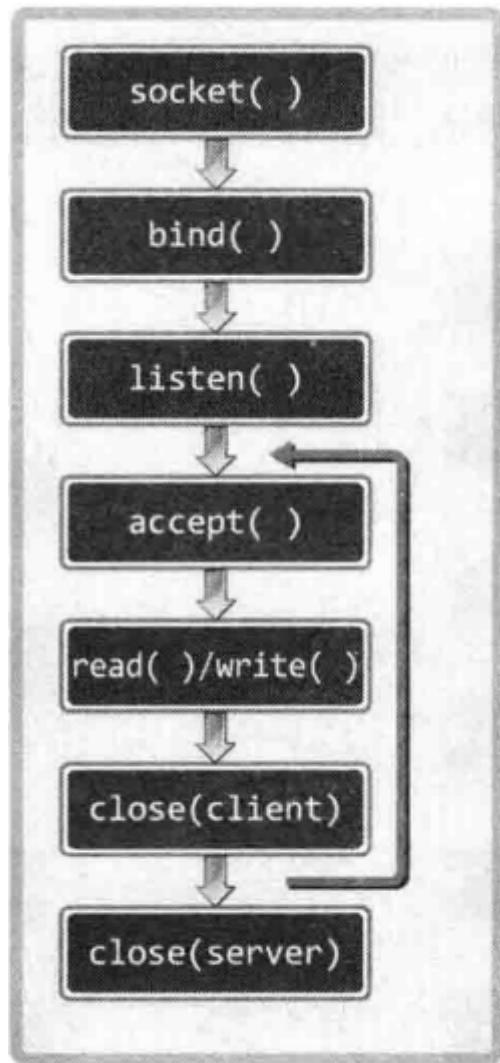
关系如上图所示。

## 4.3 实现迭代服务端/客户端

编写一个回声（echo）服务器/客户端。顾名思义，服务端将客户端传输的字符串数据原封不动的传回客户端，就像回声一样。在此之前，需要解释一下迭代服务器端。

### 4.3.1 实现迭代服务器端

在 Hello World 的例子中，等待队列的作用没有太大意义。如果想继续处理好后面的客户端请求应该怎样扩展代码？最简单的方式就是插入循环反复调用 accept 函数，如图：



## 11 迭代服务器端的函数调用顺序

可以看出，调用 accept 函数后，紧接着调用 I/O 相关的 read write 函数，然后调用 close 函数。这并非针对服务器套接字，而是针对 accept 函数调用时创建的套接字。

### 4.3.2 迭代回声服务器端/客户端

程序运行的基本方式：

- 服务器端在同一时刻只与一个客户端相连，并提供回声服务。
- 服务器端依次向 5 个客户端提供服务并退出。
- 客户端接受用户输入的字符串并发送到服务器端。
- 服务器端将接受的字符串数据传回客户端，即「回声」
- 服务器端与客户端之间的字符串回声一直执行到客户端输入 Q 为止。

以下是服务端与客户端的代码：

- [echo\\_server.c](#)
- [echo\\_client.c](#)

编译：

```
1 | gcc echo_client.c -o eclient  
2 | gcc echo_server.c -o eserver
```

分别运行：

```
1 | ./eserver 9190  
2 | ./eclient 127.0.0.1 9190
```

过程和结果：

在一个服务端开启后，用另一个终端窗口开启客户端，然后程序会让你输入字符串，然后客户端输入什么字符串，客户端就会返回什么字符串，按 q 退出。这时服务端的运行并没有结束，服务端一共要处理 5 个客户端的连接，所以另外开多个终端窗口同时开启客户端，服务器按照顺序进行处理。

server:

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch04 on git:master x  
11:21:46]  
$ ./eserver 9190  
[]
```

client:

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch04 on git:master x [  
11:22:12]  
$ ./eclient 127.0.0.1 9190  
Connected.....  
Input message(Q to quit): hello  
Message from server: hello  
Input message(Q to quit): world  
Message from server: world  
Input message(Q to quit): 中文测试 !!  
Message from server: 中文测试 !!  
Input message(Q to quit): q
```

### 4.3.3 回声客户端存在的问题

以上代码有一个假设「每次调用 read、write 函数时都会以字符串为单位执行实际 I/O 操作」

但是「第二章」中说过「TCP 不存在数据边界」，上述客户端是基于 TCP 的，因此多次调用 write 函数传递的字符串有可能一次性传递到服务端。此时客户端有可能从服务端收到多个字符串，这不是我们想要的结果。还需要考虑服务器的如下情况：

「字符串太长，需要分 2 个包发送！」

服务端希望通过调用 1 次 write 函数传输数据，但是如果数据太大，操作系统就有可能把数据分成多个数据包发送到客户端。另外，在此过程中，客户端可能在尚未收到全部数据包时就调用 read 函数。

以上的问题都是源自 TCP 的传输特性，解决方法在第 5 章。

## 4.4 基于 Windows 的实现

暂略

## 4.5 习题

答案仅代表本人个人观点，不一定是正确答案。

### 1. 请你说明 TCP/IP 的 4 层协议栈，并说明 TCP 和 UDP 套接字经过的层级结构差异。

答：TCP/IP 的四层协议分为：应用层、TCP/UDP 层、IP 层、链路层。差异是一个经过 TCP 层，一个经过 UDP 层。

### 2. 请说出 TCP/IP 协议栈中链路层和 IP 层的作用，并给出二者关系

答：链路层是物理链接领域标准化的结果，专门定义网络标准。若两台主机通过网络进行数据交换，则首先要做到的就是进行物理链接。IP 层：为了在复杂的网络中传输数据，首先需要考虑路径的选择。关系：链路层负责进行一系列物理连接，而 IP 层负责选择正确可行的物理路径。

### 3. 为何需要把 TCP/IP 协议栈分成 4 层（或 7 层）？开放式回答。

答：ARPANET 的研制经验表明，对于复杂的计算机网络协议，其结构应该是层次式的。分层的好处：①隔层之间是独立的②灵活性好③结构上可以分隔开④易于实现和维护⑤能促进标准化工作。

### 4. 客户端调用 connect 函数向服务器端发送请求。服务器端调用哪个函数后，客户端可以调用 connect 函数？

答：服务端调用 listen 函数后，客户端可以调用 connect 函数。因为，服务端调用 listen 函数后，服务端套接字才有能力接受请求连接的信号。

### 5. 什么时候创建连接请求等待队列？它有何种作用？与 accept 有什么关系？

答：服务端调用 listen 函数后，accept 函数正在处理客户端请求时，更多的客户端发来了请求连接的数据，此时，就需要创建连接请求等待队列。以便于在 accept 函数处理完手头的请求之后，按照正确的顺序处理后面正在排队的其他请求。与 accept 函数的关系：accept 函数受理连接请求等待队列中待处理的客户端连接请求。

### 6. 客户端中为何不需要调用 bind 函数分配地址？如果不调用 bind 函数，那何时、如何向套接字分配 IP 地址和端口号？

答：在调用 connect 函数时分配了地址，客户端 IP 地址和端口在调用 connect 函数时自动分配，无需调用标记的 bind 函数进行分配。

## 第 5 章 基于 TCP 的服务端/客户端（2）

本章代码，在 [TCP-IP-NetworkNote](#) 中可以找到。

上一章仅仅是从编程角度学习实现方法，并未详细讨论 TCP 的工作原理。因此，本章将依次讲解 TCP 中必要的理论知识，还将给出第 4 章客户端问题的解决方案。

### 5.1 回声客户端的完美实现

#### 5.1.1 回声服务器没有问题，只有回声客户端有问题？

问题不在服务器端，而在客户端，只看代码可能不好理解，因为 I/O 中使用了相同的函数。先回顾一下服务器端的 I/O 相关代码：

```
1 while ((str_len = read(clnt_sock, message, BUF_SIZE)) != 0)
2     write(clnt_sock, message, str_len);
```

接着是客户端代码：

```
1 write(sock, message, strlen(message));  
2 str_len = read(sock, message, BUF_SIZE - 1);
```

二者都在村换调用 read 和 write 函数。实际上之前的回声客户端将 100% 接受字节传输的数据，只不过接受数据时的单位有些问题。扩展客户端代码回顾范围，下面是，客户端的代码：

```
1 while (1)  
2 {  
3     fputs("Input message(Q to quit): ", stdout);  
4     fgets(message, BUF_SIZE, stdin);  
5  
6     if (!strcmp(message, "q\n") || !strcmp(message, "Q\n"))  
7         break;  
8  
9     write(sock, message, strlen(message));  
10    str_len = read(sock, message, BUF_SIZE - 1);  
11    message[str_len] = 0;  
12    printf("Message from server: %s", message);  
13 }
```

现在应该理解了问题，回声客户端传输的是字符串，而且是通过调用 write 函数一次性发送的。之后还调用一次 read 函数，期待着接受自己传输的字符串，这就是问题所在。

### 5.1.2 回声客户端问题的解决办法

这个问题其实很容易解决，因为可以提前接受数据的大小。若之前传输了20字节长的字符串，则再接收时循环调用 read 函数读取 20 个字节即可。既然有了解决办法，那么代码如下：

- [echo\\_client2.c](#)

这样修改为了接收所有传输数据而循环调用 read 函数。测试及运行结果可参考第四章。

### 5.1.3 如果问题不在于回声客户端：定义应用层协议

回声客户端可以提前知道接收数据的长度，这在大多数情况下是不可能的。那么此时无法预知接收数据长度时应该如何手法数据？这是需要的是应用层协议的定义。在收发过程中定好规则（协议）以表示数据边界，或者提前告知需要发送的数据的大小。服务端/客户端实现过程中逐步定义的规则集合就是应用层协议。

现在写一个小程序来体验应用层协议的定义过程。要求：

1. 服务器从客户端获得多个数组和运算符信息。
2. 服务器接收到数字候对齐进行加减乘运算，然后把结果传回客户端。

例：

1. 向服务器传递3,5,9的同时请求加法运算，服务器返回3+5+9的结果
2. 请求做乘法运算，客户端会收到 `3*5*9` 的结果
3. 如果向服务器传递4,3,2的同时要求做减法，则返回4-3-2的运算结果。

请自己实现一个程序来实现功能。

我自己的实现：

- [My\\_op\\_server.c](#)
- [My\\_op\\_client.c](#)

编译:

```
1 gcc My_op_client.c -o myclient
2 gcc My_op_server.c -o myserver
```

结果:

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch05 on git:master x [16:13:09]
$ ./myclient 127.0.0.1 9190
连接成功!
请输入你要计算的数字个数:
5
请输入第 1 个数字 :1
请输入第 2 个数字 :2
请输入第 3 个数字 :3
请输入第 4 个数字 :4
请输入第 5 个数字 :5
请输入你要进行的运算符 (+, -, *):
*
运算的结果是 : 120
```

其实主要是对程序的一点点小改动，只需要再客户端固定好发送的格式，服务端按照固定格式解析，然后返回结果即可。

书上的实现:

- [op\\_client.c](#)
- [op\\_server.c](#)

阅读代码要注意一下，`int*`与`char`之间的转换。TCP 中不存在数据边界。

编译:

```
1 gcc op_client.c -o opclient
2 gcc op_server.c -o opserver
```

运行:

```
1 ./opserver 9190
2 ./opclient 127.0.0.1 9190
```

结果:

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch05 on git:master x
11:17:47]
$ ./opclient 127.0.0.1 9190
Connected.....
Operand count: 4
Operand 1: 2
Operand 2: 3
Operand 3: 6
Operand 4: 5
Operator: *
Operation result: 180
```

## 5.2 TCP 原理

### 5.2.1 TCP 套接字中的 I/O 缓冲

TCP 套接字的数据收发无边界。服务器即使调用 1 次 `write` 函数传输 40 字节的数据，客户端也有可能通过 4 次 `read` 函数调用每次读取 10 字节。但此处也有一些一问，服务器一次性传输了 40 字节，而客户端竟然可以缓慢的分批接受。客户端接受 10 字节后，剩下的 30 字节在何处等候呢？

实际上，`write` 函数调用后并非立即传输数据，`read` 函数调用后也并非马上接收数据。如图所示，`write` 函数滴啊用瞬间，数据将移至输出缓冲；`read` 函数调用瞬间，从输入缓冲读取数据。

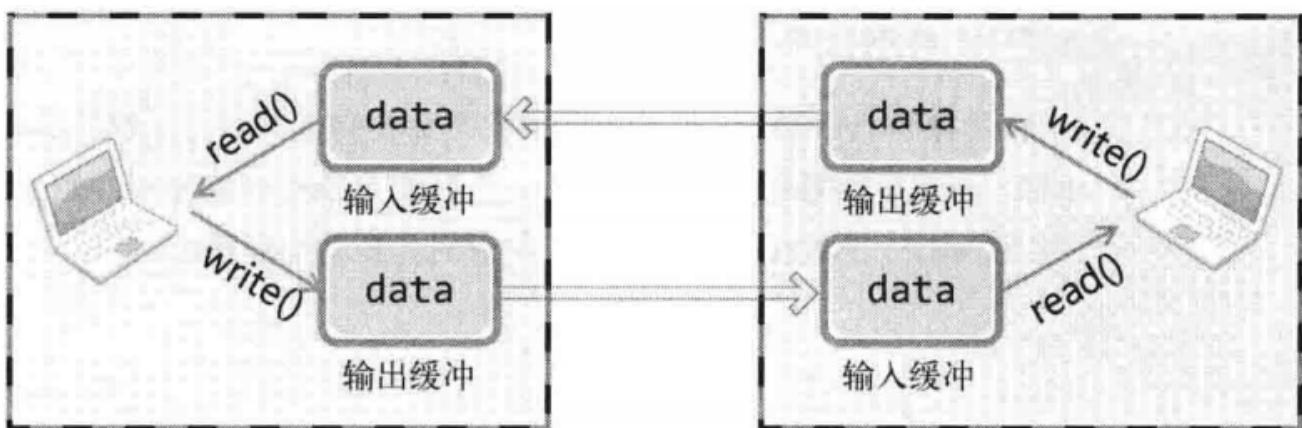


图5-2 TCP套接字的I/O缓冲

I/O 缓冲特性可以整理如下：

- I/O 缓冲在每个 TCP 套接字中单独存在
- I/O 缓冲在创建套接字时自动生成
- 即使关闭套接字也会继续传递输出缓冲中遗留的数据
- 关闭套接字将丢失输入缓冲中的数据

假设发生以下情况，会发生什么事呢？

客户端输入缓冲为 50 字节，而服务器端传输了 100 字节。

因为 TCP 不会发生超过输入缓冲大小的数据传输。也就是说，根本不会发生这类问题，因为 TCP 会控制数据流。TCP 中有滑动窗口（Sliding Window）协议，用对话方式如下：

- A: 你好，最多可以向我传递 50 字节
- B: 好的
- A: 我腾出了 20 字节的空间，最多可以接受 70 字节
- B: 好的

数据收发也是如此，因此 TCP 中不会因为缓冲溢出而丢失数据。

`write` 函数在数据传输完成时返回。

### 5.2.2 TCP 内部工作原理 1：与对方套接字的连接

TCP 套接字从创建到消失所经过的过程分为如下三步：

- 与对方套接字建立连接
- 与对方套接字进行数据交换
- 断开与对方套接字的连接

首先讲解与对方套接字建立连接的过程。连接过程中，套接字的对话如下：

- 套接字A：你好，套接字 B。我这里有数据给你，建立连接吧
- 套接字B：好的，我这边已就绪
- 套接字A：谢谢你受理我的请求

TCP 在实际通信中也会经过三次对话过程，因此，该过程又被称为 **Three-way handshaking**（三次握手）。接下来给出连接过程中实际交换的信息方式：

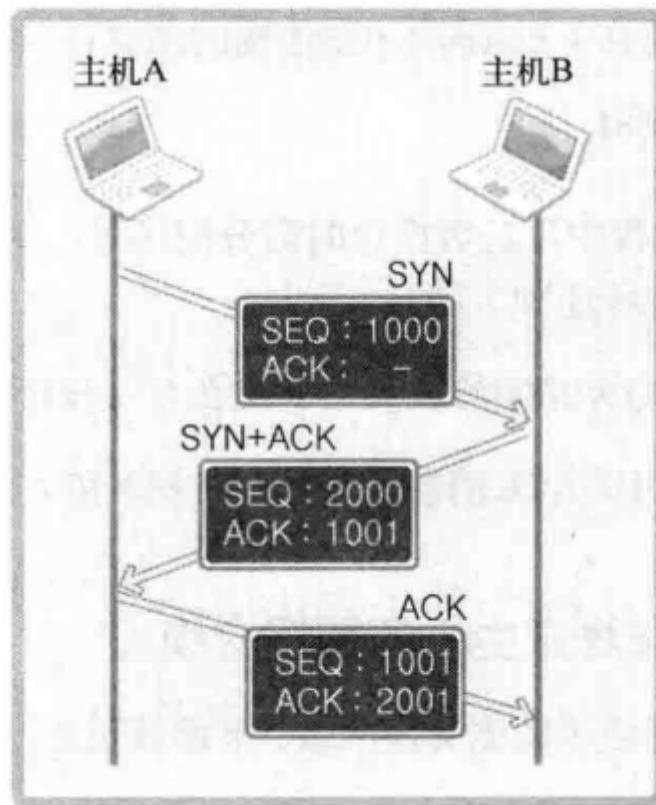


图5-3 TCP套接字的连接设置过程

套接字是全双工方式工作的。也就是说，它可以双向传递数据。因此，收发数据前要做一些准备。首先请求连接的主机 A 要给主机 B 传递以下信息：

[SYN] SEQ : 1000 , ACK:-

该消息中的 SEQ 为 1000 , ACK 为空，而 SEQ 为1000 的含义如下：

现在传递的数据包的序号为 1000，如果接收无误，请通知我向您传递 1001 号数据包。

这是首次请求连接时使用的消息，又称为 SYN。SYN 是 Synchronization 的简写，表示收发数据前传输的同步消息。接下来主机 B 向 A 传递以下信息：

[SYN+ACK] SEQ: 2000, ACK: 1001

此时 SEQ 为 2000，ACK 为 1001，而 SEQ 为 2000 的含义如下：

现传递的数据包号为 2000，如果接受无误，请通知我向您传递 2001 号数据包。

而 ACK 1001 的含义如下：

刚才传输的 SEQ 为 1000 的数据包接受无误，现在请传递 SEQ 为 1001 的数据包。

对于主机 A 首次传输的数据包的确认消息（ACK 1001）和为主机 B 传输数据做准备的同步消息（SEQ 2000）捆绑发送。因此，此种类消息又称为 SYN+ACK。

收发数据前向数据包分配序号，并向对方通报此序号，这都是为了防止数据丢失做的准备。通过项数据包分配序号并确认，可以在数据包丢失时马上查看并重传丢失的数据包。因此 TCP 可以保证可靠的数据传输。

通过这三个过程，这样主机 A 和主机 B 就确认了彼此已经准备就绪。

### 5.2.3 TCP 内部工作原理 2：与对方主机的数据交换

通过第一步三次握手过程完成了数据交换准备，下面就开始正式收发数据，其默认方式如图所示：

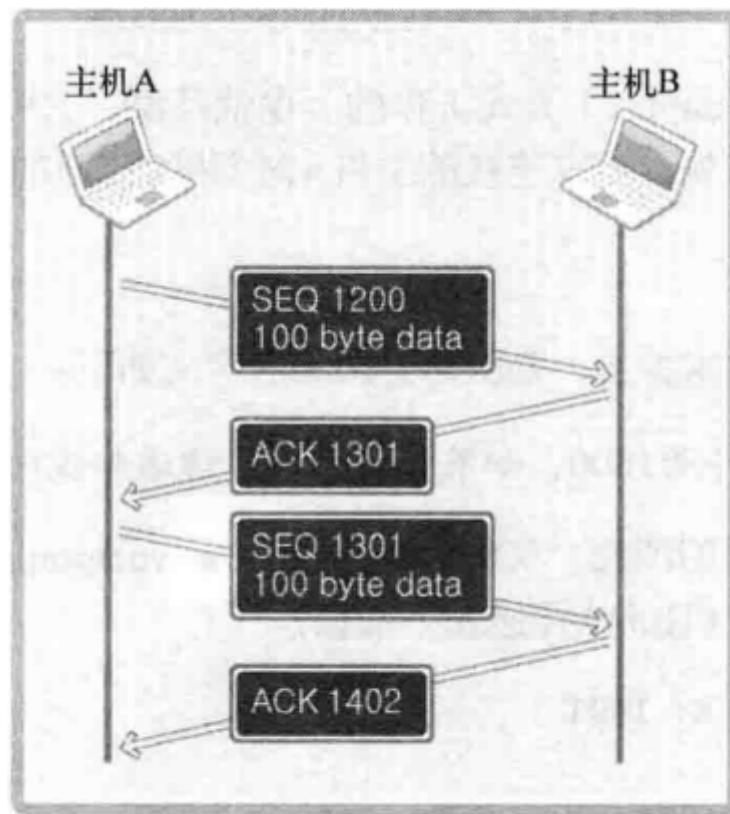


图5-4 TCP套接字的数据交换过程

图上给出了主机 A 分成 2 个数据包向主机 B 传输 200 字节的过程。首先，主机 A 通过 1 个数据包发送 100 个字节的数据，数据包的 SEQ 为 1200。主机 B 为了确认这一点，向主机 A 发送 ACK 1301 消息。

此时的 ACK 号为 1301 而不是 1201，原因在于 ACK 号的增量为传输的数据字节数。假设每次 ACK 号不加传输的字节数，这样虽然可以确认数据包的传输，但无法明确 100 个字节全都正确传递还是丢失了一部分，比如只传递了 80 字节。因此按照如下公式传递 ACK 信息：

$$\text{ACK 号} = \text{SEQ 号} + \text{传递的字节数} + 1$$

与三次握手协议相同，最后 + 1 是为了告知对方下次要传递的 SEQ 号。下面分析传输过程中数据包丢失的情况：

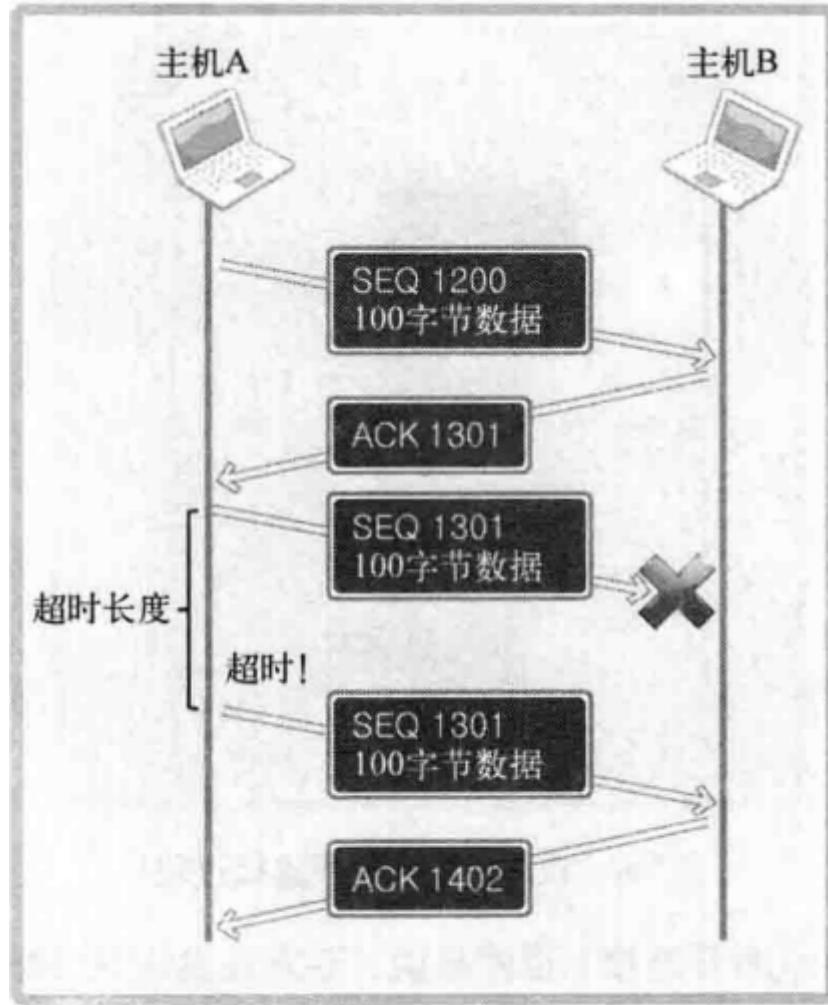


图5-5 TCP套接字数据传输中发生错误

上图表示了通过 SEQ 1301 数据包向主机 B 传递 100 字节数据。但中间发生了错误，主机 B 未收到，经过一段时间后，主机 A 仍然未收到对于 SEQ 1301 的 ACK 的确认，因此试着重传该数据包。为了完成该数据包的重传，TCP 套接字启动计时器以等待 ACK 应答。若相应计时器发生超时（Time-out!）则重传。

#### 5.2.4 TCP 内部工作原理 3：断开套接字的连接

TCP 套接字的结束过程也非常优雅。如果对方还有数据需要传输时直接断掉该连接会出问题，所以断开连接时需要双方协商，断开连接时双方的对话如下：

- 套接字A：我希望断开连接
- 套接字B：哦，是吗？请稍后。
- 套接字A：我也准备就绪，可以断开连接。
- 套接字B：好的，谢谢合作。

先由套接字 A 向套接字 B 传递断开连接的信息，套接字 B 发出确认收到的消息，然后向套接字 A 传递可以断开连接的消息，套接字 A 同样发出确认消息。

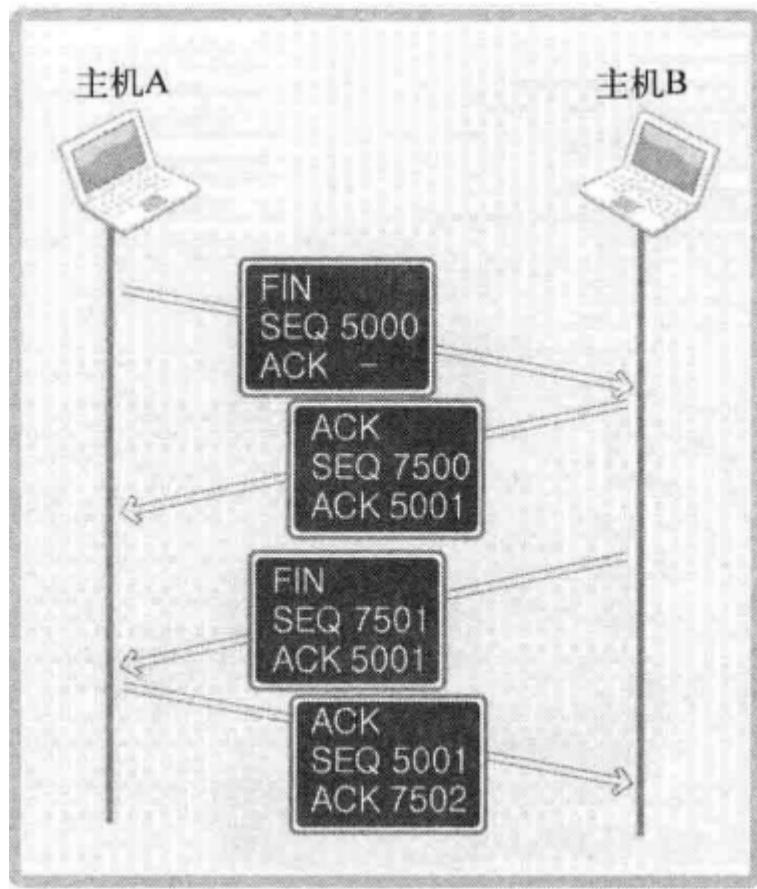


图5-6 TCP套接字断开连接过程

图中数据包内的 FIN 表示断开连接。也就是说，双方各发送 1 次 FIN 消息后断开连接。此过程经历 4 个阶段，因此又称四次握手（Four-way handshaking）。SEQ 和 ACK 的含义与之前讲解的内容一致，省略。图中，主机 A 传递了两次 ACK 5001，也许这里会有困惑。其实，第二次 FIN 数据包中的 ACK 5001 只是因为接收了 ACK 消息后未接收到的数据重传的。

## 5.3 基于 Windows 的实现

暂略

## 5.4 习题

答案仅代表本人个人观点，可能不是正确答案。

1. 请说明 TCP 套接字连接设置的三次握手过程。尤其是 3 次数据交换过程每次收发的数据内容。

答：三次握手主要分为：①与对方套接字建立连接②与对方套接字进行数据交换③断开与对方套接字的连接。每次收发的数据内容主要有：①由主机1给主机2发送初始的SEQ，首次连接请求是关键字是SYN，表示收发数据前同步传输的消息。②主机2收到报文以后，给主机1传递信息，用一个新的SEQ表示自己的序号，然后ACK代表已经接受到主机1的消息，希望接受下一个消息③主机1收到主机2的确认以后，还需要给主机2给出确认，此时再发送一次SEQ和ACK。

2. TCP 是可靠的数据传输协议，但在通过网络通信的过程中可能丢失数据。请通过 ACK 和 SEQ 说明 TCP 通过和何种机制保证丢失数据的可靠传输。

答：通过超时重传机制来保证，如果报文发出去的特定时间内，发送消息的主机没有收到另一个主机的回复，那么就继续发送这条消息，直到收到回复为止。

### 3. TCP 套接字中调用 write 和 read 函数时数据如何移动？结合 I/O 缓冲进行说明。

答：TCP 套接字调用 write 函数时，数据将移至输出缓冲，在适当的时候，传到对方输入缓冲。这时对方将调用 read 函数从输入缓冲中读取数据。

### 4. 对方主机的输入缓冲剩余 50 字节空间时，若本主机通过 write 函数请求传输 70 字节，请问 TCP 如何处理这种情况？

答：TCP 中有滑动窗口控制协议，所以传输的时候会保证传输的字节数小于等于自己能接受的字节数。

## 第 6 章 基于 UDP 的服务端/客户端

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

TCP 是内容较多的一个协议，而本章中的 UDP 内容较少，但是也很重要。

### 6.1 理解 UDP

#### 6.1.1 UDP 套接字的特点

通过寄信来说明 UDP 的工作原理，这是讲解 UDP 时使用的传统示例，它与 UDP 的特点完全相同。寄信前应现在信封上填好寄信人和收信人的地址，之后贴上邮票放进邮筒即可。当然，信件的特点使我们无法确认信件是否被收到。邮寄过程中也可能发生信件丢失的情况。也就是说，信件是一种不可靠的传输方式，UDP 也是一种不可靠的数据传输方式。

因为 UDP 没有 TCP 那么复杂，所以编程难度比较小，性能也比 TCP 高。在更重视性能的情况下可以选择 UDP 的传输方式。

TCP 与 UDP 的区别很大一部分来源于流控制。也就是说 TCP 的生命在于流控制。

#### 6.1.2 UDP 的工作原理

如图所示：

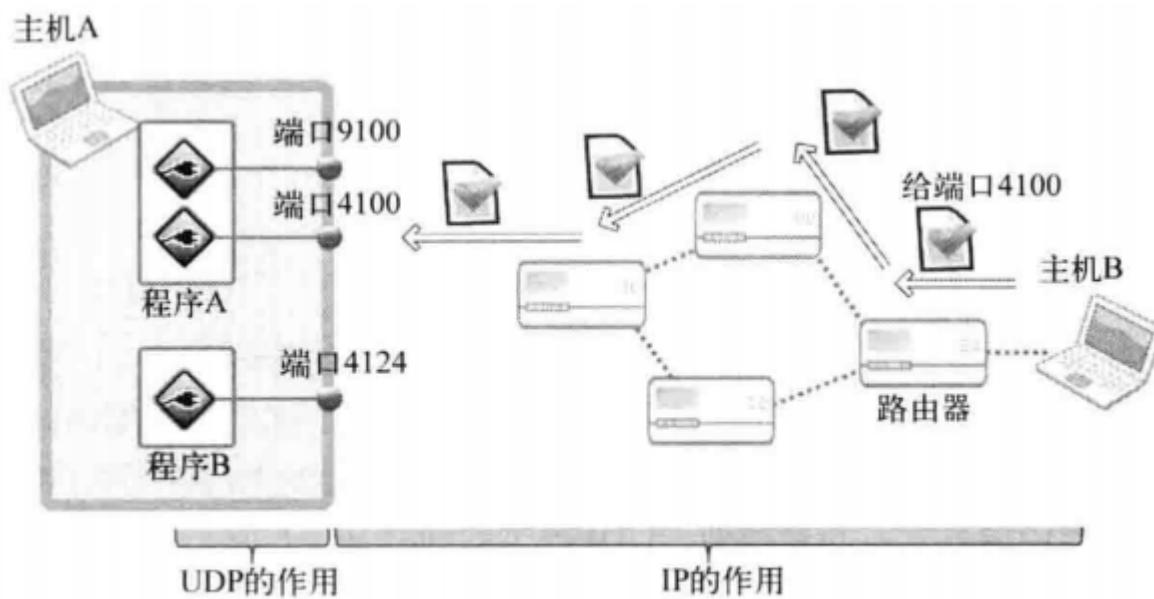


图6-1 数据包传输中UDP和IP的作用

从图中可以看出，IP 的作用就是让离开主机 B 的 UDP 数据包准确传递到主机 A。但是把 UDP 数据包最终交给主机 A 的某一 UDP 套接字的过程是由 UDP 完成的。UDP 的最重要的作用就是根据端口号将传到主机的数据包交付给最终的 UDP 套接字。

### 6.1.3 UDP 的高效使用

UDP 也具有一定的可靠性。对于通过网络实时传递的视频或者音频时情况有所不同。对于多媒体数据而言，丢失一部分数据也没有太大问题，这只是会暂时引起画面抖动，或者出现细微的杂音。但是要提供实时服务，速度就成为了一个很重要的因素。因此流控制就显得有一点多余，这时就要考虑使用 UDP。TCP 比 UDP 慢的原因主要有以下两点：

- 收发数据前后进行的连接设置及清楚过程。
- 收发过程中为保证可靠性而添加的流控制。

如果收发的数据量小但是需要频繁连接时，UDP 比 TCP 更高效。

## 6.2 实现基于 UDP 的服务端/客户端

### 6.2.1 UDP 中的服务端和客户端没有连接

UDP 中的服务端和客户端不像 TCP 那样在连接状态下交换数据，因此与 TCP 不同，无需经过连接过程。也就是说，不必调用 TCP 连接过程中调用的 listen 和 accept 函数。UDP 中只有创建套接字和数据交换的过程。

### 6.2.2 UDP 服务器和客户端均只需一个套接字

TCP 中，套接字之间应该是一对一的关系。若要向 10 个客户端提供服务，除了守门的服务器套接字之外，还需要 10 个服务器套接字。但在 UDP 中，不管事服务器端还是客户端都只需要 1 个套接字。只需要一个 UDP 套接字就可以向任意主机传输数据，如图所示：

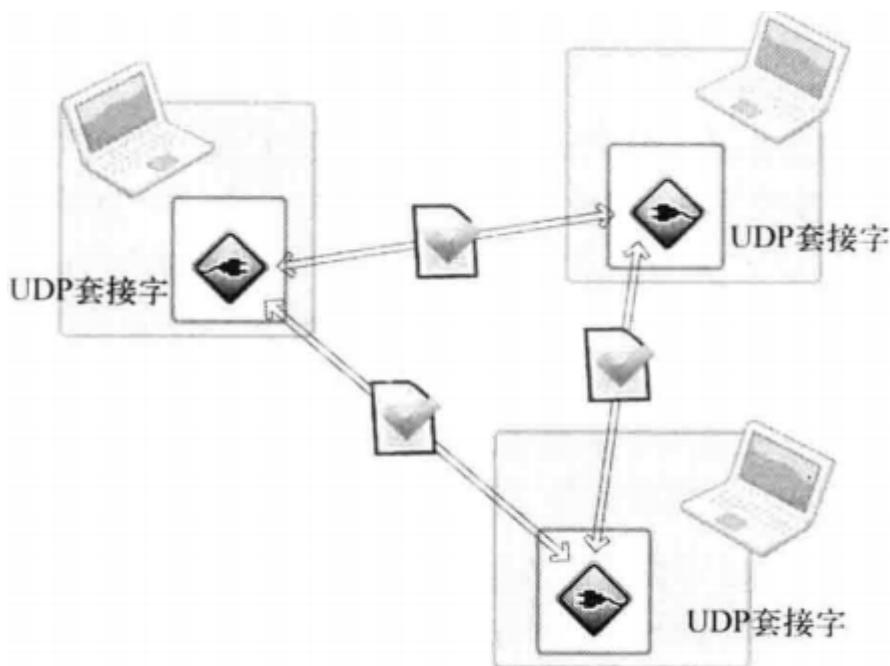


图6-2 UDP套接字通信模型

图中展示了 1 个 UDP 套接字与 2 个不同主机交换数据的过程。也就是说，只需 1 个 UDP 套接字就能和多台主机进行通信。

### 6.2.3 基于 UDP 的数据 I/O 函数

创建好 TCP 套接字以后，传输数据时无需加上地址信息。因为 TCP 套接字将保持与对方套接字的连接。换言之，TCP 套接字知道目标地址信息。但 UDP 套接字不会保持连接状态（UDP 套接字只有简单的邮筒功能），因此每次传输数据时都需要添加目标的地址信息。这相当于寄信前在信件中填写地址。接下来是 UDP 的相关函数：

```
1 #include <sys/socket.h>
2 ssize_t sendto(int sock, void *buff, size_t nbytes, int flags,
3                 struct sockaddr *to, socklen_t addrlen);
4 /*
5  成功时返回传输的字节数，失败是返回 -1
6  sock: 用于传输数据的 UDP 套接字
7  buff: 保存待传输数据的缓冲地址值
8  nbytes: 待传输的数据长度，以字节为单位
9  flags: 可选项参数，若没有则传递 0
10 to: 存有目标地址的 sockaddr 结构体变量的地址值
11 addrlen: 传递给参数 to 的地址值结构体变量长度
12 */
```

上述函数与之前的 TCP 输出函数最大的区别在于，此函数需要向它传递目标地址信息。接下来介绍接收 UDP 数据的函数。UDP 数据的发送并不固定，因此该函数定义为可接受发送端信息的形式，也就是将同时返回 UDP 数据包中的发送端信息。

```
1 #include <sys/socket.h>
2 ssize_t recvfrom(int sock, void *buff, size_t nbytes, int flags,
3                   struct sockaddr *from, socklen_t *addrlen);
4 /*
5  成功时返回传输的字节数，失败是返回 -1
6  sock: 用于传输数据的 UDP 套接字
7  buff: 保存待传输数据的缓冲地址值
8  nbytes: 待传输的数据长度，以字节为单位
9  flags: 可选项参数，若没有则传递 0
10 from: 存有发送端地址信息的 sockaddr 结构体变量的地址值
11 addrlen: 保存参数 from 的结构体变量长度的变量地址值。
12 */
```

编写 UDP 程序的最核心的部分就在于上述两个函数，这也说明二者在 UDP 数据传输中的地位。

#### 6.2.4 基于 UDP 的回声服务器端/客户端

下面是实现的基于 UDP 的回声服务器的服务器端和客户端：

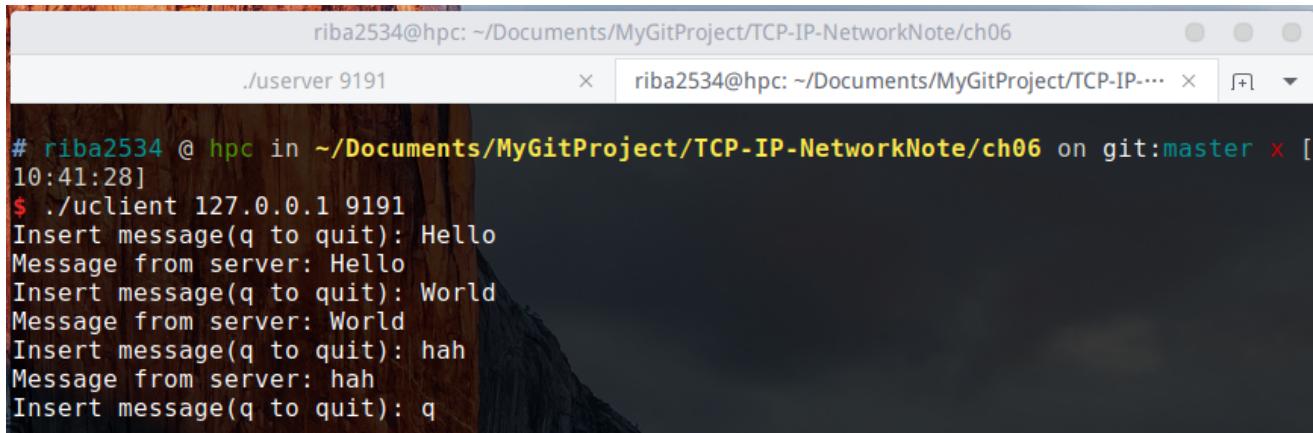
代码：

- [uecho\\_client.c](#)
- [uecho\\_server.c](#)

编译运行：

```
1 gcc uecho_client.c -o uclient
2 gcc uecho_server.c -o userver
3 ./server 9190
4 ./uclient 127.0.0.1 9190
```

结果：



```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch06 on git:master x [10:41:28]
$ ./uclient 127.0.0.1 9191
Insert message(q to quit): Hello
Message from server: Hello
Insert message(q to quit): World
Message from server: World
Insert message(q to quit): hah
Message from server: hah
Insert message(q to quit): q
```

TCP 客户端套接字在调用 `connect` 函数时自动分配IP地址和端口号，既然如此， UDP 客户端何时分配IP地址和端口号？

### 6.2.5 UDP 客户端套接字的地址分配

仔细观察 UDP 客户端可以发现， UDP 客户端缺少了把IP和端口分配给套接字的过程。TCP 客户端调用 `connect` 函数自动完成此过程，而 UDP 中连接能承担相同功能的函数调用语句都没有。究竟在什么时候分配IP和端口号呢？

UDP 程序中，调用 `sendto` 函数传输数据前应该完成对套接字的地址分配工作，因此调用 `bind` 函数。当然， `bind` 函数在 TCP 程序中出现过，但 `bind` 函数不区分 TCP 和 UDP，也就是说，在 UDP 程序中同样可以调用。另外，如果调用 `sendto` 函数尚未分配地址信息，则在首次调用 `sendto` 函数时给相应套接字自动分配 IP 和端口。而且此时分配的地址一直保留到程序结束为止，因此也可以用来和其他 UDP 套接字进行数据交换。当然，IP 用主机IP，端口号用未选用的任意端口号。

综上所述，调用 `sendto` 函数时自动分配IP和端口号，因此， UDP 客户端中通常无需额外的地址分配过程。所以之前的示例中省略了该过程。这也是普遍的实现方式。

## 6.3 UDP 的数据传输特性和调用 `connect` 函数

### 6.3.1 存在数据边界的 UDP 套接字

前面说得 TCP 数据传输中不存在数据边界，这表示「数据传输过程中调用 I/O 函数的次数不具有任何意义」

相反， UDP 是具有数据边界的下一，传输中调用 I/O 函数的次数非常重要。因此，输入函数的调用次数和输出函数的调用次数完全一致，这样才能保证接收全部已经发送的数据。例如，调用 3 次输出函数发送的数据必须通过调用 3 次输入函数才能接收完。通过一个例子来进行验证：

- [bound\\_host1.c](#)
- [bound\\_host2.c](#)

编译运行：

```
1 gcc bound_host1.c -o host1
2 gcc bound_host2.c -o host2
3 ./host1 9190
4 ./host2 127.0.0.1 9190
```

运行结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch06 on git:master x [11:39:03]
$ ./host1 9190
Message 1: Hi!
Message 2: I'm another UDP host!
Message 3: Nice to meet you
```

host1 是服务端， host2 是客户端， host2 一次性把数据发给服务端后，结束程序。但是因为服务端每隔五秒才接收一次，所以服务端每隔五秒接收一次消息。

从运行结果也可以证明 UDP 通信过程中 I/O 的调用次数必须保持一致

### 6.3.2 已连接 (connect) UDP 套接字与未连接 (unconnected) UDP 套接字

TCP 套接字中需注册待传传输数据的目标IP和端口号，而在 UDP 中无需注册。因此通过 sendto 函数传输数据的过程大概可以分为以下 3 个阶段：

- 第 1 阶段：向 UDP 套接字注册目标 IP 和端口号
- 第 2 阶段：传输数据
- 第 3 阶段：删除 UDP 套接字中注册的目标地址信息。

每次调用 sendto 函数时重复上述过程。每次都变更目标地址，因此可以重复利用同一 UDP 套接字向不同目标传递数据。这种未注册目标地址信息的套接字称为未连接套接字，反之，注册了目标地址的套接字称为连接 connected 套接字。显然，UDP 套接字默认属于未连接套接字。当一台主机向另一台主机传输很多信息时，上述的三个阶段中，第一个阶段和第三个阶段占整个通信过程中近三分之一的时间，缩短这部分的时间将会大大提高整体性能。

### 6.3.3 创建已连接 UDP 套接字

创建已连接 UDP 套接字过程格外简单，只需针对 UDP 套接字调用 connect 函数。

```
1 sock = socket(PF_INET, SOCK_DGRAM, 0);
2 memset(&adr, 0, sizeof(adr));
3 adr.sin_family = AF_INET;
4 adr.sin_addr.s_addr = inet_addr(argv[1]);
5 adr.sin_port = htons(atoi(argv[2]));
6 connect(sock, (struct sockaddr *)&adr, sizeof(adr));
```

上述代码看似与 TCP 套接字创建过程一致，但 socket 函数的第二个参数分明是 SOCK\_DGRAM 。也就是说，创建的的确是 UDP 套接字。当然针对 UDP 调用 connect 函数并不是意味着要与对方 UDP 套接字连接，这只是向 UDP 套接字注册目标IP和端口信息。

之后就与 TCP 套接字一致，每次调用 sendto 函数时只需传递信息数据。因为已经指定了收发对象，所以不仅可以使用 sendto、recvfrom 函数，还可以使用 write、read 函数进行通信。

下面的例子把之前的 [uecho client.c](#) 程序改成了基于已连接 UDP 的套接字的程序，因此可以结合 [uecho server.c](#) 程序运行。代码如下：

- [uecho con client.c](#)

编译运行过程与上面一样，故省略。

上面的代码中用 write、read 函数代替了 sendto、recvfrom 函数。

## 6.4 基于 Windows 的实现

暂略

## 6.5 习题

以下答案仅代表本人个人观点，可能不是正确答案。

### 1. UDP 为什么比 TCP 快？为什么 TCP 传输可靠而 UDP 传输不可靠？

答：为了提供可靠的数据传输服务，TCP 在不可靠的IP层进行流控制，而 UDP 缺少这种流控制。所以 UDP 是不可靠的连接。

### 2. 下面不属于 UDP 特点的是？

下面加粗的代表此句话正确

1. **UDP 不同于 TCP，不存在连接概念，所以不像 TCP 那样只能进行一对一的数据传输。**
2. 利用 UDP 传输数据时，如果有 2 个目标，则需要 2 个套接字。
3. UDP 套接字中无法使用已分配给 TCP 的同一端口号
4. **UDP 套接字和 TCP 套接字可以共存。若需要，可以同时在同一主机进行 TCP 和 UDP 数据传输。**
5. 针对 UDP 函数也可以调用 `connect` 函数，此时 UDP 套接字跟 TCP 套接字相同，也需要经过 3 次握手阶段。

### 3. UDP 数据报向对方主机的 UDP 套接字传递过程中，IP 和 UDP 分别负责哪些部分？

答：IP 的作用就是让离开主机的 UDP 数据包准确传递到另一个主机。但把 UDP 包最终交给主机的某一 UDP 套接字的过程则是由 UDP 完成的。UDP 的最重要的作用就是根据端口号将传到主机的数据包交付给最终的 UDP 套接字。

### 4. UDP 一般比 TCP 快，但根据交换数据的特点，其差异可大可小。请你说明何种情况下 UDP 的性能优于 TCP？

答：如果收发数据量小但需要频繁连接时，UDP 比 TCP 更高效。

### 5. 客户端 TCP 套接字调用 `connect` 函数时自动分配 IP 和端口号。UDP 中不调用 `bind` 函数，那何时分配 IP 和端口号？

答：在首次调用 `sendto` 函数时自动给相应的套接字分配 IP 和端口号。而且此时分配的地址一直保留到程序结束为止。

### 6. TCP 客户端必须调用 `connect` 函数，而 UDP 可以选择性调用。请问，在 UDP 中调用 `connect` 函数有哪些好处？

答：要与同一个主机进行长时间通信时，将 UDP 套接字变成已连接套接字会提高效率。因为三个阶段中，第一个阶段和第三个阶段占用了一大部分时间，调用 `connect` 函数可以节省这些时间。

## 第 7 章 优雅的断开套接字的连接

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

本章讨论如何优雅的断开套接字的连接，之前用的方法不够优雅是因为，我们是调用 `close` 函数或 `closesocket` 函数单方面断开连接的。

## 7.1 基于 TCP 的半关闭

TCP 的断开连接过程比建立连接更重要，因为连接过程中一般不会出现大问题，但是断开过程可能发生预想不到的情况。因此应该准确掌控。所以要掌握半关闭（**Half-close**），才能明确断开过程。

### 7.1.1 单方面断开连接带来的问题

Linux 和 Windows 的 `closesocket` 函数意味着完全断开连接。完全断开不仅指无法传输数据，而且也不能接收数据。因此在某些情况下，通信一方单方面的断开套接字连接，显得不太优雅。如图所示：

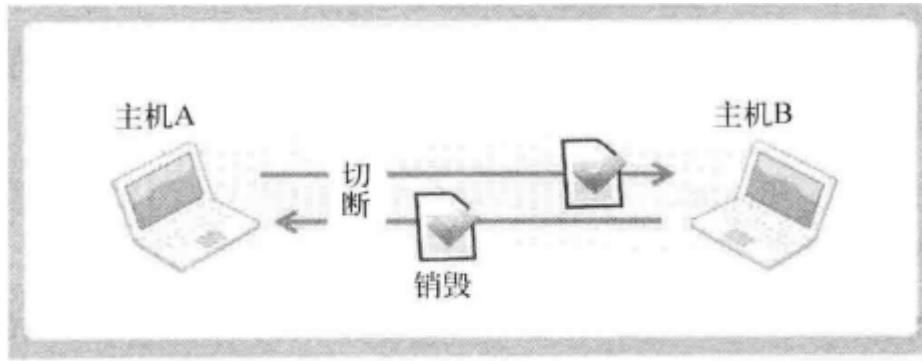


图7-1 单方面断开连接

图中描述的是 2 台主机正在进行双向通信，主机 A 发送完最后的数据后，调用 `close` 函数断开了最后的连接，之后主机 A 无法再接受主机 B 传输的数据。实际上，是完全无法调用与接受数据相关的函数。最终，由主机 B 传输的、主机 A 必须要接受的数据也销毁了。

为了解决这类问题，「只关闭一部分数据交换中使用的流」的方法应运而生。断开一部分连接是指，可以传输数据但是无法接收，或可以接受数据但无法传输。顾名思义就是只关闭流的一半。

### 7.1.2 套接字和流（Stream）

两台主机通过套接字建立连接后进入可交换数据的状态，又称「流形成的状态」。也就是把建立套接字后可交换数据的状态看作一种流。

此处的流可以比作水流。水朝着一个方向流动，同样，在套接字的流中，数据也只能向一个方向流动。因此，为了进行双向通信，需要如图所示的两个流：

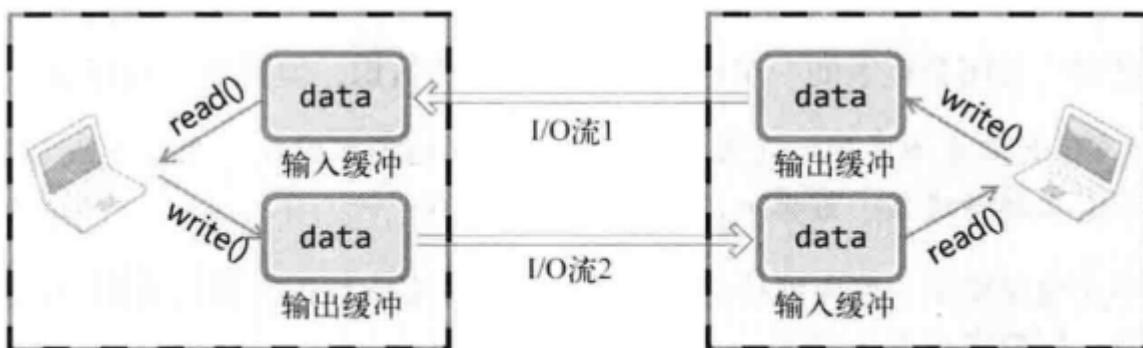


图7-2 套接字中生成的两个流

一旦两台主机之间建立了套接字连接，每个主机就会拥有单独的输入流和输出流。当然，其中一个主机的输入流与另一个主机的输出流相连，而输出流则与另一个主机的输入流相连。另外，本章讨论的「优雅的断开连接方式」只断开其中 1 个流，而非同时断开两个流。Linux 和 Windows 的 `closesocket` 函数将同时断开这两个流，因此与「优雅」二字还有一段距离。

### 7.1.3 针对优雅断开的 `shutdown` 函数

`shutdown` 用来关闭其中一个流：

```
1 #include <sys/socket.h>
2 int shutdown(int sock, int howto);
3 /*
4 成功时返回 0 , 失败时返回 -1
5 sock: 需要断开套接字文件描述符
6 howto: 传递断开方式信息
7 */
```

调用上述函数时，第二个参数决定断开连接的方式，其值如下所示：

- `SHUT_RD` : 断开输入流
- `SHUT_WR` : 断开输出流
- `SHUT_RDWR` : 同时断开 I/O 流

若向 `shutdown` 的第二个参数传递 `SHUT_RD`，则断开输入流，套接字无法接收数据。即使输入缓冲收到数据也回抹去，而且无法调用相关函数。如果向 `shutdown` 的第二个参数传递 `SHUT_WR`，则中断输出流，也就无法传输数据。若如果输出缓冲中还有未传输的数据，则将传递给目标主机。最后，若传递关键字 `SHUT_RDWR`，则同时中断 I/O 流。这相当于分 2 次调用 `shutdown`，其中一次以 `SHUT_RD` 为参数，另一次以 `SHUT_WR` 为参数。

#### 7.1.4 为何要半关闭

考虑以下情况：

一旦客户端连接到服务器，服务器将约定的文件传输给客户端，客户端收到后发送字符串「Thank you」给服务器端。

此处「Thank you」的传递是多余的，这只是用来模拟客户端断开连接前还有数据要传输的情况。此时程序的还嫌难度并不小，因为传输文件的服务器端只需连续传输文件数据即可，而客户端无法知道需要接收数据到何时。客户端也没办法无休止的调用输入函数，因为这有可能导致程序阻塞。

是否可以让服务器和客户端约定一个代表文件尾的字符？

这种方式也有问题，因为这意味着文件中不能有与约定字符相同的内容。为了解决该问题，服务端应最后向客户端传递 EOF 表示文件传输结束。客户端通过函数返回值接受 EOF，这样可以避免与文件内容冲突。那么问题来了，服务端如何传递 EOF ？

断开输出流时向主机传输 EOF。

当然，调用 `close` 函数的同时关闭 I/O 流，这样也会向对方发送 EOF。但此时无法再接受对方传输的数据。换言之，若调用 `close` 函数关闭流，就无法接受客户端最后发送的字符串「Thank you」。这时需要调用 `shutdown` 函数，只关闭服务器的输出流。这样既可以发送 EOF，同时又保留了输入流。下面实现收发文件的服务器端/客户端。

#### 7.1.5 基于半关闭的文件传输程序

上述文件传输服务器端和客户端的数据流可以整理如图：



图7-3 文件传输数据流程图

下面的代码为编程简便，省略了大量错误处理代码。

- [file\\_client.c](#)
- [file\\_server.c](#)

编译运行：

```

1 gcc file_client.c -o fclient
2 gcc file_server.c -o fserver
3 ./fserver 9190
4 ./fclient 127.0.0.1 9190

```

结果：

```

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch07 on git:master x [
10:53:49]
$ ./fserver 9191
Message from client: Thank you

```

客户端接受完成后，服务器会接收到来自客户端的感谢信息。

## 7.2 基于 Windows 的实现

暂略

## 7.3 习题

以下答案仅代表本人个人观点，可能不是正确答案

1. 解释 TCP 中「流」的概念。UDP 中能否形成流？请说明原因。

答：两台主机中通过套接字建立连接后进入可交换数据的状态，又称「流形成的状态」。也就是把建立套接字后可交换数据的状态看做一种流。UDP 没有建立连接的过程，所以不能形成流。

2. Linux 中的 `close` 函数或 Windows 中的 `closesocket` 函数属于单方面断开连接的方法，有可能带来一些问题。什么是单方面断开连接？什么情形下会出现问题？

答：单方面断开连接就是两台主机正在通信，其中一台主机关闭了所有连接，那么一台主机向另一台主机传输的数据可能会没有接收到而损毁。传输文件的服务器只需连续传输文件数据即可，而客户端不知道需要接收数据到何时。客户端也没有办法无休止的调用输入函数。现在需要一个 EOF 代表数据已经传输完毕，那么这时就需要半关闭，服务端把自己的输出流关了，这时客户端就知数据已经传输完毕，因为服务端的输入流还没关，客户端可以给服务器汇报，接收完毕。

3. 什么是半关闭？针对输出流执行半关闭的主机处于何种状态？半关闭会导致对方主机接收什么消息？

答：半关闭就是把输入流或者输出流关了。针对输出流执行半关闭的主机处于可以接收数据而不能发送数据。半关闭会导致对方主机接收一个 EOF 文件结束符。对方就知道你的数据已经传输完毕。

## 第 8 章 域名及网络地址

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

### 8.1 域名系统

DNS 是对 IP 地址和域名进行相互转换的系统，其核心是 DNS 服务器

#### 8.1.1 什么是域名

域名就是我们常常在地址栏里面输入的地址，将比较难记忆的 IP 地址变成人类容易理解的信息。

#### 8.1.2 DNS 服务器

相当于一个字典，可以查询出某一个域名对应的 IP 地址

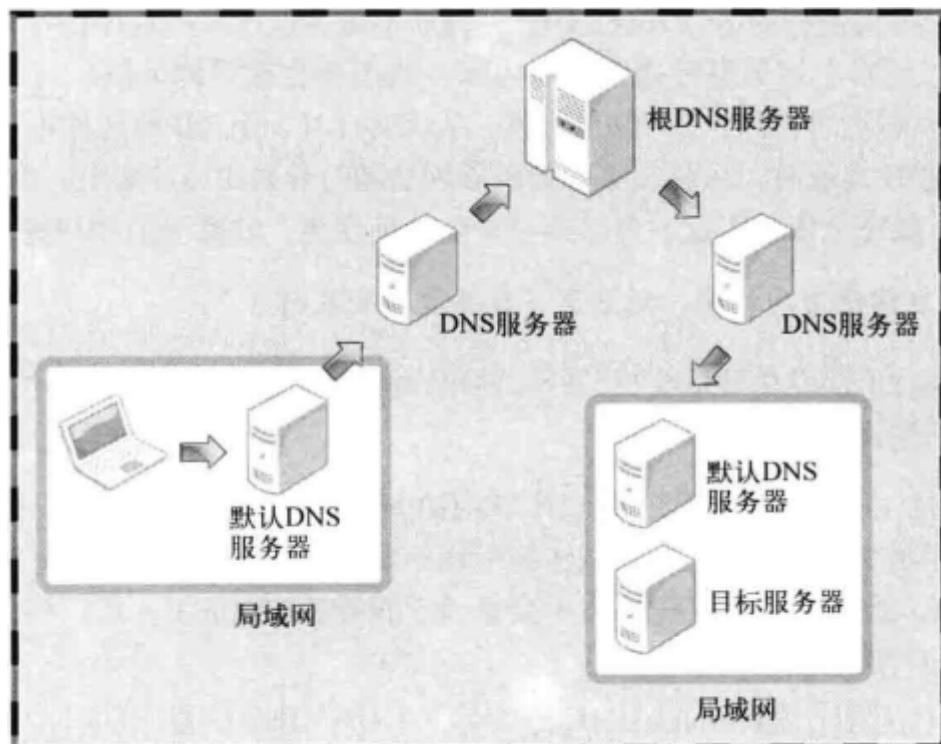


图8-1 DNS和请求获取IP地址信息

如图所示，显示了 DNS 服务器的查询路径。

## 8.2 IP地址和域名之间的转换

### 8.2.1 程序中有必要使用域名吗？

一句话，需要，因为IP地址可能经常改变，而且也不容易记忆，通过域名可以随时更改解析，达到更换IP的目的

### 8.2.2 利用域名获取IP地址

使用以下函数可以通过传递字符串格式的域名获取IP地址

```
1 #include <netdb.h>
2 struct hostent *gethostbyname(const char *hostname);
3 /*
4 成功时返回 hostent 结构体地址，失败时返回 NULL 指针
5 */
```

这个函数使用方便，只要传递字符串，就可以返回域名对应的IP地址。只是返回时，地址信息装入 hostent 结构体。此结构体的定义如下：

```
1 struct hostent
2 {
3     char *h_name;          /* Official name of host. */
4     char **h_aliases;      /* Alias list. */
5     int h_addrtype;        /* Host address type. */
6     int h_length;          /* Length of address. */
7     char **h_addr_list;    /* List of addresses from name server. */
8 };
```

从上述结构体可以看出，不止返回IP信息，同时还带着其他信息一起返回。域名转换成IP时只需要关注 h\_addr\_list 。下面简要说明上述结构体的成员：

- h\_name：该变量中存有官方域名（Official domain name）。官方域名代表某一主页，但实际上，一些著名的公司的域名并没有用官方域名注册。
- h\_aliases：可以通过多个域名访问同一主页。同一IP可以绑定多个域名，因此，除官方域名外还可以指定其他域名。这些信息可以通过 h\_aliases 获得。
- h\_addrtype：gethostbyname 函数不仅支持 IPV4 还支持 IPV6。因此可以通过此变量获取保存在 h\_addr\_list 的IP地址族信息。若是 IPV4，则此变量中存有 AF\_INET。
- h\_length：保存IP地址长度。若是 IPV4 地址，因为是 4 个字节，则保存4；IPV6 时，因为是 16 个字节，故保存 16
- h\_addr\_list：这个是最重要的的成员。通过此变量以整数形式保存域名相对应的IP地址。另外，用户比较多的网站有可能分配多个IP地址给同一个域名，利用多个服务器做负载均衡，。此时可以通过此变量获取IP地址信息。

调用 gethostbyname 函数后，返回的结构体变量如图所示：

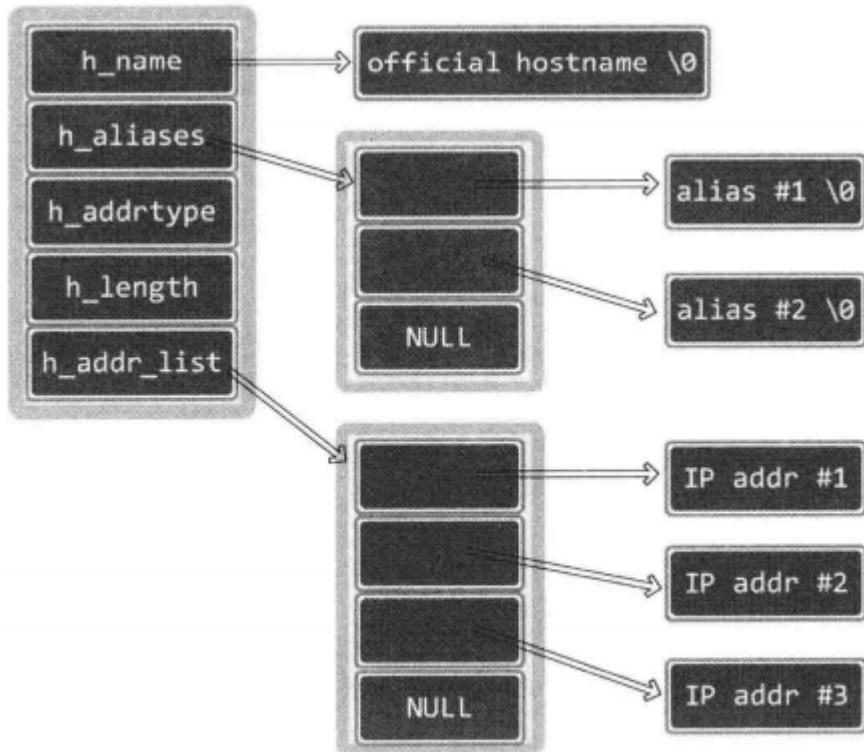


图8-2 hostent结构体变量

下面的代码通过一个例子来演示 gethostbyname 的应用，并说明 hostent 结构体变量特性。

- [gethostbyname.c](#)

编译运行：

```

1 | gcc gethostbyname.c -o hostname
2 | ./hostname www.baidu.com

```

结果：

```

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch08 on git:master x [16:34:20]
$ ./hostname www.baidu.com
Official name: www.a.shifen.com
Aliases 1: www.baidu.com
Address type: AF_INET
IP addr 1: 111.13.100.92
IP addr 2: 111.13.100.91

```

如图所示，显示出了对百度的域名解析

可以看出，百度有一个域名解析是 CNAME 解析的，指向了 [shifen.com](#)，关于百度具体的解析过程。

这一部分牵扯到了很多关于DNS解析的过程，还有 Linux 下关于域名解析的一些命令，我找了一部分资料，可以点下面的链接查看比较详细的：

- [关于百度DNS的解析过程](#)
- [DNS解析的过程是什么，求详细的？](#)
- [Linux DNS 查询剖析](#)
- [Linux DNS查询命令](#)

- [Linux中DNS服务器地址查询命令nslookup使用教程](#)
- [DNS 原理入门](#)

仔细阅读这一段代码：

```
1 inet_ntoa(*(struct in_addr *)host->h_addr_list[i])
```

若只看 hostent 的定义，结构体成员 h\_addr\_list 指向字符串指针数组（由多个字符串地址构成的数组）。但是字符串指针数组保存的元素实际指向的是 in\_addr 结构体变量中地址值而非字符串，也就是说 `(struct in_addr *)host->h_addr_list[i]` 其实是一个指针，然后用 \* 符号取具体的值。如图所示：

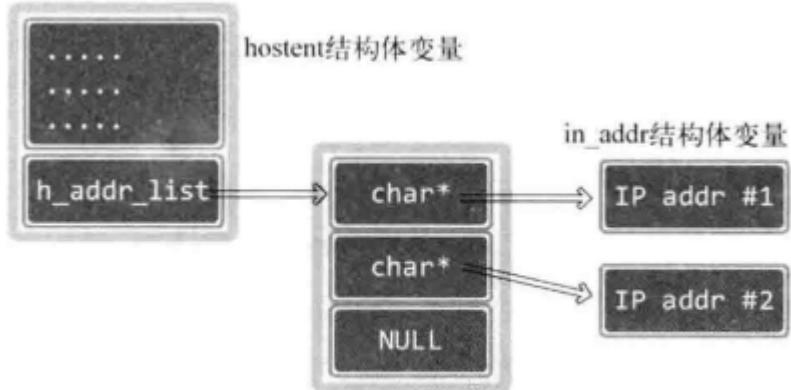


图8-3 h\_addr\_list结构体成员

### 8.2.3 利用IP地址获取域名

请看下面的函数定义：

```

1 #include <netdb.h>
2 struct hostent *gethostbyaddr(const char *addr, socklen_t len, int family);
3 /*
4 成功时返回 hostent 结构体变量地址值，失败时返回 NULL 指针
5 addr：含有IP地址信息的 in_addr 结构体指针。为了同时传递 IPV4 地址之外的全部信息，该变量的类型声明为
char 指针
6 len：向第一个参数传递的地址信息的字节数，IPV4时为 4 ， IPV6 时为16.
7 family：传递地址族信息， ipv4 是 AF_INET ， IPV6是 AF_INET6
8 */
  
```

下面的代码演示使用方法：

- [gethostbyaddr.c](#)

编译运行：

```
1 gcc gethostbyaddr.c -o hostaddr
2 ./hostaddr 8.8.8.8
```

结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch08 on git:master x [17:44:13]
$ ./hostaddr 8.8.8.8
Official name: google-public-dns-a.google.com
Address type: AF_INET
IP addr 1: 8.8.8.8
```

从图上可以看出，`8.8.8.8`这个IP地址是谷歌的。

## 8.3 基于 Windows 的实现

暂略

## 8.4 习题

以下答案仅代表本人个人观点，可能不是正确答案。

### 1. 下列关于DNS的说法错误的是？

答：字体加粗的表示正确答案。

1. 因为**DNS**从存在，故可以使用域名代替**IP**
2. DNS服务器实际上是路由器，因为路由器根据域名决定数据的路径
3. 所有域名信息并非集中与**1台 DNS 服务器**，但可以获取某一 **DNS 服务器**中未注册的所有地址
4. DNS 服务器根据操作系统进行区分，Windows 下的 DNS 服务器和 Linux 下的 DNS 服务器是不同的。

### 2. 阅读如下对话，并说明东秀的方案是否可行。（因为对话的字太多，用图代替）

- 静洙：“东秀吗？我们学校网络中使用的默认DNS服务器发生了故障，无法访问我要投简历的公司主页！有没有办法解决？”
- 东秀：“网络连接正常，但DNS服务器发生了故障？”
- 静洙：“恩！有没有解决方法？是不是要去周围的网吧？”
- 东秀：“有那必要吗？我把我们学校的DNS服务器IP地址告诉你，你改一下你的默认DNS服务器地址。”
- 静洙：“这样可以吗？默认DNS服务器必须连接到本地网络吧！”
- 东秀：“不是！上次我们学校DNS服务器发生故障时，网管就给了我们其他DNS服务器的IP地址呢。”
- 静洙：“那是因为你们学校有多台DNS服务器！”
- 东秀：“是吗？你的话好像也有道理。那你快去网吧吧！”

答：答案就是可行，DNS 服务器是分布式的，一台坏了可以找其他的。

### 3. 再浏览器地址输入 [www.orentec.co.kr](http://www.orentec.co.kr)，并整理出主页显示过程。假设浏览器访问默认 DNS 服务器中并没有关于 [www.orentec.co.kr](http://www.orentec.co.kr) 的地址信息。

答：可以参考一下知乎回答，[在浏览器地址栏输入一个URL后回车，背后会进行哪些技术步骤？](#)，我用我自己的理解，简单说一下，首先会去向上一级的 DNS 服务器去查询，通过这种方式逐级向上传递信息，一直到达根服务器时，它知道应该向哪个 DNS 服务器发起询问。向下传递解析请求，得到IP地址后原路返回，最后会将解析的IP地址传递到发起请求的主机。

## 第 9 章 套接字的多种可选项

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

## 9.1 套接字可选项和 I/O 缓冲大小

我们进行套接字编程时往往只关注数据通信，而忽略了套接字具有的不同特性。但是，理解这些特性并根据实际需要进行更改也很重要。

### 9.1.1 套接字多种可选项

我们之前写得程序都是创建好套接字之后直接使用的，此时通过默认的套接字特性进行数据通信，这里列出了一些套接字可选项。

协议层	选项名	读取	设置
SOL_SOCKET	SO_SNDBUF	○	○
SOL_SOCKET	SO_RCVBUF	○	○
SOL_SOCKET	SO_REUSEADDR	○	○
SOL_SOCKET	SO_KEEPALIVE	○	○
SOL_SOCKET	SO_BROADCAST	○	○
SOL_SOCKET	SO_DONTROUTE	○	○
SOL_SOCKET	SO_OOBINLINE	○	○
SOL_SOCKET	SO_ERROR	○	×
SOL_SOCKET	SO_TYPE	○	×
IPPROTO_IP	IP_TOS	○	○
IPPROTO_IP	IP_TTL	○	○
IPPROTO_IP	IP_MULTICAST_TTL	○	○
IPPROTO_IP	IP_MULTICAST_LOOP	○	○
IPPROTO_IP	IP_MULTICAST_IF	○	○
IPPROTO_TCP	TCP_KEEPALIVE	○	○
IPPROTO_TCP	TCP_NODELAY	○	○
IPPROTO_TCP	TCP_MAXSEG	○	○

从表中可以看出，套接字可选项是分层的。

- IPPROTO\_IP 可选项是IP协议相关事项
- IPPROTO\_TCP 层可选项是 TCP 协议的相关事项
- SOL\_SOCKET 层是套接字的通用可选项。

### 9.1.2 `getsockopt` & `setsockopt`

可选项的读取和设置通过以下两个函数来完成

```
1 #include <sys/socket.h>
2
3 int getsockopt(int sock, int level, int optname, void *optval, socklen_t *optlen);
4 /*
5 成功时返回 0，失败时返回 -1
6 sock: 用于查看选项套接字文件描述符
7 level: 要查看的可选项协议层
8 optname: 要查看的可选项名
9 optval: 保存查看结果的缓冲地址值
10 optlen: 向第四个参数传递的缓冲大小。调用函数后，该变量中保存通过第四个参数返回的可选项信息的字节数。
11 */
```

上述函数可以用来读取套接字可选项，下面的函数可以更改可选项：d

```
1 #include <sys/socket.h>
2
3 int setsockopt(int sock, int level, int optname, const void *optval, socklen_t optlen);
4 /*
5 成功时返回 0，失败时返回 -1
6 sock: 用于更改选项套接字文件描述符
7 level: 要更改的可选项协议层
8 optname: 要更改的可选项名
9 optval: 保存更改结果的缓冲地址值
10 optlen: 向第四个参数传递的缓冲大小。调用函数后，该变量中保存通过第四个参数返回的可选项信息的字节数。
11 */
```

下面的代码可以看出 `getsockopt` 的使用方法。下面示例用协议层为 `SOL_SOCKET`、名为 `SO_TYPE` 的可选项查看套接字类型（TCP 和 UDP）。

- [sock\\_type.c](#)

编译运行：

```
1 gcc sock_type.c -o sock_type
2 ./sock_type
```

结果：

```
1 SOCK_STREAM: 1
2 SOCK_DGRAM: 2
3 Socket type one: 1
4 Socket type two: 2
```

首先创建了一个 TCP 套接字和一个 UDP 套接字。然后通过调用 `getsockopt` 函数来获得当前套接字的状态。

验证套接类型的 `SO_TYPE` 是只读可选项，因为套接字类型只能在创建时决定，以后不能再更改。

### 9.1.3 `SO_SNDBUF` & `SO_RCVBUF`

创建套接字的同时会生成 I/O 缓冲。关于 I/O 缓冲，可以去看第五章。

`SO_RCVBUF` 是输入缓冲大小相关可选项，`SO_SNDBUF` 是输出缓冲大小相关可选项。用这 2 个可选项既可以读取当前 I/O 大小，也可以进行更改。通过下列示例读取创建套接字时默认的 I/O 缓冲大小。

- [get\\_buf.c](#)

编译运行：

```
1 | gcc get_buf.c -o getbuf
2 | ./getbuf
```

运行结果：

```
1 | Input buffer size: 87380
2 | Output buffer size: 16384
```

可以看出本机的输入缓冲和输出缓冲大小。

下面的代码演示了，通过程序设置 I/O 缓冲区的大小

- [set\\_buf.c](#)

编译运行：

```
1 | gcc get_buf.c -o setbuf
2 | ./setbuf
```

结果：

```
1 | Input buffer size: 6144
2 | Output buffer size: 6144
```

输出结果和我们预想的不是很相同，缓冲大小的设置需谨慎处理，因此不会完全按照我们的要求进行。

## 9.2 SO\_REUSEADDR

### 9.2.1 发生地址分配错误（Binding Error）

在学习 `SO_REUSEADDR` 可选项之前，应该好好理解 Time-wait 状态。看以下代码的示例：

- [reuseadr eserver.c](#)

这是一个回声服务器的服务端代码，可以配合第四章的 [echo\\_client.c](#) 使用，在这个代码中，客户端通知服务器终止程序。在客户端控制台输入 Q 可以结束程序，向服务器发送 FIN 消息并经过四次握手过程。当然，输入 CTRL+C 也会向服务器传递 FIN 信息。强制终止程序时，由操作系统关闭文件套接字，此过程相当于调用 `close` 函数，也会向服务器发送 FIN 消息。

这样看不到是什么特殊现象，考虑以下情况：

服务器端和客户端都已经建立连接的状态下，向服务器控制台输入 CTRL+C，强制关闭服务端

如果用这种方式终止程序，如果用同一端口号再次运行服务端，就会输出「bind() error」消息，并且无法再次运行。但是在这种情况下，再过大约 3 分钟就可以重新运行服务端。

### 9.2.2 Time-wait 状态

观察以下过程：

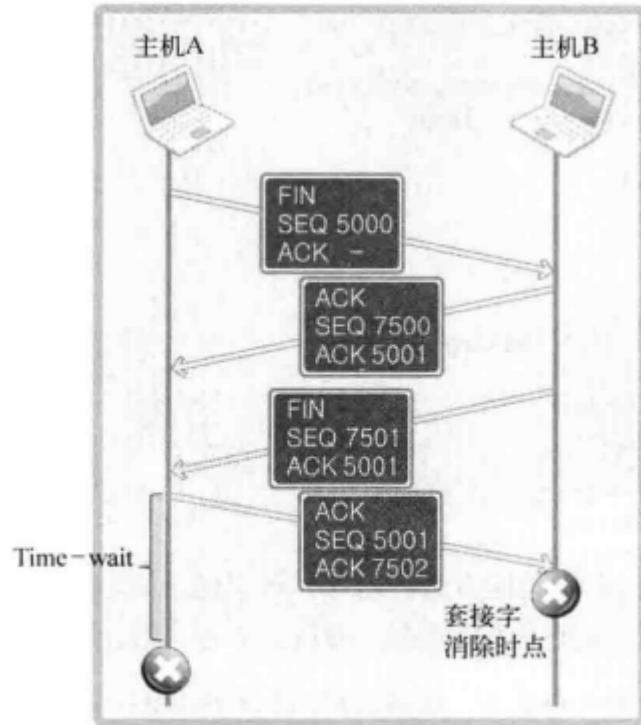


图9-1 Time-wait状态下的套接字

假设图中主机 A 是服务器，因为是主机 A 向 B 发送 FIN 消息，故可想象成服务器端在控制台中输入 `CTRL+C`。但是问题是，套接字经过四次握手后并没有立即消除，而是要经过一段时间的 Time-wait 状态。当然，只有先断开连接的（先发送 FIN 消息的）主机才经过 Time-wait 状态。因此，若服务器端先断开连接，则无法立即重新运行。套接字处在 Time-wait 过程时，相应端口是正在使用的状态。因此，就像之前验证过的，`bind` 函数调用过程中会发生错误。

实际上，不论是服务端还是客户端，都要经过一段时间的 Time-wait 过程。先断开连接的套接字必然会经过 Time-wait 过程，但是由于客户端套接字的端口是任意制定的，所以无需过多关注 Time-wait 状态。

那到底为什么会有 Time-wait 状态呢，在图中假设，主机 A 向主机 B 传输 ACK 消息 (SEQ 5001, ACK 7502) 后立刻消除套接字。但是最后这条 ACK 消息在传递过程中丢失，没有传递到主机 B，这时主机 B 就会试图重传。但是此时主机 A 已经是完全终止状态，因为主机 B 永远无法收到从主机 A 最后传来的 ACK 消息。基于这些问题的考虑，所以要设计 Time-wait 状态。

### 9.2.3 地址再分配

Time-wait 状态看似重要，但是不一定讨人喜欢。如果系统发生故障紧急停止，这时需要尽快重启服务以提供服务，但因处于 Time-wait 状态而必须等待几分钟。因此，Time-wait 并非只有优点，这些情况下容易引发大问题。下图中展示了四次握手时不得不延长 Time-wait 过程的情况。

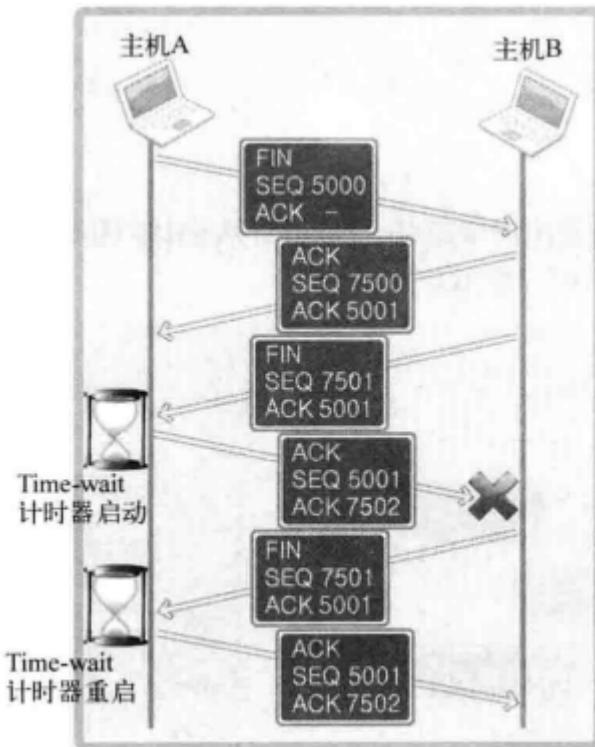


图9-2 重启Time-wait计时器

从图上可以看出，在主机 A 四次握手的过程中，如果最后的数据丢失，则主机 B 会认为主机 A 未能收到自己发送的 FIN 信息，因此重传。这时，收到的 FIN 消息的主机 A 将重启 Time-wait 计时器。因此，如果网络状况不理想，Time-wait 将持续。

解决方案就是在套接字的可选项中更改 SO\_REUSEADDR 的状态。适当调整该参数，可将 Time-wait 状态下的套接字端口号重新分配给新的套接字。SO\_REUSEADDR 的默认值为 0。这就意味着无法分配 Time-wait 状态下的套接字端口号。因此需要将这个值改成 1。具体作法已在示例 [reuseaddr eserver.c](#) 给出，只需要把注释掉的东西接解除注释即可。

```

1 optlen = sizeof(option);
2 option = TRUE;
3 setsockopt(serv_sock, SOL_SOCKET, SO_REUSEADDR, (void *)&option, optlen);

```

此时，已经解决了上述问题。

## 9.3 TCP\_NODELAY

### 9.3.1 Nagle 算法

为了防止因数据包过多而发生网络过载，Nagle 算法诞生了。它应用于 TCP 层。它是否使用会导致如图所示的差异：

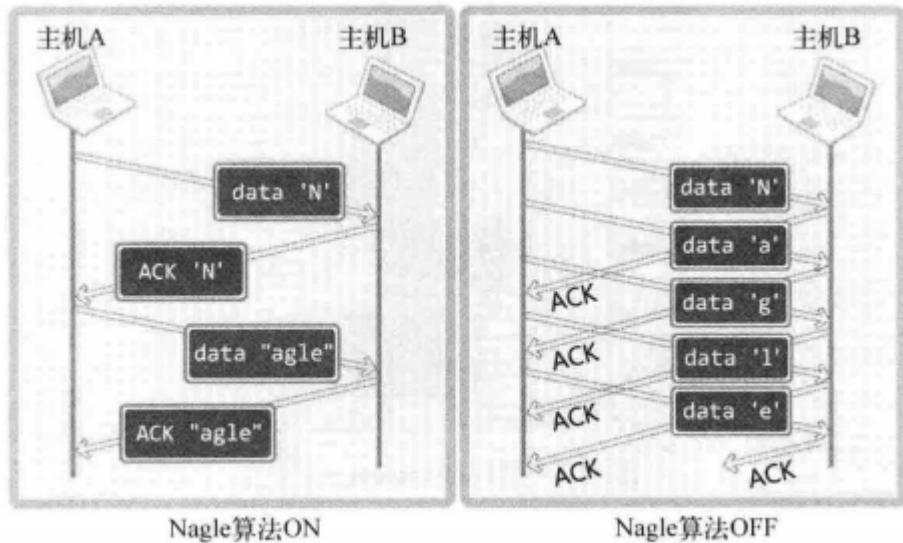


图9-3 Nagle算法

图中展示了通过 Nagle 算法发送字符串 Nagle 和未使用 Nagle 算法的差别。可以得到一个结论。

只有接收到前一数据的 ACK 消息，Nagle 算法才发送下一数据。

TCP 套接字默认使用 Nagle 算法交换数据，因此最大限度的进行缓冲，直到收到 ACK。左图也就是说一共传递 4 个数据包以传输一个字符串。从右图可以看出，发送数据包一共使用了 10 个数据包。由此可知，不使用 Nagle 算法将对网络流量产生负面影响。即使只传输一个字节的数据，其头信息都可能是几十个字节。因此，为了提高网络传输效率，必须使用 Nagle 算法。

Nagle 算法并不是什么情况下都适用，网络流量未受太大影响时，不使用 Nagle 算法要比使用它时传输速度快。最典型的就是「传输大文数据」。将文件数据传入输出缓冲不会花太多时间，因此，不使用 Nagle 算法，也会在装满输出缓冲时传输数据包。这不仅不会增加数据包的数量，反而在无需等待 ACK 的前提下连续传输，因此可以大大提高传输速度。

所以，未准确判断数据性质时不应禁用 Nagle 算法。

### 9.3.2 禁用 Nagle 算法

禁用 Nagle 算法应该使用：

```
1 int opt_val = 1;
2 setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (void *)&opt_val, sizeof(opt_val));
```

通过 TCP\_NODELAY 的值来查看 Nagle 算法的设置状态。

```
1 opt_len = sizeof(opt_val);
2 getsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (void *)&opt_val, opt_len);
```

如果正在使用 Nagle 算法，那么 opt\_val 值为 0，如果禁用则为 1。

关于这个算法，可以参考这个回答：[TCP连接中启用和禁用TCP\\_NODELAY有什么影响？](#)

## 9.4 基于 Windows 的实现

暂略

## 9.5 习题

以下答案仅代表本人个人观点，可能不是正确答案。

### 1. 下列关于 Time-wait 状态的说法错误的是？

答：以下字体加粗的代表正确。

1. Time-wait 状态只在服务器的套接字中发生
  2. 断开连接的四次握手过程中，先传输 FIN 消息的套接字将进入 Time-wait 状态。
  3. Time-wait 状态与断开连接的过程无关，而与请求连接过程中 SYN 消息的传输顺序有关
  4. Time-wait 状态通常并非必要，应尽可能通过更改套接字可选项来防止其发生
2. **TCP\_NODELAY** 可选项与 Nagle 算法有关，可通过它禁用 Nagle 算法。请问何时应考虑禁用 Nagle 算法？结合收发数据的特性给出说明。

答：当网络流量未受太大影响时，不使用 Nagle 算法要比使用它时传输速度快，比如说在传输大文件时。

## 第 10 章 多进程服务器端

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

### 10.1 进程概念及应用

#### 10.1.1 并发服务端的实现方法

通过改进服务端，使其同时向所有发起请求的客户端提供服务，以提高平均满意度。而且，网络程序中数据通信时间比 CPU 运算时间占比更大，因此，向多个客户端提供服务是一种有效的利用 CPU 的方式。接下来讨论同时向多个客户端提供服务的并发服务器端。下面列出的是具有代表性的并发服务端的实现模型和方法：

- 多进程服务器：通过创建多个进程提供服务
- 多路复用服务器：通过捆绑并统一管理 I/O 对象提供服务
- 多线程服务器：通过生成与客户端等量的线程提供服务

先是第一种方法：多进程服务器

#### 10.1.2 理解进程

进程的定义如下：

占用内存空间的正在运行的程序

假如你下载了一个游戏到电脑上，此时的游戏不是进程，而是程序。只有当游戏被加载到主内存并进入运行状态，这是才可称为进程。

#### 10.1.3 进程 ID

在说进程创建方法之前，先要简要说明进程 ID。无论进程是如何创建的，所有的进程都会被操作系统分配一个 ID。此 ID 被称为「进程ID」，其值为大于 2 的证书。1 要分配给操作系统启动后的（用于协助操作系统）首个进程，因此用户无法得到 ID 值为 1。接下来观察在 Linux 中运行的进程。

1 | ps au

通过上面的命令可查看当前运行的所有进程。需要注意的是，该命令同时列出了 PID（进程ID）。参数 a 和 u列出了所有进程的详细信息。

```
# riba2534 @ hpc in ~ [10:06:22]
$ ps au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START  TIME COMMAND
gdm      1022  0.0  0.0 197912  5424  tty1     Ssl+ 09:28  0:00 /usr/lib/gdm3/gdm-way
gdm      1026  0.0  0.1 706652 14116  tty1     Sl+   09:28  0:00 /usr/lib/gnome-session
gdm      1032  0.1  1.6 3651516 135748  tty1     Sl+   09:28  0:03 /usr/bin/gnome-shell
gdm      1074  0.0  0.6 842840  53732  tty1     Sl+   09:28  0:00 /usr/bin/Xwayland :10
gdm      1115  0.0  0.0 361720  7924  tty1     Sl   09:28  0:00 ibus-daemon --xim --p
gdm      1118  0.0  0.0 280980  5992  tty1     Sl   09:28  0:00 /usr/lib/ibus/ibus-dc
gdm      1120  0.0  0.3 403080  28356  tty1     Sl   09:28  0:00 /usr/lib/ibus/ibus-xl
gdm      1139  0.0  0.3 553836  29748  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1142  0.0  0.0 278404  6008  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1144  0.0  0.3 402724  27952  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1145  0.0  0.3 810176  31156  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1149  0.0  0.1 393852  13664  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1150  0.0  0.0 283888  5172  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1151  0.0  0.3 557020  28604  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1152  0.0  0.3 999600  31040  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1160  0.0  0.0 202144  4580  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1162  0.0  0.3 576320  29896  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1165  0.0  0.1 267152  8924  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1167  0.0  0.0 202164  4528  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1171  0.0  0.0 275880  5256  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1172  0.0  0.1 305488  8620  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1179  0.0  0.1 378184  8276  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
gdm      1180  0.0  0.1 333104  8120  tty1     Sl+   09:28  0:00 /usr/lib/gnome-settings-daemon
```

#### 10.1.4 通过调用 fork 函数创建进程

创建进程的方式很多，此处只介绍用于创建多进程服务端的 fork 函数。

```
1 #include <unistd.h>
2 pid_t fork(void);
3 // 成功时返回进程ID,失败时返回 -1
```

fork 函数将创建调用的进程副本。也就是说，并非根据完全不同的程序创建进程，而是复制正在运行的、调用 fork 函数的进程。另外，两个进程都执行 fork 函数调用后的语句（准确的说是在 fork 函数返回后）。但因为是通过同一个进程、复制相同的内存空间，之后的程序流要根据 fork 函数的返回值加以区分。即利用 fork 函数的如下特点区分程序执行流程。

- 父进程：fork 函数返回子进程 ID
- 子进程：fork 函数返回 0

此处，「父进程」（Parent Process）指原进程，即调用 fork 函数的主体，而「子进程」（Child Process）是通过父进程调用 fork 函数复制出的进程。接下来是调用 fork 函数后的程序运行流程。如图所示：

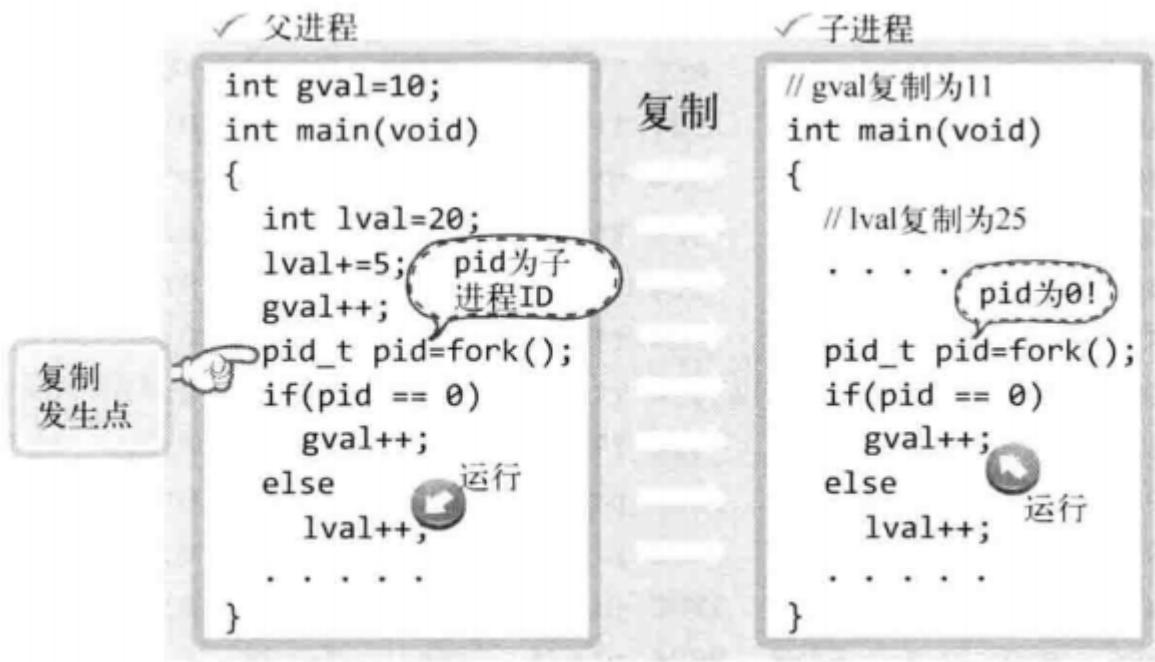


图10-1 fork函数的调用

从图中可以看出，父进程调用 fork 函数的同时复制出子进程，并分别得到 fork 函数的返回值。但复制前，父进程将全局变量 gval 增加到 11，将局部变量 lval 的值增加到 25，因此在这种状态下完成进程复制。复制完成后根据 fork 函数的返回类型区分父子进程。父进程的 lval 的值增加 1，但这不会影响子进程的 lval 值。同样子进程将 gval 的值增加 1 也不会影响到父进程的 gval。因为 fork 函数调用后分成了完全不同的进程，只是二者共享同一段代码而已。接下来给出一个例子：

- [fork.c](#)

```

1 #include <stdio.h>
2 #include <unistd.h>
3 int gval = 10;
4 int main(int argc, char *argv[])
5 {
6     pid_t pid;
7     int lval = 20;
8     gval++, lval += 5;
9     pid = fork();
10    if (pid == 0)
11        gval += 2, lval += 2;
12    else
13        gval -= 2, lval -= 2;
14    if (pid == 0)
15        printf("Child Proc: [%d,%d] \n", gval, lval);
16    else
17        printf("Parent Proc: [%d,%d] \n", gval, lval);
18    return 0;
19 }
```

编译运行：

```
1 gcc fork.c -o fork  
2 ./fork
```

运行结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch10 on git:master x [  
10:43:06]  
$ ./fork  
Parent Proc: [9,23]  
Child Proc: [13,27]
```

可以看出，当执行了 fork 函数之后，此后就相当于有了两个程序在执行代码，对于父进程来说，fork 函数返回的是子进程的ID，对于子进程来说，fork 函数返回 0。所以这两个变量，父进程进行了 +2 操作，而子进程进行了 -2 操作，所以结果是这样。

## 10.2 进程和僵尸进程

文件操作中，关闭文件和打开文件同等重要。同样，进程销毁和进程创建也同等重要。如果未认真对待进程销毁，他们将变成僵尸进程。

### 10.2.1 僵尸（Zombie）进程

进程的工作完成后（执行完 main 函数中的程序后）应被销毁，但有时这些进程将变成僵尸进程，占用系统中的重要资源。这种状态下的进程称作「僵尸进程」，这也是给系统带来负担的原因之一。

僵尸进程是当子进程比父进程先结束，而父进程又没有回收子进程，释放子进程占用的资源，此时子进程将成为一个僵尸进程。如果父进程先退出，子进程被 init 接管，子进程退出后 init 会回收其占用的相关资源。

维基百科：

在类UNIX系统中，僵尸进程是指完成执行（通过exit系统调用，或运行时发生致命错误或收到终止信号所致）但在操作系统的进程表中仍然有一个表项（进程控制块PCB），处于“终止状态”的进程。这发生于子进程需要保留表项以允许其父进程读取子进程的exit status：一旦退出态通过wait系统调用读取，僵尸进程条目就从进程表中删除，称之为“回收（reaped）”。正常情况下，进程直接被其父进程wait并由系统回收。进程长时间保持僵尸状态一般是错误的并导致资源泄漏。

英文术语zombie process源自丧尸 — 不死之人，隐喻子进程已死但仍然没有被收割。与正常进程不同，kill 命令对僵尸进程无效。孤儿进程不同于僵尸进程，其父进程已经死掉，但孤儿进程仍能正常执行，但并不会变为僵尸进程，因为被init（进程ID号为1）收养并wait其退出。

子进程死后，系统会发送SIGCHLD 信号给父进程，父进程对其默认处理是忽略。如果想响应这个消息，父进程通常在SIGCHLD 信号事件处理程序中，使用wait系统调用来响应子进程的终止。

僵尸进程被收割后，其进程号(PID)与在进程表中的表项都可以被系统重用。但如果父进程没有调用wait，僵尸进程将保留进程表中的表项，导致了资源泄漏。某些情况下这反倒是期望的：父进程创建了另外两个子进程，并希望具有不同的进程号。如果父进程通过设置事件处理函数为SIG\_IGN 显式忽略SIGCHLD信号，而不是隐式默认忽略该信号，或者具有SA\_NOCLDWAIT 标志，所有子进程的退出状态信息将被抛弃并且直接被系统回收。

UNIX命令ps列出的进程的状态（"STAT"）栏标示为 "Z" 则为僵尸进程。[\[1\]](#)

收割僵尸进程的方法是通过kill命令手工向其父进程发送SIGCHLD信号。如果其父进程仍然拒绝收割僵尸进程，则终止父进程，使得init进程收养僵尸进程。init进程周期执行wait系统调用收割其收养的所有僵尸进程。

## 10.2.2 产生僵尸进程的原因

为了防止僵尸进程产生，先解释产生僵尸进程的原因。利用如下两个示例展示调用 fork 函数产生子进程的终止方式。

- 传递参数并调用 exit() 函数
- main 函数中执行 return 语句并返回值

向 **exit** 函数传递的参数值和 **main** 函数的 **return** 语句返回的值都回传递给操作系统。而操作系统不会销毁子进程，直到把这些值传递给产生该子进程的父进程。处在这种状态下的进程就是僵尸进程。也就是说将子进程变成僵尸进程的正是操作系统。既然如此，僵尸进程何时被销毁呢？

应该向创建子进程册父进程传递子进程的 exit 参数值或 return 语句的返回值。

如何向父进程传递这些值呢？操作系统不会主动把这些值传递给父进程。只有父进程主动发起请求（函数调用）的时候，操作系统才会传递该值。换言之，如果父进程未主动要求获得子进程结束状态值，操作系统将一直保存，并让子进程长时间处于僵尸进程状态。也就是说，父母要负责收回自己生的孩子。接下来的示例是创建僵尸进程：

- [zombie.c](#)

```
1 #include <stdio.h>
2 #include <unistd.h>
3 int main(int argc, char *argv[])
4 {
5     pid_t pid = fork();
6     if (pid == 0)
7     {
8         puts("Hi, I am a child Process");
9     }
10    else
11    {
12        printf("Child Process ID: %d \n", pid);
13        sleep(30);
14    }
15    if (pid == 0)
16        puts("End child process");
17    else
18        puts("End parent process");
19    return 0;
20 }
```

编译运行：

```
1 gcc zombie.c -o zombie
2 ./zombie
```

结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch10 on git:master x [16:59:15]
$ ./zombie
Child Process ID: 6846
Hi, I am a child Process
End child process
```

因为暂停了 30 秒，所以在这个时间内可以验证一下子进程是否为僵尸进程。

```
riba2534@hpc: ~
riba2534 4202 0.0 0.0 26632 4336 tty2 S+ 16:37 0:00 /opt/google/chrome/na
riba2534 4205 0.0 0.1 438532 14140 tty2 S+ 16:37 0:00 /opt/google/chrome/ch
riba2534 4229 0.2 1.3 491036 105028 tty2 Sl+ 16:37 0:03 /opt/google/chrome/ch
riba2534 4286 0.0 0.1 403164 12028 tty2 S+ 16:37 0:00 /opt/google/chrome/ch
riba2534 4369 0.4 1.9 1956584 156708 tty2 Sl+ 16:37 0:06 /opt/google/chrome/ch
riba2534 4400 0.0 0.5 642512 44088 tty2 Sl+ 16:37 0:00 /opt/google/chrome/ch
riba2534 4410 0.0 1.0 735296 83864 tty2 Sl+ 16:37 0:00 /opt/google/chrome/ch
riba2534 4441 0.0 1.0 742024 85236 tty2 Sl+ 16:37 0:00 /opt/google/chrome/ch
riba2534 4474 0.0 1.1 754828 90836 tty2 Sl+ 16:37 0:00 /opt/google/chrome/ch
riba2534 4477 0.0 0.9 709144 74508 tty2 Sl+ 16:37 0:00 /opt/google/chrome/ch
riba2534 4496 0.0 1.0 736940 85544 tty2 Sl+ 16:37 0:00 /opt/google/chrome/ch
riba2534 4503 0.0 1.9 833664 152628 tty2 Sl+ 16:37 0:01 /opt/google/chrome/ch
riba2534 4508 0.2 1.4 799224 114712 tty2 Sl+ 16:37 0:04 /opt/google/chrome/ch
riba2534 5377 0.1 0.3 1503472 25992 tty2 Sl+ 16:43 0:01 /home/riba2534/.vscode
riba2534 5841 0.0 0.1 836840 12496 tty2 Sl+ 16:45 0:00 /home/riba2534/.vscode
riba2534 6046 0.0 0.0 54928 6868 pts/0 Ss+ 16:54 0:00 /usr/bin/zsh
riba2534 6584 17.0 4.8 1933504 389116 tty2 SLl+ 16:58 0:53 c:\Program Files\Tenc
riba2534 6987 1.7 1.8 870728 146864 tty2 Sl+ 16:59 0:04 /opt/google/chrome/ch
riba2534 7625 0.0 0.6 681492 49732 tty2 Sl+ 17:03 0:00 /opt/google/chrome/ch
riba2534 7811 1.4 0.0 54896 6536 pts/1 Ss 17:03 0:00 zsh
riba2534 7862 0.0 0.0 4508 796 pts/1 S+ 17:03 0:00 ./zombie
riba2534 7863 0.0 0.0 0 0 pts/1 Z+ 17:03 0:00 [zombie] <defunct>
riba2534 7875 8.0 0.0 52432 6104 pts/2 Ss 17:03 0:00 zsh
riba2534 7902 0.0 0.0 46780 3644 pts/2 R+ 17:03 0:00 ps au
```

```
# riba2534 @ hpc in ~ [17:03:49]
```

```
$ 
```

通过 `ps au` 命令可以看出，子进程仍然存在，并没有被销毁，僵尸进程在这里显示为 `Z+ .30` 秒后，红框里面的两个进程会同时被销毁。

利用 `./zombie &` 可以使程序在后台运行，不用打开新的命令行窗口。

### 10.2.3 销毁僵尸进程 1：利用 `wait` 函数

如前所述，为了销毁子进程，父进程应该主动请求获取子进程的返回值。下面是发起请求的具体方法。有两种，下面的函数是其中一种。

```
1 #include <sys/wait.h>
2 pid_t wait(int *statloc);
3 /*
4 成功时返回终止的子进程 ID，失败时返回 -1
5 */
```

调用此函数时如果已有子进程终止，那么子进程终止时传递的返回值（`exit` 函数的参数返回值，`main` 函数的 `return` 返回值）将保存到该函数的参数所指的内存空间。但函数参数指向的单元中还包含其他信息，因此需要用下列宏进行分离：

- WIFEXITED 子进程正常终止时返回「真」
- WEXITSTATUS 返回子进程的返回值

也就是说，向 wait 函数传递变量 status 的地址时，调用 wait 函数后应编写如下代码：

```

1 if (WIFEXITED(status))
2 {
3     puts("Normal termination");
4     printf("Child pass num: %d", WEXITSTATUS(status));
5 }
```

根据以上内容，有如下示例：

- [wait.c](#)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int main(int argc, char *argv[])
7 {
8     int status;
9     pid_t pid = fork(); //这里的子进程将在第13行通过 return 语句终止
10
11    if (pid == 0)
12    {
13        return 3;
14    }
15    else
16    {
17        printf("Child PID: %d \n", pid);
18        pid = fork(); //这里的子进程将在 21 行通过 exit() 函数终止
19        if (pid == 0)
20        {
21            exit(7);
22        }
23        else
24        {
25            printf("Child PID: %d \n", pid);
26            wait(&status);           //之间终止的子进程相关信息将被保存到 status 中，同时相关子进程
被完全销毁
27            if (WIFEXITED(status)) //通过 WIFEXITED 来验证子进程是否正常终止。如果正常终止，则调
用 WEXITSTATUS 宏输出子进程返回值
28            printf("Child send one: %d \n", WEXITSTATUS(status));
29
30            wait(&status); //因为之前创建了两个进程，所以再次调用 wait 函数和宏
31            if (WIFEXITED(status))
32                printf("Child send two: %d \n", WEXITSTATUS(status));
33                sleep(30);
34        }
35    }
```

```
36     return 0;
37 }
```

编译运行：

```
1 gcc wait.c -o wait
2 ./wait
```

结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch10 on git:master x
17:38:04]
$ ./wait
Child PID: 12687
Child PID: 12688
Child send one: 3
Child send two: 7
□
```

此时，系统中并没有上述 PID 对应的进程，这是因为调用了 `wait` 函数，完全销毁了该子进程。另外两个子进程返回时返回的 3 和 7 传递到了父进程。

这就是通过 `wait` 函数消灭僵尸进程的方法，调用 `wait` 函数时，如果没有已经终止的子进程，那么程序将阻塞（Blocking）直到有子进程终止，因此要谨慎调用该函数。

#### 10.2.4 销毁僵尸进程 2：使用 `waitpid` 函数

`wait` 函数会引起程序阻塞，还可以考虑调用 `waitpid` 函数。这是防止僵尸进程的第二种方法，也是防止阻塞的方法。

```
1 #include <sys/wait.h>
2 pid_t waitpid(pid_t pid, int *statloc, int options);
3 /*
4 成功时返回终止的子进程ID 或 0，失败时返回 -1
5 pid: 等待终止的目标子进程的ID,若传 -1，则与 wait 函数相同，可以等待任意子进程终止
6 statloc: 与 wait 函数的 statloc 参数具有相同含义
7 options: 传递头文件 sys/wait.h 声明的常量 WNOHANG，即使没有终止的子进程也不会进入阻塞状态，而是返回
8 0 退出函数。
*/
```

以下是 `waitpid` 的使用示例：

- [waitpid.c](#)

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4 int main(int argc, char *argv[])
5 {
6     int status;
7     pid_t pid = fork();
8     if (pid == 0)
```

```

9     {
10        sleep(15); //用 sleep 推迟子进程的执行
11        return 24;
12    }
13 else
14 {
15     //调用waitpid 传递参数 WNOHANG , 这样之前有没有终止的子进程则返回0
16     while (!waitpid(-1, &status, WNOHANG))
17     {
18         sleep(3);
19         puts("sleep 3 sec.");
20     }
21     if (WIFEXITED(status))
22         printf("Child send %d \n", WEXITSTATUS(status));
23 }
24 return 0;
25 }
```

编译运行：

```

1 | gcc waitpid.c -o waitpid
2 | ./waitpid
```

结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch10 on git:master x
18:03:09]
$ ./waitpid
sleep 3 sec.
Child send 24
```

可以看出来，在 while 循环中正好执行了 5 次。这也证明了 waitpid 函数并没有阻塞

## 10.3 信号处理

我们已经知道了进程的创建及销毁的办法，但是还有一个问题没有解决。

子进程究竟何时终止？调用 waitpid 函数后要无休止的等待吗？

### 10.3.1 向操作系统求助

子进程终止的识别主题是操作系统，因此，若操作系统能把如下信息告诉正忙于工作的父进程，将有助于构建更高效的程序

为了实现上述的功能，引入信号处理机制（Signal Handling）。此处「信号」是在特定事件发生时由操作系统向进程发送的消息。另外，为了响应该消息，执行与消息相关的自定义操作的过程被称为「处理」或「信号处理」。

### 10.3.2 信号与 signal 函数

下面进程和操作系统的对话可以帮助理解信号处理。

进程：操作系统，如果我之前创建的子进程终止，就帮我调用 zombie\_handler 函数。

操作系统：好的，如果你的子进程终止，我帮你调用 zombie\_handler 函数，你先把要函数要执行的语句写好。

上述的对话，相当于「注册信号」的过程。即进程发现自己的子进程结束时，请求操作系统调用的特定函数。该请求可以通过如下函数调用完成：

```
1 #include <signal.h>
2 void (*signal(int signo, void (*func)(int)))(int);
3 /*
4 为了在产生信号时调用，返回之前注册的函数指针
5 函数名: signal
6 参数: int signo,void(*func)(int)
7 返回类型: 参数类型为int型，返回 void 型函数指针
8 */
```

调用上述函数时，第一个参数为特殊情况信息，第二个参数为特殊情况下将要调用的函数的地址值（指针）。发生第一个参数代表的情况时，调用第二个参数所指的函数。下面给出可以在 signal 函数中注册的部分特殊情况和对应的函数。

- SIGALRM: 已到通过调用 alarm 函数注册时间
- SIGINT: 输入 ctrl+c
- SIGCHLD: 子进程终止

接下来编写调用 signal 函数的语句完成如下请求：

「子进程终止则调用 mychild 函数」

此时 mychild 函数的参数应为 int，返回值类型应为 void。只有这样才能称为 signal 函数的第二个参数。另外，常数 SIGCHLD 定义了子进程终止的情况，应成为 signal 函数的第一个参数。也就是说，signal 函数调用语句如下：

```
1 signal(SIGCHLD , mychild);
```

接下来编写 signal 函数的调用语句，分别完成如下两个请求：

1. 已到通过 alarm 函数注册时间，请调用 timeout 函数
2. 输入 ctrl+c 时调用 keycontrol 函数

代表这 2 种情况的常数分别为 SIGALRM 和 SIGINT，因此按如下方式调用 signal 函数。

```
1 signal(SIGALRM , timeout);
2 signal(SIGINT , keycontrol);
```

以上就是信号注册过程。注册好信号之后，发生注册信号时（注册的情况发生时），操作系统将调用该信号对应的函数。先介绍 alarm 函数。

```
1 #include <unistd.h>
2 unsigned int alarm(unsigned int seconds);
3 // 返回0或以秒为单位的距 SIGALRM 信号发生所剩时间
```

如果调用该函数的同时向它传递一个正整型参数，相应时间后（以秒为单位）将产生 SIGALRM 信号。若向该函数传递为 0，则之前对 SIGALRM 信号的预约将取消。如果通过改函数预约信号后未指定该信号对应的处理函数，则（通过调用 signal 函数）终止进程，不做任何处理。

- [signal.c](#)

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <signal.h>
4 void timeout(int sig) //信号处理器
5 {
6     if (sig == SIGALRM)
7         puts("Time out!");
8     alarm(2); //为了每隔 2 秒重复产生 SIGALRM 信号，在信号处理器中调用 alarm 函数
9 }
10 void keycontrol(int sig) //信号处理器
11 {
12     if (sig == SIGINT)
13         puts("CTRL+C pressed");
14 }
15 int main(int argc, char *argv[])
16 {
17     int i;
18     signal(SIGALRM, timeout); //注册信号及相应处理器
19     signal(SIGINT, keycontrol);
20     alarm(2); //预约 2 秒候发生 SIGALRM 信号
21
22     for (i = 0; i < 3; i++)
23     {
24         puts("wait...");
25         sleep(100);
26     }
27     return 0;
28 }
```

编译运行：

```
1 gcc signal.c -o signal
2 ./signal
```

结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch10 on git:master
20:41:08]
$ ./signal
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

上述结果是没有任何输入的运行结果。当输入 `ctrl+c` 时：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch10 on git:master x
20:42:44]
$ ./signal
wait...
Time out!
wait...
^CCTRL+C pressed
wait...
Time out!
```

就可以看到 `CTRL+C pressed` 的字符串。

发生信号时将唤醒由于调用 `sleep` 函数而进入阻塞状态的进程。

调用函数的主题的确是操作系统，但是进程处于睡眠状态时无法调用函数，因此，产生信号时，为了调用信号处理器，将唤醒由于调用 `sleep` 函数而进入阻塞状态的进程。而且，进程一旦被唤醒，就不会再进入睡眠状态。即使还未到 `sleep` 中规定的时间也是如此。所以上述示例运行不到 10 秒后就会结束，连续输入 `CTRL+C` 可能连一秒都不到。

简言之，就是本来系统要睡眠100秒，但是到了 `alarm(2)` 规定的两秒之后，就会唤醒睡眠的进程，进程被唤醒了就不会再进入睡眠状态了，所以就不用等待100秒。如果把 `timeout()` 函数中的 `alarm(2)` 注释掉，就会先输出 `wait...`，然后再输出 `Time out!` (这时已经跳过了第一次的 `sleep(100)` 秒)，然后就真的会睡眠100秒，因为没有再发出 `alarm(2)` 的信号。

### 10.3.3 利用 `sigaction` 函数进行信号处理

前面所学的内容可以防止僵尸进程，还有一个函数，叫做 `sigaction` 函数，他类似于 `signal` 函数，而且可以完全代替后者，也更稳定。之所以稳定，是因为：

signal 函数在 Unix 系列的不同操作系统可能存在区别，但 `sigaction` 函数完全相同

实际上现在很少用 `signal` 函数编写程序，他只是为了保持对旧程序的兼容，下面介绍 `sigaction` 函数，只讲解可以替换 `signal` 函数的功能。

```
1 #include <signal.h>
2
3 int sigaction(int signo, const struct sigaction *act, struct sigaction *oldact);
4 /*
5 成功时返回 0，失败时返回 -1
6 act: 对于第一个参数的信号处理函数（信号处理器）信息。
7 oldact: 通过此参数获取之前注册的信号处理函数指针，若不需要则传递 0
8 */
```

声明并初始化 `sigaction` 结构体变量以调用上述函数，该结构体定义如下：

```
1 struct sigaction
2 {
3     void (*sa_handler)(int);
4     sigset_t sa_mask;
5     int sa_flags;
6 };
```

此结构体的成员 `sa_handler` 保存信号处理的函数指针值（地址值）。`sa_mask` 和 `sa_flags` 的所有位初始化 0 即可。这 2 个成员用于指定信号相关的选项和特性，而我们的目的主要是防止产生僵尸进程，故省略。

下面的示例是关于 `sigaction` 函数的使用方法。

- [sigaction.c](#)

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <signal.h>
4
5 void timeout(int sig)
6 {
7     if (sig == SIGALRM)
8         puts("Time out!");
9     alarm(2);
10 }
11
12 int main(int argc, char *argv[])
13 {
14     int i;
15     struct sigaction act;
16     act.sa_handler = timeout; //保存函数指针
17     sigemptyset(&act.sa_mask); //将 sa_mask 函数的所有位初始化成0
18     act.sa_flags = 0; //sa_flags 同样初始化成 0
19     sigaction(SIGALRM, &act, 0); //注册 SIGALRM 信号的处理器。
20
21     alarm(2); //2 秒后发生 SIGALRM 信号
22
23     for (int i = 0; i < 3; i++)
24     {
25         puts("wait...");
26         sleep(100);
27     }
28     return 0;
29 }
30
```

编译运行：

```
1 gcc sigaction.c -o sigaction
2 ./sigaction
```

结果：

```
1 wait...
2 Time out!
3 wait...
4 Time out!
5 wait...
6 Time out!
```

可以发现，结果和之前用 signal 函数的结果没有什么区别。以上就是信号处理的相关理论。

### 10.3.4 利用信号处理技术消灭僵尸进程

下面利用子进程终止时产生 SIGCHLD 信号这一点，来用信号处理来消灭僵尸进程。看以下代码：

- [remove\\_zombie.c](#)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <signal.h>
5 #include <sys/wait.h>
6
7 void read_childproc(int sig)
8 {
9     int status;
10    pid_t id = waitpid(-1, &status, WNOHANG);
11    if (WIFEXITED(status))
12    {
13        printf("Removed proc id: %d \n", id);           //子进程的 pid
14        printf("Child send: %d \n", WEXITSTATUS(status)); //子进程的返回值
15    }
16 }
17
18 int main(int argc, char *argv[])
19 {
20     pid_t pid;
21     struct sigaction act;
22     act.sa_handler = read_childproc;
23     sigemptyset(&act.sa_mask);
24     act.sa_flags = 0;
25     sigaction(SIGCHLD, &act, 0);
26
27     pid = fork();
28     if (pid == 0) //子进程执行阶段
29     {
30         puts("Hi I'm child process");
31         sleep(10);
32         return 12;
33     }
34     else //父进程执行阶段
35     {
```

```
36     printf("Child proc id: %d\n", pid);
37     pid = fork();
38     if (pid == 0)
39     {
40         puts("Hi! I'm child process");
41         sleep(10);
42         exit(24);
43     }
44     else
45     {
46         int i;
47         printf("Child proc id: %d \n", pid);
48         for (i = 0; i < 5; i++)
49         {
50             puts("wait");
51             sleep(5);
52         }
53     }
54 }
55 return 0;
56 }
```

编译运行：

```
1 | gcc remove_zombie.c -o zombie
2 | ./zombie
```

结果：

```
1 Child proc id: 11211
2 Hi I'm child process
3 Child proc id: 11212
4 wait
5 Hi! I'm child process
6
7 wait
8
9 wait
10 Removed proc id: 11211
11 Child send: 12
12 wait
13 Removed proc id: 11212
14 Child send: 24
15 wait
```

请自习观察结果，结果中的每一个空行代表间隔了5秒，程序是先创建了两个子进程，然后子进程10秒之后会返回值，第一个wait由于子进程在执行，所以直接被唤醒，然后这两个子进程正在睡10秒，所以5秒之后第二个wait开始执行，又过了5秒，两个子进程同时被唤醒。所以剩下的wait也被唤醒。

所以在本程序的过程中，当子进程终止时候，会向系统发送一个信号，然后调用我们提前写好的处理函数，在处理函数中使用`waitpid`来处理僵尸进程，获取子进程返回值。

## 10.4 基于多任务的并发服务器

### 10.4.1 基于进程的并发服务器模型

之前的回声服务器每次只能同时向 1 个客户端提供服务。因此，需要扩展回声服务器，使其可以同时向多个客户端提供服务。下图是基于多进程的回声服务器的模型。

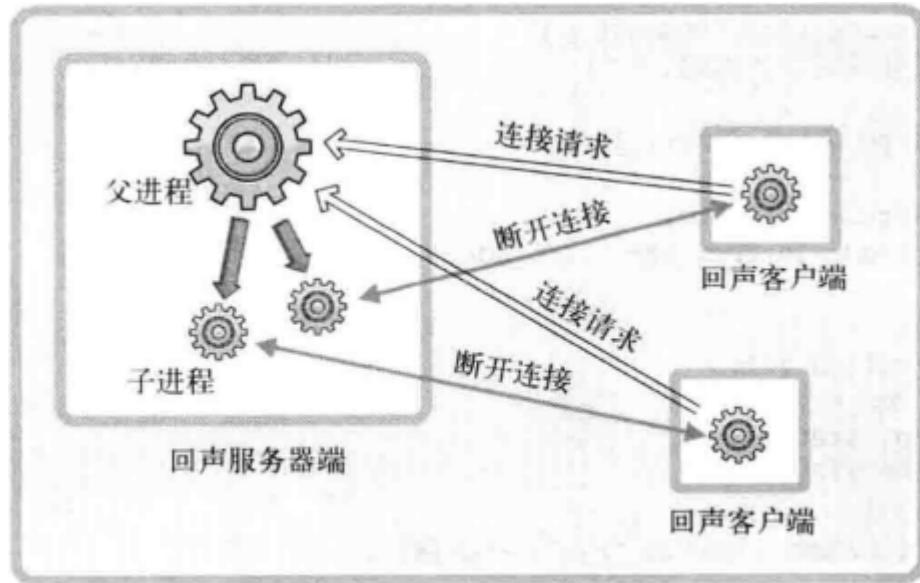


图10-2 并发服务器模型

从图中可以看出，每当有客户端请求时（连接请求），回声服务器都创建子进程以提供服务。如果请求的客户端有 5 个，则将创建 5 个子进程来提供服务，为了完成这些任务，需要经过如下过程：

- 第一阶段：回声服务器端（父进程）通过调用 accept 函数受理连接请求
- 第二阶段：此时获取的套接字文件描述符创建并传递给子进程
- 第三阶段：进程利用传递来的文件描述符提供服务

### 10.4.2 实现并发服务器

下面是基于多进程实现的并发服务器的服务端，可以结合第四章的 [echo\\_client.c](#) 回声客户端来运行。

- [echo\\_mperv.c](#)

编译运行：

```
1 | gcc echo_mperv.c -o eserver
2 | ./eserver
```

结果：

和第四章的几乎一样，可以自己测试，此时的服务端支持同时给多个客户端进行服务，每有一个客户端连接服务端，就会多开一个子进程，所以可以同时提供服务。

### 10.4.3 通过 fork 函数复制文件描述符

示例中给出了通过 fork 函数复制文件描述符的过程。父进程将 2 个套接字（一个是服务端套接字另一个是客户端套接字）文件描述符复制给了子进程。

调用 fork 函数时复制父进程的所有资源，但是套接字不是归进程所有的，而是归操作系统所有，只是进程拥有代表相应套接字的文件描述符。

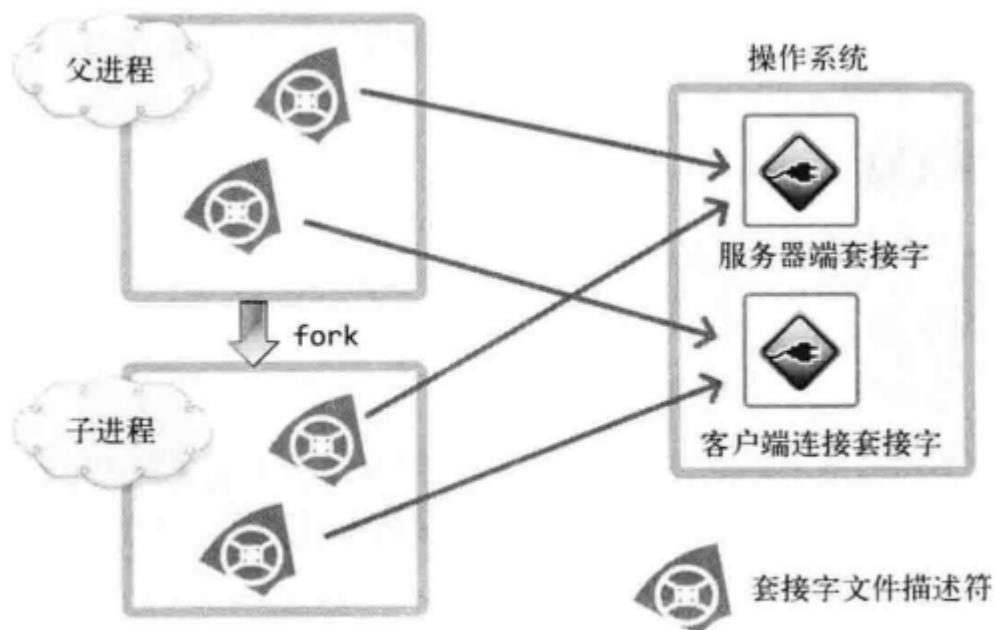


图10-3 调用fork函数并复制文件描述符

如图所示，1个套接字存在2个文件描述符时，只有2个文件描述符都终止（销毁）后，才能销毁套接字。如果维持图中的状态，即使子进程销毁了与客户端连接的套接字文件描述符，也无法销毁套接字（服务器套接字同样如此）。因此调用 fork 函数时，要将无关紧要的套接字文件描述符关掉，如图所示：

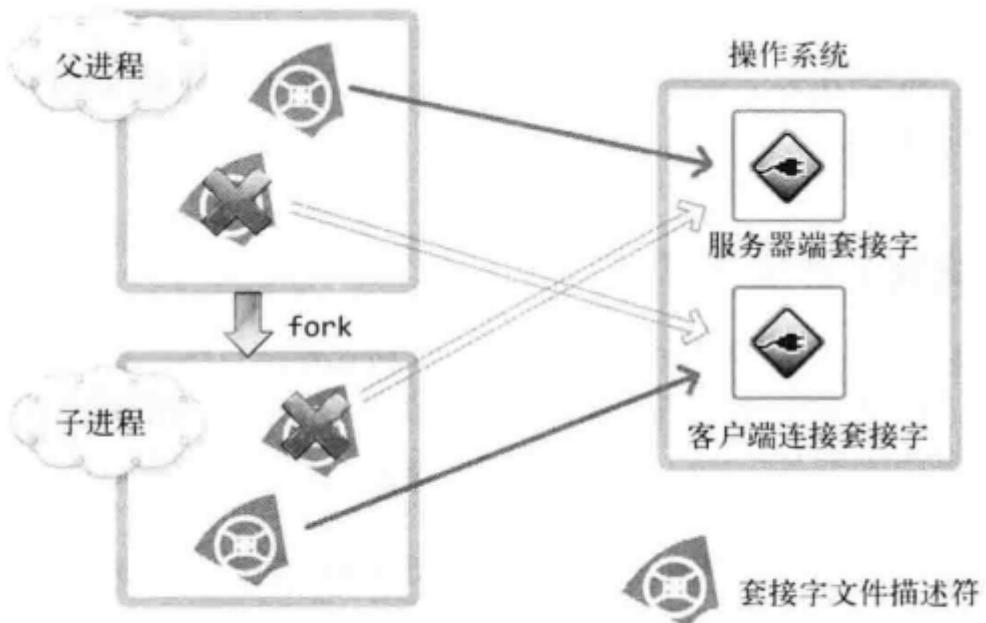


图10-4 整理复制的文件描述符

## 10.5 分割 TCP 的 I/O 程序

### 10.5.1 分割 I/O 的优点

我们已经实现的回声客户端的数据回声方式如下：

向服务器传输数据，并等待服务器端回复。无条件等待，直到接收完服务器端的回声数据后，才能传输下一批数据。

传输数据后要等待服务器端返回的数据，因为程序代码中重复调用了 `read` 和 `write` 函数。只能这么写的原因之一是，程序在 1 个进程中运行，现在可以创建多个进程，因此可以分割数据收发过程。默认分割过程如下图所示：

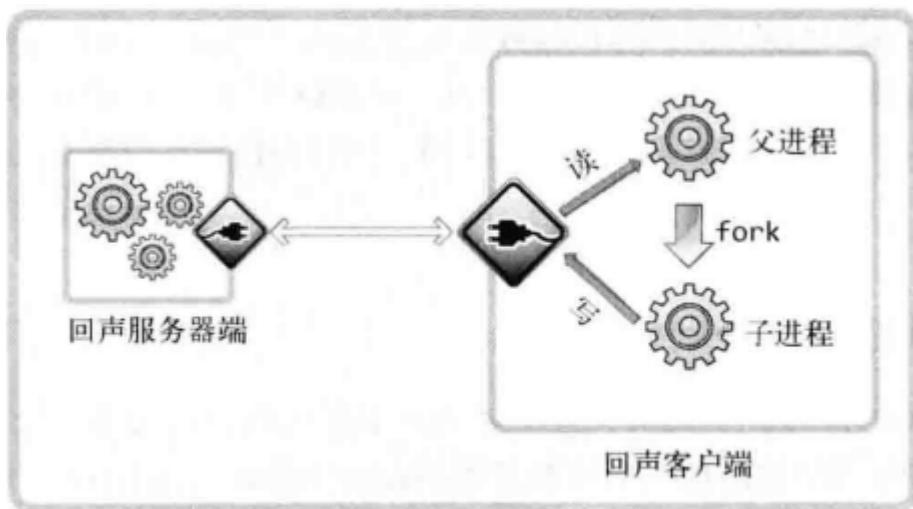


图10-5 回声客户端I/O分割模型

从图中可以看出，客户端的父进程负责接收数据，额外创建的子进程负责发送数据，分割后，不同进程分别负责输入输出，这样，无论客户端是否从服务器端接收完数据都可以进程传输。

分割 I/O 程序的另外一个好处是，可以提高频繁交换数据的程序性能，图下图所示：

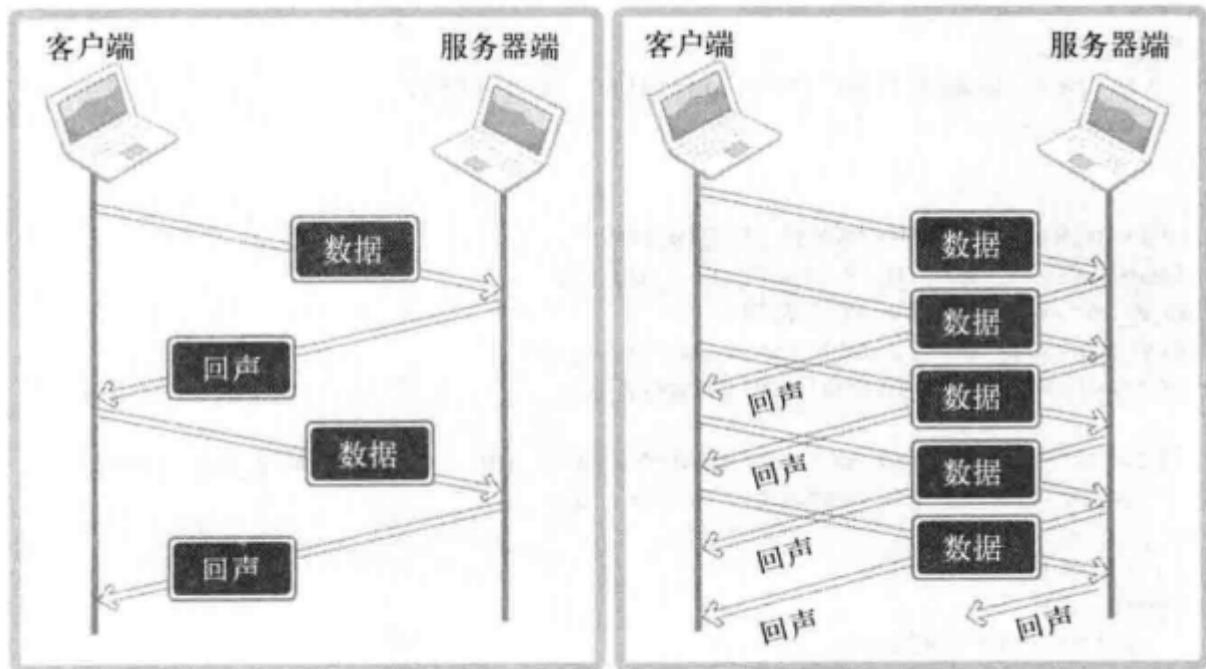


图10-6 数据交换方法比较

根据上图显示可以看出，在网络不好的情况下，明显提升速度。

## 10.5.2 回声客户端的 I/O 程序分割

下面是回声客户端的 I/O 分割的代码实现：

- [echo\\_mpclient.c](#)

可以配合刚才的并发服务器进行执行。

编译运行：

```
1 | gcc echo_mpclient.c -o eclient
2 | ./eclient 127.0.0.1 9190
```

结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch10 on git:master x [15:23:46] C:130
$ ./eclient 127.0.0.1 9190
Hello
Message from server: Hello
哈哈哈
Message from server: 哈哈哈
q
```

可以看出，基本和以前的一样，但是里面的内部结构却发生了很大的变化

## 10.6 习题

以下答案仅代表本人个人观点，可能不是正确答案。

1. 下列关于进程的说法错误的是？

答：以下加粗的内容为正确的

1. **从操作系统的角度上说，进程是程序运行的单位**
2. 进程根据创建方式建立父子关系
3. 进程可以包含其他进程，即一个进程的内存空间可以包含其他进程
4. 子进程可以创建其他子进程，而创建出来的子进程还可以创建其他子进程，但所有这些进程只与一个父进程建立父子关系。

2. 调用 **fork** 函数将创建子进程，一下关于子进程错误的是？

答：以下加粗的内容为正确的

1. 父进程销毁时也会同时销毁子进程
2. 子进程是复制父进程所有资源创建出的进程
3. 父子进程共享全局变量
4. 通过 **fork** 函数创建的子进程将执行从开始到 **fork** 函数调用为止的代码。

3. 创建子进程时复制父进程所有内容，此时复制对象也包含套接字文件描述符。编写程序验证复制的文件描述符整数值是否与原文件描述符数值相同。

答：代码为多进程服务器修改而来，代码：[test\\_server.c](#)

运行截图：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch10 on git:master x [15:49:33]
$ ./test_server 9190
父进程的 serv_sock: 3,clnt_sock:4
new client connected...
子进程的 serv_sock: 3,clnt_sock:4
client disconnected...
removed proc id: 31707
父进程的 serv_sock: 3,clnt_sock:-1
^C
```

从图上可以看出，数值相同。

#### 4. 请说明进程变为僵尸进程的过程以及预防措施。

答：当一个父进程以fork()系统调用建立一个新的子进程后，核心进程就会在进程表中给这个子进程分配一个进入点，然后将相关信息存储在该进入点所对应的进程表内。这些信息中有一项是其父进程的识别码。而当这个子进程结束的时候（比如调用exit命令结束），其实他并没有真正的被销毁，而是留下一个称为僵尸进程（Zombie）的数据结构（系统调用exit的作用是使进程退出，但是也仅仅限于一个正常的进程变成了一个僵尸进程，并不能完全将其销毁）。预防措施：通过wait和waitpid函数加上信号函数写代码来预防。

## 第 11 章 进程间通信

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

进程间通信，意味着两个不同的进程中可以交换数据

### 11.1 进程间通信的基本概念

#### 11.1.1 通过管道实现进程间通信

下图是基于管道（PIPE）的进程间通信的模型：

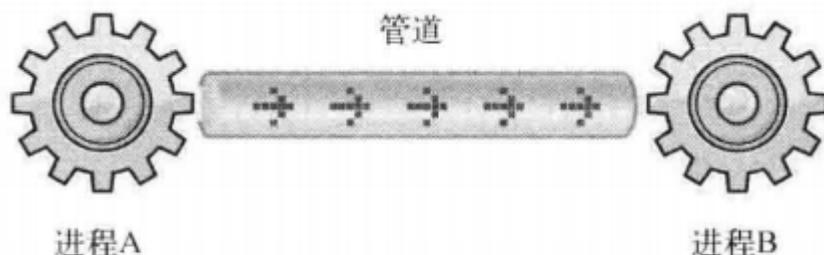


图11-1 基于管道的进程间通信模型

可以看出，为了完成进程间通信，需要创建进程。管道并非属于进程的资源，而是和套接字一样，属于操作系统（也就不是fork函数的复制对象）。所以，两个进程通过操作系统提供的内存空间进行通信。下面是创建管道的函数。

```
1 #include <unistd.h>
2 int pipe(int filedes[2]);
3 /*
4 成功时返回 0，失败时返回 -1
5 filedes[0]: 通过管道接收数据时使用的文件描述符，即管道出口
6 filedes[1]: 通过管道传输数据时使用的文件描述符，即管道入口
7 */
```

父进程创建函数时将创建管道，同时获取对应于出入口的文件描述符，此时父进程可以读写同一管道。但父进程的目的是与子进程进行数据交换，因此需要将入口或出口中的 1 个文件描述符传递给子进程。下面的例子是关于该函数的使用方法：

- [pipe1.c](#)

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #define BUF_SIZE 30
4
5 int main(int argc, char *argv[])
6 {
7     int fds[2];
8     char str[] = "Who are you?";
9     char buf[BUF_SIZE];
10    pid_t pid;
11    // 调用 pipe 函数创建管道, fds 数组中保存用于 I/O 的文件描述符
12    pipe(fds);
13    pid = fork(); //子进程将同时拥有创建管道获取的2个文件描述符, 复制的并非管道, 而是文件描述符
14    if (pid == 0)
15    {
16        write(fds[1], str, sizeof(str));
17    }
18    else
19    {
20        read(fds[0], buf, BUF_SIZE);
21        puts(buf);
22    }
23    return 0;
24 }
```

编译运行：

```
1 gcc pipe1.c -o pipe1
2 ./pipe1
```

结果：

```
1 Who are you?
```

可以从程序中看出，首先创建了一个管道，子进程通过 `fds[1]` 把数据写入管道，父进程从 `fds[0]` 再把数据读出来。可以从下图看出：

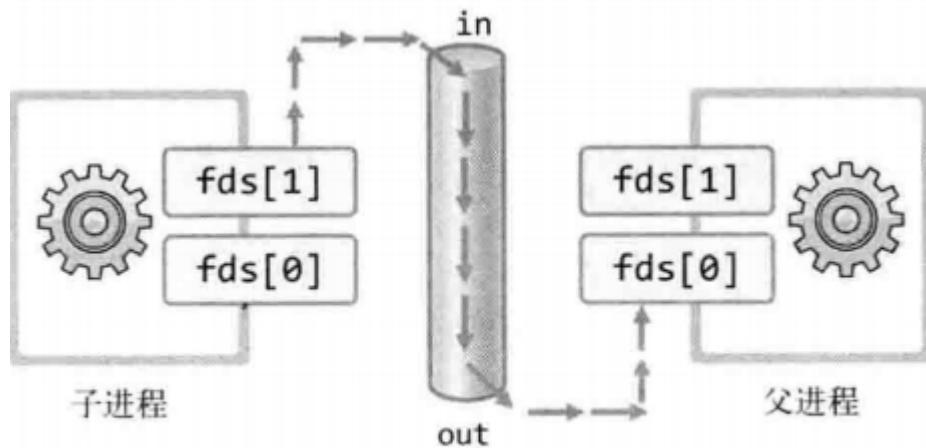


图11-2 示例pipe1.c的通信路径

### 11.1.2 通过管道进行进程间双向通信

下图可以看出双向通信模型：

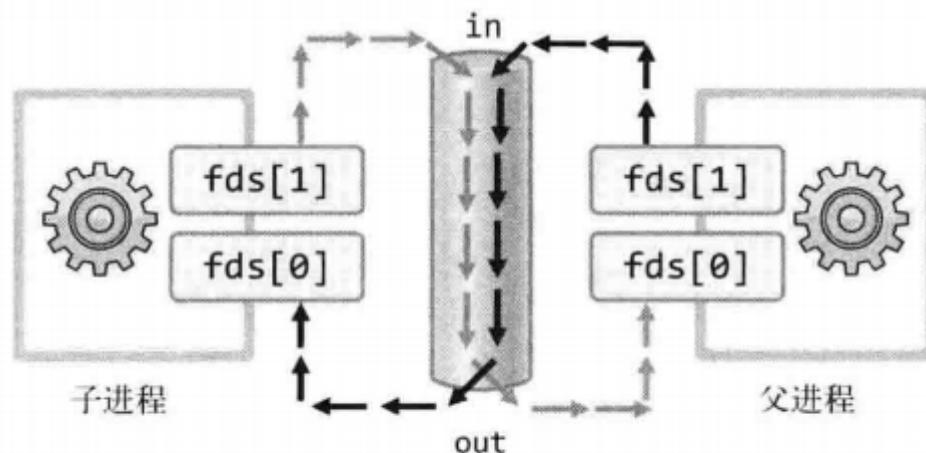


图11-3 双向通信模型1

下面是双向通信的示例：

- [pipe2.c](#)

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #define BUF_SIZE 30
4
5 int main(int argc, char *argv[])
6 {
7     int fds[2];
8     char str1[] = "Who are you?";
9     char str2[] = "Thank you for your message";
10    char buf[BUF_SIZE];
11    pid_t pid;
12
13    pipe(fds);

```

```

14     pid = fork();
15     if (pid == 0)
16     {
17         write(fds[1], str1, sizeof(str1));
18         sleep(2);
19         read(fds[0], buf, BUF_SIZE);
20         printf("Child proc output: %s \n", buf);
21     }
22     else
23     {
24         read(fds[0], buf, BUF_SIZE);
25         printf("Parent proc output: %s \n", buf);
26         write(fds[1], str2, sizeof(str2));
27         sleep(3);
28     }
29     return 0;
30 }
31

```

编译运行：

```

1 | gcc pipe2.c -o pipe2
2 | ./pipe2

```

结果：

```

1 | Parent proc output: Who are you?
2 | Child proc output: Thank you for your message

```

运行结果是正确的，但是如果注释掉第18行的代码，就会出现问题，导致一直等待下去。因为数据进入管道后变成了无主数据。也就是通过 `read` 函数先读取数据的进程将得到数据，即使该进程将数据传到了管道。因为，注释第18行会产生问题。第19行，自己成将读回自己在第 17 行向管道发送的数据。结果父进程调用 `read` 函数后，无限期等待数据进入管道。

当一个管道不满足需求时，就需要创建两个管道，各自负责不同的数据流动，过程如下图所示：

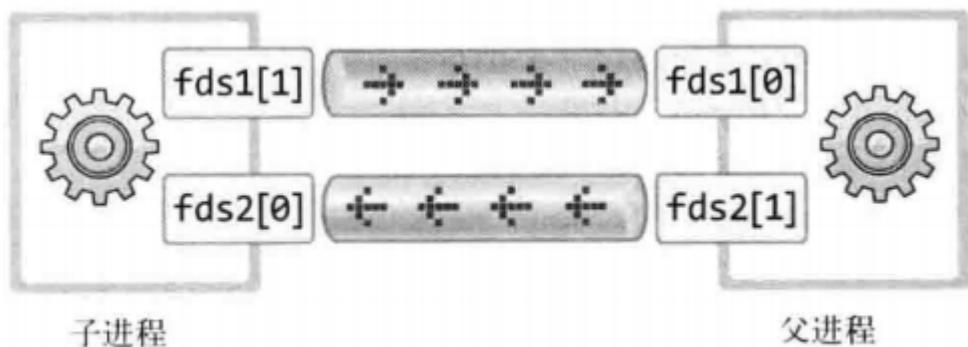


图11-4 双向通信模型2

下面采用上述模型改进 `pipe2.c`。

- [pipe3.c](#)

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #define BUF_SIZE 30
4
5 int main(int argc, char *argv[])
6 {
7     int fds1[2], fds2[2];
8     char str1[] = "Who are you?";
9     char str2[] = "Thank you for your message";
10    char buf[BUF_SIZE];
11    pid_t pid;
12
13    pipe(fds1), pipe(fds2);
14    pid = fork();
15    if (pid == 0)
16    {
17        write(fds1[1], str1, sizeof(str1));
18        read(fds2[0], buf, BUF_SIZE);
19        printf("Child proc output: %s \n", buf);
20    }
21    else
22    {
23        read(fds1[0], buf, BUF_SIZE);
24        printf("Parent proc output: %s \n", buf);
25        write(fds2[1], str2, sizeof(str2));
26    }
27    return 0;
28 }
```

上面通过创建两个管道实现了功能，此时，不需要额外再使用 sleep 函数。运行结果和上面一样。

## 11.2 运用进程间通信

### 11.2.1 保存消息的回声服务器

下面对第 10 章的 [echo\\_mperv.c](#) 进行改进，添加一个功能：

将回声客户端传输的字符串按序保存到文件中

实现该任务将创建一个新进程，从向客户端提供服务的进程读取字符串信息，下面是代码：

- [echo\\_storeserv.c](#)

编译运行：

```

1 gcc echo_storeserv.c -o serv
2 ./serv 9190
```

此服务端配合第 10 章的客户端 [echo\\_mpclient.c](#) 使用，运行结果如下图：

```

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch11 on git:master x [11:15:32] C:130
$ ./serv 9190
zsh: 没有那个文件或目录: ./serv
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch11 on git:master x [11:17:56] C:127
$ ./client 127.0.0.1 9190
Message from server: 1
2
echo_storeserv.c pipe1.c pipe2.c pipe3 README.md
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch11 on git:master x [11:17:58]
$ ls
Message from server: 4
echo_storeserv.c pipe1.c pipe2.c pipe3 README.md
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch11 on git:master x [11:18:35]
$ gcc echo_storeserv.c -o serv
Message from server: 5
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch11 on git:master x [11:18:41]
$ ./serv 9190
new client connected...
removed proc id: 17227
client disconnected...
removed proc id: 17230
Message from server: 6
0
Message from server: 7
q
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch10 on git:master x [11:18:56]

```

```

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch11 on git:master x [11:19:27]
$ cat echomsg.txt
1
2
3
4
5
6
7
8
9
0
4
Message from server: 4
5
Message from server: 5
6
Message from server: 6
7
Message from server: 7
8
Message from server: 8
9
Message from server: 9

```

从图上可以看出，服务端已经生成了文件，把客户端的消息保存下来，只保存了10次消息。

## 11.3 习题

以下答案仅代表本人个人观点，可能不是正确答案。

### 1. 什么是进程间通信？分别从概念和内存的角度进行说明。

答：进程间通信意味着两个不同的进程间可以交换数据。从内存上来说，就是两个进程可以访问同一个内存区域，然后通过这个内存区域数据的变化来进行通信。

### 2. 进程间通信需要特殊的 IPC 机制，这是由于操作系统提供的。进程间通信时为何需要操作系统的帮助？

答：为了进行进程间通信，需要管道的帮助，但是管道不是进程的资源，它属于从操作系统，所以，两个进程通过操作系统提供的内存空间进行通信。

### 3. 「管道」是典型的 IPC 技法。关于管道，请回答以下问题：

#### 1. 管道是进程间交换数据的路径。如何创建此路径？由谁创建？

答：使用 pipe 函数进行创建，由操作系统创建。父进程调用该函数时将创建管道。

#### 2. 为了完成进程间通信。2 个进程要同时连接管道。那2 个进程如何连接到同一管道？

答：数组中有两个文件描述符，父子进程调用相关函数时，通过 fork 函数，把 1 个文件描述符传递给子进程。

#### 3. 管道允许 2 个进程间的双向通信。双向通信中需要注意哪些内容？

答：向管道传输数据时，先读的进程会把数据取走。简言之，就是数据进入管道候会变成无主数据，所以有时候为了防止错误，需要多个管道来进程通信。

## 第 12 章 I/O 复用

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

### 12.1 基于 I/O 复用的服务器端

#### 12.1.1 多进程服务端的缺点和解决方法

为了构建并发服务器，只要有客户端连接请求就会创建新进程。这的确是实际操作中采用的一种方案，但并非十全十美，因为创建进程要付出很大的代价。这需要大量的运算和内存空间，由于每个进程都具有独立的内存空间，所以相互间的数据交换也要采用相对复杂的方法（IPC 属于相对复杂的通信方法）

I/O 复用技术可以解决这个问题。

#### 12.1.2 理解复用

「复用」在电子及通信工程领域很常见，向这些领域的专家询问其概念，可能会得到如下答复：

在 1 个通信频道中传递多个数据（信号）的技术

「复用」的含义：

为了提高物理设备的效率，只用最少的物理要素传递最多数据时使用的技术

上述两种方法的内容完全一致。可以用纸电话模型做一个类比：



图12-1 3方对话纸杯电话系统

上图是一个纸杯电话系统，为了使得三人同时通话，说话时要同时对着两个纸杯，接听时也需要耳朵同时对准两个纸杯。为了完成 3 人通话，可以进行如下图的改进：



图12-2 纸杯电话系统中引入复用技术

如图做出改进，就是引入了复用技术。

复用技术的优点：

- 减少连线长度
- 减少纸杯个数

即使减少了连线和纸杯的量仍然可以进行三人同时说话，但是如果碰到以下情况：

「好像不能同时说话？」

实际上，因为是在进行对话，所以很少发生同时说话的情况。也就是说，上述系统采用的是「时分复用」技术。因为说话人声频率不同，即使在同时说话也能进行一定程度上的区分（杂音也随之增多）。因此，也可以说是「频分复用技术」。

### 12.1.3 复用技术在服务器端的应用

纸杯电话系统引入复用技术之后可以减少纸杯数量和连线长度。服务器端引入复用技术可以减少所需进程数。下图是多进程服务端的模型：

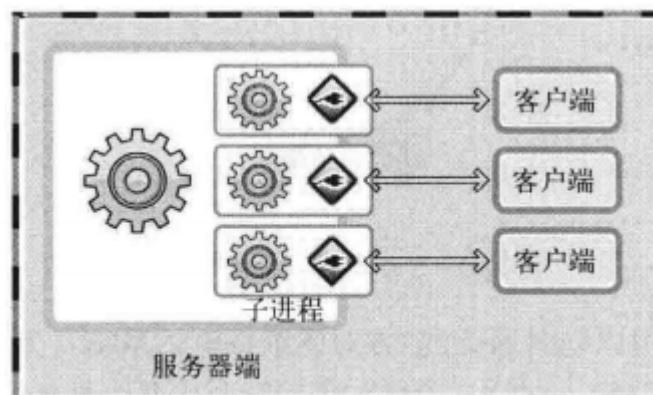


图12-3 多进程服务器端模型

下图是引入复用技术之后的模型：

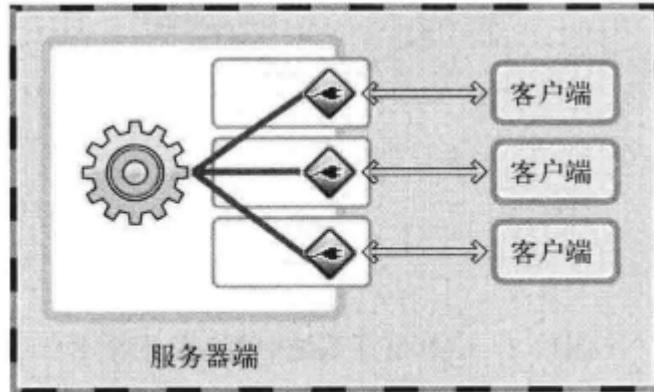


图12-4 I/O复用服务器端模型

从图上可以看出，引入复用技术之后，可以减少进程数。重要的是，无论连接多少客户端，提供服务的进程只有一个。

## 12.2 理解 select 函数并实现服务端

`select` 函数是最具代表性的实现复用服务器的方法。在 Windows 平台下也有同名函数，所以具有很好的移植性。

### 12.2.1 select 函数的功能和调用顺序

使用 `select` 函数时可以将多个文件描述符集中到一起统一监视，项目如下：

- 是否存在套接字接收数据？
- 无需阻塞传输数据的套接字有哪些？
- 哪些套接字发生了异常？

术语：「事件」。当发生监视项对应情况时，称「发生了事件」。

`select` 函数的使用方法与一般函数的区别并不大，更准确的说，他很难使用。但是为了实现 I/O 复用服务器端，我们应该掌握 `select` 函数，并运用于套接字编程当中。认为「`select` 函数是 I/O 复用的全部内容」也并不为过。

`select` 函数的调用过程如下图所示：

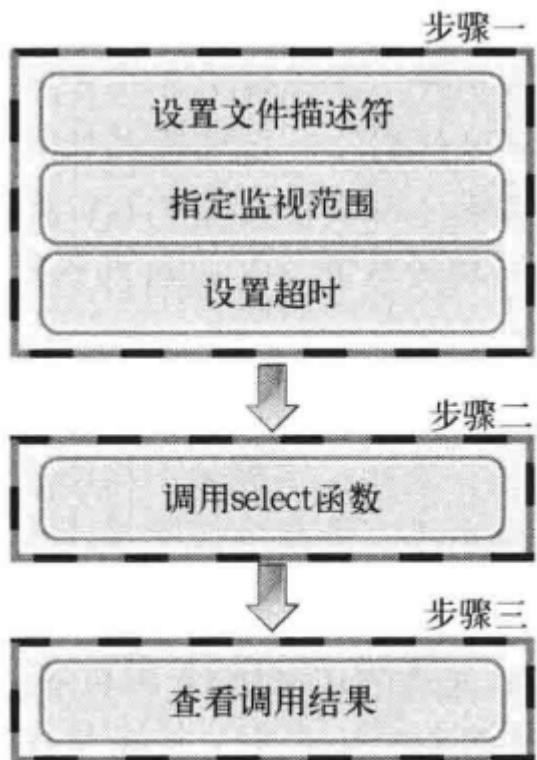


图12-5 select函数调用过程

### 12.2.2 设置文件描述符

利用 select 函数可以同时监视多个文件描述符。当然，监视文件描述符可以视为监视套接字。此时首先需要将要监视的文件描述符集中在一起。集中时也要按照监视项（接收、传输、异常）进行区分，即按照上述 3 种监视项分成 3 类。

利用 fd\_set 数组变量执行此操作，如图所示，该数组是存有 0 和 1 的位数组。



图12-6 fd\_set结构体

图中最左端的位表示文件描述符 0（所在位置）。如果该位设置为 1，则表示该文件描述符是监视对象。那么图中哪些文件描述符是监视对象呢？很明显，是描述符 1 和 3。在 fd\_set 变量中注册或更改值的操作都由下列宏完成。

- `FD_ZERO(fd_set *fdset)`：将 fd\_set 变量所指的位全部初始化成 0
- `FD_SET(int fd, fd_set *fdset)`：在参数 fdset 指向的变量中注册文件描述符 fd 的信息
- `FD_CLR(int fd, fd_set *fdset)`：从参数 fdset 指向的变量中清除文件描述符 fd 的信息
- `FD_ISSET(int fd, fd_set *fdset)`：若参数 fdset 指向的变量中包含文件描述符 fd 的信息，则返回「真」

上述函数中，`FD_ISSET` 用于验证 `select` 函数的调用结果，通过下图解释这些函数的功能：

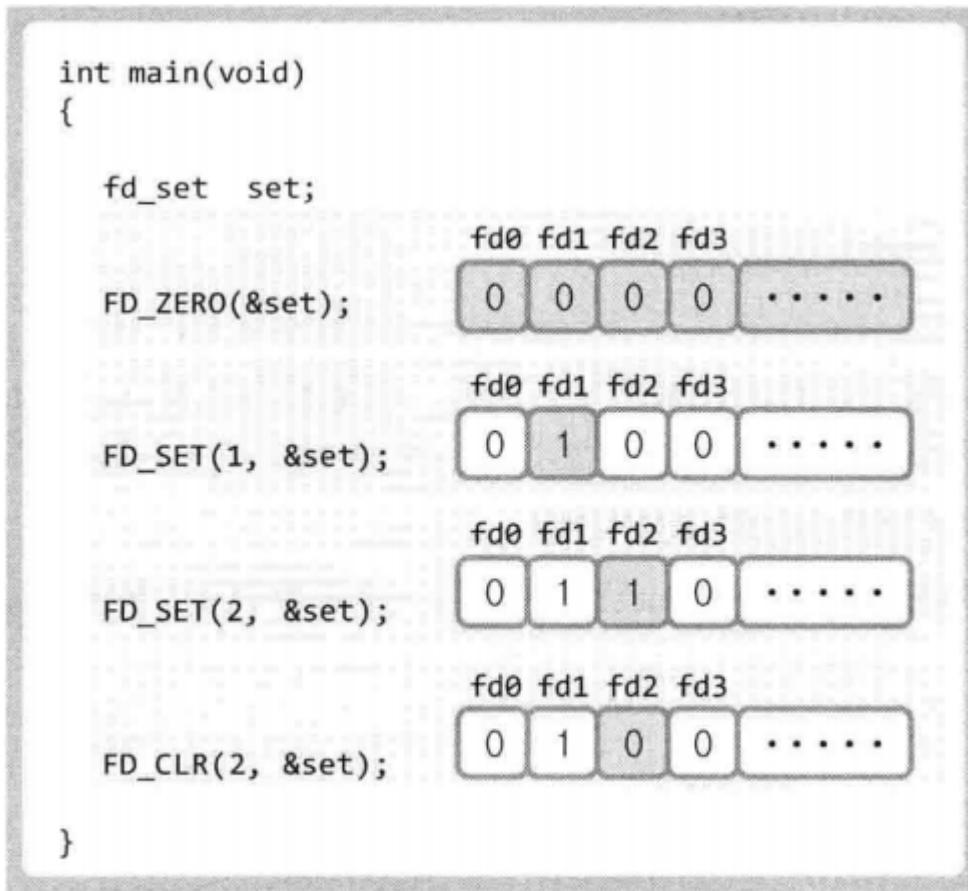


图12-7 fd\_set相关函数的功能

### 12.2.3 设置检查（监视）范围及超时

下面是 select 函数的定义：

```

1 #include <sys/select.h>
2 #include <sys/time.h>
3
4 int select(int maxfd, fd_set *readset, fd_set *writeset,
5            fd_set *exceptset, const struct timeval *timeout);
6 /*
7 成功时返回大于 0 的值，失败时返回 -1
8 maxfd: 监视对象文件描述符数量
9 readset: 将所有关注「是否存在待读取数据」的文件描述符注册到 fd_set 型变量，并传递其地址值。
10 writeset: 将所有关注「是否可传输无阻塞数据」的文件描述符注册到 fd_set 型变量，并传递其地址值。
11 exceptset: 将所有关注「是否发生异常」的文件描述符注册到 fd_set 型变量，并传递其地址值。
12 timeout: 调用 select 函数后，为防止陷入无限阻塞的状态，传递超时(timeout)信息
13 返回值：发生错误时返回 -1，超时时返回0。因发生关注的时间返回时，返回大于0的值，该值是发生事件的文件描
述符数。
14 */

```

如上所述，select 函数用来验证 3 种监视的变化情况，根据监视项声明 3 个 fd\_set 型变量，分别向其注册文件描述符信息，并把变量的地址值传递到上述函数的第二到第四个参数。但在此之前（调用 select 函数之前）需要决定下面两件事：

1. 文件描述符的监视（检查）范围是？

## 2. 如何设定 select 函数的超时时间？

第一，文件描述符的监视范围和 select 的第一个参数有关。实际上，select 函数要求通过第一个参数传递监视对象文件描述符的数量。因此，需要得到注册在 fd\_set 变量中的文件描述符数。但每次新建文件描述符时，其值就会增加 1，故只需将最大的文件描述符值加 1 再传递给 select 函数即可。加 1 是因为文件描述符的值是从 0 开始的。

第二，select 函数的超时时间与 select 函数的最后一个参数有关，其中 timeval 结构体定义如下：

```
1 struct timeval
2 {
3     long tv_sec;
4     long tv_usec;
5 };
```

本来 select 函数只有在监视文件描述符发生变化时才返回。如果未发生变化，就会进入阻塞状态。指定超时时间就是为了防止这种情况的发生。通过上述结构体变量，将秒数填入 tv\_sec 的成员，将微妙数填入 tv\_usec 的成员，然后将结构体的地址值传递到 select 函数的最后一个参数。此时，即使文件描述符未发生变化，只要过了指定时间，也可以从函数中返回。不过这种情况下，select 函数返回 0。因此，可以通过返回值了解原因。如果不向设置超时，则传递 NULL 参数。

### 12.2.4 调用 select 函数查看结果

select 返回正整数时，怎样获知哪些文件描述符发生了变化？向 select 函数的第二到第四个参数传递的 fd\_set 变量中将产生如图所示的变化：

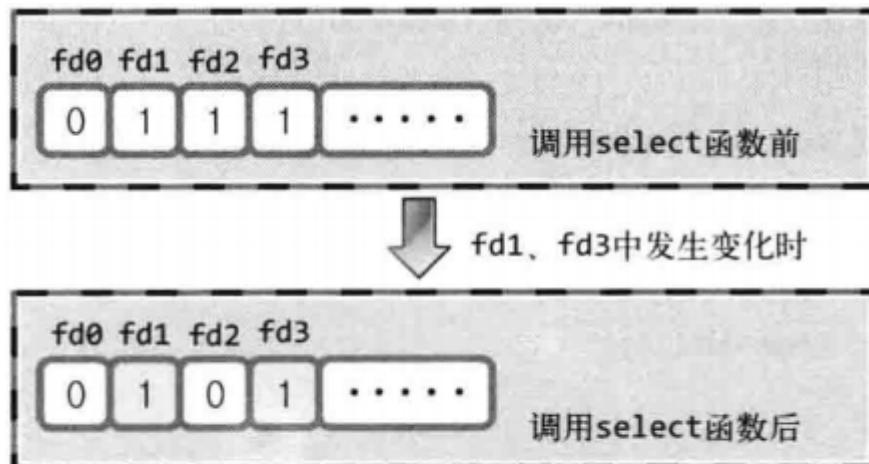


图12-8 fd\_set变量的变化

由图可知，select 函数调用完成候，向其传递的 fd\_set 变量将发生变化。原来为 1 的所有位将变成 0，但是发生了变化的文件描述符除外。因此，可以认为值仍为 1 的位置上的文件描述符发生了变化。

### 12.2.5 select 函数调用示例

下面是一个 select 函数的例子：

- [select.c](#)

编译运行：

```
1 gcc select.c -o select  
2 ./select
```

结果:

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch12 on git:master x [  
14:47:26] C:130  
$ ./select  
hello  
message from console: hello  
how are you  
message from console: how are you  
Time-out!  
^C
```

可以看出，如果运行后在标准输入流输入数据，就会在标准输出流输出数据，但是如果 5 秒没有输入数据，就提示超时。

## 12.2.6 实现 I/O 复用服务器端

下面通过 select 函数实现 I/O 复用服务器端。下面是基于 I/O 复用的回声服务器端。

- [echo\\_selectserv.c](#)

编译运行:

```
1 gcc echo_selectserv.c -o selserv  
2 ./selserv 9190
```

结果:

```
./selserv 9190  
$ ls  
echo_selectserv.c README.md select.c selserv  
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch12 on git:master x [16:05:47]  
$ gcc echo_selectserv.c -o selserv  
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch12 on git:master x [16:05:53]  
$ ls  
echo_selectserv.c README.md select.c selserv  
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch12 on git:master x [16:05:53]  
$ ./selserv 9190  
Connected client: 4  
Connected client: 5  
closed client: 5  
closed client: 4  
^C  
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch12 on git:master x [16:06:20] C:130  
$ ./client 127.0.0.1 9190  
Connected.....  
Input message(Q to quit): 123  
Message from server: 123  
Input message(Q to quit): hello  
Message from server: hello  
Input message(Q to quit): 1  
Message from server: 1  
Input message(Q to quit): 2  
Message from server: 2  
Input message(Q to quit):
```

从图上可以看出，虽然只用了一个进程，但是却实现了可以和多个客户端进行通信，这都是利用了 select 的特点。

## 12.3 基于 Windows 的实现

暂略

## 12.4 习题

以下答案仅代表本人个人观点，可能不是正确答案。

1. 请解释复用技术的通用含义，并说明何为 I/O 复用。

答：通用含义：在 1 个通信频道中传递多个数据（信号）的技术。IO 复用就是进程预先告诉内核需要监视的 IO 条件，使得内核一旦发现进程指定的一个或多个 IO 条件就绪，就通过进程处理，从而不会在单个 IO 上阻塞了。

参考文章：[Linux 网络编程-IO 复用技术](#)

## 2. 多进程并发服务器的缺点有哪些？如何在 I/O 复用服务器中弥补？

答：多进程需要进行大量的运算和大量的内存空间。在 I/O 复用服务器中通过 select 函数监视文件描述符，通过判断变化的文件描述符，来得知变化的套接字是哪个，从而实时应答来自多个客户端的请求。

## 3. 复用服务器端需要 select 函数。下列关于 select 函数使用方法的描述错误的是？

答：以下加粗的为正确的描述。

1. 调用 select 函数前需要集中 I/O 监视对象的文件描述符
  2. 若已通过 **select** 函数注册为监视对象，则后续调用 **select** 函数时无需重复注册
  3. 复用服务器端同一时间只能服务于 1 个客户端，因此，需要服务的客户端接入服务器端后只能等待
  4. 与多线程服务端不同，基于 **select** 的复用服务器只需要 1 个进程。因此，可以减少因创建多进程产生的服务器端的负担。
4. **select** 函数的观察对象中应包含服务端套接字（监听套接字），那么应将其包含到哪一类监听对象集合？请说明原因。

答：应该包含到「是否存在待读取数据」，因为服务器端需要查看套接字中有没有可以读取的数据。

# 第 13 章 多种 I/O 函数

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

## 13.1 send & recv 函数

### 13.1.1 Linux 中的 send & recv

首先看 send 函数定义：

```
1 #include <sys/socket.h>
2 ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
3 /*
4 成功时返回发送的字节数，失败时返回 -1
5 sockfd：表示与数据传输对象的连接的套接字和文件描述符
6 buf：保存带传输数据的缓冲地址值
7 nbytes：待传输字节数
8 flags：传输数据时指定的可选项信息
9 */
```

下面是 recv 函数的定义：

```

1 #include <sys/socket.h>
2 ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
3 /*
4 成功时返回接收的字节数（收到 EOF 返回 0），失败时返回 -1
5 sockfd: 表示数据接受对象的连接的套接字文件描述符
6 buf: 保存接受数据的缓冲地址值
7 nbytes: 可接收的最大字节数
8 flags: 接收数据时指定的可选项参数
9 */

```

send 和 recv 函数都是最后一个参数是收发数据的可选项，该选项可以用位或（bit OR）运算符（| 运算符）同时传递多个信息。

send & recv 函数的可选项意义：

可选项 (Option)	含义	send	recv
MSG_OOB	用于传输带外数据 (Out-of-band data)	O	O
MSG_PEEK	验证输入缓冲中是否存在接受的数据	X	O
MSG_DONTROUTE	数据传输过程中不参照本地路由 (Routing) 表，在本地 (Local) 网络中寻找目的地	O	X
MSG_DONTWAIT	调用 I/O 函数时不阻塞，用于使用非阻塞 (Non-blocking) I/O	O	O
MSG_WAITALL	防止函数返回，直到接收到全部请求的字节数	X	O

### 13.1.2 MSG\_OOB：发送紧急消息

MSG\_OOB 可选项用于创建特殊发送方法和通道以发送紧急消息。下面为 MSG\_OOB 的示例代码：

- [oob\\_recv.c](#)
- [oob\\_send.c](#)

编译运行：

```

1 gcc oob_send.c -o send
2 gcc oob_recv.c -o recv

```

运行结果：

```

11:45:16] $ gcc oob_send.c -o send
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13 on git:master x [11:47:27]
$ gcc oob_recv.c -o recv
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13 on git:master x [11:47:35]
$ ./recv
Usage : ./recv <port>
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13 on git:master x [11:48:30] C:1
$ ./recv
Urgent message: 0
123
56789
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13 on git:master x [11:48:36] C:1
$ ./recv 9190
Urgent message: 4
Urgent message: 0
123
56789

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13 on git:master x [11:49:02]
$ 

```

```

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13 on git:master x [11:58:58]
$ ./recv 9190
Urgent message: 4
Urgent message: 0
123
56789

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13 on git:master x [11:59:14]
$ ./recv 9190
123456789

```

从运行结果可以看出，`send` 是客户端，`recv` 是服务端，客户端给服务端发送消息，服务端接收完消息之后显示出来。可以从图中看出，每次运行的效果，并不是一样的。

代码中关于：

```
1 | fcntl(recv_sock, F_SETOWN, getpid());
```

的意思是：

文件描述符 `recv_sock` 指向的套接字引发的 `SIGURG` 信号处理进程变为 `getpid` 函数返回值用作 ID 进程。

上述描述中的「处理 `SIGURG` 信号」指的是「调用 `SIGURG` 信号处理函数」。但是之前讲过，多个进程可以拥有 1 个套接字的文件描述符。例如，通过调用 `fork` 函数创建子进程并同时复制文件描述符。此时如果发生 `SIGURG` 信号，应该调用哪个进程的信号处理函数呢？可以肯定的是，不会调用所有进程的信号处理函数。因此，处理 `SIGURG` 信号时必须指定处理信号所用的进程，而 `getpid` 返回的是调用此函数的进程 ID。上述调用语句指当前为处理 `SIGURG` 信号的主体。

输出结果，可能出乎意料：

通过 `MSG_OOB` 可选项传递数据时只返回 1 个字节，而且也不快

的确，通过 `MSG_OOB` 并不会加快传输速度，而通过信号处理函数 `urg_handler` 也只能读取一个字节。剩余数据只能通过未设置 `MSG_OOB` 可选项的普通输入函数读取。因为 TCP 不存在真正意义上的「外带数据」。实际上，`MSG_OOB` 中的 `OOB` 指的是 `Out-of-band`，而「外带数据」的含义是：

通过完全不同的通信路径传输的数据

即真正意义上的 `Out-of-band` 需要通过单独的通信路径高速传输数据，但是 TCP 不另外提供，只利用 TCP 的紧急模式（`Urgent mode`）进行传输。

### 13.1.3 紧急模式工作原理

MSG\_OOB 的真正意义在于督促数据接收对象尽快处理数据。这是紧急模式的全部内容，而 TCP 「保持传输顺序」的传输特性依然成立。TCP 的紧急消息无法保证及时到达，但是可以要求急救。下面是 MSG\_OOB 可选项状态下的数据传输过程，如图：

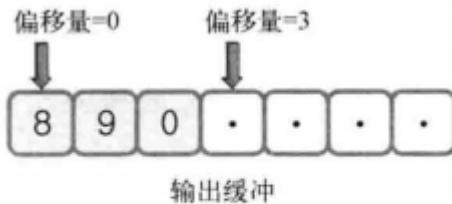


图13-1 紧急消息传输阶段的输出缓冲

上面是：

```
1 | send(sock, "890", strlen("890"), MSG_OOB);
```

图上是调用这个函数的缓冲状态。如果缓冲最左端的位置视作偏移量 0。字符 0 保存于偏移量 2 的位置。另外，字符 0 右侧偏移量为 3 的位置存有紧急指针（Urgent Pointer）。紧急指针指向紧急消息的下一个位置（偏移量加一），同时向对方主机传递一下信息：

紧急指针指向的偏移量为 3 之前的部分就是紧急消息。

也就是说，实际上只用了一个字节表示紧急消息。这一点可以通过图中用于传输数据的 TCP 数据包（段）的结构看得更清楚，如图：

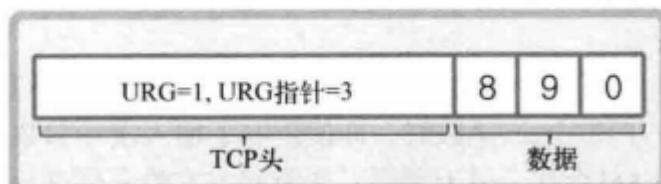


图13-2 设置URG的数据包

TCP 数据包实际包含更多信息。TCP 头部包含如下两种信息：

- URG=1：载有紧急消息的数据包
- URG指针：紧急指针位于偏移量为 3 的位置。

指定 MSG\_OOB 选项的数据包本身就是紧急数据包，并通过紧急指针表示紧急消息所在的位置。

紧急消息的意义在于督促消息处理，而非紧急传输形式受限的信息。

### 13.1.4 检查输入缓冲

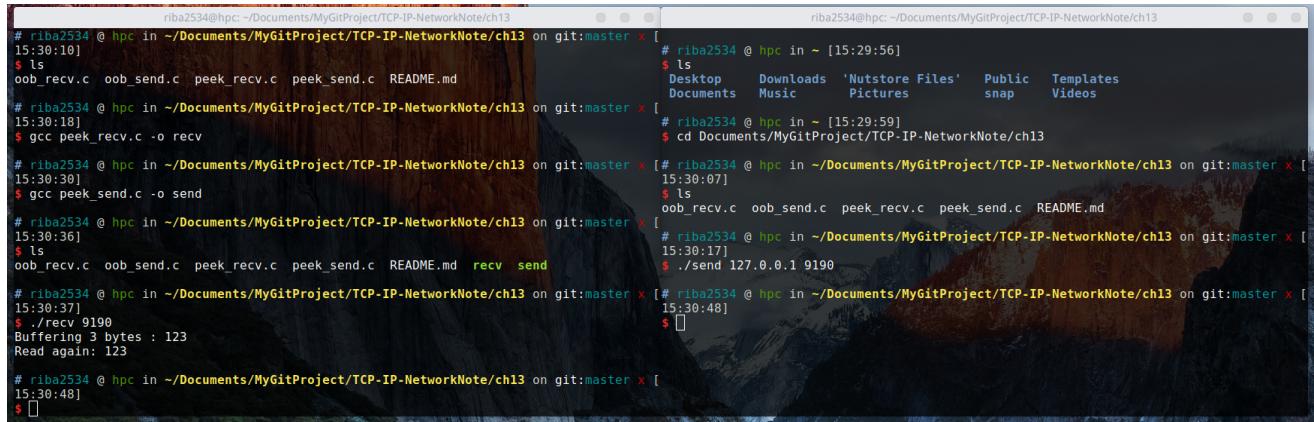
同时设置 MSG\_PEEK 选项和 MSG\_DONTWAIT 选项，以验证输入缓冲是否存在接收的数据。设置 MSG\_PEEK 选项并调用 recv 函数时，即使读取了输入缓冲的数据也不会删除。因此，该选项通常与 MSG\_DONTWAIT 合作，用于调用以非阻塞方式验证待读数据存与否的函数。下面的示例是二者的含义：

- [peek\\_recv.c](#)
- [peek\\_send.c](#)

编译运行：

```
1 gcc peek_recv.c -o recv
2 gcc peek_send.c -o send
3 ./recv 9190
4 ./send 127.0.0.1 9190
```

结果：



```
rib2534@hpc:~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13$ gcc peek_recv.c -o recv
rib2534@hpc:~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13$ gcc peek_send.c -o send
rib2534@hpc:~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13$ ./recv 9190
Buffering 3 bytes : 123
Read again: 123
rib2534@hpc:~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13$ ./send 127.0.0.1 9190
rib2534@hpc:~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13$ ls
Desktop Downloads 'Nutstore Files' Public Templates
Documents Music Pictures snap Videos
rib2534@hpc:~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13$ cd Documents/MyGitProject/TCP-IP-NetworkNote/ch13
rib2534@hpc:~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13$ ./recv 9190
Buffering 3 bytes : 123
Read again: 123
rib2534@hpc:~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13$ ./send 127.0.0.1 9190
rib2534@hpc:~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13$
```

可以通过结果验证，仅发送了一次的数据被读取了 2 次，因为第一次调用 recv 函数时设置了 MSG\_PEEK 可选项。

## 13.2 ready & writev 函数

### 13.2.1 使用 ready & writev 函数

ready & writev 函数的功能可概括如下：

对数据进行整合传输及发送的函数

也就是说，通过 writev 函数可以将分散保存在多个缓冲中的数据一并发送，通过 ready 函数可以由多个缓冲分别接收。因此，适用这 2 个函数可以减少 I/O 函数的调用次数。下面先介绍 writev 函数。

```
1 #include <sys/uio.h>
2 ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);
3 /*
4 成功时返回发送的字节数，失败时返回 -1
5 filedes：表示数据传输对象的套接字文件描述符。但该函数并不仅限于套接字，因此，可以像 read 一样向其传递文件或标准输出描述符。
6 iov：iovec 结构体数组的地址值，结构体 iovec 中包含待发送数据的位置和大小信息
7 iovcnt：向第二个参数传递数组长度
8 */
```

上述第二个参数中出现的数组 iovec 结构体的声明如下：

```
1 struct iovec
2 {
3     void *iov_base; //缓冲地址
4     size_t iov_len; //缓冲大小
5 };
```

下图是该函数的使用方法：

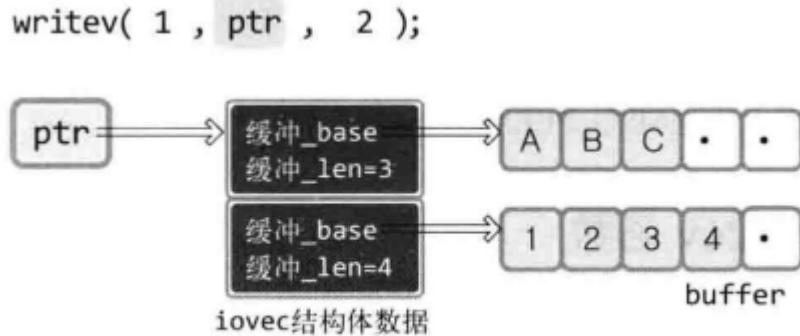


图13-4 write & iovec

`writev`的第一个参数，是文件描述符，因此向控制台输出数据，`ptr`是存有待发送数据信息的 `iovec` 数组指针。第三个参数为 2，因此，从 `ptr` 指向的地址开始，共浏览 2 个 `iovec` 结构体变量，发送这些指针指向的缓冲数据。

下面是 `writev` 函数的使用方法：

- [writev.c](#)

```
1 #include <stdio.h>
2 #include <sys/uio.h>
3 int main(int argc, char *argv[])
4 {
5     struct iovec vec[2];
6     char buf1[] = "ABCDEFG";
7     char buf2[] = "1234567";
8     int str_len;
9
10    vec[0].iov_base = buf1;
11    vec[0].iov_len = 3;
12    vec[1].iov_base = buf2;
13    vec[1].iov_len = 4;
14
15    str_len = writev(1, vec, 2);
16    puts("");
17    printf("Write bytes: %d \n", str_len);
18    return 0;
19 }
```

编译运行：

```
1 gcc writev.c -o writev
2 ./writevi
```

结果：

```
1 ABC1234
2 Write bytes: 7
```

下面介绍 `readv` 函数，功能和 `writev` 函数正好相反。函数为：

```
1 #include <sys/uio.h>
2 ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);
3 /*
4 成功时返回接收的字节数，失败时返回 -1
5 filedes：表示数据传输对象的套接字文件描述符。但该函数并不仅限于套接字，因此，可以像 read 一样向其传递文件或标准输出描述符。
6 iov：iovec 结构体数组的地址值，结构体 iovec 中包含待发送数据的位置和大小信息
7 iovcnt：向第二个参数传递数组长度
8 */
```

下面是示例代码：

- [readv.c](#)

```
1 #include <stdio.h>
2 #include <sys/uio.h>
3 #define BUF_SIZE 100
4
5 int main(int argc, char *argv[])
6 {
7     struct iovec vec[2];
8     char buf1[BUF_SIZE] = {
9         0,
10    };
11     char buf2[BUF_SIZE] = {
12         0,
13    };
14     int str_len;
15
16     vec[0].iov_base = buf1;
17     vec[0].iov_len = 5;
18     vec[1].iov_base = buf2;
19     vec[1].iov_len = BUF_SIZE;
20
21     str_len = readv(0, vec, 2);
22     printf("Read bytes: %d \n", str_len);
23     printf("First message: %s \n", buf1);
24     printf("Second message: %s \n", buf2);
25     return 0;
26 }
```

编译运行：

```
1 gcc readv.c -o rv
2 ./rv
```

运行结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch13 on git:master x [22:40:39]
$ ./rv
123456789
Read bytes: 10
First message: 12345
Second message: 6789
```

从图上可以看出，首先截取了长度为 5 的数据输出，然后再输出剩下的。

### 13.2.2 合理使用 readv & writev 函数

实际上，能使用该函数的所有情况都适用。例如，需要传输的数据分别位于不同缓冲（数组）时，需要多次调用 write 函数。此时可通过 1 次 writev 函数调用替代操作，当然会提高效率。同样，需要将输入缓冲中的数据读入不同位置时，可以不必多次调用 read 函数，而是利用 1 次 readv 函数就能大大提高效率。

其意义在于减少数据包个数。假设为了提高效率在服务器端明确禁用了 Nagle 算法。其实 writev 函数在不采用 Nagle 算法时更有价值，如图：

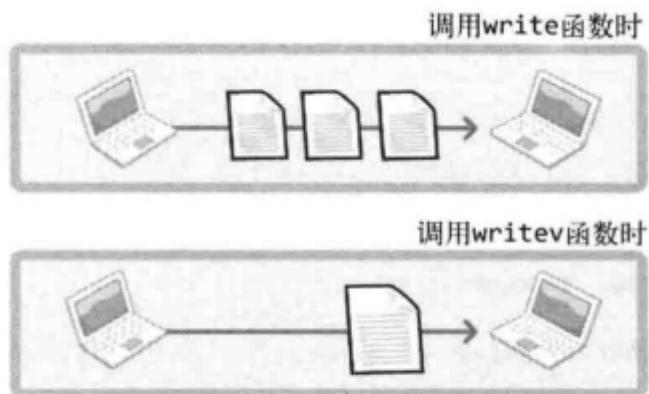


图13-5 Nagle算法关闭状态下的数据传输

## 13.3 基于 Windows 的实现

暂略

## 13.4 习题

以下答案仅代表本人个人观点，可能不是正确答案。

1. 下列关于 **MSG\_OOB** 可选项的说法错误的是？

答：以下加粗的字体代表说法正确。

1. MSG\_OOB 指传输 Out-of-band 数据，是通过其他路径高速传输数据

2. **MSG\_OOB** 指通过其他路径高速传输数据，因此 TCP 中设置该选项的数据先到达对方主机
  3. **设置 **MSG\_OOB**** 是数据先到达对方主机后，以普通数据的形式和顺序读取。也就是说，只是提高了传输速度，接收方无法识别这一点。
  4. **MSG\_OOB** 无法脱离 TCP 的默认数据传输方式，即使脱离了 **MSG\_OOB**，也会保持原有的传输顺序。该选项只用于要求接收方紧急处理。
2. 利用 **readv & writev** 函数收发数据有何优点？分别从函数调用次数和 I/O 缓冲的角度给出说明。

答：需要传输的数据分别位于不同缓冲（数组）时，需要多次调用 **write** 函数。此时可通过 1 次 **writev** 函数调用替代操作，当然会提高效率。同样，需要将输入缓冲中的数据读入不同位置时，可以不必多次调用 **read** 函数，而是利用 1 次 **readv** 函数就能大大提高效率。

3. 通过 **recv** 函数验证输入缓冲中是否存在数据时（确认后立即返回时），如何设置 **recv** 函数最后一个参数中的可选项？分别说明各可选项的含义。

答：使用 **MSG\_PEEK** 来验证输入缓冲中是否存在待接收的数据。各个可选项的意义参见上面对应章节的表格。

## 第 14 章 多播与广播

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

### 14.1 多播

多播（Multicast）方式的数据传输是基于 UDP 完成的。因此，与 UDP 服务器端/客户端的实现方式非常接近。区别在于，UDP 数据传输以单一目标进行，而多播数据同时传递到加入（注册）特定组的大量主机。换言之，采用多播方式时，可以同时向多个主机传递数据。

#### 14.1.1 多播的数据传输方式以及流量方面的优点

多播的数据传输特点可整理如下：

- 多播服务器端针对特定多播组，只发送 1 次数据。
- 即使只发送 1 次数据，但该组内的所有客户端都会接收数据
- 多播组数可以在 IP 地址范围内任意增加

多播组是 D 类 IP 地址（224.0.0.0~239.255.255.255），「加入多播组」可以理解为通过程序完成如下声明：

在 D 类 IP 地址中，我希望接收发往目标 239.234.218.234 的多播数据

多播是基于 UDP 完成的，也就是说，多播数据包的格式与 UDP 数据包相同。只是与一般的 UDP 数据包不同。向网络传递 1 个多播数据包时，路由器将复制该数据包并传递到多个主机。像这样，多播需要借助路由器完成。如图所示：

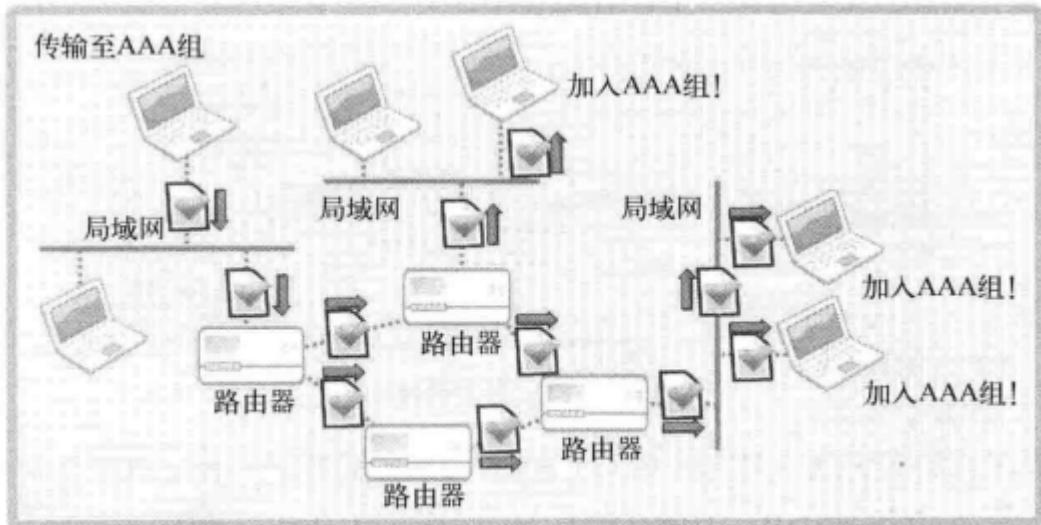


图14-1 多播路由

若通过 TCP 或 UDP 向 1000 个主机发送文件，则共需要传递 1000 次。但是此时如果用多播网络传输文件，则只需要发送一次。这时由 1000 台主机构成的网络中的路由器负责复制文件并传递到主机。就因为这种特性，多播主要用于「多媒体数据实时传输」。

另外，理论上可以完成多播通信，但是不少路由器并不支持多播，或即便支持也因网络拥堵问题故意阻断多播。因此，为了在不支持多播的路由器中完成多播通信，也会使用隧道（Tunneling）技术。

### 14.1.2 路由（Routing）和 TTL（Time to Live, 生存时间），以及加入组的办法

为了传递多播数据包，必须设置 TTL。TTL 是 Time to Live 的简写，是决定「数据包传递距离」的主要因素。TTL 用整数表示，并且每经过一个路由器就减一。TTL 变为 0 时，该数据包就无法再被传递，只能销毁。因此，TTL 的值设置过大将影响网络流量。当然，设置过小，也无法传递到目标。

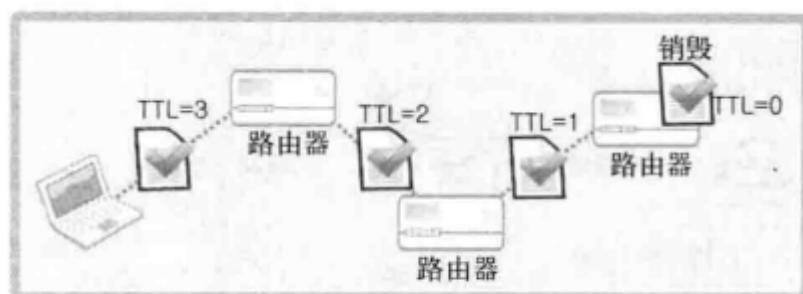


图14-2 TTL和多播路由

接下来是 TTL 的设置方法。TTL 是可以通过第九章的套接字可选项完成的。与设置 TTL 相关的协议层为 IPPROTO\_IP，选项名为 IP\_MULTICAST\_TTL。因此，可以用如下代码把 TTL 设置为 64

```

1 int send_sock;
2 int time_live = 64;
3 ...
4 send_sock=socket(PF_INET,SOCK_DGRAM,0);
5 setsockopt(send_sock,IPPROTO_IP,IP_MULTICAST_TTL,(void*)&time_live,sizeof(time_live));
6 ...

```

加入多播组也通过设置设置套接字可选项来完成。加入多播组相关的协议层为 IPPROTO\_IP，选项名为 IP\_ADD\_MEMBERSHIP。可通过如下代码加入多播组：

```
1 int recv_sock;
2 struct ip_mreq join_addr;
3 ...
4 recv_sock=socket(PF_INET,SOCK_DGRAM,0);
5 ...
6 join_addr.imr_multiaddr.s_addr="多播组地址信息";
7 join_addr.imr_interface.s_addr="加入多播组的主机地址信息";
8 setsockopt(recv_sock,IPPROTO_IP,IP_ADD_MEMBERSHIP,(void*)&join_addr,sizeof(time_live));
9 ...
```

下面是 ip\_mreq 结构体的定义：

```
1 struct ip_mreq
2 {
3     struct in_addr imr_multiaddr; //写入加入组的IP地址
4     struct in_addr imr_interface; //加入该组的套接字所属主机的IP地址
5 };
```

### 14.1.3 实现多播 Sender 和 Receiver

多播中用「发送者」（以下称为 Sender）和「接收者」（以下称为 Receiver）替代服务器端和客户端。顾名思义，此处的 Sender 是多播数据的发送主体，Receiver 是需要多播组加入过程的数据接收主体。下面是示例，示例的运行场景如下：

- Sender : 向 AAA 组广播（Broadcasting）文件中保存的新闻信息
- Receiver : 接收传递到 AAA 组的新闻信息。

下面是两个代码：

- [news\\_sender.c](#)
- [news\\_receiver.c](#)

编译运行：

```
1 gcc news_sender.c -o sender
2 gcc news_receiver.c -o receiver
3 ./sender 224.1.1.2 9190
4 ./receiver 224.1.1.2 9190
```

结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch14 on git:master x [12:27:08] C:130
$ ./receiver 224.1.1.2 9190
这是一段测试信息！
1
2
3
4
5
!!!!!!!!!!!!!!!
!!!!!!!!!!!
...

```

通过结果可以看出，使用 sender 多播信息，通过 receiver 接收广播，如果延迟运行 receiver 将无法接受之前发送的信息。

## 14.2 广播

广播（Broadcast）在「一次性向多个主机发送数据」这一点上与多播类似，但传输数据的范围有区别。多播即使在跨越不同网络的情况下，只要加入多播组就能接受数据。相反，广播只能向同一网络中的主机传输数据。

### 14.2.1 广播的理解和实现方法

广播是向同一网络中的所有主机传输数据的方法。与多播相同，广播也是通过 UDP 来完成的。根据传输数据时使用的IP地址形式，广播分为以下两种：

- 直接广播（Directed Broadcast）
- 本地广播（Local Broadcast）

二者在实现上的差别主要在于IP地址。直接广播的IP地址中除了网络地址外，其余主机地址全部设置成 1。例如，希望向网络地址 192.12.34 中的所有主机传输数据时，可以向 192.12.34.255 传输。换言之，可以采取直接广播的方式向特定区域内所有主机传输数据。

反之，本地广播中使用的IP地址限定为 255.255.255.255。例如，192.32.24 网络中的主机向 255.255.255.255 传输数据时，数据将传输到 192.32.24 网络中所有主机。

数据通信中使用的IP地址是与 UDP 示例的唯一区别。默认生成的套接字会阻止广播，因此，只需通过如下代码更改默认设置。

```
1 int send_sock;
2 int bcast;
3 ...
4 send_sock=socket(PF_INET,SOCK_DGRAM,0);
5 ...
6 setsockopt(send_sock,SOL_SOCKET,SO_BROADCAST,(void*)&bcast,sizeof(bcast));
7 ...
```

### 14.2.2 实现广播数据的 Sender 和 Receiver

下面是广播数据的 Sender 和 Receiver 的代码：

- [news\\_sender\\_brd.c](#)
- [news\\_receiver\\_brd.c](#)

编译运行：

```

1 gcc news_receiver_brd.c -o receiver
2 gcc news_sender_brd.c -o sender
3 ./sender 255.255.255.255 9190
4 ./receiver 9190

```

结果：

```

ribal2534@hpc: ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch14 [13:15:09]
$ ls
news_receiver_brd.c news_sender_brd.c news.txt
news_receiver.c news_sender.c README.md

ribal2534@hpc: ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch14 [13:15:10]
$ ls
news_receiver_brd.c news_sender_brd.c news.txt
news_receiver.c news_sender.c README.md
1
2
3
4
5
!!!!!!!!!!!!!!!!

ribal2534@hpc: ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch14 [13:15:14]
$ gcc news_receiver_brd.c -o receiver
# ribal2534@hpc: ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch14 [13:15:27]
$ gcc news_sender_brd.c -o sender
# ribal2534@hpc: ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch14 [13:15:37]
$ ./sender 255.255.255.255 9190
# ribal2534@hpc: ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch14 [13:17:47]
$ ./receiver 9190

```

## 14.3 基于 Windows 的实现

暂略

## 14.4 习题

以下答案仅代表本人个人观点，可能不是正确答案。

1. TTL 的含义是什么？请从路由器的角度说明较大的 TTL 值与较小的 TTL 值之间的区别及问题。

答：TTL 是决定「数据包传递距离」的主要因素。TTL 每经过一个路由器就减一。TTL 变为 0 时，数据包就无法再被传递，只能销毁。因此，TTL 设置过大会影响网络流量。当然，设置过小无法传递到目标。

2. 多播与广播的异同点是什么？请从数据通信的角度进行说明。

答：在「一次性向多个主机发送数据」这一点上与多播类似，但传输的数据范围有区别。多播即使在跨越不同网络的情况下，只要加入多播组就能接受数据。相反，广播只能向同意网络中的主机传输数据。

3. 下面关于多播的说法描述错误的是？

答：以下内容加粗的为描述正确

1. 多播是用来加入多播组的所有主机传输数据的协议
2. 主机连接到同一网络才能加入到多播组，也就是说，多播组无法跨越多个网络
- 3. 能够加入多播组的主机数并无限制，但只能有 1 个主机（Sender）向该组发送数据**
4. 多播时使用的套接字是 **UDP** 套接字，因为多播是基于 **UDP** 进行数据通信的。

4. 多播也对网络流量有利，请比较 TCP 交换方式解释其原因

答：TCP 是必须建立一对一的连接，如果要向 1000 个主机发送文件，就得传递 1000 次。但是此时用多播方式传输数据，就只需要发送一次。

5. 多播方式的数据通信需要 **MBone** 虚拟网络。换言之，**MBone** 是用于多播的网络，但它是虚拟网络。请解释此处的「虚拟网络」

答：可以理解为「通过网络中的特殊协议工作的软件概念上的网络」。也就是说，MBone 并非可以触及的物理网络。他是以物理网络为基础，通过软件方法实现的多播通信必备虚拟网络。

# 第 15 章 套接字和标准 I/O

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

## 15.1 标准 I/O 的优点

### 15.1.1 标准 I/O 函数的两个优点

下面是标准 I/O 函数的两个优点：

- 标准 I/O 函数具有良好的移植性
- 标准 I/O 函数可以利用缓冲提高性能

创建套接字时，操作系统会准备 I/O 缓冲。此缓冲在执行 TCP 协议时发挥着非常重要的作用。此时若使用标准 I/O 函数，将得到额外的缓冲支持。如下图：

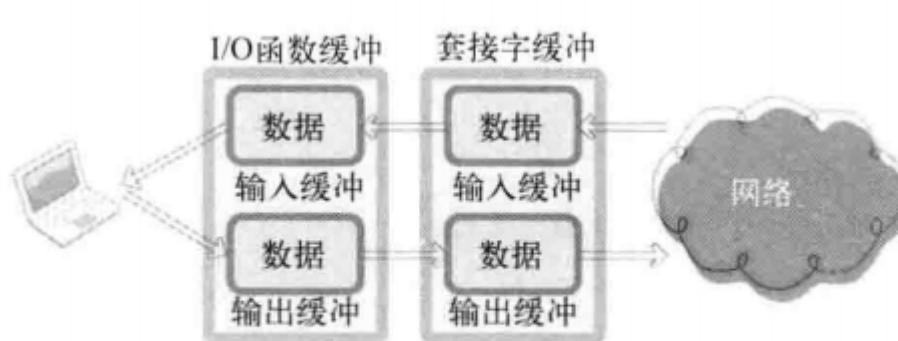


图15-1 缓冲的关系

假设使用 `fputs` 函数进行传输字符串「Hello」时，首先将数据传递到标准 I/O 缓冲，然后将数据移动到套接字输出缓冲，最后将字符串发送到对方主机。

设置缓冲的主要目的是为了提高性能。从以下两点可以说明性能的提高：

- 传输的数据量
- 数据向输出缓冲移动的次数。

比较 1 个字节的数据发送 10 次的情况和 10 个数据包发送 1 次的情况。发送数据时，数据包中含有头信息。头信与数据大小无关，是按照一定的格式填入的。假设头信息占 40 个字节，需要传输的数据量也存在较大区别：

- 1 个字节 10 次： $40 \times 10 = 400$  字节
- 10 个字节 1 次： $40 \times 1 = 40$  字节。

### 15.1.2 标准 I/O 函数和系统函数之间的性能对比

下面是利用系统函数的示例：

- [syscpy.c](#)

下面是使用标准 I/O 函数复制文件

- [stdcpy.c](#)

对于以上两个代码进行测试，明显基于标准 I/O 函数的代码跑的更快

### 15.1.3 标准 I/O 函数的几个缺点

标准 I/O 函数存在以下几个缺点：

- 不容易进行双向通信
- 有时可能频繁调用 `fflush` 函数
- 需要以 `FILE` 结构体指针的形式返回文件描述符。

## 15.2 使用标准 I/O 函数

### 15.2.1 利用 `fdopen` 函数转换为 `FILE` 结构体指针

函数原型如下：

```
1 #include <stdio.h>
2 FILE *fdopen(int fildes, const char *mode);
3 /*
4 成功时返回转换的 FILE 结构体指针，失败时返回 NULL
5 fildes : 需要转换的文件描述符
6 mode : 将要创建的 FILE 结构体指针的模式信息
7 */
```

以下为示例：

- [desto.c](#)

```
1 #include <stdio.h>
2 #include <fcntl.h>
3
4 int main()
5 {
6     FILE *fp;
7     int fd = open("data.dat", O_WRONLY | O_CREAT | O_TRUNC); // 创建文件并返回文件描述符
8     if (fd == -1)
9     {
10         fputs("file open error", stdout);
11         return -1;
12     }
13     fp = fdopen(fd, "w"); // 返回写模式的 FILE 指针
14     fputs("Network C programming \n", fp);
15     fclose(fp);
16     return 0;
17 }
```

编译运行：

```
1 gcc desto.c -o desto
2 ./desto
3 cat data.dat
```

运行结果：



```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch15 on git:master x [17:11:44]
$ ./desto

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch15 on git:master x [17:11:47]
$ cat data.dat
Network C programming

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch15 on git:master x [17:11:54]
$ 
```

文件描述符转换为 FILE 指针，并可以通过该指针调用标准 I/O 函数。

### 15.2.2 利用 fileno 函数转换为文件描述符

函数原型如下：

```
1 #include <stdio.h>
2 int fileno(FILE *stream);
3 /*
4 成功时返回文件描述符，失败时返回 -1
5 */
```

示例：

- [todes.c](#)

```
1 #include <stdio.h>
2 #include <fcntl.h>
3
4 int main()
5 {
6     FILE *fp;
7     int fd = open("data.dat", O_WRONLY | O_CREAT | O_TRUNC);
8     if (fd == -1)
9     {
10         fputs("file open error");
11         return -1;
12     }
13
14     printf("First file descriptor : %d \n", fd);
15     fp = fdopen(fd, "w"); //转成 file 指针
16     fputs("TCP/IP SOCKET PROGRAMMING \n", fp);
17     printf("Second file descriptor: %d \n", fileno(fp)); //转回文件描述符
18     fclose(fp);
19     return 0;
20 }
```

## 15.3 基于套接字的标准 I/O 函数使用

把第四章的回声客户端和回声服务端的内容改为基于标准 I/O 函数的数据交换形式。

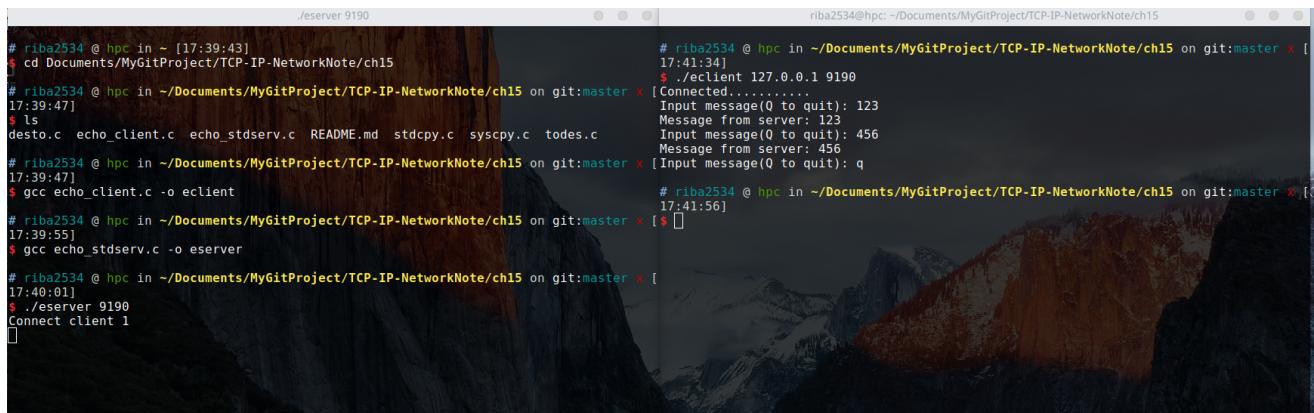
代码如下：

- [echo\\_client.c](#)
- [echo\\_stdserv.c](#)

编译运行：

```
1 | gcc echo_client.c -o eclient
2 | gcc echo_stdserv.c -o eserver
```

结果：



```
# riba2534 @ hpc in ~ [17:39:43]
$ cd Documents/MyGitProject/TCP-IP-NetworkNote/ch15
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch15 on git:master x [17:39:47]
$ ls
desto.c echo_client.c echo_stdserv.c README.md stdcpy.c syscpy.c todes.c
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch15 on git:master x [17:39:47]
$ gcc echo_client.c -o eclient
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch15 on git:master x [17:39:55]
$ ./eclient 127.0.0.1 9190
[Connected.....]
Input message(0 to quit): 123
Message from server: 123
Input message(0 to quit): 456
Message from server: 456
Input message(0 to quit): q
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch15 on git:master x [17:40:01]
$ ./eserver 9190
[ $ ]
Connect client 1
```

可以看出，运行结果和第四章相同，这是利用标准 I/O 实现的。

## 15.4 习题

以下答案仅代表本人个人观点，可能不是正确答案。

### 1. 请说明标准 I/O 的 2 个优点。他为何拥有这 2 个优点？

答：①具有很高的移植性②有良好的缓冲提高性能。因为这些函数是由 ANSI C 标准定义的。适合所有编程领域。

### 2. 利用标准 I/O 函数传输数据时，下面的说法是错误的：

调用 fputs 函数传输数据时，调用后应立即开始发送！

为何上述说法是错误的？为达到这种效果应该添加哪些处理过程？

答：只是传输到了缓冲中，应该利用 fflush 来刷新缓冲区。

## 第 16 章 关于 I/O 流分离的其他内容

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

## 16.1 分离 I/O 流

「分离 I/O 流」是一种常用表达。有 I/O 工具可区分二者，无论采用哪种方法，都可以认为是分离了 I/O 流。

### 16.1.1 2 次 I/O 流分离

之前有两种分离方法：

- 第一种是第 10 章的「TCP I/O 过程」分离。通过调用 `fork` 函数复制出一个文件描述符，以区分输入和输出中使用的文件描述符。虽然文件描述符本身不会根据输入和输出进行区分，但我们分开了 2 个文件描述符的用途，因此，这也属于「流」的分离。
- 第二种分离是在第 15 章。通过 2 次 `fdopen` 函数的调用，创建读模式 FILE 指针（FILE 结构体指针）和写模式 FILE 指针。换言之，我们分离了输入工具和输出工具，因此也可视为「流」的分离。下面是分离的理由。

### 16.1.2 分离「流」的好处

首先是第 10 章「流」的分离目的：

- 通过分开输入过程（代码）和输出过程降低实现难度
- 与输入无关的输出操作可以提高速度

下面是第 15 章「流」分离的目的：

- 为了将 FILE 指针按读模式和写模式加以区分
- 可以通过区分读写模式降低实现难度
- 通过区分 I/O 缓冲提高缓冲性能

### 16.1.3 「流」分离带来的 EOF 问题

第 7 章介绍过 EOF 的传递方法和半关闭的必要性。有一个语句：

```
1 | shutdown(sock,SHUT_WR);
```

当时说过调用 `shutdown` 函数的基于半关闭的 EOF 传递方法。第十章的 [echo\\_mpclient.c](#) 添加了半关闭的相关代码。但是还没有讲采用 `fdopen` 函数怎么半关闭。那么是否是通过 `fclose` 函数关闭流呢？我们先试试

下面是服务端和客户端码：

- [sep\\_clnt.c](#)
- [sep\\_serv.c](#)

编译运行：

```
1 | gcc sep_clnt.c -o clnt
2 | gcc sep_serv.c -o serv
3 | ./serv 9190
4 | ./clnt 127.0.0.1 9190
```

结果：

```

# riba2534 @ hpc in ~ [11:55:06]
$ cd Documents/MyGitProject/TCP-IP-NetworkNote/ch16
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [11:55:10]
$ ls
README.md sep_clnt.c sep_serv.c
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [11:55:11]
$ gcc sep_clnt.c -o clnt
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [11:55:23]
$ gcc sep_serv.c -o serv
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [11:55:29]
$ ./serv 9198
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [11:55:36]
$ clnt 127.0.0.1 9198
[FROM SERVER: Hi~ client!
 I love all of the world
 You are awesome!]

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [11:55:56]
$ 

```

从运行结果可以看出，服务端最终没有收到客户端发送的信息。那么这是什么原因呢？

原因是：服务端代码的 `fclose(writefp);` 这一句，完全关闭了套接字而不是半关闭。这才是这一章需要解决的问题。

## 16.2 文件描述符的复制和半关闭

### 16.2.1 终止「流」时无法半关闭原因

下面的图描述的是服务端代码中的两个FILE指针、文件描述符和套接字中的关系。

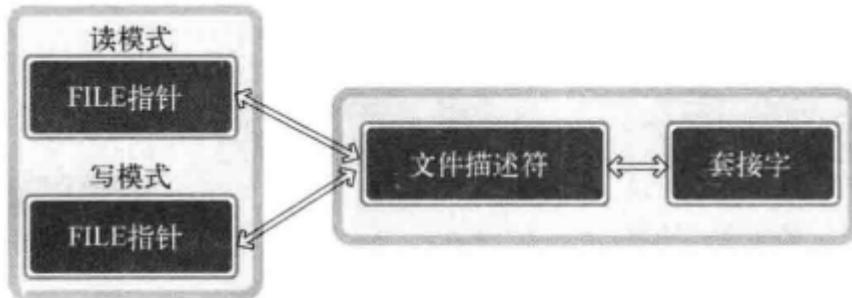


图16-1 FILE指针的关系

从图中可以看到，两个指针都是基于同一文件描述符创建的。因此，针对任何一个 FILE 指针调用 `fclose` 函数都会关闭文件描述符，如图所示：

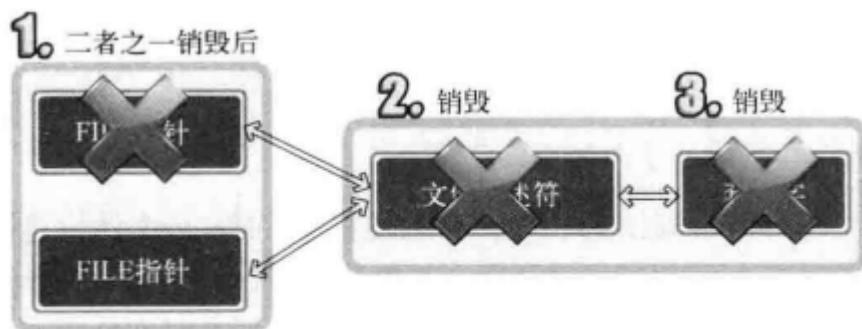


图16-2 调用fclose函数的结果

从图中看到，销毁套接字时再也无法进行数据交换。那如何进入可以进入但是无法输出的半关闭状态呢？如下图所示：

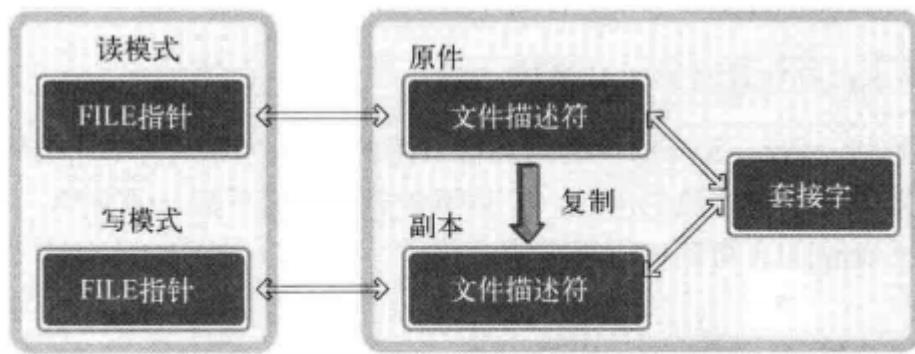


图16-3 半关闭模型1

只需要创建 FILE 指针前先复制文件描述符即可。复制后另外创建一个文件描述符，然后利用各自的文件描述符生成读模式的 FILE 指针和写模式的 FILE 指针。这就为半关闭创造了环境，因为套接字和文件描述符具有如下关系：

销毁所有文件描述符后才能销毁套接字

也就是说，针对写模式 FILE 指针调用 fclose 函数时，只能销毁与该 FILE 指针相关的文件描述符，无法销毁套接字，如下图：

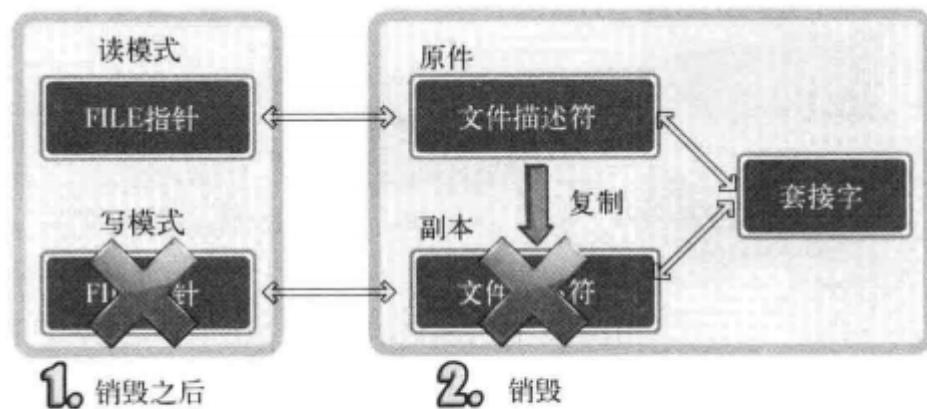


图16-4 半关闭模型2

那么调用 fclose 函数后还剩下 1 个文件描述符，因此没有销毁套接字。那此时的状态是否为半关闭状态？不是！只是准备好了进入半关闭状态，而不是已经进入了半关闭状态。仔细观察，还剩下一个文件描述符。而该文件描述符可以同时进行 I/O。因此，不但没有发送 EOF，而且仍然可以利用文件描述符进行输出。

## 16.2.2 复制文件描述符

与调用 fork 函数不同，调用 fork 函数将复制整个进程，此处讨论的是同一进程中完成对文件描述符的复制。如图：

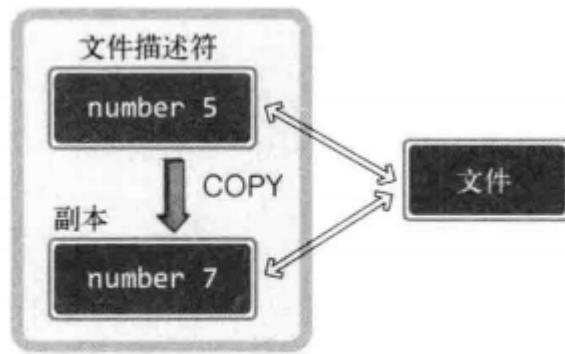


图16-5 文件描述符的复制

复制完成后，两个文件描述符都可以访问文件，但是编号不同。

### 16.2.3 dup 和 dup2

下面给出两个函数原型：

```

1 #include <unistd.h>
2 int dup(int fildes);
3 int dup2(int fildes, int fildes2);
4 /*
5 成功时返回复制的文件描述符，失败时返回 -1
6 fildes : 需要复制的文件描述符
7 fildes2 : 明确指定的文件描述符的整数值。
8 */

```

dup2 函数明确指定复制的文件描述符的整数值。向其传递大于 0 且小于进程能生成的最大文件描述符值时，该值将成为复制出的文件描述符值。下面是代码示例：

- [dup.c](#)

```

1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(int argc, char *argv[])
5 {
6     int cfd1, cfd2;
7     char str1[] = "Hi~ \n";
8     char str2[] = "It's nice day~ \n";
9
10    cfd1 = dup(1);           //复制文件描述符 1
11    cfd2 = dup2(cfd1, 7); //再次复制文件描述符,定为数值 7
12
13    printf("fd1=%d , fd2=%d \n", cfd1, cfd2);
14    write(cfd1, str1, sizeof(str1));
15    write(cfd2, str2, sizeof(str2));
16
17    close(cfd1);
18    close(cfd2); //终止复制的文件描述符，但是仍有一个文件描述符
19    write(1, str1, sizeof(str1));

```

```

20     close(1);
21     write(1, str2, sizeof(str2)); //无法完成输出
22     return 0;
23 }
24

```

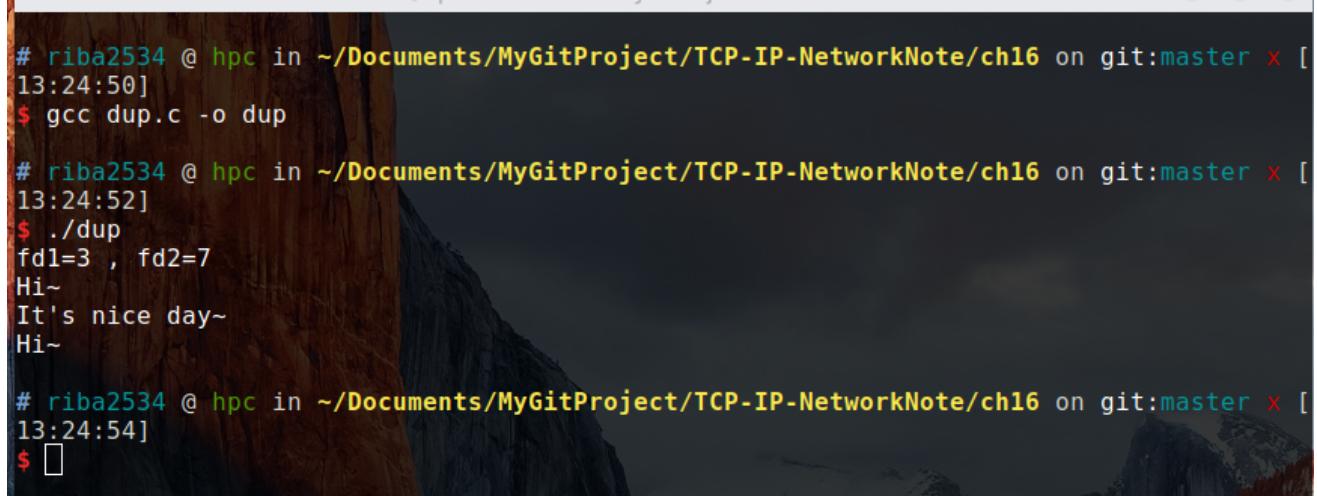
编译运行：

```

1 gcc dup.c -o dup
2 ./dup

```

结果：



```

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [13:24:50]
$ gcc dup.c -o dup
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [13:24:52]
$ ./dup
fd1=3 , fd2=7
Hi~
It's nice day~
Hi~

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [13:24:54]
$ 

```

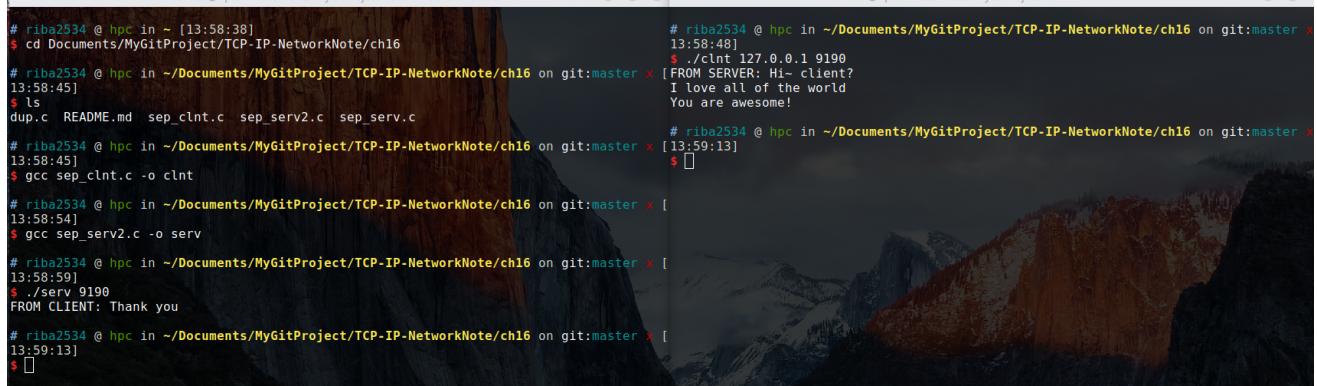
#### 16.2.4 复制文件描述符后「流」的分离

下面更改 [sep\\_clnt.c](#) 和 [sep\\_serv.c](#) 可以使得让它正常工作，正常工作是指通过服务器的半关闭状态接收客户端最后发送的字符串。

下面是代码：

- [sep\\_serv2.c](#)

这个代码可以与 [sep\\_clnt.c](#) 配合起来食用，编译过程和上面一样，运行结果为：



```

# riba2534 @ hpc in ~ [13:58:38]
$ cd Documents/MyGitProject/TCP-IP-NetworkNote/ch16
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [13:58:45]
$ ls
dup.c README.md sep_clnt.c sep_serv2.c sep_serv.c
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [13:58:45]
$ gcc sep_clnt.c -o clnt
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [13:58:54]
$ gcc sep_serv2.c -o serv
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [13:58:59]
$ ./serv 9190
FROM CLIENT: Thank you

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch16 on git:master x [13:59:13]
$ 

```

### 16.3 习题

以下答案仅代表本人个人观点，可能不是正确答案。

## 1. 下列关于 FILE 结构体指针和文件描述符的说法错误的是?

答: 以下加粗内容代表说法正确。

1. 与 FILE 结构体指针相同, 文件描述符也分输入描述符和输出描述符
2. 复制文件描述符时将生成相同值的描述符, 可以通过这 2 个描述符进行 I/O
3. **可以利用创建套接字时返回的文件描述符进行 I/O, 也可以不通过文件描述符, 直接通过 FILE 结构体指针完成**
4. **可以从文件描述符生成 FILE 结构体指针, 而且可以利用这种 FILE 结构体指针进行套接字 I/O**
5. 若文件描述符为读模式, 则基于该描述符生成的 FILE 结构体指针同样是读模式; 若文件描述符为写模式, 则基于该描述符生成的 FILE 结构体指针同样是写模式

## 2. EOF 的发送相关描述中错误的是?

答: 以下加粗内容代表说法正确。

1. 终止文件描述符时发送 EOF
2. **即使未完成终止文件描述符, 关闭输出流时也会发送 EOF**
3. 如果复制文件描述符, 则包括复制的文件描述符在内, 所有文件描述符都终止时才会发送 EOF
4. 即使复制文件描述符, 也可以通过调用 shutdown 函数进入半关闭状态并发送 EOF

# 第 17 章 优于 select 的 epoll

本章代码, 在[TCP-IP-NetworkNote](#)中可以找到。

## 17.1 epoll 理解及应用

select 复用方法由来已久, 因此, 利用该技术后, 无论如何优化程序性能也无法同时介入上百个客户端。这种 select 方式并不适合以 web 服务器端开发为主流的现代开发环境, 所以需要学习 Linux 环境下的 epoll

### 17.1.1 基于 select 的 I/O 复用技术速度慢的原因

第 12 章实现了基于 select 的 I/O 复用技术服务端, 其中有不合理的设计如下:

- 调用 select 函数后常见的针对所有文件描述符的循环语句
- 每次调用 select 函数时都需要向该函数传递监视对象信息

上述两点可以从[echo\\_selectserv.c](#)得到确认, 调用 select 函数后, 并不是把发生变化的文件描述符单独集中在一起, 而是通过作为监视对象的 fd\_set 变量的变化, 找出发生变化的文件描述符(54,56行), 因此无法避免针对所有监视对象的循环语句。而且, 作为监视对象的 fd\_set 会发生变化, 所以调用 select 函数前应该复制并保存原有信息, 并在每次调用 select 函数时传递新的监视对象信息。

select 性能上最大的弱点是: 每次传递监视对象信息, 准确的说, select 是监视套接字变化的函数。而套接字是操作系统管理的, 所以 select 函数要借助操作系统才能完成功能。select 函数的这一缺点可以通过如下方式弥补:

仅向操作系统传递一次监视对象, 监视范围或内容发生变化时只通知发生变化的事项

这样就无需每次调用 select 函数时都向操作系统传递监视对象信息, 但是前提操作系统支持这种处理方式。Linux 的支持方式是 epoll, Windows 的支持方式是 IOCP。

### 17.1.2 select 也有有点

select 的兼容性比较高, 这样就可以支持很多的操作系统, 不受平台的限制, 使用 select 函数满足以下两个条件:

- 服务器接入者少
- 程序应该具有兼容性

### 17.1.3 实现 epoll 时必要的函数和结构体

能够克服 select 函数缺点的 epoll 函数具有以下优点，这些优点正好与之前的 select 函数缺点相反。

- 无需编写以监视状态变化为目的的针对所有文件描述符的循环语句
- 调用对应于 select 函数的 epoll\_wait 函数时无需每次传递监视对象信息。

下面是 epoll 函数的功能：

- epoll\_create：创建保存 epoll 文件描述符的空间
- epoll\_ctl：向空间注册并注销文件描述符
- epoll\_wait：与 select 函数类似，等待文件描述符发生变化

select 函数中为了保存监视对象的文件描述符，直接声明了 fd\_set 变量，但 epoll 方式下的操作系统负责保存监视对象文件描述符，因此需要向操作系统请求创建保存文件描述符的空间，此时用的函数就是 epoll\_create。

此外，为了添加和删除监视对象文件描述符，select 方式中需要 FD\_SET、FD\_CLR 函数。但在 epoll 方式中，通过 epoll\_ctl 函数请求操作系统完成。最后，select 方式下调用 select 函数等待文件描述符的变化，而 epoll\_wait 调用 epoll\_wait 函数。还有，select 方式中通过 fd\_set 变量查看监视对象的状态变化，而 epoll 方式通过如下结构体 epoll\_event 将发生变化的文件描述符单独集中在一起。

```
1 struct epoll_event
2 {
3     __uint32_t events;
4     epoll_data_t data;
5 };
6 typedef union epoll_data {
7     void *ptr;
8     int fd;
9     __uint32_t u32;
10    __uint64_t u64;
11 } epoll_data_t;
12
```

声明足够大的 epoll\_event 结构体数组候，传递给 epoll\_wait 函数时，发生变化的文件描述符信息将被填入数组。因此，无需像 select 函数那样针对所有文件描述符进行循环。

### 17.1.4 epoll\_create

epoll 是从 Linux 的 2.5.44 版内核开始引入的。通过以下命令可以查看 Linux 内核版本：

```
1 cat /proc/sys/kernel/osrelease
```

下面是 epoll\_create 函数的原型：

```
1 #include <sys/epoll.h>
2 int epoll_create(int size);
3 /*
4 成功时返回 epoll 的文件描述符，失败时返回 -1
5 size: epoll 实例的大小
6 */
```

调用 `epoll_create` 函数时创建的文件描述符保存空间称为「`epoll` 例程」，但有些情况下名称不同，需要稍加注意。通过参数 `size` 传递的值决定 `epoll` 例程的大小，但该值只是向操作系统提出的建议。换言之，`size` 并不用来决定 `epoll` 的大小，而仅供操作系统参考。

Linux 2.6.8 之后的内核将完全传入 `epoll_create` 函数的 `size` 函数，因此内核会根据情况调整 `epoll` 例程大小。但是本书程序并没有忽略 `size`

`epoll_create` 函数创建的资源与套接字相同，也由操作系统管理。因此，该函数和创建套接字的情况相同，也会返回文件描述符，也就是说返回的文件描述符主要用于区分 `epoll` 例程。需要终止时，与其他文件描述符相同，也要调用 `close` 函数

### 17.1.5 `epoll_ctl`

生成例程后，应在其内部注册监视对象文件描述符，此时使用 `epoll_ctl` 函数。

```
1 #include <sys/epoll.h>
2 int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
3 /*
4 成功时返回 0，失败时返回 -1
5 epfd: 用于注册监视对象的 epoll 例程的文件描述符
6 op: 用于制定监视对象的添加、删除或更改等操作
7 fd: 需要注册的监视对象文件描述符
8 event: 监视对象的事件类型
9 */
```

与其他 `epoll` 函数相比，该函数看起来有些复杂，但通过调用语句就很容易理解，假设按照如下形式调用 `epoll_ctl` 函数：

```
1 epoll_ctl(A,EPOLL_CTL_ADD,B,C);
```

第二个参数 `EPOLL_CTL_ADD` 意味着「添加」，上述语句有如下意义：

`epoll` 例程 A 中注册文件描述符 B，主要目的是为了监视参数 C 中的事件

再介绍一个调用语句。

```
1 epoll_ctl(A,EPOLL_CTL_DEL,B,NULL);
```

上述语句中第二个参数意味这「删除」，有以下含义：

从 `epoll` 例程 A 中删除文件描述符 B

从上述示例中可以看出，从监视对象中删除时，不需要监视类型，因此向第四个参数可以传递为 `NULL`

下面是第二个参数的含义：

- `EPOLL_CTL_ADD`: 将文件描述符注册到 `epoll` 例程
- `EPOLL_CTL_DEL`: 从 `epoll` 例程中删除文件描述符
- `EPOLL_CTL_MOD`: 更改注册的文件描述符的关注事件发生情况

epoll\_event 结构体用于保存事件的文件描述符结合。但也可以在 epoll 例程中注册文件描述符时，用于注册关注的事件。该函数中 epoll\_event 结构体的定义并不显眼，因此通过掉英语剧说明该结构体在 epoll\_ctl 函数中的应用。

```
1 struct epoll_event event;
2 ...
3 event.events=EPOLLIN;//发生需要读取数据的情况时
4 event.data.fd=sockfd;
5 epoll_ctl(epfd,EPOLL_CTL_ADD,sockfd,&event);
6 ...
```

上述代码将 epfd 注册到 epoll 例程 epfd 中，并在需要读取数据的情况下产生相应事件。接下来给出 epoll\_event 的成员 events 中可以保存的常量及所指的事件类型。

- EPOLLIN: 需要读取数据的情况
- EPOLLOUT: 输出缓冲为空，可以立即发送数据的情况
- EPOLLPRI: 收到 OOB 数据的情况
- EPOLLRDHUP: 断开连接或半关闭的情况，这在边缘触发方式下非常有用
- EPOLLERR: 发生错误的情况
- EPOLLET: 以边缘触发的方式得到事件通知
- EPOLLONESHOT: 发生一次事件后，相应文件描述符不再收到事件通知。因此需要向 epoll\_ctl 函数的第二个参数传递 EPOLL\_CTL\_MOD，再次设置事件。

可通过位运算同时传递多个上述参数。

### 17.1.6 epoll\_wait

下面是函数原型：

```
1 #include <sys/epoll.h>
2 int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
3 /*
4 成功时返回发生事件的文件描述符，失败时返回 -1
5 epfd : 表示事件发生监视范围的 epoll 例程的文件描述符
6 events : 保存发生事件的文件描述符集合的结构体地址值
7 maxevents : 第二个参数中可以保存的最大事件数
8 timeout : 以 1/1000 秒为单位的等待时间，传递 -1 时，一直等待直到发生事件
9 */
```

该函数调用方式如下。需要注意的是，第二个参数所指缓冲需要动态分配。

```
1 int event_cnt;
2 struct epoll_event *ep_events;
3 ...
4 ep_events=malloc(sizeof(struct epoll_event)*EPOLL_SIZE);//EPOLL_SIZE是宏常量
5 ...
6 event_cnt=epoll_wait(epfd,ep_events,EPOLL_SIZE,-1);
7 ...
```

调用函数后，返回发生事件的文件描述符，同时在第二个参数指向的缓冲中保存发生事件的文件描述符集合。因此，无需像 select 一样插入针对所有文件描述符的循环。

### 17.1.7 基于 epoll 的回声服务器端

下面是回声服务器端的代码（修改自第 12 章 [echo\\_selectserv.c](#)）：

- [echo\\_epollserv.c](#)

编译运行：

```
1 | gcc echo_epollserv.c -o serv
2 | ./serv 9190
```

运行结果：

The screenshot shows a terminal window with two panes. The left pane contains the command-line interface for the server, and the right pane shows the interaction with multiple clients. The server starts with 'gcc echo\_epollserv.c -o serv' and './serv 9190'. It then lists files and starts listening. The right pane shows three client connections (clnt) being established from different IP addresses (127.0.0.1, 127.0.0.0.1, and 127.0.0.0.1) to port 9190. Each client sends messages to the server, which then echoes them back.

```
# riba2534 @ hpc in ~ [15:28:51]
$ cd Documents/MyGitProject/TCP-IP-NetworkNote/ch17
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [15:29:00]
$ ls
echo_epollserv.c README.md serv
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [15:29:00]
$ ./serv 9190
connected client : 5
closed client : 5
connected client : 5
connected client : 6
[...]
/jclnt 127.0.0.1 9190
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch04 on git:master x [15:29:40]
$ ./clnt 127.0.0.1 9190
Connected.....
Input message(0 to quit): ds
Message from server: ds
Input message(0 to quit): ds
Message from server: ds
Input message(0 to quit): q
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch04 on git:master x [15:29:53]
$ ./clnt 127.0.0.1 9190
Connected.....
Input message(0 to quit): fds
Message from server: fds
Input message(0 to quit): sdf
Message from server: sdf
[...]
```

可以看出运行结果和以前 select 实现的和 fork 实现的结果一样，都可以支持多客户端同时运行。

但是这里运用了 epoll 效率高于 select

总结一下 epoll 的流程：

1. epoll\_create 创建一个保存 epoll 文件描述符的空间，可以没有参数
2. 动态分配内存，给将要监视的 epoll\_wait
3. 利用 epoll\_ctl 控制 添加 删除， 监听事件
4. 利用 epoll\_wait 来获取改变的文件描述符,来执行程序

select 和 epoll 的区别：

- 每次调用 select 函数都会向操作系统传递监视对象信息，浪费大量时间
- epoll 仅向操作系统传递一次监视对象，监视范围或内容发生变化时只通知发生变化的事项

## 17.2 条件触发和边缘触发

学习 epoll 时要了解条件触发（Level Trigger）和边缘触发（Edge Trigger）。

### 17.2.1 条件触发和边缘触发的区别在于发生事件的时间点

## 条件触发的特性:

条件触发方式中，只要输入缓冲有数据就会一直通知该事件

例如，服务器端输入缓冲收到 50 字节数据时，服务器端操作系统将通知该事件（注册到发生变化的文件描述符）。但是服务器端读取 20 字节后还剩下 30 字节的情况下，仍会注册事件。也就是说，条件触发方式中，只要输入缓冲中还剩有数据，就将以事件方式再次注册。

边缘触发特性：

边缘触发中输入缓冲收到数据时仅注册1次该事件。即使输入缓冲中还留有数据，也不会再进行注册。

## 17.2.2 掌握条件触发的事件特性

下面代码修改自 [echo\\_epollserv.c](#)。epoll 默认以条件触发的方式工作，因此可以通过该示例验证条件触发的特性。

- echo EPLTserv.c

上面的代码把调用 `read` 函数时使用的缓冲大小缩小到了 4 个字节，插入了验证 `epoll_wait` 调用次数的验证函数。减少缓冲大小是为了阻止服务器端一次性读取接收的数据。换言之，调用 `read` 函数后，输入缓冲中仍有数据要读取，而且会因此注册新的事件并从 `epoll_wait` 函数返回时将循环输出「`return epoll_wait`」字符串。

编译运行：

```
1 gcc echo_EPLTserv.c -o serv  
2 ./serv 9190
```

运行结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [16:47:32]
$ ./serv 9190
return epoll_wait
connected client : 5
return epoll_wait
closed client : 5
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [16:46:44]
$ cd ..\ch04
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch04 on git:master x [16:47:40]
$ ./clnt 127.0.0.1 9190
Connected.....
Input message(Q to quit): Ha aaaaaaa
Message From server: HaInput message(Q to quit): h
Message From server: aaaaaaa
Input message(Q to quit): hg
Message From server: h
Input message(Q to quit): gfgf
Message From server: hg
Input message(Q to quit): oggg
Message From server: gfgf
Input message(Q to quit): fffff
Message From server: oggg
Input message(Q to quit): q
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch04 on git:master x [16:48:28]
$ █
```

从结果可以看出，每当收到客户端数据时，都回注册该事件，并因此调用 `epoll_wait` 函数。

下面的代码是修改后的边缘触发方式的代码，仅仅是把上面的代码改为：

```
1     event.events = EPOLLIN | EPOLLET;
```

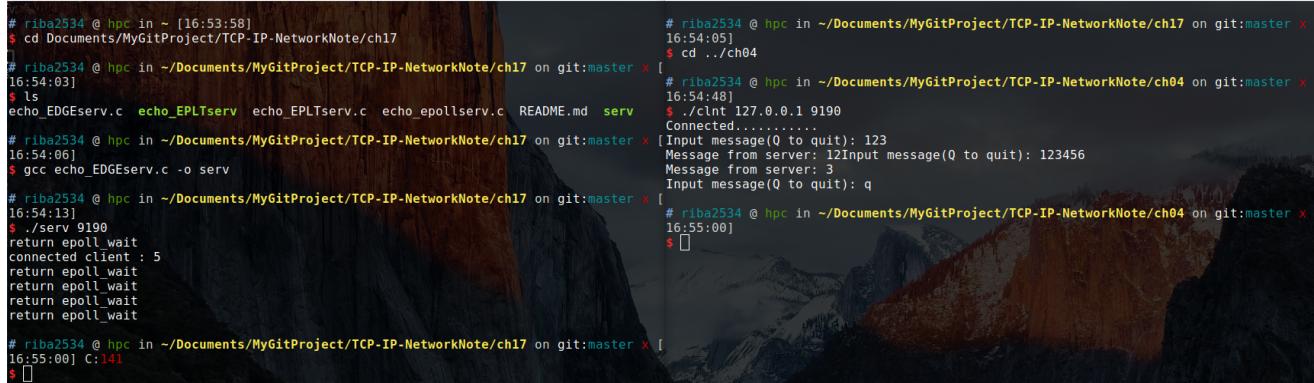
代码：

- echo EDGEServ.c

编译运行：

```
1 gcc echo_EDGEserv.c -o serv
2 ./serv 9190
```

结果：



```
# riba2534 @ hpc in ~ [16:53:58]
$ cd Documents/MyGitProject/TCP-IP-NetworkNote/ch17
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [16:54:03]
$ ls
echo_EDGEserv.c echo_EPLTserv echo_EPLTserv.c echo_epollserv.c README.md serv
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [16:54:06]
$ gcc echo_EDGEserv.c -o serv
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [16:54:13]
$ ./serv 9190
return epoll_wait
connected client : 5
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [16:55:00] C:141
$ 
```

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch04 on git:master x [16:54:05]
$ cd ..//ch04
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch04 on git:master x [16:54:48]
$ ./clnt 127.0.0.1 9190
Connected.....
[Input message(0 to quit): 123
Message from server: 123
Input message(0 to quit): 123456
Message from server: 3
Input message(0 to quit): q
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch04 on git:master x [16:55:00]
$ 
```

从上面的例子看出，接收到客户端的消息时，只输出一次「return epoll\_wait」字符串，这证明仅注册了一次事件。

**select** 模型是以条件触发的方式工作的。

### 17.2.3 边缘触发的服务器端必知的两点

- 通过 `errno` 变量验证错误原因
- 为了完成非阻塞（Non-blocking）I/O，更改了套接字特性。

Linux 套接字相关函数一般通过 -1 通知发生了错误。虽然知道发生了错误，但仅凭这些内容无法得知产生错误的原因。因此，为了在发生错误的时候提额外的信息，Linux 声明了如下全局变量：

```
1 int errno;
```

为了访问该变量，需要引入 `error.h` 头文件，因此此头文件有上述变量的 `extern` 声明。另外，每种函数发生错误时，保存在 `errno` 变量中的值都不同。

read 函数发现输入缓冲中没有数据可读时返回 -1，同时在 `errno` 中保存 `EAGAIN` 常量

下面是 Linux 中提供的改变和更改文件属性的办法：

```
1 #include <fcntl.h>
2 int fcntl(int fields, int cmd, ...);
3 /*
4 成功时返回 cmd 参数相关值，失败时返回 -1
5 filedes : 属性更改目标的文件描述符
6 cmd : 表示函数调用目的
7 */
```

从上述声明可以看出 `fcntl` 有可变参数的形式。如果向第二个参数传递 `F_GETFL`，可以获得第一个参数所指的文件描述符属性（int 型）。反之，如果传递 `F_SETFL`，可以更改文件描述符属性。若希望将文件（套接字）改为非阻塞模式，需要如下 2 条语句。

```
1 int flag = fcntl(fd, F_GETFL, 0);
2 fcntl(fd, F_SETFL | O_NONBLOCK)
```

通过第一条语句，获取之前设置的属性信息，通过第二条语句在此基础上添加非阻塞 O\_NONBLOCK 标志。调用 read/write 函数时，无论是否存在数据，都会形成非阻塞文件（套接字）。fcntl 函数的适用范围很广。

### 17.2.4 实现边缘触发回声服务器端

通过 errno 确认错误的原因是：边缘触发方式中，接收数据仅注册一次该事件。

因为这种特点，一旦发生输入相关事件时，就应该读取输入缓冲中的全部数据。因此需要验证输入缓冲是否为空。

read 函数返回 -1，变量 errno 中的值变成 EAGAIN 时，说明没有数据可读。

既然如此，为什么要将套接字变成非阻塞模式？边缘触发条件下，以阻塞方式工作的 read & write 函数有可能引起服务端的长时间停顿。因此，边缘触发方式中一定要采用非阻塞 read & write 函数。

下面是以边缘触发方式工作的回声服务端代码：

- [echo\\_EPETserv.c](#)

编译运行：

```
1 gcc echo_EPETserv.c -o serv
2 ./serv
```

结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [18:35:37] C:127
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [18:35:41] C:127
$ cd ch04
cd: 没有那个文件或目录: ch04
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [18:35:41] C:127
$ ls
clnt echo_client.c echo_server.c hello_client.c hello_server.c README.md
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch04 on git:master x [18:35:42]
$ ./clnt 127.0.0.1 9191
Connected.....
Input message(Q to quit): 123456
Message from server: 123456
Input message(Q to quit): 12
Message from server: 56
Input message(Q to quit): 12
Message from server: 12
Input message(Q to quit): 123
Message from server: 123
Input message(Q to quit):
Message from server: 123
Input message(Q to quit): ^C
Input message(Q to quit): ^C
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch17 on git:master x [18:36:19] C:130
$
```

### 17.2.5 条件触发和边缘触发孰优孰劣

边缘触发方式可以做到这点：

可以分离接收数据和处理数据的时间点！

下面是边缘触发的图

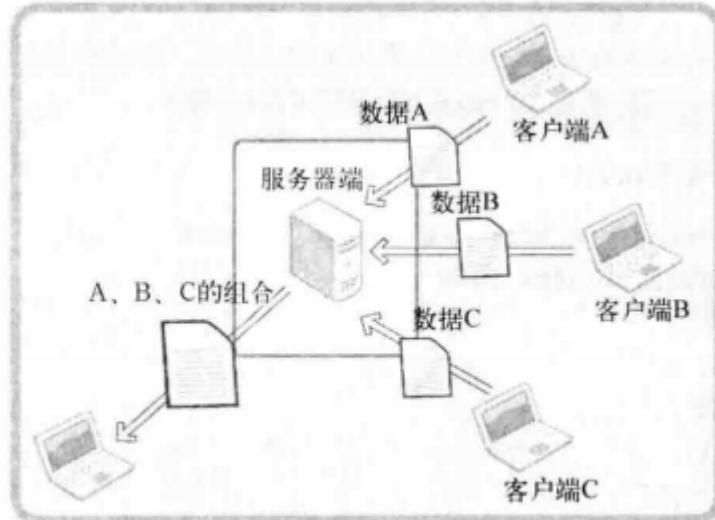


图17-1 理解边缘触发

运行流程如下：

- 服务器端分别从 A B C 接收数据
- 服务器端按照 A B C 的顺序重新组合接收到的数据
- 组合的数据将发送给任意主机。

为了完成这个过程，如果可以按照如下流程运行，服务端的实现并不难：

- 客户端按照 A B C 的顺序连接服务器，并且按照次序向服务器发送数据
- 需要接收数据的客户端应在客户端 A B C 之前连接到服务器端并等待

但是实际情况中可能是下面这样：

- 客户端 C 和 B 正在向服务器发送数据，但是 A 并没有连接到服务器
- 客户端 A B C 乱序发送数据
- 服务端已经接收到数据，但是要接收数据的目标客户端并没有连接到服务器端。

因此，即使输入缓冲收到数据，服务器端也能决定读取和处理这些数据的时间点，这样就给服务器端的实现带来很大灵活性。

### 17.3 习题

以下答案仅代表本人个人观点，可能不是正确答案。

- 利用 select 函数实现服务器端时，代码层面存在的两个缺点是？

答：①调用 select 函数后常见的针对所有文件描述符的循环语句②每次调用 select 函数时都要传递监视对象信息。

- 无论是 select 方式还是 epoll 方式，都需要将监视对象文件描述符信息通过函数调用传递给操作系统。请解释传递该信息的原因。

答：文件描述符是由操作系统管理的，所以必须要借助操作系统才能完成。

- select 方式和 epoll 方式的最大差异在于监视对象文件描述符传递给操作系统的方式。请说明具体差异，并解释为何存在这种差异。

答：select 函数每次调用都要传递所有的监视对象信息，而 epoll 函数仅向操作系统传递 1 次监视对象，监视范围或内容发生变化时只通知发生变化的事项。select 采用这种方法是为了保持兼容性。

4. 虽然 epoll 是 select 的改进反感，但 select 也有自己的优点。在何种情况下使用 select 更加合理。

答：①服务器端接入者少②程序应具有兼容性。

5. epoll 是以条件触发和边缘触发方式工作。二者有何差别？从输入缓冲的角度说明这两种方式通知事件的时间点差异。

答：在条件触发中，只要输入缓冲有数据，就会一直通知该事件。边缘触发中输入缓冲收到数据时仅注册 1 次该事件，即使输入缓冲中还留有数据，也不会再进行注册。

6. 采用边缘触发时可以分离数据的接收和处理时间点。请说明其优点和原因。

答：分离接收数据和处理数据的时间点，给服务端的实现带来很大灵活性。

## 第 18 章 多线程服务器端的实现

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

### 18.1 理解线程的概念

#### 18.1.1 引入线程背景

第 10 章介绍了多进程服务端的实现方法。多进程模型与 select 和 epoll 相比的确有自身的优点，但同时也存在问题。如前所述，创建（复制）进程的工作本身会给操作系统带来相当沉重的负担。而且，每个进程都具有独立的内存空间，所以进程间通信的实现难度也会随之提高。换言之，多进程的缺点可概括为：

- 创建进程的过程会带来一定的开销
- 为了完成进程间数据交换，需要特殊的 IPC 技术。

但是更大的缺点是下面的：

- 每秒少则 10 次，多则千次的「上下文切换」是创建进程的最大开销

只有一个 CPU 的系统是将时间分成多个微小的块后分配给了多个进程。为了分时使用 CPU，需要「上下文切换」的过程。「上下文切换」是指运行程序前需要将相应进程信息读入内存，如果运行进程 A 后紧接着需要运行进程 B，就应该将进程 A 相关信息移出内存，并读入进程 B 相关信息。这就是上下文切换。但是此时进程 A 的数据将被移动到硬盘，所以上下文切换要很长时间，即使通过优化加快速度，也会存在一定的局限。

为了保持多进程的优点，同时在一定程度上克服其缺点，人们引入的线程（Thread）的概念。这是为了将进程的各种劣势降至最低程度（不是直接消除）而设立的一种「轻量级进程」。线程比进程具有如下优点：

- 线程的创建和上下文切换比进程的创建和上下文切换更快
- 线程间交换数据无需特殊技术

#### 18.1.2 线程和进程的差异

线程是为了解决：为了得到多条代码执行流而复制整个内存区域的负担太重。

每个进程的内存空间都由保存全局变量的「数据区」、向 malloc 等函数动态分配提供空间的堆（Heap）、函数运行时间使用的栈（Stack）构成。每个进程都有独立的这种空间，多个进程的内存结构如图所示：

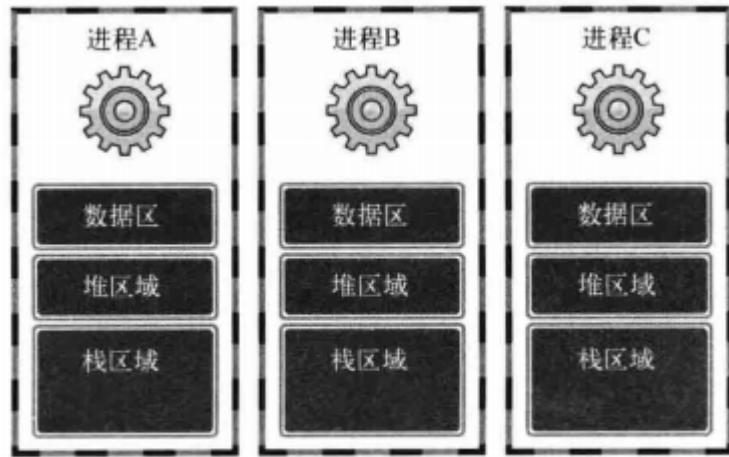


图18-1 进程间独立的内存

但如果以获得多个代码执行流为目的，则不应该像上图那样完全分离内存结构，而只需分离栈区域。通过这种方式可以获得如下优势：

- 上下文切换时不需要切换数据区和堆
- 可以利用数据区和堆交换数据

实际上这就是线程。线程为了保持多条代码执行流而隔开了栈区域，因此具有如下图所示的内存结构：



图18-2 线程的内存结构

如图所示，多个线程共享数据区和堆。为了保持这种结构，线程将在进程内创建并运行。也就是说，进程和线程可以定义为如下形式：

- 进程：在操作系统构成单独执行流的单位
- 线程：在进程构成单独执行流的单位

如果说进程在操作系统内部生成多个执行流，那么线程就在同一进程内部创建多条执行流。因此，操作系统、进程、线程之间的关系可以表示为下图：

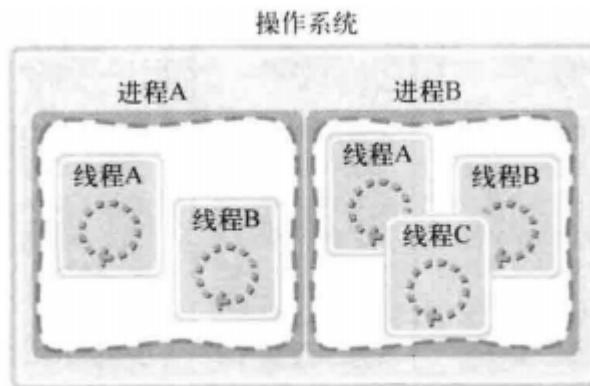


图18-3 操作系统、进程、线程之间的关系

## 18.2 线程创建及运行

可移植操作系统接口（英语：Portable Operating System Interface，缩写为POSIX）是IEEE为要在各种UNIX操作系统上运行软件，而定义API的一系列互相关联的标准的总称，其正式称呼为IEEE Std 1003，而国际标准名称为ISO/IEC 9945。此标准源于一个大约开始于1985年的项目。POSIX这个名称是由理查德·斯托曼（RMS）应IEEE的要求而提议的一个易于记忆的名称。它基本上是Portable Operating System Interface（可移植操作系统接口）的缩写，而X则表明其对Unix API的传承。

Linux基本上逐步实现了POSIX兼容，但并没有参加正式的POSIX认证。

微软的Windows NT声称部分实现了POSIX标准。

当前的POSIX主要分为四个部分：Base Definitions、System Interfaces、Shell and Utilities和Rationale。

### 18.2.1 线程的创建和执行流程

线程具有单独的执行流，因此需要单独定义线程的 main 函数，还需要请求操作系统在单独的执行流中执行该函数，完成函数功能的函数如下：

```

1 #include <pthread.h>
2
3 int pthread_create(pthread_t *restrict thread,
4                     const pthread_attr_t *restrict attr,
5                     void *(*start_routine)(void *),
6                     void *restrict arg);
7 /*
8 成功时返回 0，失败时返回 -1
9 thread : 保存新创建线程 ID 的变量地址值。线程与进程相同，也需要用于区分不同线程的 ID
10 attr : 用于传递线程属性的参数，传递 NULL 时，创建默认属性的线程
11 start_routine : 相当于线程 main 函数的、在单独执行流中执行的函数地址值（函数指针）
12 arg : 通过第三个参数传递的调用函数时包含传递参数信息的变量地址值
13 */

```

下面通过简单示例了解该函数功能：

- [thread1.c](#)

```

1 #include <stdio.h>
2 #include <pthread.h>

```

```

3 #include <unistd.h>
4 void *thread_main(void *arg);
5
6 int main(int argc, char *argv[])
7 {
8     pthread_t t_id;
9     int thread_param = 5;
10    // 请求创建一个线程，从 thread_main 调用开始，在单独的执行流中运行。同时传递参数
11    if (pthread_create(&t_id, NULL, thread_main, (void *)&thread_param) != 0)
12    {
13        puts("pthread_create() error");
14        return -1;
15    }
16    sleep(10); //延迟进程终止时间
17    puts("end of main");
18    return 0;
19 }
20 void *thread_main(void *arg) //传入的参数是 pthread_create 的第四个
21 {
22     int i;
23     int cnt = *((int *)arg);
24     for (int i = 0; i < cnt; i++)
25     {
26         sleep(1);
27         puts("running thread");
28     }
29     return NULL;
30 }
```

编译运行：

```

1 gcc thread1.c -o tr1 -lpthread # 线程相关代码编译时需要添加 -lpthread 选项声明需要连接到线程库
2 ./tr1
```

运行结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch18 on git:master x
23:21:22]
$ ./tr1
running thread
running thread
running thread
running thread
running thread
running thread
end of main
```

上述程序的执行如图所示：

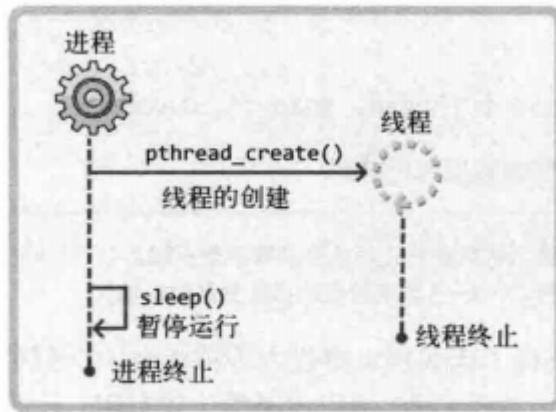


图18-4 示例thread1.c的执行流程

可以看出，程序在主进程没有结束时，生成的线程每隔一秒输出一次 `running thread`，但是如果主进程没有等待十秒，而是直接结束，这样也会强制结束线程，不论线程有没有运行完毕。

那是否意味着主进程必须每次都 `sleep` 来等待线程执行完毕？并不需要，可以通过以下函数解决。

```

1 #include <pthread.h>
2 int pthread_join(pthread_t thread, void **status);
3 /*
4 成功时返回 0，失败时返回 -1
5 thread：该参数值 ID 的线程终止后才会从该函数返回
6 status：保存线程的 main 函数返回值的指针的变量地址值
7 */

```

作用就是调用该函数的进程（或线程）将进入等待状态，知道第一个参数为 ID 的线程终止为止。而且可以得到线程的 `main` 函数的返回值。下面是该函数的用法代码：

- [thread2.c](#)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <pthread.h>
5 void *thread_main(void *arg);
6
7 int main(int argc, char *argv[])
8 {
9     pthread_t t_id;
10    int thread_param = 5;
11    void *thr_ret;
12    // 请求创建一个线程，从 thread_main 调用开始，在单独的执行流中运行。同时传递参数
13    if (pthread_create(&t_id, NULL, thread_main, (void *)&thread_param) != 0)
14    {
15        puts("pthread_create() error");
16        return -1;
17    }
18    //main函数将等待 ID 保存在 t_id 变量中的线程终止
19    if (pthread_join(t_id, &thr_ret) != 0)
20    {

```

```

21     puts("pthread_join() error");
22     return -1;
23 }
24 printf("Thread return message : %s \n", (char *)thr_ret);
25 free(thr_ret);
26 return 0;
27 }
28 void *thread_main(void *arg) //传入的参数是 pthread_create 的第四个
29 {
30     int i;
31     int cnt = *((int *)arg);
32     char *msg = (char *)malloc(sizeof(char) * 50);
33     strcpy(msg, "Hello,I'am thread~ \n");
34     for (int i = 0; i < cnt; i++)
35     {
36         sleep(1);
37         puts("running thread");
38     }
39     return (void *)msg; //返回值是 thread_main 函数中内部动态分配的内存空间地址值
40 }
```

编译运行：

```

1 gcc thread2.c -o tr2 -lpthread
2 ./tr2
```

运行结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch18 on git:master x [23:54:30]
$ ./tr2
running thread
running thread
running thread
running thread
running thread
Thread return message : Hello,I'am thread~
```

可以看出，线程输出了5次字符串，并且把返回值给了主进程

下面是该函数的执行流程图：

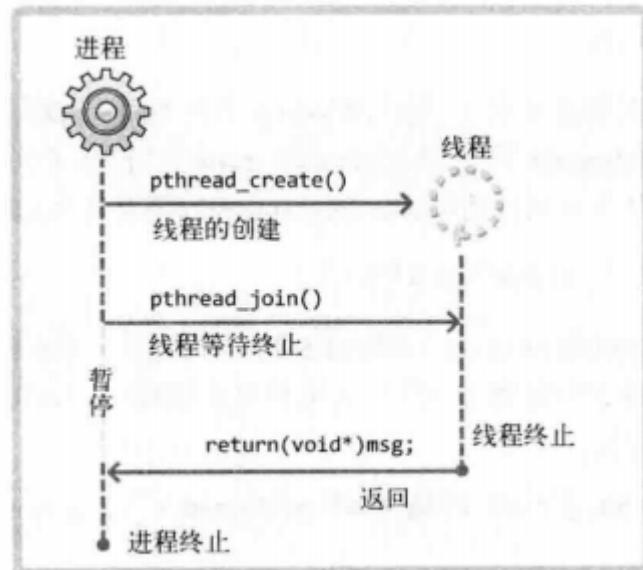


图18-6 调用pthread\_join函数

### 18.2.2 可在临界区内调用的函数

在同步的程序设计中，临界区块（Critical section）指的是一个访问共享资源（例如：共享设备或是共享存储器）的程序片段，而这些共享资源有无法同时被多个线程访问的特性。

当有线程进入临界区块时，其他线程或是进程必须等待（例如： bounded waiting 等待法），有一些同步的机制必须在临界区块的进入点与离开点实现，以确保这些共享资源是被异或的使用，例如：semaphore。

只能被单一线程访问的设备，例如：打印机。

一个最简单的实现方法就是当线程（Thread）进入临界区块时，禁止改变处理器；在uni-processor系统上，可以用“禁止中断（CLI）”来完成，避免发生系统调用（System Call）导致的上下文交换（Context switching）；当离开临界区块时，处理器恢复原先的状态。

根据临界区是否引起问题，函数可以分为以下 2 类：

- 线程安全函数（Thread-safe function）
- 非线程安全函数（Thread-unsafe function）

线程安全函数被多个线程同时调用也不会发生问题。反之，非线程安全函数被同时调用时会引发问题。但这并非有关于临界区的讨论，线程安全的函数中同样可能存在临界区。只是在线程安全的函数中，同时被多个线程调用时可通过一些措施避免问题。

幸运的是，大多数标准函数都是线程安全函数。操作系统在定义非线程安全函数的同时，提供了具有相同功能的线程安全的函数。比如，第 8 章的：

```
1 | struct hostent *gethostbyname(const char *hostname);
```

同时，也提供了同一功能的安全函数：

```
1 struct hostent *gethostbyname_r(const char *name,
2                                 struct hostent *result,
3                                 char *buffer,
4                                 int intbuflen,
5                                 int *h_errnop);
```

线程安全函数结尾通常是 `_r`。但是使用线程安全函数会给程序员带来额外的负担，可以通过以下方法自动将 `gethostbyname` 函数调用改为 `gethostbyname_r` 函数调用。

声明头文件前定义 `_REENTRANT` 宏。

无需特意更改源代码加，可以在编译的时候指定编译参数定义宏。

```
1 gcc -D_REENTRANT mythread.c -o mthread -lpthread
```

### 18.2.3 工作（Worker）线程模型

下面的示例是计算从 1 到 10 的和，但并不是通过 `main` 函数进行运算，而是创建两个线程，其中一个线程计算 1 到 5 的和，另一个线程计算 6 到 10 的和，`main` 函数只负责输出运算结果。这种方式的线程模型称为「工作线程」。显示该程序的执行流程图：

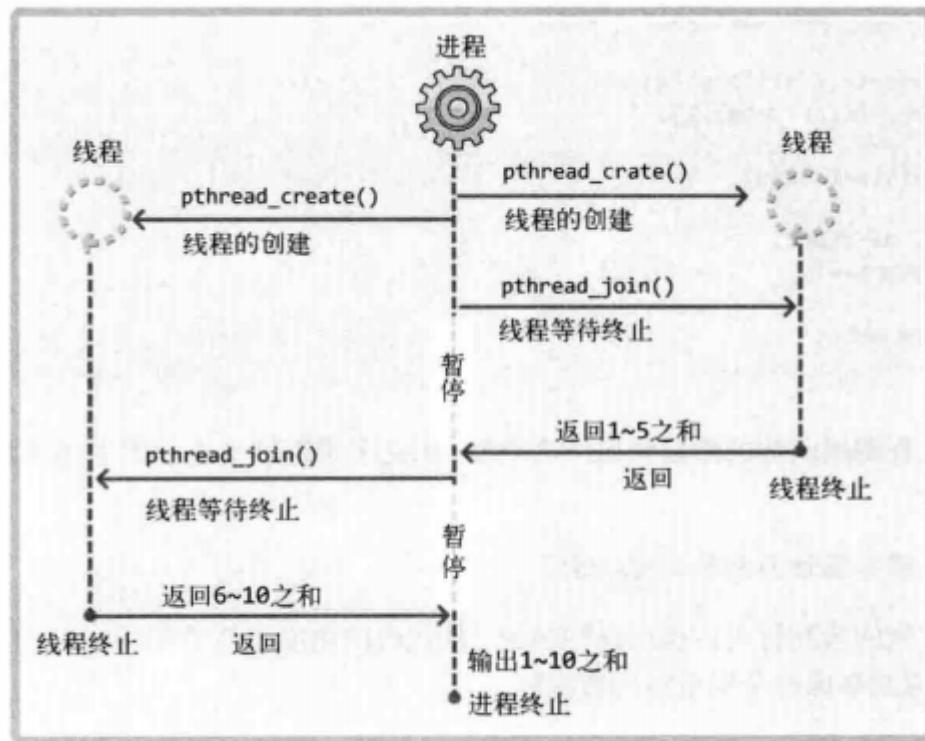


图18-7 示例thread3.c的执行流程

下面是代码：

- [thread3.c](#)

```
1 #include <stdio.h>
2 #include <pthread.h>
3 void *thread_summation(void *arg);
```

```

4 int sum = 0;
5
6 int main(int argc, char *argv[])
7 {
8     pthread_t id_t1, id_t2;
9     int range1[] = {1, 5};
10    int range2[] = {6, 10};
11
12    pthread_create(&id_t1, NULL, thread_summation, (void *)range1);
13    pthread_create(&id_t2, NULL, thread_summation, (void *)range2);
14
15    pthread_join(id_t1, NULL);
16    pthread_join(id_t2, NULL);
17    printf("result: %d \n", sum);
18    return 0;
19 }
20 void *thread_summation(void *arg)
21 {
22     int start = ((int *)arg)[0];
23     int end = ((int *)arg)[1];
24     while (start <= end)
25     {
26         sum += start;
27         start++;
28     }
29     return NULL;
30 }
```

编译运行：

```

1 gcc thread3.c -D_REENTRANT -o tr3 -lpthread
2 ./tr3
```

结果：

```

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch18 on git:master x [0:27:23]
$ gcc thread3.c -D_REENTRANT -o tr3 -lpthread

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch18 on git:master x [0:27:56]
$ ./tr3
result: 55
```

可以看出计算结果正确，两个线程都用了全局变量 sum，证明了 2 个线程共享保存全局变量的数据区。

但是本例子本身存在问题。存在临界区相关问题，可以从下面的代码看出，下面的代码和上面的代码相似，只是增加了发生临界区错误的可能性，即使在高配置系统环境下也容易产生的错误：

- [thread4.c](#)

```

1 #include <stdio.h>
2 #include <unistd.h>
```

```

3 #include <stdlib.h>
4 #include <pthread.h>
5 #define NUM_THREAD 100
6
7 void *thread_inc(void *arg);
8 void *thread_des(void *arg);
9 long long num = 0;
10
11 int main(int argc, char *argv[])
12 {
13     pthread_t thread_id[NUM_THREAD];
14     int i;
15
16     printf("sizeof long long: %d \n", sizeof(long long));
17     for (i = 0; i < NUM_THREAD; i++)
18     {
19         if (i % 2)
20             pthread_create(&(thread_id[i]), NULL, thread_inc, NULL);
21         else
22             pthread_create(&(thread_id[i]), NULL, thread_des, NULL);
23     }
24
25     for (i = 0; i < NUM_THREAD; i++)
26         pthread_join(thread_id[i], NULL);
27
28     printf("result: %lld \n", num);
29     return 0;
30 }
31
32 void *thread_inc(void *arg)
33 {
34     int i;
35     for (i = 0; i < 50000000; i++)
36         num += 1;
37     return NULL;
38 }
39 void *thread_des(void *arg)
40 {
41     int i;
42     for (i = 0; i < 50000000; i++)
43         num -= 1;
44     return NULL;
45 }
```

编译运行：

```

1 gcc thread4.c -D_REENTRANT -o tr4 -lpthread
2 ./tr4
```

结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch18 on git:master x [0:40:15]
$ gcc thread4.c -D_REENTRANT -o tr4 -lpthread
thread4.c: In function 'main':
thread4.c:16:32: warning: format '%d' expects argument of type 'int', but argument 2 has type 'long unsigned int' [-Wformat=]
    printf("sizeof long long: %d \n", sizeof(long long));
                           ^~~~~
                           %ld

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch18 on git:master x [0:40:42]
$ ./tr4
sizeof long long: 8
result: 570832

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch18 on git:master x [0:41:22]
$ ./tr4
sizeof long long: 8
result: -3986187

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch18 on git:master x [0:41:48]
$ 
```

从图上可以看出，每次运行的结果竟然不一样。理论上来说，上面代码的最后结果应该是 0。原因暂时不得而知，但是可以肯定的是，这对于线程的应用是个大问题。

## 18.3 线程存在的问题和临界区

下面分析 [thread4.c](#) 中产生问题的原因，并给出解决方案。

### 18.3.1 多个线程访问同一变量是问题

[thread4.c](#) 的问题如下：

2 个线程正在同时访问全局变量 num

任何内存空间，只要被同时访问，都有可能发生问题。

因此，线程访问变量 num 时应该阻止其他线程访问，直到线程 1 运算完成。这就是同步（Synchronization）

### 18.3.2 临界区位置

那么在刚才代码中的临界区位置是：

函数内同时运行多个线程时引发问题的多条语句构成的代码块

全局变量 num 不能视为临界区，因为他不是引起问题的语句，只是一个内存区域的声明。下面是刚才代码的两个 main 函数

```
1 void *thread_inc(void *arg)
2 {
3     int i;
4     for (i = 0; i < 50000000; i++)
5         num += 1;//临界区
6     return NULL;
7 }
```

```
8 void *thread_des(void *arg)
9 {
10     int i;
11     for (i = 0; i < 50000000; i++)
12         num -= 1;//临界区
13     return NULL;
14 }
```

由上述代码可知，临界区并非 num 本身，而是访问 num 的两条语句，这两条语句可能由多个线程同时运行，也是引起这个问题的直接原因。产生问题的原因可以分为以下三种情况：

- 2个线程同时执行 thread\_inc 函数
- 2个线程同时执行 thread\_des 函数
- 2个线程分别执行 thread\_inc 和 thread\_des 函数

比如发生以下情况：

线程 1 执行 thread\_inc 的 num+=1 语句的同时，线程 2 执行 thread\_des 函数的 num-=1 语句

也就是说，两条不同的语句由不同的线程执行时，也有可能构成临界区。前提是这 2 条语句访问同一内存空间。

## 18.4 线程同步

前面讨论了线程中存在的问题，下面就是解决方法，线程同步。

### 18.4.1 同步的两面性

线程同步用于解决线程访问顺序引发的问题。需要同步的情况可以从如下两方面考虑。

- 同时访问同一内存空间时发生的情况
- 需要指定访问同一内存空间的线程顺序的情况

情况一之前已经解释过，下面讨论情况二。这是「控制线程执行的顺序」的相关内容。假设有 A B 两个线程，线程 A 负责向指定的内存空间内写入数据，线程 B 负责取走该数据。所以这是有顺序的，不按照顺序就可能发生问题。所以这种也需要进行同步。

### 18.4.2 互斥量

互斥锁（英语：英语：Mutual exclusion，缩写 Mutex）是一种用于多线程编程中，防止两条线程同时对同一公共资源（比如全局变量）进行读写的机制。该目的通过将代码切片成一个一个的临界区域（critical section）达成。临界区域指的是一块对公共资源进行访问的代码，并非一种机制或是算法。一个程序、进程、线程可以拥有多个临界区域，但是并不一定会应用互斥锁。

通俗的说就互斥量就是一把优秀的锁，当临界区被占据的时候就上锁，等占用完毕然后再放开。

下面是互斥量的创建及销毁函数。

```
1 #include <pthread.h>
2 int pthread_mutex_init(pthread_mutex_t *mutex,
3                         const pthread_mutexattr_t *attr);
4 int pthread_mutex_destroy(pthread_mutex_t *mutex);
5 /*
6 成功时返回 0, 失败时返回其他值
7 mutex : 创建互斥量时传递保存互斥量的变量地址值, 销毁时传递需要销毁的互斥量地址
8 attr : 传递即将创建的互斥量属性, 没有特别需要指定的属性时传递 NULL
9 */
```

从上述函数声明中可以看出，为了创建相当于锁系统的互斥量，需要声明如下 pthread\_mutex\_t 型变量：

```
1 pthread_mutex_t mutex
```

该变量的地址值传递给 pthread\_mutex\_init 函数，用来保存操作系统创建的互斥量（锁系统）。调用 pthread\_mutex\_destroy 函数时同样需要该信息。如果不配置特殊的互斥量属性，则向第二个参数传递 NULL 时，可以利用 PTHREAD\_MUTEX\_INITIALIZER 进行如下声明：

```
1 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

推荐尽可能的使用 pthread\_mutex\_init 函数进行初始化，因为通过宏进行初始化时很难发现发生的错误。

下面是利用互斥量锁住或释放临界区时使用的函数。

```
1 #include <pthread.h>
2 int pthread_mutex_lock(pthread_mutex_t *mutex);
3 int pthread_mutex_unlock(pthread_mutex_t *mutex);
4 /*
5 成功时返回 0, 失败时返回其他值
6 */
```

函数本身含有 lock unlock 等词汇，很容易理解其含义。进入临界区前调用的函数就是 pthread\_mutex\_lock。调用该函数时，发现有其他线程已经进入临界区，则 pthread\_mutex\_lock 函数不会返回，直到里面的线程调用 pthread\_mutex\_unlock 函数退出临界区位置。也就是说，其他线程让出临界区之前，当前线程一直处于阻塞状态。接下来整理一下代码的编写方式：

```
1 pthread_mutex_lock(&mutex);
2 //临界区开始
3 //...
4 //临界区结束
5 pthread_mutex_unlock(&mutex);
```

简言之，就是利用 lock 和 unlock 函数围住临界区的两端。此时互斥量相当于一把锁，阻止多个线程同时访问，还有一点要注意，线程退出临界区时，如果忘了调用 pthread\_mutex\_unlock 函数，那么其他为了进入临界区而调用 pthread\_mutex\_lock 的函数无法摆脱阻塞状态。这种情况称为「死锁」。需要格外注意，下面是利用互斥量解决示例 [thread4.c](#) 中遇到的问题代码：

- [mutex.c](#)

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <pthread.h>
5 #define NUM_THREAD 100
6 void *thread_inc(void *arg);
7 void *thread_des(void *arg);
8
9 long long num = 0;
10 pthread_mutex_t mutex; //保存互斥量读取值的变量
11
12 int main(int argc, char *argv[])
13 {
14     pthread_t thread_id[NUM_THREAD];
15     int i;
16
17     pthread_mutex_init(&mutex, NULL); //创建互斥量
18
19     for (i = 0; i < NUM_THREAD; i++)
20     {
21         if (i % 2)
22             pthread_create(&(thread_id[i]), NULL, thread_inc, NULL);
23         else
24             pthread_create(&(thread_id[i]), NULL, thread_des, NULL);
25     }
26
27     for (i = 0; i < NUM_THREAD; i++)
28         pthread_join(thread_id[i], NULL);
29
30     printf("result: %lld \n", num);
31     pthread_mutex_destroy(&mutex); //销毁互斥量
32     return 0;
33 }
34
35 void *thread_inc(void *arg)
36 {
37     int i;
38     pthread_mutex_lock(&mutex); //上锁
39     for (i = 0; i < 50000000; i++)
40         num += 1;
41     pthread_mutex_unlock(&mutex); //解锁
42     return NULL;
43 }
44 void *thread_des(void *arg)
45 {
46     int i;
47     pthread_mutex_lock(&mutex);
48     for (i = 0; i < 50000000; i++)
49         num -= 1;
50     pthread_mutex_unlock(&mutex);
51     return NULL;
```

编译运行：

```
1 | gcc mutex.c -D_REENTRANT -o mutex -lpthread
2 | ./mutex
```

运行结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch18 on git:master x [13:37:38]
$ ./mutex
result: 0
```

从运行结果可以看出，通过互斥量机制得出了正确的运行结果。

在代码中：

```
1 | void *thread_inc(void *arg)
2 | {
3 |     int i;
4 |     pthread_mutex_lock(&mutex); //上锁
5 |     for (i = 0; i < 50000000; i++)
6 |         num += 1;
7 |     pthread_mutex_unlock(&mutex); //解锁
8 |     return NULL;
9 | }
```

以上代码的临界区划分范围较大，但这是考虑如下优点所做的决定：

最大限度减少互斥量 lock unlock 函数的调用次数

### 18.4.3 信号量

信号量（英语：Semaphore）又称为信号标，是一个同步对象，用于保持在0至指定最大值之间的一个计数值。当线程完成一次对该semaphore对象的等待（wait）时，该计数值减一；当线程完成一次对semaphore对象的释放（release）时，计数值加一。当计数值为0，则线程等待该semaphore对象不再能成功直至该semaphore对象变成signaled状态。semaphore对象的计数值大于0，为signaled状态；计数值等于0，为nonsignaled状态。

semaphore对象适用于控制一个仅支持有限个用户的共享资源，是一种不需要使用忙碌等待（busy waiting）的方法。

信号量的概念是由荷兰计算机科学家艾兹赫尔·戴克斯特拉（Edsger W. Dijkstra）发明的，广泛的应用于不同的操作系统中。在系统中，给予每一个进程一个信号量，代表每个进程当前的状态，未得到控制权的进程会在特定地方被强迫停下来，等待可以继续进行的信号到来。如果信号量是一个任意的整数，通常被称为计数信号量（Counting semaphore），或一般信号量（general semaphore）；如果信号量只有二进制的0或1，称为二进制信号量（binary semaphore）。在linux系统中，二进制信号量（binary semaphore）又称互斥锁（Mutex）。

下面介绍信号量，在互斥量的基础上，很容易理解信号量。此处只涉及利用「二进制信号量」（只用0和1）完成「控制线程顺序」为中心的同步方法。下面是信号量的创建及销毁方法：

```

1 #include <semaphore.h>
2 int sem_init(sem_t *sem, int pshared, unsigned int value);
3 int sem_destroy(sem_t *sem);
4 /*
5 成功时返回 0，失败时返回其他值
6 sem : 创建信号量时保存信号量的变量地址值，销毁时传递需要销毁的信号量变量地址值
7 pshared : 传递其他值时，创建可由多个继承共享的信号量；传递 0 时，创建只允许 1 个进程内部使用的信号
量。需要完成同一进程的线程同步，故为0
8 value : 指定创建信号量的初始值
9 */

```

上述的 shared 参数超出了我们的关注范围，故默认向其传递为 0。下面是信号量中相当于互斥量 lock unlock 的函数。

```

1 #include <semaphore.h>
2 int sem_post(sem_t *sem);
3 int sem_wait(sem_t *sem);
4 /*
5 成功时返回 0，失败时返回其他值
6 sem : 传递保存信号量读取值的变量地址值，传递给 sem_post 的信号量增1，传递给 sem_wait 时信号量减一
7 */

```

调用 sem\_init 函数时，操作系统将创建信号量对象，此对象中记录这「信号量值」（Semaphore Value）整数。该值在调用 sem\_post 函数时增加 1，调用 wait\_wait 函数时减一。但信号量的值不能小于 0，因此，在信号量为 0 的情况下调用 sem\_wait 函数时，调用的线程将进入阻塞状态（因为函数未返回）。当然，此时如果有其他线程调用 sem\_post 函数，信号量的值将变为 1，而原本阻塞的线程可以将该信号量重新减为 0 并跳出阻塞状态。实际上就是通过这种特性完成临界区的同步操作，可以通过如下形式同步临界区（假设信号量的初始值为 1）

```

1 sem_wait(&sem); //信号量变为0...
2 // 临界区的开始
3 //...
4 //临界区的结束
5 sem_post(&sem); //信号量变为1...

```

上述代码结构中，调用 sem\_wait 函数进入临界区的线程在调用 sem\_post 函数前不允许其他线程进入临界区。信号量的值在 0 和 1 之间跳转，因此，具有这种特性的机制称为「二进制信号量」。接下来的代码是信号量机制的代码。下面代码并非是同时访问的同步，而是关于控制访问顺序的同步，该场景为：

线程 A 从用户输入得到值后存入全局变量 num，此时线程 B 将取走该值并累加。该过程一共进行 5 次，完成后输出总和并退出程序。

下面是代码：

- [semaphore.c](#)

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4

```

```

5 void *read(void *arg);
6 void *accu(void *arg);
7 static sem_t sem_one;
8 static sem_t sem_two;
9 static int num;
10
11 int main(int argc, char const *argv[])
12 {
13     pthread_t id_t1, id_t2;
14     sem_init(&sem_one, 0, 0);
15     sem_init(&sem_two, 0, 1);
16
17     pthread_create(&id_t1, NULL, read, NULL);
18     pthread_create(&id_t2, NULL, accu, NULL);
19
20     pthread_join(id_t1, NULL);
21     pthread_join(id_t2, NULL);
22
23     sem_destroy(&sem_one);
24     sem_destroy(&sem_two);
25     return 0;
26 }
27
28 void *read(void *arg)
29 {
30     int i;
31     for (i = 0; i < 5; i++)
32     {
33         fputs("Input num: ", stdout);
34
35         sem_wait(&sem_two);
36         scanf("%d", &num);
37         sem_post(&sem_one);
38     }
39     return NULL;
40 }
41 void *accu(void *arg)
42 {
43     int sum = 0, i;
44     for (i = 0; i < 5; i++)
45     {
46         sem_wait(&sem_one);
47         sum += num;
48         sem_post(&sem_two);
49     }
50     printf("Result: %d \n", sum);
51     return NULL;
52 }
```

编译运行：

```
1 gcc semaphore.c -D_REENTRANT -o sema -lpthread  
2 ./sema
```

结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch18 on git:master x [  
14:35:42]  
$ gcc semaphore.c -D_REENTRANT -o sema -lpthread  
  
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch18 on git:master x [  
14:36:20]  
$ ./sema  
Input num: 1  
Input num: 2  
Input num: 3  
Input num: 4  
Input num: 5  
Result: 15
```

从上述代码可以看出，设置了两个信号量 one 的初始值为 0， two 的初始值为 1，然后在调用函数的时候，「读」的前提是 two 可以减一，如果不能减一就会阻塞在这里，一直等到「计算」操作完毕后，给 two 加一，然后就可以继续执行下一句输入。对于「计算」函数，也一样。

## 18.5 线程的销毁和多线程并发服务器端的实现

先介绍线程的销毁，然后再介绍多线程服务端

### 18.5.1 销毁线程的 3 种方法

Linux 的线程并不是在首次调用的线程 main 函数返回时自动销毁，所以利用如下方法之一加以明确。否则由线程创建的内存空间将一直存在。

- 调用 pthread\_join 函数
- 调用 pthread\_detach 函数

之前调用过 pthread\_join 函数。调用该函数时，不仅会等待线程终止，还会引导线程销毁。但该函数的问题是，线程终止前，调用该函数的线程将进入阻塞状态。因此，通过如下函数调用引导线程销毁。

```
1 #include <pthread.h>  
2 int pthread_detach(pthread_t th);  
3 /*  
4 成功时返回 0，失败时返回其他值  
5 thread : 终止的同时需要销毁的线程 ID  
6 */
```

调用上述函数不会引起线程终止或进入阻塞状态，可以通过该函数引导销毁线程创建的内存空间。调用该函数后不能再针对相应线程调用 pthread\_join 函数。

### 18.5.2 多线程并发服务器端的实现

下面是多个客户端之间可以交换信息的简单聊天程序。

- [chat\\_server.c](#)
- [chat\\_clnt.c](#)

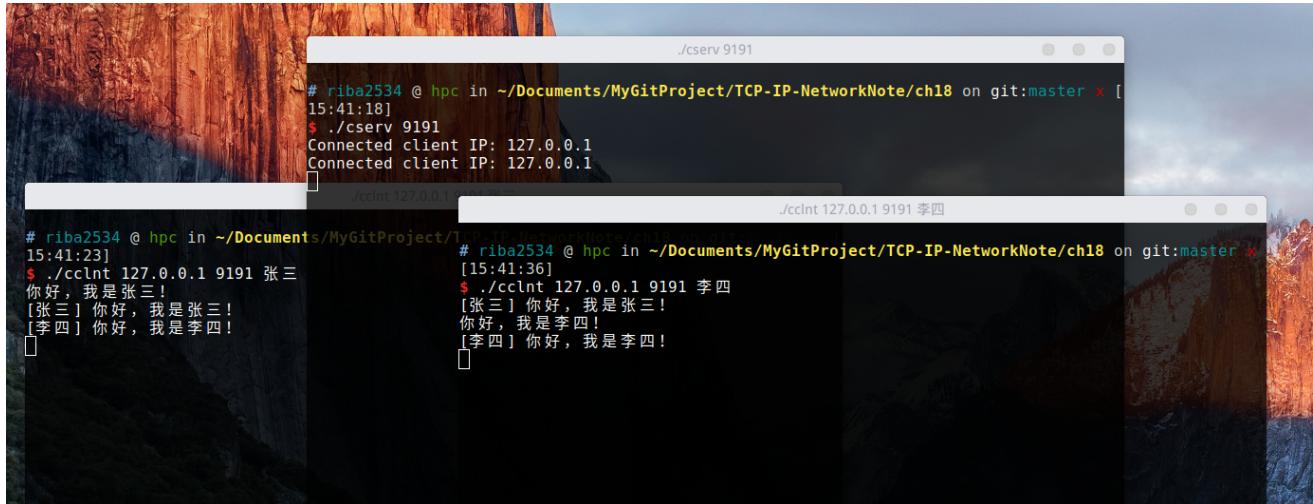
上面的服务端示例中，需要掌握临界区的构成，访问全局变量 clnt\_cnt 和数组 clnt\_socks 的代码将构成临界区，添加和删除客户端时，变量 clnt\_cnt 和数组 clnt\_socks 将同时发生变化。因此下列情形会导致数据不一致，从而引发错误：

- 线程 A 从数组 clnt\_socks 中删除套接字信息，同时线程 B 读取 clnt\_cnt 变量
- 线程 A 读取变量 clnt\_cnt，同时线程 B 将套接字信息添加到 clnt\_socks 数组

编译运行：

```
1 gcc chat_server.c -D_REENTRANT -o cserv -lpthread
2 gcc chat_clnt.c -D_REENTRANT -o cclnt -lpthread
3 ./cserv 9191
4 ./cclnt 127.0.0.1 9191 张三
5 ./cclnt 127.0.0.1 9191 李四
```

结果：



## 18.6 习题

以下答案仅代表本人个人观点，可能不是正确答案。

### 1. 单 CPU 系统中如何同时执行多个进程？请解释该过程中发生的上下文切换。

答：系统将 CPU 时间分成多个微小的块后分配给了多个进程。为了分时使用 CPU，需要「上下文切换」过程。运行程序前需要将相应进程信息读入内存，如果运行进程 A 后需要紧接着运行进程 B，就应该将进程 A 相关代码移出内存，并读入进程 B 的信息。这就是上下文切换

### 2. 为何线程的上下文切换速度相对更快？线程间数据交换为何不需要类似 IPC 特别技术。

答：线程上下文切换过程不需要切换数据区和堆。可以利用数据区和堆交换数据。

### 3. 请从执行流角度说明进程和线程的区别。

答：进程：在操作系统构成单独执行流的单位。线程：在进程内部构成单独执行流的单位。线程为了保持多条代码执行流而隔开了栈区域。

### 4. 下面关于临界区的说法错误的是？

答：下面加粗的选项为说法正确。（全错）

1. 临界区是多个线程同时访问时发生问题的区域
2. 线程安全的函数中不存在临界区，即便多个线程同时调用也不会发生问题

3. 1 个临界区只能由 1 个代码块，而非多个代码块构成。换言之，线程 A 执行的代码块 A 和线程 B 执行的代码块 B 之间绝对不会构成临界区。
  4. 临界区由访问全局变量的代码构成。其他变量中不会发生问题。
5. 下列关于线程同步的说法错误的是？

答：下面加粗的选项为说法正确。

1. 线程同步就是限制访问临界区
2. 线程同步也具有控制线程执行顺序的含义
3. 互斥量和信号量是典型的同步技术
4. 线程同步是代替进程 IPC 的技术。

6. 请说明完全销毁 Linux 线程的 2 种办法

答：①调用 `pthread_join` 函数②调用 `pthread_detach` 函数。第一个会阻塞调用的线程，而第二个不阻塞。都可以引导线程销毁。

## 第 19 章 Windows 平台下线程的使用

---

暂略

## 第 20 章 Windows 中的线程同步

---

暂略

## 第 21 章 异步通知 I/O 模型

---

暂略

## 第 22 章 重叠 I/O 模型

---

暂略

## 第 23 章 IOCP

---

暂略

## 第 24 章 制作 HTTP 服务器端

---

本章代码，在[TCP-IP-NetworkNote](#)中可以找到。

### 24.1 HTTP 概要

本章将编写 HTTP（HyperText Transfer Protocol，超文本传输协议）服务器端，即 Web 服务器端。

#### 24.1.1 理解 Web 服务器端

web服务器端就是要基于 HTTP 协议，将网页对应文件传输给客户端的服务器端。

#### 24.1.2 HTTP

无状态的 Stateless 协议

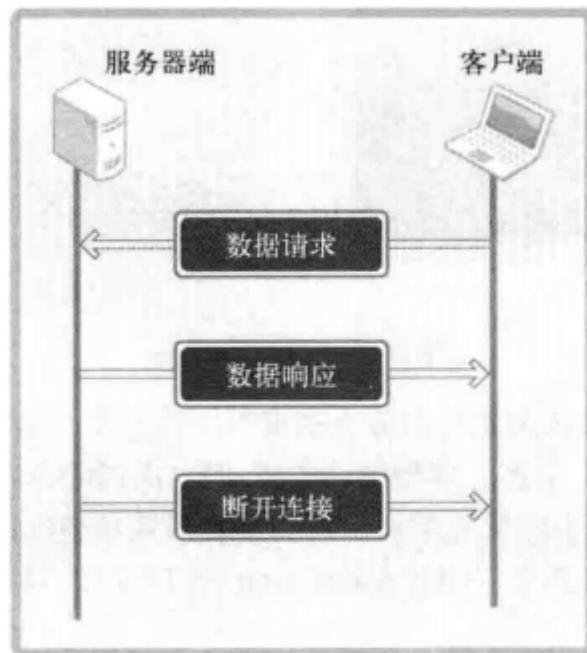


图24-1 HTTP请求/响应过程

从上图可以看出，服务器端相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」。

### 24.1.3 请求消息（Request Message）的结构

下面是客户端向服务端发起请求消息的结构：

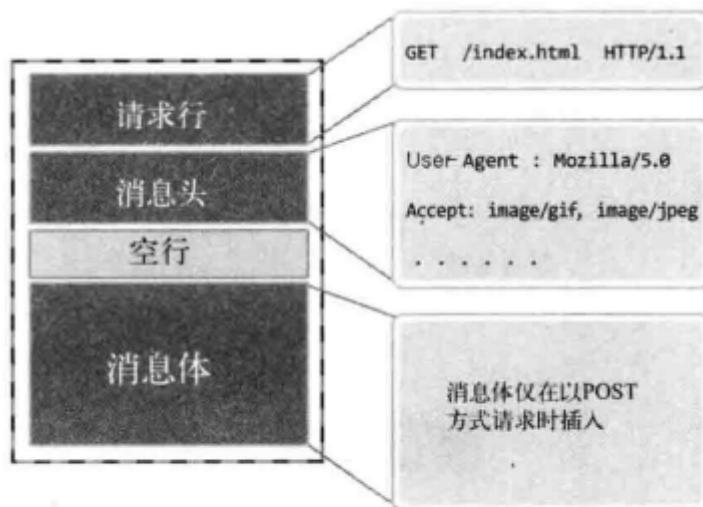


图24-2 HTTP请求头

从图中可以看出，请求消息可以分为请求头、消息体 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，通常以 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，并分别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端传输的数据，为了装入数据，需要以 POST 方式发送请求。但是我们的目标是实现 GET 方式的服务器端，所以可以忽略这部分内容。另外，消息体和消息头与之间以空行隔开，因此不会发生边界问题

#### 24.1.4 响应消息 (Response Message) 的结构

下面是 Web 服务器端向客户端传递的响应信息的结构。从图中可以看出，该响应消息由状态行、头信息、消息体等 3 个部分组成。状态行中有关于请求的状态信息，这是与请求消息相比最为显著地区别。

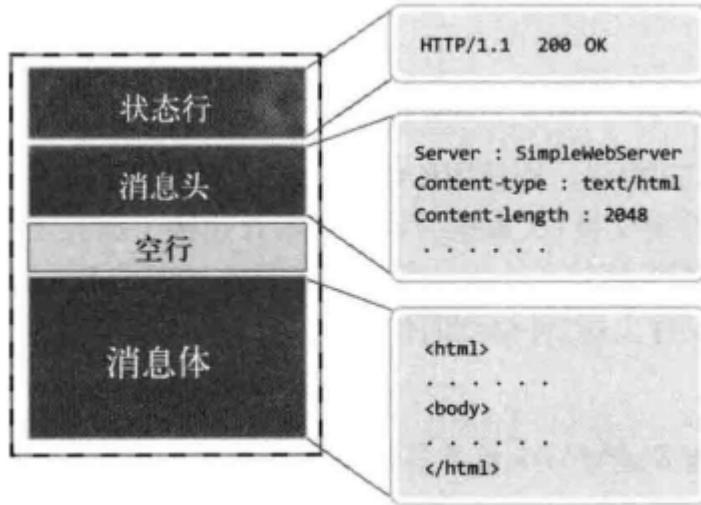


图24-3 HTTP响应头

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、服务端是否发生问题而无法响应等不同情况的信息写入状态行。图中的「HTTP/1.1 200 OK」具有如下含义：

- 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。图中的消息头含有如下信息：

服务端名为 SimpleWebServer，传输的数据类型为 text/html。数据长度不超过 2048 个字节。

最后插入一个空行后，通过消息体发送客户端请求的文件数据。以上就是实现 Web 服务端过程中必要的 HTTP 协议。

### 24.2 实现简单的 Web 服务器端

#### 24.2.1 实现基于 Windows 的多线程 Web 服务器端

暂略

#### 24.2.2 实现基于 Linux 的多线程 Web 服务器端

下面是代码：

- [webserv\\_linux.c](#)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <arpa/inet.h>
6 #include <sys/socket.h>
7 #include <pthread.h>
8
9 #define BUF_SIZE 1024
10#define SMALL_BUF 100
11
12 void *request_handler(void *arg);
13 void send_data(FILE *fp, char *ct, char *file_name);
14 char *content_type(char *file);
15 void send_error(FILE *fp);
16 void error_handling(char *message);
17
18 int main(int argc, char *argv[])
19 {
20     int serv_sock, clnt_sock;
21     struct sockaddr_in serv_addr, clnt_addr;
22     int clnt_addr_size;
23     char buf[BUF_SIZE];
24     pthread_t t_id;
25     if (argc != 2)
26     {
27         printf("Usage : %s <port>\n", argv[0]);
28         exit(1);
29     }
30
31     serv_sock = socket(PF_INET, SOCK_STREAM, 0);
32     memset(&serv_addr, 0, sizeof(serv_addr));
33     serv_addr.sin_family = AF_INET;
34     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
35     serv_addr.sin_port = htons(atoi(argv[1]));
36     if (bind(serv_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == -1)
37         error_handling("bind() error");
38     if (listen(serv_sock, 20) == -1)
39         error_handling("listen() error");
40
41     while (1)
42     {
43         clnt_addr_size = sizeof(clnt_addr);
44         clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr, &clnt_addr_size);
45         printf("Connection Request : %s:%d\n",
46                inet_ntoa(clnt_addr.sin_addr), ntohs(clnt_addr.sin_port));
47         pthread_create(&t_id, NULL, request_handler, &clnt_sock);
48         pthread_detach(t_id);
49     }
50     close(serv_sock);
51     return 0;
52 }
53

```

```
54 void *request_handler(void *arg)
55 {
56     int clnt_sock = *((int *)arg);
57     char req_line[SMALL_BUF];
58     FILE *clnt_read;
59     FILE *clnt_write;
60
61     char method[10];
62     char ct[15];
63     char file_name[30];
64
65     clnt_read = fdopen(clnt_sock, "r");
66     clnt_write = fdopen(dup(clnt_sock), "w");
67     fgets(req_line, SMALL_BUF, clnt_read);
68     if (strstr(req_line, "HTTP/")) == NULL)
69     {
70         send_error(clnt_write);
71         fclose(clnt_read);
72         fclose(clnt_write);
73         return;
74     }
75     strcpy(method, strtok(req_line, " /"));
76     strcpy(file_name, strtok(NULL, " /"));
77     strcpy(ct, content_type(file_name));
78     if (strcmp(method, "GET") != 0)
79     {
80         send_error(clnt_write);
81         fclose(clnt_read);
82         fclose(clnt_write);
83         return;
84     }
85     fclose(clnt_read);
86     send_data(clnt_write, ct, file_name);
87 }
88 void send_data(FILE *fp, char *ct, char *file_name)
89 {
90     char protocol[] = "HTTP/1.0 200 OK\r\n";
91     char server[] = "Server:Linux Web Server \r\n";
92     char cnt_len[] = "Content-length:2048\r\n";
93     char cnt_type[SMALL_BUF];
94     char buf[BUF_SIZE];
95     FILE *send_file;
96
97     sprintf(cnt_type, "Content-type:%s\r\n\r\n", ct);
98     send_file = fopen(file_name, "r");
99     if (send_file == NULL)
100    {
101        send_error(fp);
102        return;
103    }
104
105    //传输头信息
106    fputs(protocol, fp);
```

```

107     fputs(server, fp);
108     fputs(cnt_len, fp);
109     fputs(cnt_type, fp);
110
111     //传输请求数据
112     while (fgets(buf, BUF_SIZE, send_file) != NULL)
113     {
114         fputs(buf, fp);
115         fflush(fp);
116     }
117     fflush(fp);
118     fclose(fp);
119 }
120 char *content_type(char *file)
121 {
122     char extension[SMALL_BUF];
123     char file_name[SMALL_BUF];
124     strcpy(file_name, file);
125     strtok(file_name, ".");
126     strcpy(extension, strtok(NULL, "."));
127
128     if (!strcmp(extension, "html") || !strcmp(extension, "htm"))
129         return "text/html";
130     else
131         return "text/plain";
132 }
133 void send_error(FILE *fp)
134 {
135     char protocol[] = "HTTP/1.0 400 Bad Request\r\n";
136     char server[] = "Server:Linux Web Server \r\n";
137     char cnt_len[] = "Content-length:2048\r\n";
138     char cnt_type[] = "Content-type:text/html\r\n\r\n";
139     char content[] = "<html><head><title>NETWORK</title></head>"
140                     "<body><font size=+5><br>发生错误！ 查看请求文件名和请求方式！"
141                     "</font></body></html>";
142     fputs(protocol, fp);
143     fputs(server, fp);
144     fputs(cnt_len, fp);
145     fputs(cnt_type, fp);
146     fflush(fp);
147 }
148 void error_handling(char *message)
149 {
150     fputs(message, stderr);
151     fputc('\n', stderr);
152     exit(1);
153 }
```

编译运行：

```

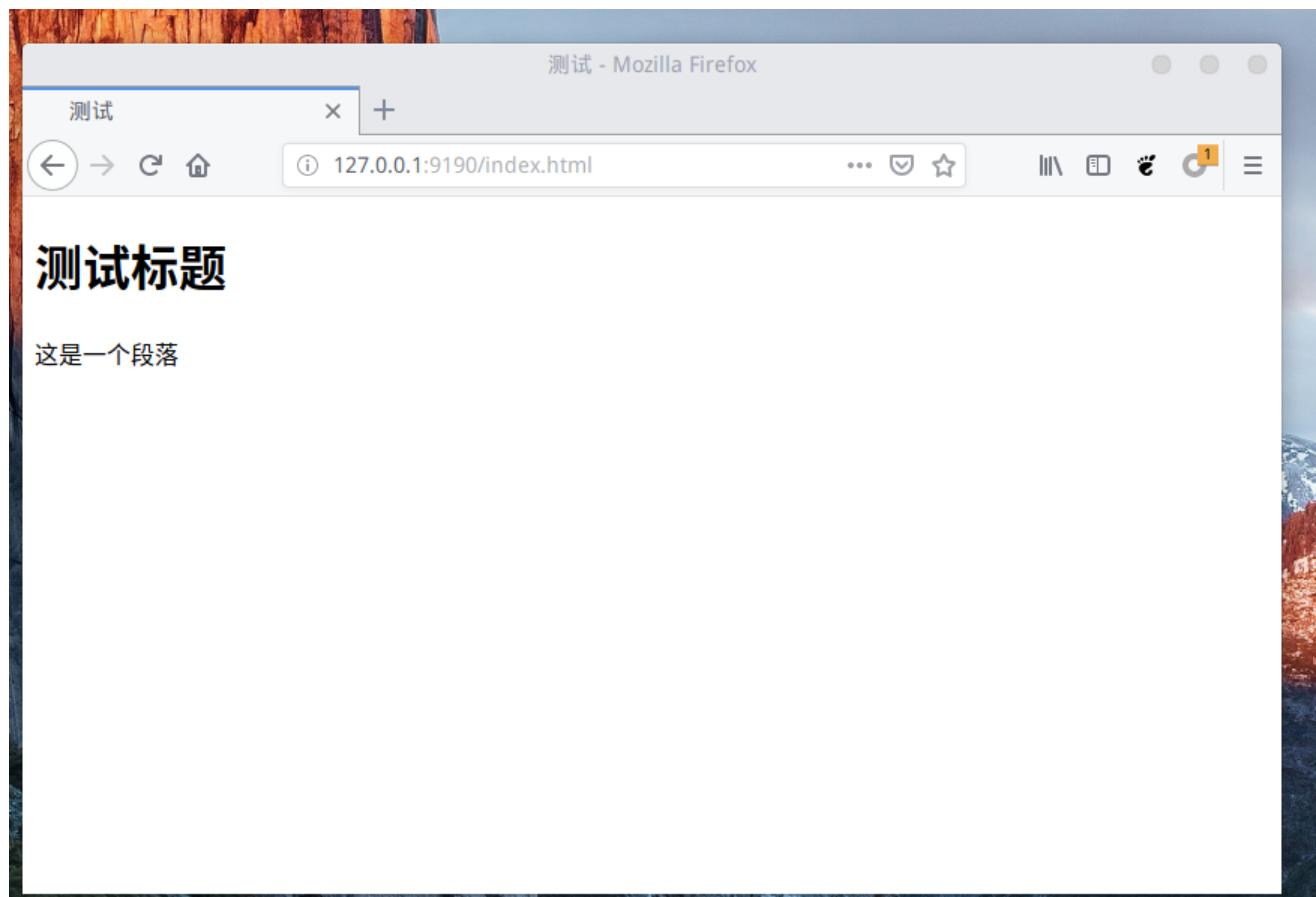
1 | gcc webserve_linux.c -D_REENTRANT -o web_serv -lpthread
2 | ./web_serv 9190
```

结果：

```
# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch24 on git:master x [19:02:20]
$ gcc webserv_linux.c -D REENTRANT -o web_serv -lpthread
webserv_linux.c: In function 'request_handler':
webserv_linux.c:73:9: warning: 'return' with no value, in function returning non-void
    return;
    ^
webserv_linux.c:54:7: note: declared here
void *request_handler(void *arg)
^~~~~~
webserv_linux.c:83:9: warning: 'return' with no value, in function returning non-void
    return;
    ^
webserv_linux.c:54:7: note: declared here
void *request_handler(void *arg)
^~~~~~

# riba2534 @ hpc in ~/Documents/MyGitProject/TCP-IP-NetworkNote/ch24 on git:master x [19:02:22]
$ ./web_serv 9190

```



经过测试，这个简单的 HTTP 服务器可以正常的显示出页面。

## 24.3 习题

以下答案仅代表本人个人观点，可能不是正确答案。

1. 下列关于 Web 服务器端和 Web 浏览器端的说法错误的是？

答：以下加粗选项代表正确。

1. **Web 浏览器并不是通过自身创建的套接字连接服务端的客户端**
  2. Web 服务器端通过 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
  3. 超文本与普通文本的最大区别是其具有可跳转的特性
  4. Web 浏览器可视为向浏览器提供请求文件的文件传输服务器端
  5. 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？

答：以下加粗选项代表正确。

1. HTTP 协议是无状态的 **Stateless** 协议，不仅可以通过 TCP 实现，还可以通过 UDP 来实现
2. **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，则意味着需要经过 **3** 次套接字的创建过程。
3. 服务端向客户端传递的状态码中含有请求处理结果的信息。
4. **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** 协议被设计为 **Stateless** 协议。

我的笔记到此结束 ☺



本仓库遵循 CC BY-NC-SA 4.0（署名 - 非商业性使用）协议，转载请注明出处。

