高等学校计算机科学与技术教材

TCP/IP 网络编程技术基础

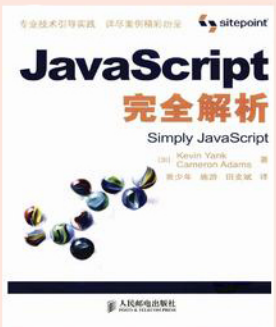
COMPUTER Science and Technology

□ 王雷 编著

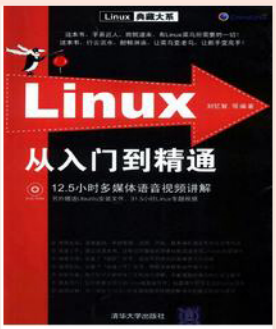
- 原理与技术的完美结合
- 教学与科研的最新成果
- 语言精练，实例丰富
- 可操作性强，实用性突出

清华大学出版社 ● 北京交通大学出版社

热读推荐



免费下载



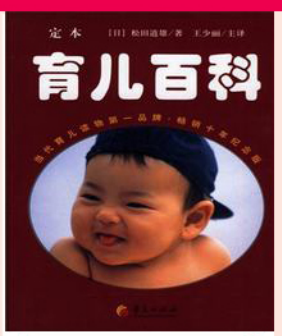
免费下载

免费下载

免费下载

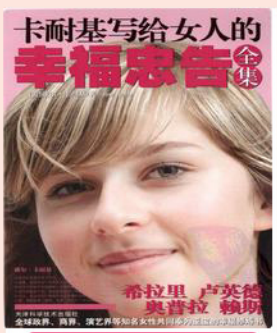
免费下载

免费下载



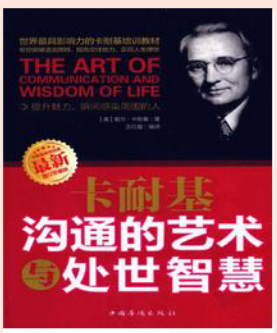
免费下载

免费下载

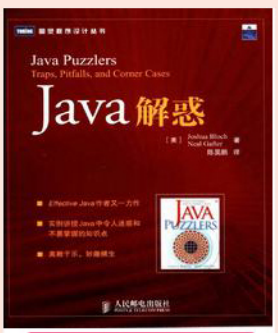


免费下载

免费下载



免费下载



免费下载

高等学校计算机科学与技术教材

TCP/IP 网络编程技术基础

王 雷 编著

清华大学出版社
北京交通大学出版社
·北京·

内 容 简 介

本书是一本基于TCP/IP协议进行计算机网络编程的教科书。全书通过原理介绍与例程剖析的形式，系统介绍了LINUX环境下如何使用C语言基于TCP/IP协议进行网络编程的详细步骤与过程。

与国内外出版的同类教材相比，本书主要的特点为：在注重阐述TCP/IP网络通信原理与套接字API编程原理的基础上，通过对例程的深入剖析，深入浅出地介绍服务器与客户软件的编程技巧；同时，在章节的编排上更加富有衔接性。本书第1章和第2章主要介绍TCP/IP网络通信原理与套接字API编程原理，第3章和第4章主要介绍循环服务器软件的设计方法，第5章介绍服务器的并发机制，第6章到第8章则主要介绍并发服务器的设计方法，第9章主要介绍服务器并发性的统一与高效管理技术，第10章主要介绍客户进程中的并发机制，第11章主要介绍客户－服务器系统中的死锁问题，第12章则介绍了GCC编译器的安装与使用方法，整个12章按照“原理→循环服务器软件设计→并发服务器软件设计→并发客户软件设计→客户－服务器系统中的死锁问题→客户－服务器软件编译环境”的顺序，通过C语言例程剖析，由浅入深地介绍了基于TCP/IP协议进行计算机网络编程的方法。通过以上连贯的章节编排，使得读者能够更加简洁、系统地掌握网络编程技术。

本书可供计算机与通信专业的本科生、从事计算机网络编程的技术人与网络编程爱好者使用，同时，也可供其他专业的学生、计算机网络技术的爱好者，以及计算机应用技术相关的工程技术人员参考。

图书在版编目(CIP)数据

TCP/IP网络编程技术基础/王雷编著．—北京：清华大学出版社；北京交通大学出版社，2012.3
（高等学校计算机科学与技术教材）
ISBN 978－7－5121－0903－2

Ⅰ．①T…　Ⅱ．①王…　Ⅲ．①计算机网络－通信协议－高等学校－教材　②计算机网络－网络编程－高等学校－教材　Ⅳ．①TN915.04　②TP393.09

中国版本图书馆CIP数据核字（2012）第018322号

责任编辑：谭文芳　　特邀编辑：胡花蕾
出版发行：清华大学出版社　　邮编：100084　　电话：010－62776969
　　北京交通大学出版社　　邮编：100044　　电话：010－51686414
印 刷 者：北京交大印刷厂
经　　销：全国新华书店
开　　本：185×260　　印张：12.25　　字数：313千字
版　　次：2012年3月第1版　　2012年3月第1次印刷
书　　号：ISBN 978－7－5121－0903－2/TN·82
印　　数：1～4 000册　　定价：23.00元

前　言

背景动机

现代社会是信息社会，随着 Internet 在全球范围内的迅速普及，网络对人们的学习、工作、生活以及对社会的影响越来越大。计算机网络技术被誉为是“近代最深刻的技术革命”，人们用“网络时代”、“网络经济”等术语来描述计算机网络对社会信息化与经济发展的影响。

目前，国内各主要高校的计算机应用技术与软件工程专业本科生均开设了“TCP/IP 网络编程技术基础”这一门课程，但传统的教材缺乏对所给例程的深入剖析，从而导致初学者在采用这些教材进行学习时难以轻松掌握所学内容。为此，本书在作者多年讲授 TCP/IP 网络编程技术课程的基础上，首先在传统教材所介绍的 Socket 网络编程相关概念与技术的基础之上，进行了大幅度的内容增减与结构调整；其次，为了使不同层次的读者均能够更加方便地掌握所学内容，本书还在各章节中对所给出的例程新增了全面深入的剖析；最后，本书还新增了对 GCC 编译器的有关介绍，使得全书内容更加完整。

上述这些原因构成了编著本书的一个主要背景动机。

目标读者

本书的目标读者包括计算机相关专业的本科生与研究生、计算机网络编程技术与 C 语言的爱好者，以及计算机应用技术相关的工程技术人员。

组织结构

考虑到读者在阅读本书之前对计算机网络编程技术的了解程度不尽相同，特将本书分为以下四大部分，其中：

① 第 1 章和第 2 章为第一部分，主要介绍 TCP/IP 网络通信原理与套接字 API 编程原理；

② 第 3 章至第 9 章为第二部分，其中，第 3 章主要介绍循环服务器软件的设计方法，第 4 章介绍服务器的并发机制，第 5 章到第 8 章则主要介绍并发服务器的设计方法，第 9 章主要介绍服务器并发性的统一与高效管理技术；

③ 第 10 章至第 11 章为第三部分，其中，第 10 章主要介绍客户进程中的并发机制，第 11 章主要介绍客户 - 服务器系统中的死锁问题；

④ 第 12 章为第四部分，主要介绍了 GCC 编译器的安装与使用方法。

编　者

2011 年 12 月

目　录

第 1 章　TCP/IP 网络通信原理

TCP/IP 协议是实现网络通信的基础，本章将在简要介绍 TCP/IP 协议及 TCP/IP 参考模型的基础上，深入介绍 TCP/IP 网络通信的基本原理与 TCP/IP 网络通信中所采用的客户 - 服务器通信模型，然后在此基础上，进一步系统介绍客户 - 服务器通信模型中的客户软件与服务器软件的设计流程。

1.1　TCP/IP 协议概述

1.1.1　TCP/IP 参考模型

TCP/IP（Transmission Control Protocol/Internet Protocol），即传输控制协议/因特网协议，是一个由多种协议组成的协议族（Protocol Family），定义了计算机通过网络互相通信及协议族各层次之间通信的规范。

TCP/IP 参考模型是一个抽象的分层模型，这个模型中，属于 TCP/IP 协议族的所有网络协议都被归类到了以下四个抽象的“层”之中。

1. 主机 - 网络层（Host to Network Layer）

主机 - 网络层是 TCP/IP 参考模型的最低层，也称为网络接口层，它主要负责接收从互联网络层交来的 IP 数据报并将其通过低层物理网络发送出去，或者从低层物理网络上接收物理帧并从中抽出 IP 数据报交给互联网络层。其中，网络接口主要有以下两种类型：第一种是设备驱动程序，如局域网的网络接口；第二种是含自身数据链路协议的复杂子系统。在 TCP/IP 参考模型中未定义数据链路层，这主要是因为在 TCP/IP 最初的设计中已经使其可以使用各种典型的数据链路层协议。

2. 互联网层（Internet Layer）

也称为网际互联层或 IP 层，主要负责将源主机的报文分组发送到目的主机，源主机与目的主机可以在一个网络上，也可以在不同的网络上。由于 TCP/IP 参考模型中网络层协议是 IP 协议，因此互联网络层也称为 IP 层。其中，IP 协议是一种不可靠、无连接的数据报传送服务的协议，它提供的是一种“尽力而为（Best Effort）”的服务。IP 协议的协议数据单元是 IP 分组，由于在 IP 层提供数据报服务，因此，也常将 IP 分组称为 IP 数据报。

3. 传输层（Transport Layer）

传输层主要负责在互联网中源主机与目的主机的对等进程实体之间提供可靠的端到端的数据传输。在 TCP/IP 参考模型的传输层中定义了以下这两种协议。

（1）TCP 协议。TCP 协议是一种可靠的面向连接的协议，它允许将一台主机的字节流（Byte Stream）无差错地传送到目的主机。TCP 协议将应用层的字节流分成多个字节段

(Byte Segment)，然后将一个个的字节段传送到互联网络层，并最终发送到目的主机。当互联网络层将接收到的字节段传送给传输层时，传输层再将多个字节段还原成原始的字节流，并传送到应用层。TCP 协议同时要完成流量控制功能，协调收发双方的发送与接收速度，以达到正确传输的目的。

(2) UDP 协议 (User Datagram Protocol，用户数据报协议)。UDP 协议是一种不可靠的无连接协议，它主要用于不要求分组顺序到达的传输服务之中，在基于 UDP 协议的传输服务中，分组的传输顺序检查与排序由应用层完成。UDP 协议主要面向请求 - 应答式的交易型应用，一次交易往往只有一来一回两次报文交换，假如为此而建立和撤销连接将导致网络开销过大，因此，在这种情况下使用 UDP 就非常有效。另外，UDP 协议也常用于那些对可靠性要求不高，但要求网络的延迟较小的场合，如语音和视频数据的传送等。

4. 应用层 (Application Layer)

应用层包括了所有的高层协议，目前 TCP/IP 参考模型中的应用层协议主要包括以下几种：

(1) 网络终端协议 Telnet；
(2) 文件传输协议 (File Transfer Protocol，FTP)；
(3) 简单邮件传输协议 (Simple Mail Transfer Protocol，SMTP)；
(4) 域名系统 (Domain Name System，DNS)；
(5) 简单网络管理协议 (Simple Network Management Protocol，SNMP)；
(6) 超文本传输协议 (Hyper Text Transfer Protocol，HTTP)。

1.1.2 TCP/IP 参考模型的通信原理

TCP/IP 参考模型的通信原理如图 1.1 所示，其中，第一至二层为串联的，而第三至四层则是端到端 (End to End) 的。

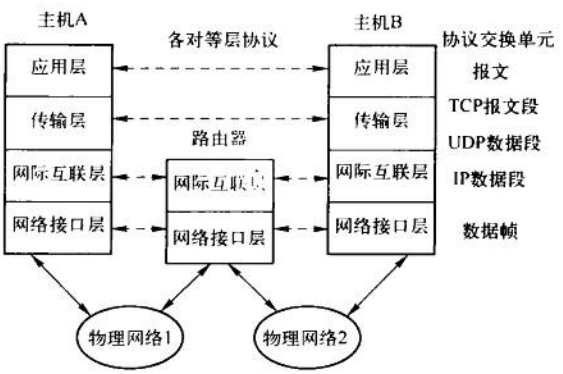


图 1.1 TCP/IP 参考模型的通信原理

由图 1.1 可知，网际互联层与网络接口层实现了计算机网络中处于不同位置的主机之间的数据通信，但是数据通信不是计算机网络的最终目的，计算机网络最本质的活动是实现分布在不同地理位置的主机之间的进程通信，以实现各种网络服务功能。而设置传输层的主要目的就是要实现上述这种分布式进程之间的通信功能。

在单机系统中，由于每个进程都在自己的地址范围内运行，为保证两个相互通信的进程之间既互不干扰又协调一致工作，LINUX 操作系统为进程之间的通信提供了相应的设施，如管道（Pipe）、命名管道（Named Pipe）、软中断信号（Signal）和信号量（Semaphore）等，但上述这些设施都仅限于用在本机进程之间的通信。而网间分布式进程通信要解决的是不同主机进程间的相互通信问题（同机进程通信只是其中的一个特例）。为此，传输层需要解决在网络环境中分布式进程通信所需解决的以下两个方面的问题。

（1）进程命名与寻址。在同一台计算机中，每一个进程均被分配了一个唯一的端口（Port）与端口号（Port Number，也称为进程标识），其中，端口是一个信息缓存区，用于保留 Socket 中的输入/输出信息，端口号是一个 16 位无符号整数，范围是 0 ~ 65 535，以区别主机上的每一个应用程序进程（如果将端口类比为房间，则端口号就是房间号。端口号一般有以下两种基本的分配方式：第一种为全局分配，这是一种集中分配方式，由一个公认权威的中央机构根据用户的需要进行统一分配，并将结果公布于众，通过这种方式分配的端口号也称为熟知端口号；第二种为本地分配，又称动态连接，是当进程需要访问传输层服务时，向本地操作系统提出申请，再由操作系统返回本地唯一的端口号）。由于同一台机器上的不同进程所分配到的端口号不同，因此，同一台机器上的不同进程就可以用端口号来唯一标识；但在网络环境中，显然，若要标识一个完整的进程，除了端口号之外，还需使用到本地主机的 IP 地址（本地地址）来唯一标识进程所在的本地主机（这是由于不同机器上的进程可以拥有相同的端口号）。

（2）多重协议的识别。由于 LINUX 操作系统支持的网络协议众多，不同协议的工作方式与地址格式均不相同。因此在网络环境中，一个应用程序进程最终需要使用一个三元组 < 协议，本地地址，本地端口号 > 来唯一地标识；另外，在 TCP/IP 网络环境中，一个完整的网间通信需要由两个进程组成，并且这两个进程之间只能使用相同的传输层协议才能进行通信，也就是说，不可能通信的一端使用 TCP 协议，而另一端使用 UDP 协议。因此，一个完整的网间通信需要用一个五元组 < 协议，本地地址，本地端口号，远程地址，远程端口号 > 才能唯一地标识。其中，二元组 < 本地地址，本地端口号 > 称为网间进程通信中的本地端点地址（End-point Address），二元组 < 远程地址，远程端口号 > 称为网间进程通信中的远程端点地址，而三元组 < 协议，本地地址，本地端口号 > 称为一个半相关（Half-Association），五元组 < 协议，本地地址，本地端口号，远程地址，远程端口号 > 则称为一个相关（Association）。

1.1.3　LINUX 系统实现网络通信的基本原理

1. LINUX 中提供的基本 I/O 功能

操作系统是一个用来和计算机硬件打交道并为用户程序提供一个有限服务集的低级支撑软件。一个计算机系统是一个由硬件和软件组成的共生体，它们互相依赖，不可分割。但是硬件若没有软件来操作和控制，它们自身是不能工作的，而完成上述控制工作的软件就称为操作系统（Operation System）。

LINUX 操作系统是最受欢迎的计算机操作系统之一，它是一个用 C 语言写成并符合 POSIX 标准的类 UNIX 操作系统。LINUX 操作系统最早是 1991 年由芬兰黑客 Linus Torvalds 为尝试在英特尔 x86 架构上提供自由免费的类 UNIX 操作系统而开发的。为了让 LINUX 系统支持网络通信功能，TCP/IP 协议被集成到了 LINUX 操作系统的内核之中。

在 LINUX 操作系统中，所有的设备都看作是一种具体的文件，LINUX 操作系统提供了以下一组（共包括六个）基本的系统函数，用来对本地设备或文件进行 I/O 操作.

(1) open( ) 函数

该函数用于打开一个文件（设备），其返回值为一个整型变量，若返回值为 -1，则表示调用 open 函数打开文件时出错；否则，则表示打开文件成功，此时该返回值也称为所打开文件的文件描述符（File Descriptor）。其函数原型如下：

```
#include < sys/types. h >      /* 基本数据类型头文件,含有基本系统数据类型的定义 */
#include < sys/stat. h >       /* 文件状态头文件,含有文件或文件系统状态结构和常量的定义 */
#include < fcntl. h >          /* 文件控制头文件,含有文件及其描述符的操作控制常数符号的
                                定义 */
int open( const char * pathname, int flags, mode_t mode);
```

在上述 open( ) 函数的原型中，各参数的含义如下。

pathname：指向欲打开的文件路径字符串。

flags：文件打开方式的标志，主要包括以下几种。

- O_RDONLY：以只读方式打开文件。
- O_WRONLY：以只写方式打开文件 。
- O_RDWR：以可读写方式打开文件。

上述三种文件打开方式标志是互斥的，不可以同时使用，但可与下列文件打开方式标志利用 OR(|)运算符进行组合。

- O_CREAT：若欲打开的文件不存在则自动建立该文件。
- O_APPEND：当写文件时会从文件尾开始移动，也就是所写入的数据会以附加的方式加入到文件末尾。
- O_TRUNC：若文件存在并且以可写的方式打开时，此标志会令文件长度清 0。

mode：被打开文件的存取权限，只有在建立新文件时才会生效，即只有当 flags 取值中有 O_CREAT 时才有效。

(2) close( ) 函数

该函数用于关闭 open( ) 函数所打开一个文件（设备），若调用成功时将返回 0；若调用出错则返回 -1。其函数原型如下：

```
#include < unistd. h > /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型、符号常数和
                        类型的定义 */
int close( int fd);
```

在上述 close( ) 函数的原型中，主要参数的含义如下。

fd：欲关闭文件的文件描述符。

(3) read( ) 函数

该函数用于从指定的文件（由文件描述符参数 fd 指定）中读取指定的字节数据（由参数 count 指定）存放到指定的缓存区（由参数 buf 指定）之中。调用成功时返回读取的字节数；若返回 0 表示已到达文件尾；若返回 -1 则表示调用出错。其函数原型如下：

```
#include < unistd. h >
```

```
ssize_t read(int fd, void * buf, size_t count);
```

在上述 read( ) 函数的原型中，各参数的含义如下。

fd：指定文件的文件描述符。

buf：指定用来存储所读取的数据的缓冲区。

count：指定要读取的字节数。在读取普通文件时，若读到要求的字节数之前已达到文件尾部，则返回的字节数将会小于希望读取的字节数 count。

注：数据类型 ssize_t 是 signed size_t，而数据类型 size_t 是标准 C 库中定义的，表示 unsigned int（无符号整形）。

(4) write( ) 函数

该函数用于向指定的文件（由文件描述符参数 fd 指定）中写入指定的字节数据（由参数 count 指定）存放到指定的缓存区（由参数 buf 指定）之中。调用成功时返回实际写入的字节数；若调用出错则返回 -1。其函数原型如下：

```
#include < unistd. h >
ssize_t write(int fd, const void * buf, size_t count);
```

在上述 write( ) 函数的原型中，各参数的含义如下。

fd：指定文件的文件描述符。

buf：指定存储写入数据的缓冲区。

count：指定写入的字节数。

(5) lseek( ) 函数

该函数用于移动文件的读写指针，更改打开文件的偏移量，实现在文件内部的定位。调用成功时返回所设置的新的偏移量；若调用出错则返回 -1。其函数原型如下：

```
#include < unistd. h >
#include < sys/types. h >
off_t lseek(int fd, off_t offset, int whence);
```

在上述 lseek( ) 函数的原型中，各参数的含义如下。

fd：文件描述符。

offset：偏移量，每一次读写操作所需移动的距离，单位是字节的数量，可正可负（为正时表示从当前基点位置向前移动，为负时表示从当前基点位置向后移动）。

whence：表示当前基点位置，取值如下。

- SEEK_SET：当前基点位置为文件的开头，新位置为偏移量的大小。
- SEEK_CUR：当前基点位置为文件指针位置，新位置为当前位置加上偏移量。
- SEEK_END：当前基点位置为文件的结尾，新位置为文件的大小加上偏移量。

(6) ioctl( ) 函数

该函数是设备驱动程序中对设备的 I/O 通道进行管理的函数。所谓对设备的 I/O 通道进行管理，就是对设备的一些特性进行控制。尽管多数情况下硬件设备的操作可以通过文件的 read、write、lseek 等操作实现，但总有一些特例，如弹出光盘、让磁带机倒带、设置声卡的采样率等，当需要对设备的上述这类特性进行控制时，就需要用到 ioctl( ) 函数。若调用成

功时将返回 0；若调用出错则返回 -1。其函数原型如下：

```
#include <sys/ioctl.h> //该头文件中包含了 I/O 控制函数的定义
int ioctl(int fd, int cmd, …);
```

在上述 ioctl( ) 函数的原型中，各参数的含义如下。

fd：文件（设备）的文件描述符。

cmd：用户程序对设备的控制命令。

注：ioctl 是一个可变长参数的函数。可变长参数函数的原型为 type VAFunction（type arg1, type arg2, …）；参数可以分为个数确定的固定参数和个数可变的可选参数两部分。函数至少需要一个固定参数，固定参数的声明和普通函数一样；可选参数由于个数不确定、可有可无，需要根据实际情况而定，声明时用“…”表示。固定参数和可选参数公同构成一个函数的参数列表。可变长参数的函数的定义方法如下例 1-1 所示。

例 1-1：求任意个自然数的平方和。

```
int SqSum(int n1, …){
    va_list arg_ptr; //定义一个指向个数可变的参数列表指针
    int nSqSum = 0, n = n1;
    va_start(arg_ptr, n1); /* va_start(arg_ptr, argN):使参数列表指针 arg_ptr 指向函数参数列表
                              中的第一个可选参数,说明:argN 是位于第一个可选参数之前的那个
                              固定参数,(即最后一个固定参数;亦即"…"之前的那个参数),函数
                              参数列表中参数在内存中的顺序与函数声明时的顺序是一致的。如
                              果有一个 va 函数的声明是 void va_test(char a, char b, char c, …),
                              则它的固定参数依次是 a,b,c,最后一个固定参数 argN 为 c,因此就
                              是 va_start(arg_ptr, c) */
    while(n>0){
        nSqSum += (n*n);
        n = va_arg(arg_ptr, int); /* va_arg(arg_ptr, type):返回参数列表中指针 arg_ptr 所指的参
                                     数,其返回类型为 type,并使得指针 arg_ptr 指向参数列表中的
                                     下一个参数 */
    }
     va_end(arg_ptr); /* va_end(arg_ptr):清空参数列表,并置参数指针 arg_ptr 无效 */
    return nSqSum;
}
```

针对上述定义的可变长参数的函数 SqSum，可通过如下方法进行调用：

```
int totalsum;
int totalsum = SqSum(7,2,7,11,-1); /* 调用 SqSum(7,2,7,11,-1)将计算 7*7+2*2+7*7+
                                      11*11 之和,当 n=-1 是将跳出 while(n>0)循环 */
```

2. LINUX 中基本 I/O 系统函数的用法示例

(1) open( )、read( )、write( )、lseek( )、close( ) 函数的用法示例

```
#include <unistd.h>        //调用 close( )、read( )、write( )、lseek( ) 函数所需的头文件
```

```
#include <sys/types.h>      //调用 open()、lseek()函数所需的头文件
#include <sys/stat.h>       //调用 open()函数所需的头文件
#include <fcntl.h>          //调用 open()函数所需的头文件
#include <stdlib.h>         /*C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()等)原型
                              的定义*/
#include <stdio.h>          /*标准输入输出头文件,包含了标准输入输出函数(如 perror()和 printf
                              ()等)的定义*/
#include <string.h>         /*字符串头文件,包含了字符串操作函数(如 strlen()等)的定义*/
int main(void){
    char *buf="Hello! I'm writing to this file!";/*定义了一个指向包含有 10 个字符的字符常
                                                   量"Hello! I'm writing to this file!"的字符型
                                                   指针*/
    char buf_r[11];/*定义了一个长度为 11 个字符的字符型数组变量,字符在计算机中以其
                     ASCII 码方式表示,其长度为 1 个字节,因此,计算机在编译时将留出连
                     续 11 个字节的空间,即 buf_r[0]到 buf_r[10]共 10 个字符变量,但只有
                     前 10 个变量可供用户使用,第 11 个字符变量 buf_r[10]将用来存放字
                     符串终止符 NULL 即"\0",其中,终止符是编编译程序自动加上的*/
```

注：字符型指针与字符型数组变量是不同的，当两者在用 sizeof 取长度时，在 32 位机器下，字符型指针的长度为 4 个字节，如 sizeof(buf)的长度为 4；而字符型数组变量的长度为其中缓存的字符串的长度，如 sizeof(buf_r)的长度为 11。

```
    int fd, size, len;
    len=strlen(buf);            /* strlen 函数的原型为 int strlen(char *s);在头文件<string.h>中定
                                   义,其功能为计算字符型指针 s 所指向的字符串的长度*/
    if((fd=open("/tmp/hello.c", O_CREAT | O_TRUNC | O_RDWR,0666))<0){
    //调用 open 函数打开 hello.c 文件,并指定打开方式与相应的操作权限
        perror("open failed:");     /* perror 函数的原型为 void perror(const char *s);打印 s
                                       所指的字符串和标准出错信息,相当于 printf("%s: %s
                                       \n", *s, strerror(errno));调用该函数需要包括头文
                                       件<stdlib.h>和 <errno.h>*/
        exit(1);                    /* exit 的函数原型为 void exit(int status);其功能是中止程序执行并
                                       返回一个状态值 status 给操作系统,status 为 0 时表示正常终止,为
                                       -1 表示异常终止*/
    } else
        printf("open and create file:hello.c with file descriptor %d OK\n",fd);
                                    /* printf 函数的原型为 int printf(const char *format, …);在头文件
                                       <stdio.h>中定义,是一种可变长参数的函数,参数 format 表示如
                                       何来格式字符串的指令*/
    if((size=write(fd, buf, len))<0){       /*调用 write 函数,将 buf 中的内容写入到打开的
                                              文件中*/
        perror("write:");
        exit(1);
    } else
```

```
        printf("Write:%s OK\n",buf);
    lseek(fd, 0, SEEK_SET ); //定位到文件的开头位置
    if((size=read( fd, buf_r, 10))<0){      /*调用 read 函数从文件中读取 10 个字节到缓存区
                                              buf_r 中*/
        perror("read failed:");
        exit(1);
    } else
        printf("read form file:%s OK\n",buf_r);
    if(close(fd)<0 ){
        perror("close failed:");
        exit(1);
    } else printf("Close hello.c OK\n");
        exit(0);
    }
```

(2) ioctl()函数的用法示例

声卡是普通个人计算机的标准设备。在 LINUX 下，有几种设备文件用来控制声卡的功能。一种是声音混合设备/dev/mixer，用来控制各个声道的音量；另一种设备是声音的采集和播放设备，包括/dev /audio、/dev/dsp 等，以下例程介绍了如何通过对/dev/audio 编程来对声卡的采样频率进行设置。

声卡编程必须包含以下头文件，因为这些头文件中包含了必须的函数声明和变量说明：

```
#include <sys/ioctl.h> //提供对 I/O 控制的函数原型的定义
#include <unistd.h>
#include <sys/soundcard.h>      /*提供对声卡支持的所有采样格式和采样频率等的定义*/
```

对声卡采样频率进行设置：

```
int audio_fd;
int format;
audio_fd=open("/dev/audio", O_WRONLY, 0); //打开声音采集/播放设备
format=AFMT_S16_LE;                       //设置 16 位采样频率 AFMT_S16_LE
ioctl(audio_fd, SNDCTL_DSP_SETFMT, &format);
            // SNDCTL_DSP_SETFMT 命令用于设置声卡的采样频率
```

3. 将 LINUX 中的基本 I/O 功能用于 TCP/IP 网络通信

为了访问 TCP/IP 协议，LINUX 操作系统对上述传统的基本 I/O 调用进行了扩展。首先，LINUX 操作系统扩展了文件描述符集，使得应用程序进程既可以创建用于本机设备访问的文件描述符，又可以创建能被 TCP/IP 网络通信所使用的套接字描述符（Socket Dsecriptor）。其次，LINUX 操作系统还扩展了 read 和 write 两个系统调用，使得其既可以同套接字描述符一起使用，又可以同普通的文件描述符一起使用。这样一来，当应用程序进程需要通过 TCP/IP 网络收发数据时，就可以创建相应的套接字描述符，然后再使用 read 和 write 系统调用来通过 TCP/IP 网络进行数据收发了。

其中，所谓的套接字描述符，就是一个整数类型的值，与文件描述符是用于标识一个活

动的文件一样，套接字描述符则是用于标识一个活动的套接字（Socket）。而所谓的套接字，则是 LINUX 操作系统为实现 TCP/IP 网络通信所新增加的一个抽象，类似于 UNIX 中 Pipe（管道），通信双方进程可通过它来与对方发送或接受数据。如图 1.2 所示，套接字可以看成是在两个应用程序进程进行通信连接中的一个端点：当主机 A 上的应用程序进程 A 需要和主机 B 上的应用程序进程 B 进行通信时，主机 A 上的应用程序进程 A 首先将一段信息写入其在本地主机上的 Socket A，然后由 Socket A 将该段信息通过 TCP/IP 网络发送到主机 B 上应用程序进程 B 所对应的 Socket B 中，最后再由 Socket B 将该段信息传送给应用程序进程 B。

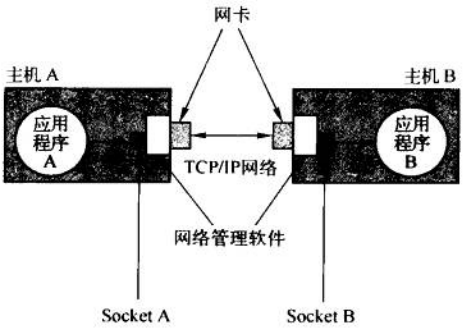


图 1.2　TCP/IP 模型的通信原理

由于在网络环境中，一个完整的网络通信连接需要由两个应用程序进程组成，而一个应用程序进程则是用一个半相关 < 协议，本地地址，本地端口号 > 来唯一地标识的，一个套接字又可以看成是在这两个应用程序进程进行通信连接中的一个端点，因此，一个套接字显然也可以用上述半相关 < 协议，本地地址，本地端口号 > 来唯一地标识，其中，二元组 < 本地地址，本地端口号 > 通常也称为套接字的端点地址。

显然，由以上描述可知，一个完整的 TCP/IP 网络通信连接可用通信双方所对应的套接字所组成的套接字对（Socket Pair）来唯一地标识，其中一个套接字运行于客户端，称为客户端套接字（Client Socket），而另一个套接字则运行于服务器端，称为服务器套接字（Server Socket）。在实际网络通信中，将由客户端套接字提出连接请求，而要连接的目标就是服务器端套接字。为此，客户端套接字必须首先描述它要连接的服务器套接字，指出服务器端套接字的端点地址（即服务器的 IP 地址和服务器进程的端口号），然后，它才能向服务器端套接字提出连接请求。这也就是说，服务器套接字的端点地址必须预先被客户端知道，即：服务器必须使用熟知的（Well-Known）端口号。

1.2　TCP/IP 网络通信中的客户 - 服务器模型

1.2.1　客户 - 服务器模型

如图 1.3 所示，在 TCP/IP 协议体系中，进程间的相互作用采用客户 - 服务器（Client-Server）模型；其中，客户与服务器分别表示相互通信的两个应用程序进程。在客户 - 服务器模型中，是根据通信发起的方向来区别一个应用程序进程是客户还是服务器的，一般来

说，发起对等通信的应用程序进程称为客户，而负责等待接收客户通信请求并为客户提供服务的应用程序进程则称为服务器。

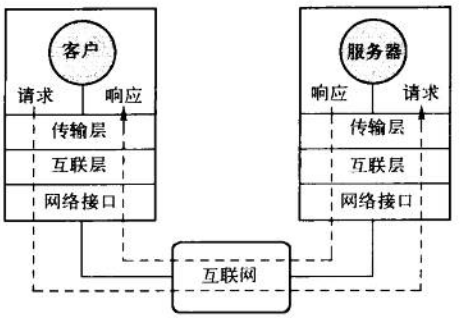


图 1.3 客户 - 服务器交互模型

1.2.2 客户 - 服务器模型中的汇聚点问题及其解决方法

在 TCP/IP 网络通信中，由于参与通信的两个应用程序进程一般位于两台处于不同地理位置的独立的计算机之上，因此将可能会导致汇聚点（Rendezvous）问题，即：当两个人在分别启动这两个处于不同地理位置的独立的计算机上的应用程序进程时，由于两个人的操作速度不同，而计算机的运行速度要比人快许多数量级，这将导致速度快的那个人在启动了一个应用程序进程之后，该应用程序进程开始执行并向其对等应用程序进程发送报文，而此时速度慢的那个人还未完成其对等应用程序进程的启动，如此一来，在几微秒之内，先启动的应用程序进程将会判断出其对等应用程序进程还不存在，于是，它将发出一条错误消息然后退出。这时，假若第二个应用程序进程正启动，它将发现其对等应用程序进程已经终止执行。按照上述方式，即便是两个应用程序进程继续尝试通信，但由于每个应用程序进程都执行得相当快，因此，在同一时刻双方能相互发送消息的概率还是会非常的低。

为了解决上述汇聚点问题，在客户 - 服务器模型中，要求每一次通信均由客户进程随机启动，而服务器进程则必须处于无限循环等待状态，以等待来自客户的服务请求，并在接收到客户的请求之后，执行必要的计算，然后再将结果返回给客户。其中，发起对等通信的一方称为客户，无限期地等待接收客户通信请求的一方则称为服务器。

1.2.3 客户 - 服务器模型中服务器设计与实现的复杂性

为了完成计算和返回结果给客户，服务器软件通常需要访问受操作系统保护的对象（如文件、数据库、设备或协议端口等），因此，服务器软件的执行通常需要带有一些系统特权。由于服务器在执行时带有系统特权，因此应特别注意不要将特权传递给使用其所提供的服务的客户。由于服务器的特权允许它访问任何文件，因此，为了保障服务器端的安全，服务器不能只依赖于那些常规的操作系统检查，通常，与客户软件不同，服务器软件除了接收客户的服务请求并返回应答给客户之外，还应含有处理以下安全问题的代码：

（1）鉴别——验证客户的身份；

（2）授权——判断某个客户是否被允许访问服务器所提供的服务；

（3）数据安全——确保数据不被无意泄露或损坏；

（4）保密——防止未经授权访问信息；

（5）保护——确保网络应用程序不能滥用系统资源。

显然，由于服务区具有系统特权并应包含处理上述安全问题的代码，从而导致了服务器软件的设计与实现比客户软件的设计与实现要更加的困难和复杂。

1.2.4　服务器中的并发问题

并发（Concurrency）是指真正的或表面呈现的同时计算。通常，一个多用户的计算机系统可以通过分时（Time Sharing）或多处理器（Multiprocessing）来获得并发，其中，分时机制是使得单个处理器在多个计算任务（或多个用户）之间快速地切换，从而使得从表面上来看这些计算（或这些用户所获得的服务）是同时进行的；而多处理器机制则是让多个处理器同时执行多个任务，因此，所实现的是真正的同时计算（即真正的并发）。

在客户－服务器模型中，很多时候会有多个客户使用服务器的一个熟知协议端口与服务器联系，如图 1.4 所示，在一台主机上有可能会运行有多个服务器进程，并且每个服务器进程也可能需要及时处理多个客户的请求，并将处理的结果返回给客户，因此，服务器软件还必须在设计中处理好并发请求。

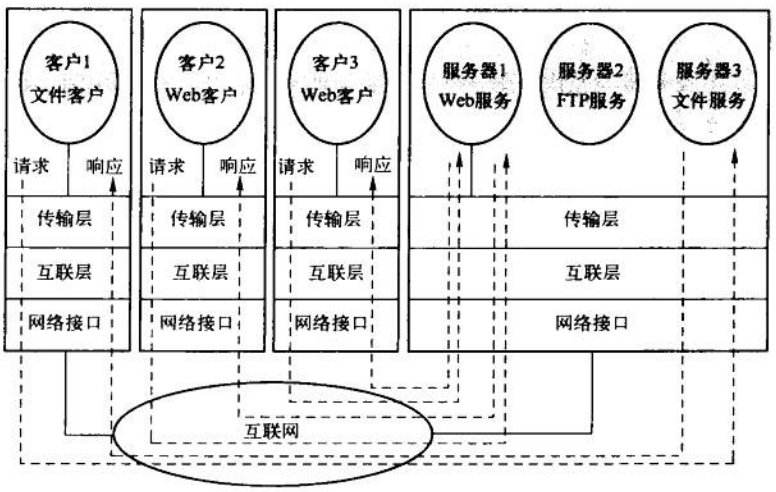


图 1.4　并发服务器模型

为了理解服务器中并发的重要性，考虑一下需要大量计算或通信的服务器操作。例如：设想一个远程登录服务器，如果其不能并发运行，而是一次只能处理一个远程登录。此时，一旦有一个客户与该服务器建立了联系，则服务器在第一个用户结束会话之前，必须忽略或拒绝所有其他客户的请求。显然，这样的设计限制了服务器的使用效率，而且还使得多个远程用户不能在同一时间对该服务器进行访问。

1.2.5　服务器并发性的实现方法

LINUX 提供了两种方法来实现服务器的并发性，其中，一种实现方法是服务器创建

多个进程（Process），每个进程都有一个线程（Thread），使得不同进程中的多个线程并行工作以完成多项任务，从而以提高系统的效率；另一种实现方法则是服务器在一个进程中创建多个线程，使得同一进程中的多个线程并行工作以完成多项任务，从而以提高系统的效率。

其中，进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程是系统进行资源分配和调度的一个独立单位，是具有一定独立功能的程序关于某个数据集合上的一次运行活动。由于每个进程都拥有自己独立的地址空间，因此，当一个进程崩溃后，在保护模式下它不会对其他进程产生影响。而线程则是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。但线程自己基本上不拥有系统资源，但可与同属一个进程内的其他线程共享该进程所拥有的全部资源。由于线程除了只拥有一点在运行中必不可少的资源（如程序计数器、一组寄存器和栈）之外，没有单独的地址空间，因此一个线程的崩溃也就等于整个进程的崩溃。这也意味着多进程的程序要比多线程的程序健壮，但线程间彼此切换所需的时间要远远小于进程间切换所需要的时间，因此，多进程服务器的效率也就要比多线程的服务器差。

1.2.6 服务器的分类

在客户 - 服务器模型中，服务器模型可依据不同方式进行分类。例如：按照使用的传输协议不同可以分为无连接的服务器与面向连接的服务器；按照是否维护与客户交互活动的信息可以分为有状态服务器与无状态服务器；按照处理与客户交互的机制不同又可以分为循环服务器与并发服务器。

（1）无连接的服务器与面向连接的服务器

程序员在设计客户 - 服务器软件时，必须从 TCP/IP 协议族所提供的两个主要的传输协议 TCP 和 UDP 中选择一个来实现客户 - 服务器之间的通信，如果选择使用 UDP 协议进行通信，由于 UDP 协议是一种不可靠的无连接协议，因此，客户 - 服务器之间的交互是无连接的，故使用 UDP 协议进行通信的服务器通常被称为无连接的服务器；若选择使用 TCP 协议进行通信，由于 TCP 协议是一种可靠的面向连接的协议，因此，客户 - 服务器之间的交互是面向连接的，故使用 TCP 协议进行通信的服务器通常也被称为面向连接的服务器。

由于使用 UDP 协议进行通信的客户和服务器在传输可靠性上没有任何保障，在客户发送请求时，这个请求可能丢失、重复、延迟或者交付失序，因此通常应用程序只在由应用程序处理可靠性，或者应用程序需要硬件进行广播或组播，或者应用程序在可靠的本地环境中运行而不需要额外的可靠性处理等情况下才使用 UDP 协议；而在其他情况下，一般推荐使用 TCP 协议。

（2）有状态服务器与无状态服务器

服务器维护的与客户交互活动的信息称为状态信息。不保存任何状态信息的服务器称为无状态服务器（Stateless Server）；反之，则称为有状态服务器（Stateful Server）。从本质上讲，状态信息让服务器记住了客户以前有过哪些请求，并在每个新请求到来时计算新的响应，在计算新的响应时，由于在服务器中保存了客户以前的状态信息，因此可以减少客户和服务器之间所交换的报文的大小，从而提高了客户和服务器之间通信的效率；但由于网络的稳定性问题，一旦存在报文丢失、重复或交付失序等情况，则将会导致服务器中维护的状态

信息不正确，从而使得服务器计算新的响应时，若使用了不正确的状态信息，就可能产生错误的响应。另外，客户计算机的崩溃或重新启动也将会导致有状态服务器的出错，例如：如果一个客户与服务器联系之后就崩溃了，而服务器已经为该客户建立了状态信息，那么服务器就可能永远也不会收到让它丢弃这些状态信息的报文，从而最终使得这些累积起来的状态信息会把服务器的存储资源耗尽。

一般来说，保持服务器中维护的状态信息的正确性这个问题只有复杂的协议才能解决，因此，采用有状态服务器的设计将会导致复杂的应用协议，特别是在不可靠的网络环境中，这种应用协议一般难于设计、理解和正确实现。

在常见的服务器应用中，传统的 Web 服务器就是一个典型的无状态服务器，在客户访问 Web 服务器时，一个客户每次发送的 HTTP 请求均和他以前发送的请求无任何关系，每次连接请求的目的都只是为了获取目标 URI（Universal Resource Identifier，通用资源标志符），而且在得到目标内容之后，这次连接就会被断开，在服务器中不会保留任何与该客户的交互活动的信息。在后来的发展过程中，Web 服务器也逐渐在无状态的过程中加入了状态信息，比如 COOKIE。当 Web 服务器在响应客户的连接请求时，会向客户端推送一个 COOKIE，这个 COOKIE 记录了 Web 服务器上的一些客户的状态信息；这样一来，客户端在后续的请求中可以携带该 COOKIE，Web 服务器则可根据该 COOKIE 来判断请求的上下文关系。COOKIE 的存在是无状态服务器向有状态服务器的一个过渡手段，它通过 COOKIE 这种外部扩展手段来维护服务器中的上下文关系。

代表性的有状态服务器则包括有即时通信服务器（如 QQ、MSN），以及网络游戏服务器等。由于在这些服务器中维护有每个客户连接的状态信息，因此服务器在接收到客户发送的连接请求时，可以依据本地存储的状态信息来重现上下文关系。比如说，当一个用户登录网络游戏服务器后，服务器就可以依据其保存的状态信息，很容易找到该用户的历史状态了。

显然，有状态服务器虽然在功能实现方面具有更大的优势，但由于它需要维护大量的状态信息，因此在性能方面要逊于无状态服务器。而无状态服务器虽然在处理简单服务方面有一定的优势，但却在处理复杂功能方面存在很多弊端，比如，用无状态服务器来实现即时通信服务器，那无疑将会是一场噩梦。因此，一个服务器到底是该采用无状态还是有状态的设计，这一问题的答案应更多地取决于应用协议而不是实现。例如：如果应用协议规定了某个报文的含义在某种方式上要依赖于先前的一些报文，这样它就不可能提供无状态的交互；而在不稳定的网络环境下，由于服务器难于维护正确的状态信息，此时，若采用有状态的服务器设计则将是不明智的。

（3）循环服务器与并发服务器

在同一时刻只能响应一个客户请求的服务器，称为循环服务器（Iterative Server）；而在同一时刻可以响应多个客户请求的服务器，则称为并发服务器（Concurrent Server）。在循环服务器中，由于服务器将会首先接受一个客户的连接请求，然后再处理该连接请求，只有在处理完该连接请求并断开连接之后，才会再接受下一个客户的连接请求。因此，在采用循环服务器设计时，如果有一个客户端占用服务器不放时，则将会导致所有其他的客户连接请求都得不到及时的响应。

1.3 TCP/IP 网络通信中的客户软件的设计流程

1.3.1 TCP 客户算法

由于 TCP 协议是一种可靠的面向连接的协议，它只需在建立连接的连接请求分组中携带目的节点地址，而在后续的实际数据传输过程中，各数据分组则无需携带目的节点地址，只需使用连接号（Connect ID）即可。因此，基于图 1.2 给出的 TCP/IP 模型的通信原理，TCP 客户算法的设计流程可描述如下。

步骤 1：找到期望通信的服务器套接字端点地址（IP 地址 + 协议端口号）。

步骤 2：创建本地客户端套接字。

步骤 3：为该套接字指明本地端点地址（由 TCP/IP 协议软件自动选取）。

步骤 4：建立该套接字到服务器套接字之间的一个 TCP 连接。

步骤 5：基于建立的 TCP 连接，与服务器进行通信（发送请求与等待应答）。

步骤 6：通信结束之后，关闭该套接字以释放与之相关的资源（包括 TCP 连接的释放）。

1.3.2 UDP 客户算法

由于 UDP 协议提供无连接的数据报服务，各分组在通信子网中是独立传送的，每个分组均需要携带完整的目的节点的地址。因此，基于图 1.2 给出的 TCP/IP 模型的通信原理，UDP 客户算法的设计流程可描述如下。

步骤 1：找到期望通信的服务器套接字端点地址（IP 地址 + 协议端口号）。

步骤 2：创建本地客户端套接字。

步骤 3：为该套接字指明本地端点地址（由 TCP/IP 协议软件自动选取）。

步骤 4：基于期望通信的服务器套接字端点地址，与服务器进行通信（发送请求与等待应答）。

步骤 5：通信结束之后，关闭该套接字以释放与之相关的资源。

1.3.3 客户算法中服务器套接字端点地址的查找问题

1. 客户软件查找服务器套接字端点地址的方法

客户软件可以使用以下多种方法来找到某个服务器套接字的端点地址：

（1）在编译程序时，将服务器套接字的端点地址说明为常量；

（2）要求用户在启动程序时输入服务器套接字的端点地址；

（3）从本地文件中获取服务器套接字端点地址的有关信息；

（4）通过某个组播或广播协议来查找服务器套接字的端点地址。

在上述四种方法中，将服务器套接字端点地址指明为常量可以使得客户软件快速执行，但当服务器套接字端点地址变化后，则客户软件就必须重新编译才可正常运行；把服务器套接字端点地址存放在本地的一个文件中可使得客户软件更加灵活，但这也意味着如果该文件损坏或丢失将导致客户软件运行失败；在本地的小环境下，通过使用某个组播或广播协议来

动态地查找服务器套接字的端点地址是可行的，但却不适合用于大的因特网环境，此外，使用这种动态的查找机制还会在增加网络的额外通信开销。因此，为了避免不必要的麻烦和对计算环境的依赖，大多数客户使用在启动客户程序时，要求用户输入服务器 IP 地址和协议端口号的方法，来获取某个服务器套接字的端点地址。按照这种方法来构建客户软件，不但可以使得客户软件更具一般性，而且还可以使得改变服务器的位置成为了可能。

2. 应用程序进程中端点地址的存储结构

为允许协议族自由地选择其地址表示方法，套接字抽象为每种类型的地址定义了一个地址族，一个协议族可以使用一种或多种地址族来定义其端点地址的表示方式。同时，套接字软件也为应用程序存储端点地址在头文件 < sys/socket. h > 中提供了以下预定义的 C 结构声明：

```
struct sockaddr {              //来存储断电地址信息的结构
    u_char sa_len;             //表示整个 sockaddr 结构体的长度
    u_short sa_family;         //地址族(长度为 2 字节)
    char sa_data[14];          //14 字节的协议端点地址
};
```

在上述 sockaddr 结构中，包含一个占 2 字节的地址族标识符和一个占 14 字节用于存储实际端点地址的数组。

需要注意的是，不是所有的地址族都定义了适合上述 sockaddr 结构的端点地址。例如：某些 UNIX 定义了一种称为命名管道（Named Pipe）的地址族 AF_UNIX，其端点地址长度要远远大于 14 字节，从而使得声明为 sockaddr 类型的变量无法装下其地址信息。因此，在编写混合协议的程序时，程序员一定要注意，有些非 TCP/IP 协议的端点地址可能要求一个更大的结构。

上述 sockaddr 结构虽然可适用于 TCP/IP 协议族中的端点地址，但由于使用套接字的每个协议族都精确地定义了它的端点地址，例如，每个 TCP/IP 端点地址是由以下字段构成：一个用来标识地址类型的 2 字节字段、一个 2 字节的端口号字段、一个 4 字节的 IP 地址字段（IPv4），以及一个未使用的 8 字节字段。因此，套接字软件在头文件 < netinet/in. h > 中还为 TCP/IP 协议族提供了以下预定义结构 sockaddr_in 来指明这种格式：

```
struct sockaddr_in {
    u_char sin_len;                  //表示整个 sockaddr_in 结构体的长度
    short int sin_family;            //地址族(长度为 2 字节)
    unsigned short int sin_port;     //端口号(长度为 2 字节)
    struct in_addr sin_addr;         //存储 IP 地址的结构(长度为 4 字节)
     unsigned char sin_zero[8];      //填充 0 以保持与 struct sockaddr 同样大小
};
```

其中，存储 IP 地址的结构 struct in_addr 的定义如下：

```
struct in_addr {                 //存储 IP 地址的结构
    unsigned long s_addr;        //IP 地址
};
```

显然，只使用 TCP/IP 协议的应用程序可以只使用上述 sockaddr_in 结构，而无需使用 sockaddr 结构。另外，由于 TCP/IP 协议族（表示为 PF_INET）中各协议均使用一种单一的地址表示方式，其地址族用符号 AF_INET 表示，因此，在上述 sockaddr_in 结构中，地址类型字段 sin_family 应赋值为 AF_INET。

3. 网络字节顺序与主机字节顺序

主机字节顺序是指占用内存多于一个字节类型的数据在主机的内存之中的存放顺序，不同的 CPU 有不同的字节顺序类型，通常有小端、大端两种字节顺序，其中：小端字节顺序（Little Endian）是指低字节数据存放在内存的低地址处，高字节数据存放在内存的高地址处；大端字节顺序（Big Endian）是指高字节数据存放在内存的低地址处，低字节数据存放在内存的高地址处。

网络字节顺序是 TCP/IP 中规定好的一种数据表示格式，它与具体的 CPU 或操作系统的类型等无关，从而可以保证数据在不同的主机之间传输时能够被正确解释。网络字节顺序采用的是 Big Endian 的排序方式。

为了进行主机字节顺序与网络字节顺序之间的转换，套接字软件提供了以下四个转换函数。

（1）htons()函数

htons 即 host-to-network-for type 'short'，该函数的功能是把 unsigned short 类型的数据从主机字节顺序转换到网络字节顺序，调用成功时，将返回一个网络字节顺序的 16 位无符号短整型（unsigned short）值；若调用出错则返回 -1。其函数原型如下：

```
#include < netinet/in. h >              //含有 sockaddr_in 结构与字节顺序转换函数的定义
uint16_t htons( uint16_t hostshort) ;
```

在上述 htons 函数的原型中，其参数的含义如下。

hostshort：一个 16 位无符号短整型值。

（2）htonl()函数

htonl 即 host-to-network-for type 'long'，该函数的功能是把 unsigned long 类型的数据从主机字节顺序转换到网络字节顺序，调用成功时返回一个网络字节顺序的 32 位无符号长整型（unsigned short）值；若调用出错则返回 -1。其函数原型如下：

```
#include < netinet/in. h >              //含有 sockaddr_in 结构与字节顺序转换函数的定义
uint32_t htonl( uint32_t hostlong) ;
```

在上述 htonl 函数的原型中，其参数的含义如下。

hostlong：一个 32 位无符号长整型值。

（3）ntohs()函数

ntohs 即 network-to-host-for type 'short'，该函数的功能是把 unsigned short 类型的数据从网络字节顺序转换到主机字节顺序，调用成功时返回一个主机字节顺序的 16 位无符号短整型值；若调用出错则返回 -1。其函数原型如下：

```
#include < netinet/in. h >              //含有 sockaddr_in 结构与字节顺序转换函数的定义
uint16_t ntohs( uint16_t netshort) ;
```

在上述 ntohs 函数的原型中，其参数的含义如下。

netshort：一个 16 位无符号短整型值。

(4) ntohl( ) 函数

ntohl 即 network-to-host-for type 'long'，该函数的功能是把 unsigned long 类型的数据从网络字节顺序转换到主机字节顺序，调用成功时返回一个主机字节顺序的 32 位无符号长整型值；若调用出错则返回 -1。其函数原型如下。

```
#include <netinet/in.h>          //含有 sockaddr_in 结构与字节顺序转换函数的定义
uint32_t ntohl(uint32_t netlong);
```

在上述 ntohl 函数的原型中，各参数的含义如下。

netlong：一个 32 位无符号长整型值。

4. 由域名查找 IP 地址及由服务名查找端口号

在前面已指出，为了避免不必要的麻烦和对计算环境的依赖，大多数客户使用在启动客户程序时，要求用户输入服务器 IP 地址和协议端口号的方法，来获取某个服务器套接字的端点地址。用户在命令行方式下输入服务器 IP 地址信息时，一般应允许用户既可输入用点分十进制表示的服务器 IP 地址（如 128.10.2.3），又可输入服务器的域名（如 merlin.cs.purdue.edu）；而在用户输入协议端口号时，也应允许用户既可输入用十进制表示的实际协议端口号（如 23），又可输入该端口号所提供的服务的服务名（如 Telnet）。

(1) 将点分十进制 IP 地址转换为网络字节顺序二进制 IP 地址

当用户输入的是用点分十进制表示的服务器的 IP 地址时，由于客户必须使用 sockaddr_in 结构来保存服务器的 IP 地址，因此，这就意味着客户需要将用点分十进制表示的服务器的 IP 地址转换为用二进制表示的 32 位的网络字节顺序的 IP 地址，套接字软件提供了库例程 inet_addr、inet_aton、inet_ntoa 来实现上述转换。

inet_addr( ) 函数：调用成功时返回一个用二进制表示的 32 位的网络字节顺序的 IP 地址；若调用出错则返回 -1。其函数原型如下：

```
#include <arpa/inet.h>           //含有 inet_addr, inet_aton, inet_ntoa 等函数的定义
in_addr_t inet_addr(const char *cp);
```

在上述 inet_addr( ) 函数的原型中，各参数的含义如下。

cp：指向一个用点分十进制表示的 IP 地址字符串。

inet_aton( ) 函数：调用成功时返回 1，表示将 cp 指向的一个用点分十进制表示的 IP 地址字符串转换为一个用二进制表示的 32 位的网络字节顺序的 IP 地址，转换后的 IP 地址存储在参数 inp 中；若调用出错则返回 0。其函数原型如下：

```
#include <arpa/inet.h>           //含有 inet_addr, inet_aton, inet_ntoa 等函数的定义
int inet_aton(const char * cp,struct in_addr * inp);
```

在上述 inet_aton( ) 函数的原型中，各参数的含义如下。

cp：指向一个用点分十进制表示的 IP 地址字符串。

inp：指向一个用二进制表示的 32 位的 IP 地址结构。

inet_ntoa( )函数：调用成功时返回一个指向字符串指针，表示将 32 位二进制形式的 IP 地址转换为用点分十进制形式表示的 IP 地址，结果在函数返回值中返回；若调用出错则返回 NULL。其函数原型如下：

```
#include < arpa/inet. h >          //含有 inet_addr, inet_aton, inet_ntoa 等函数的定义
char * inet_ntoa( struct in-addr in) ;
```

在上述 inet_ntoa( )函数的原型中，其参数的含义如下。

in：指向一个用二进制表示的 32 位的 IP 地址结构。

(2) 由域名查找 IP 地址

当用户输入的不是服务器的 IP 地址，而是服务器的域名时，由于客户必须使用 sockaddr_in 结构保存服务器的 IP 地址，因此，这就意味着客户需要将服务器的域名转换为用二进制表示的 32 位的网络字节顺序的 IP 地址，套接字软件提供了库例程 gethostbyname 来实现上述转换。该函数若调用成功时将返回一个指向包含有 IP 地址信息的 HOSTENT 结构指针；若调用出错则返回 NULL。该函数的原型如下：

```
#include < netdb. h >          //含有 hostent 结构与 gethostbyname 函数的定义
struct hostent * gethostbyname( const char * name) ;
```

在上述 gethostbyname 函数的原型中，其参数的含义如下。

name：指向一个包含域名或主机名的字符串指针。

其中，HOSTENT 结构的定义如下：

```
struct hostent {
    char * h_name;          //主机的规范名称
    char * * h_aliases;     //主机的别名
    char h_addrtype;        /* 主机 IP 地址的类型，如：是 IPv4 ( AF_INET) ，还是 IPv6 ( AF_
                               INET6) */
    char h_length; //主机 IP 地址的长度
    char * * h_addr_list;   /* 以网络字节序存储的主机的 IP 地址列表( 一个主机可能会有多
                               个 IP 地址) */
};
#define h_addr h_addr_list[0] /* 指向主机 IP 地址列表中的第一个位置，这样应用程序就可以将
                                 h_addr 当作 HOSTENT 结构中的一个字段来使用了 */
```

(3) 由服务名查找端口号

当用户输入的不是服务器所提供的特定服务的协议端口号，而是服务器所提供的特定服务的服务名时，由于客户必须使用 sockaddr_in 结构保存服务器的端口号，因此，这就意味着客户需要将服务器所提供的特定服务的服务名转换为服务器所提供的特定服务的协议端口号，套接字软件提供了库例程 getservbyname 来实现上述转换，该函数若调用成功时将返回一个指向包含有协议端口号信息的 SERVENT 结构指针；若调用出错则返回 NULL。该函数的原型如下：

```
#include < netdb. h > //含有 servent 结构与 getservbyname 函数的定义
struct servent * getservbyname( const char * name, const char * proto) ;
```

在上述 getservbyname 函数的原型中，各参数的含义如下。

name：指向一个包含服务器所提供的特定服务的服务名的字符串指针；

proto：连接该服务时用到的协议名。

其中，SERVENT 结构的定义如下：

```
struct servent {
    char *s_name;          //主机的规范名称
    char **s_aliases;      //主机的别名
    short s_port;          //以网络字节顺序存储的服务的协议端口号
    char *s_proto;         //连接该服务时用到的协议名
};
```

5. 由协议名查找协议号

套接字软件提供一种机制，允许客户或服务器将协议名映射为分配给该协议的整数常量（也称为协议号）。库函数 getprotobyname 执行这种查找，调用 getprotobyname 函数时以一个字符串参数的形式传递协议名，它返回的是一个 protoent 类型的结构地址，该函数若调用成功时将返回一个指向包含有协议号信息的 PROTOENT 结构指针；若调用出错则返回 NULL。其中，getprotobyname 的原型如下：

```
#include <netdb.h>              //含有 protoent 结构与 getprotobyname 函数的定义
struct protoent *getprotobyname(const char *name);
```

在上述 getprotobyname 函数的原型中，其参数的含义如下。

※name：指向一个包含协议名的字符串指针。

其中，PROTOENT 结构的定义如下：

```
struct protoent {
    char *p_name;          //协议的规范名称
    char **s_aliases;      //协议的别名
    int p_proto;           //规范的协议号
};
```

1.3.4　客户算法中本地端点地址的选择问题

由 1.1.3 节中介绍可知，客户套接字在能够用于通信之前，客户端应用程序进程需要事先为其指明远程（服务器）的和本地的端点地址。在客户 - 服务器模型中，由于所有客户均需知道服务器端应用程序进程的端口号，因此，要求服务器端应用程序进程必须运行于某个熟知的协议端口之上，但对客户端应用程序进程而言，却并不需要它工作于某个预分配的端口上，而只需要为其分配的端口号没有被其他的应用程序进程使用、且没有被预分配给某个熟知服务即可。

另外，对于拥有多个 IP 地址的主机，客户端应用程序进程往往难以进行 IP 地址的选择，这主要是因为正确的 IP 地址选择一般要依赖于选路信息，而应用程序却很少使用选路信息。例如：假定某台主机拥有多个网络接口（即有多个 IP 地址），在应用程序进程

使用 TCP 之前，它必须知道该连接的端点地址，当该应用程序进程基于 TCP 协议与某个远程服务器通信时，TCP 软件将 TCP 报文段封装到 IP 数据报中，并将该数据报传递给 IP 软件，然后 IP 使用远程目的地址和它自身的路由表来选择下一跳（Next Hop）的地址，以及可以用来到达下一跳的网络接口。此时，将会导致以下问题发生：在外发（Outgoing）数据报中的 IP 源地址应当与网络接口的 IP 地址相匹配，因为 IP 是通过该接口传送数据报。但若程序员为应用程序进程所选择的 IP 地址与接口不匹配的话，则将使得该应用程序进程无法工作，或者即使可以工作，也会使得网络管理变得困难和混乱，使得应用程序进程的可靠性下降。

为此，为了解决上述客户算法中本地端点地址的分配问题，客户端应用程序进程选择将本地端点地址放置不填，而是改由 TCP/IP 协议软件自动地选取正确的本地 IP 地址与未使用的协议端口号来进行本地端点地址的填充。

1.4 TCP/IP 网络通信中的服务器软件的设计流程

1.4.1 主动套接字与被动套接字

套接字是通过调用系统函数 socket( )来创建的；其中，用于等待传入连接的套接字通常称为被动套接字，如服务器套接字；而用于发起连接的套接字则通常称为主动套接字，如客户套接字。

显然，主动套接字和被动套接字的创建方式是完全相同的，即两者都是通过调用系统函数 socket( )来创建的，只是主动套接字可通过调用系统函数 socket( )直接创建，而被动套接字在调用系统函数 socket( )创建一个主动套接字后，还需要再调用系统函数 listen( )来将其转化为被动模式。

1.4.2 TCP 服务器算法

基于 TCP 协议具有的面向连接的特性，TCP 服务器算法的具体实现流程可大致描述如下。

步骤 1：创建本地服务器端套接字。

步骤 2：为该套接字指明本地端点地址（将该套接字绑定到它所提供服务的熟知端口上）。

步骤 3：将该套接字设置为被动模式（被动套接字）。

步骤 4：从该套接字上接收一个来自客户的连接请求，建立与该客户的连接。

步骤 5：构造响应，并按照应用协议将该响应通过所建立的连接发回给客户。

步骤 6：与客户完成交互之后，关闭所建立的连接，并返回步骤 3 以接收来自下一个客户的新的连接请求。

1.4.3 UDP 服务器算法

基于 UDP 协议具有的无连接的特性，UDP 服务器算法的具体实现流程可大致描述如下。

步骤 1：创建本地服务器端套接字。

步骤 2：为该套接字指明本地端点地址（将该套接字绑定到它所提供服务的熟知端口上）。

步骤 3：重复读取来自客户的请求，然后构造响应，并按照应用协议将该响应发回给客户。

1.4.4　服务器算法中熟知端口的绑定问题

在服务器算法中，如果在将套接字绑定到某个端口号时，若它指定了某个特定的 IP 地址，则该套接字将只能接收客户发送到该 IP 地址上的请求，而不能接收客户发送到该机器其他 IP 地址上的请求。

然而，在很多时候路由器或多接口机器一般会拥有多个 IP 地址，显然，要想使得服务器可以接收客户发送到其所有 IP 地址的请求，则在套接字绑定到某个端口号时，不能只指定某个特定的 IP 地址。为此，为了解决这个问题，套接字接口定义了一个特殊的常量——INADDR_ANY，它指明了一个通配地址（Wildcard Address），可以与该主机的任何一个 IP 地址都匹配。即在服务器算法中当为套接字指明本地端点时，若服务器使用常量 INADDR_ANY 来取代某个特定的 IP 地址，则表示允许该套接字接收发给该机器上的任何一个 IP 地址的客户请求。

1.5　本章小结

本章主要对 TCP/IP 参考模型及其通信原理、TCP/IP 网络通信中的客户 - 服务器模型，以及 TCP/IP 网络通信中的客户和服务器算法设计流程等内容分别进行了详细介绍。通过本章的学习，需要了解 LINUX 系统中实现网络通信的基本原理；需要熟习客户 - 服务器模型的通信实现原理；需要掌握 TCP/IP 网络通信中的 TCP 客户算法与 UDP 客户算法的设计流程，以及 TCP 服务器算法与 UDP 服务器算法的设计流程。

本章习题

1. 什么是端口号和端点地址？
2. LINUX 操作系统提供了哪些基本的系统函数用来对本地设备或文件进行 I/O 操作？
3. 什么是套接字描述符？
4. 什么是客户 - 服务器模型中的汇聚点问题？其解决方法是什么？
5. 什么是并发？服务器是如何实现并发性的？
6. 什么是无连接的服务器与面向连接的服务器？
7. 什么是有状态服务器与无状态服务器？
8. 什么是循环服务器与并发服务器？
9. 简述 TCP 客户算法的设计流程。
10. 简述 UDP 客户算法的设计流程。
11. 客户软件可以使用哪些方法来找到某个服务器套接字的端点地址？
12. 客户算法中的本地端点地址是如何分配的？

13. 简述 TCP 服务器算法的设计流程。

14. 简述 UDP 服务器算法的设计流程。

15. 在服务器算法中，如何使得套接字可以接收发给该机器上的任何一个 IP 地址的客户请求？

16. 什么是主动套接字和被动套接字？

第 2 章　套接字 API

套接字 API 已形成了事实上的网络套接字的标准，本章首先简要介绍套接字 API 的概念与起源；然后分别针对套接字 API 中提供的主要系统函数深入介绍其原型、参数及调用方法；最后，基于套接字 API 中提供的主要系统函数，简要介绍基于套接字 API 的 C/S 网络通信模型及其用 C 语言伪代码的概略实现方法。

2.1　套接字 API 概述

应用程序接口（Application Programming Interface，API）又称为应用编程接口，是一组能用来操作组件、应用程序或者操作系统的函数，其主要目的是提供应用程序与开发人员以访问一组例程的能力，而又无需访问源码或理解内部工作机制的细节，从而使得程序员通过使用 API 函数开发应用程序时可以减轻编程任务。

在 20 世纪 80 年代初，美国加利福尼亚大学伯克利分校的研究人员为方便在 BSD UNIX 系统中实现 TCP/IP 网络通信而开发了一个专门用于网络通信应用的 API，该 API 就是套接字 API（Socket API）。它包括了一个用 C 语言写成的应用程序开发库，主要用于实现进程之间的通信，目前已在计算机网络通信方面被广泛使用，形成了事实上的网络套接字的标准。套接字 API 中提供的主要系统函数如表 2.1 所示。

表 2.1　套接字 API 中提供的主要系统函数

函 数 名	功　能
socket	创建用于网络通信的套接字描述符
connect	连接远程对等实体（客户）
send（write）	通过 TCP 连接外发数据
recv（read）	从 TCP 连接中获得传入数据
close	终止通信并释放套接字描述符
bind	将本地 IP 地址和协议端口号绑定到套接字上（服务器）
listen	将套接字设置为被动模式，并设置排队的 TCP 传入连接个数（服务器）
accept	接收下一个传入连接（服务器）
recvfrom	接收下一个传入的数据报并记录其源端点地址
sendto	依据调用 recvfrom 预先记录下的端点地址，发送外发的数据报
shutdown	在一个或两个方向上终止 TCP 连接
getpeername	在连接到达后，从套接字中获得远程机器的端点地址
getsockopt	获得套接字的当前选项
setsockopt	改变/设置套接字的当前选项

2.2 套接字 API 中的主要系统函数

2.2.1 socket( ) 函数

系统函数 socket( ) 用于创建一个新套接字，该套接字可用于网络通信。socket( ) 调用若执行成功，将返回一个套接字描述符（即一个整型的数值），若出错则返回 -1。socket( ) 函数的原型如下：

```
#include < sys/types. h > / * 提供各种数据类型的定义 * /
#include < sys/socket. h > / * 提供 socket 函数及数据结构的定义 * /
int socket( int domain, int type, int protocol);
```

在上述 socket( ) 函数的原型之中，各个参数的含义如下。

domain：用于指明建立 socket 所使用的协议族，通常赋值为符号常量 PF_INET，表示互联网协议族（TCP/IP 协议族）。

type：用于指定创建该 socket 的应用程序所希望采用的通信服务类型。同一协议族可能提供多种不同的服务类型，例如，TCP/IP 协议族可提供数据流（TCP）和数据报（UDP）两种服务类型。为此，该参数通常赋值为符号常量 SOCK_STREAM（表示数据流服务类型）或 SOCK_DGRAM（表示数据报服务类型）。

protocol：用于指明该 socket 所使用的具体协议的协议号。由于有些协议族中不止一种协议支持同一类型的服务，因此单纯用前面两个参数 domain 和 type 还不能唯一确定一个协议。但在 TCP/IP 协议族中，用 domain 和 type 这两个参数一般即可唯一确定一个协议，因此在 TCP/IP 协议族中，protocol 参数通常赋值为 0。

应用程序在调用 socket( ) 函数返回一个 socket 描述符后，socket( ) 函数将建立一个 socket，这里“建立一个 socket”实际上是意味着为一个 socket 数据结构分配了存储空间，而返回的 socket 描述符则是一个指向该内部数据结构的指针。

在通过 socket( ) 函数建立了一个 socket 之后，在使用该 socket 进行网络通信以前，还必须配置该 socket。其中，面向连接的 socket 客户端是通过执行 connect( ) 函数来在 socket 数据结构中保存本地和远端信息的；而无连接的 socket 客户端和服务端，以及面向连接的 socket 服务端则都是通过调用 bind( ) 函数来配置本地信息的。

2.2.2 connect( ) 函数

系统函数 connect( ) 用于配置 socket 并与远端服务器建立一个 TCP 连接，只有面向连接的客户程序使用 socket 时才需要调用 connect( ) 函数来将 socket 与远端主机相连，而无连接协议则不需要建立直接连接，面向连接的服务器也不需启动一个连接，它只是被动地在指定的协议端口监听客户的请求。connect( ) 函数若执行成功，将返回一个整型数值，若出错则返回 -1。connect( ) 函数的原型如下：

```
#include < sys/types. h >
#include < sys/socket. h >
```

```
int connect(int sockfd, struct sockaddr * serv_addr, int addrlen);
```

在上述 connect( )函数的原型之中，各个参数的含义如下。

sockfd：套接字文件描述符，它是由系统函数 socket( )返回的。

serv_addr：指向数据结构 sockaddr 的指针，其中包含远端机器的端点地址（远端机器的端口号和 IP 地址）。

addrlen：远端地址结构 serv_addr 的长度，可以使用 sizeof（struct sockaddr）来获得。

2.2.3　bind( )函数

系统函数 bind( )用于将 socket 与本机的一个端点地址（端口号 + IP 地址）相关联。bind( )函数若调用成功，将返回一个整型数值；若出错则返回 -1。bind( )函数主要用于服务器端，其原型如下：

```
#include < sys/types. h >
#include < sys/socket. h >
int bind(int sockfd, struct sockaddr * my_addr, int addrlen);
```

在上述 bind( )函数的原型之中，各个参数的含义如下。

sockfd：系统函数 socket( )返回的套接字描述符。

my_addr：指向数据结构 sockaddr 的指针，其中包含本机的端点地址（本机的端口号和 IP 地址）。在给结构指针变量 m_addr 赋值时，可通过将 my_addr. sin_port 置为 0 来自动选择一个未占用的端口号，通过将 my_addr. sin_addr. s_addr 置为 INADDR_ANY 来自动填入本机 IP 地址。

addrlen：本机地址结构 my_addr 的长度，可以使用 sizeof（struct sockaddr）来获得。

注：*在调用 bind( )函数时，需要将 sin_port 和 s_addr 转换成为网络字节优先顺序。另外，调用 bind( )函数时一般不要将端口号设置为小于 1024 的值，因为 1 到 1024 是保留端口号。*

2.2.4　listen( )函数

系统函数 listen( )用于使 socket 处于被动的监听模式，并为该 socket 建立一个输入数据队列，将到达的客户服务请求保存在此队列中，直到程序处理它们。listen( )函数若调用成功，将返回一个整型数值，若出错则返回 -1。listen( )函数的原型如下：

```
#include < sys/types. h >
#include < sys/socket. h >
int listen(int sockfd, int backlog);
```

在上述 listen( )函数的原型之中，各个参数的含义如下。

sockfd：系统函数 socket( )返回的套接字描述符。

backlog：指定进入队列中所允许的连接的个数，进入队列的客户连接请求在使用系统调用 accept( )应答之前将在进入队列中等待。这个值是队列中最多可以拥有的客户请求的个数。大多数系统的默认设置为 20，也可以设置为 5 或 10。如果一个客户请求到来时，输入队列已满，则 socket 将拒绝客户的连接请求，客户将收到一个出错信息。

2.2.5 accept( ) 函数

系统函数 accept( ) 用于从等待连接队列（该等待连接队列为系统函数 listen( ) 所创建）中抽取第一个客户连接请求，然后建立与该客户之间的连接，并为该接连创建一个新的套接字（该新套接字将复制系统函数 socket( ) 所创建的那个原套接字中的部分信息，主要包括协议类型，协议操作等），此后，就由这个新的套接字来负责通过与该客户之间的连接与该远端客户进行通信。如果队列中没有正在等待的客户连接，且套接字为阻塞方式，则 accept( ) 函数将阻塞调用进程直至新的客户连接请求出现；如果套接字为非阻塞方式且队列中没有正在等待的客户连接请求，则 accept( ) 函数将返回一个错误代码。accept( ) 函数若调用成功，将返回一个新的套接字描述符，若出错则返回 -1。accept( ) 函数的原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

在上述 accept( ) 函数的原型之中，各个参数的含义如下。

sockfd：是系统函数 socket( ) 返回的套接字描述符。

addr：是一个用于回传的指向数据结构 sockaddr 的指针，accept( ) 函数将从客户的连接请求之中自动抽取远端客户主机的端点地址信息并存入到该 addr 指针所指向的数据结构之中。

addrlen：远端地址结构 addr 的长度，可以使用 sizeof（struct sockaddr）来获得。

2.2.6 send( ) 函数

系统函数 send( ) 用于给 TCP 连接的另一端发送数据，其中，客户程序一般用 send( ) 函数向服务器发送请求，而服务器则通常调用 send( ) 函数来向客户程序发送应答。send( ) 函数只可用于基于连接的套接字，send( ) 函数和 write( ) 函数唯一的不同点是标志的存在，当标志为 0 时，send( ) 函数等同于 write( ) 函数。send( ) 函数的原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int s, const void *buf, size_t len, int flags);
```

在上述 send( ) 函数的原型之中，各个参数的含义如下。

s：指明用来发送数据的套接字描述符（如系统函数 accept( ) 返回的套接字描述符）。

buf：指明一个存放应用程序要发送数据的缓冲区。

len：指明实际要发送的数据的字节数。

flags：指明调用执行方式，一般设置为 0。

系统函数 send( ) 若调用出错将返回 -1；若调用成功，则将返回实际发送出的字节数，该字节数可能会少于实际欲发送的数据的字节数 len，在程序中应该将 send( ) 函数的返回值与实际欲发送的数据的字节数进行比较。当 send( ) 函数的返回值与 len 不匹配时，应对这种情况进行处理。

注：即便是已成功地调用 send( ) 函数也并不意味着数据一定会正确地传送到达对端机

器，因为当调用 send( )函数时，send( )函数将首先比较待发送数据的长度 len 和套接字 s 的发送缓冲区的长度，如果 len 大于 s 的发送缓冲区的长度，则该函数返回 -1；如果 len 小于或者等于 s 的发送缓冲区的长度，那么 send( )函数先检查协议是否正在发送 s 的发送缓冲区中的已有数据，如果是，就等待协议把 s 的发送缓冲区中的已有数据发送完，如果协议还没有开始发送 s 的发送缓冲区中的已有数据或者 s 的发送缓冲区中没有数据，那么 send( )函数将比较 s 的发送缓冲区的剩余空间和 len 的大小，如果 len 大于剩余空间大小，则 send( )函数将一直等待协议把 s 的发送缓冲区中的已有数据发送完，如果 len 小于剩余空间的大小，则 send( )函数将仅仅把 buf 中的数据拷贝到剩余空间里（注意：这里 send( )函数仅仅是把 buf 中的数据拷贝到 s 的发送缓冲区的剩余空间里，而不是将 buf 中的数据传到连接的另一端）；如果 send( )函数已成功地将 buf 中的数据拷贝到了 s 的发送缓冲区的剩余空间之中，则立即返回实际拷贝的字节数。即 send( )函数仅仅只是在把 buf 中的数据成功拷贝到了 s 的发送缓冲区的剩余空间里之后就返回了，因此，如果协议在后续的数据传送过程中出现网络错误的话，那么 send( )函数并不能保证这些数据会被正确传到连接的另一端。

2.2.7　recv( )函数

系统函数 recv( )用于客户端和服务器程序从 TCP 连接的接收来自另一端的数据，recv( )函数只可用于基于连接的套接字，recv( )函数和 read( )函数唯一的不同点是标志的存在，当标志为 0 时，recv( )函数等同于 read( )函数。recv( )函数的原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int s, void *buf, int len, unsigned int flags);
```

在上述 recv( )函数的原型之中，各个参数的含义如下。

s：指明用来接收数据的套接字描述符（如系统函数 accept( )返回的套接字描述符）。

buf：指明一个应用程序用来存放接收到的数据的缓冲区。

len：指明缓冲区的长度。

flags：指明调用执行方式，一般设置为 0。

系统函数 recv( )若调用成功，将返回实际接收的字节数；若出现错误，则返回 -1。当应用程序调用 recv( )函数时，recv( )函数将首先等待 s 的发送缓冲区中的数据被协议传送完毕，如果协议在传送 s 的发送缓冲区中的数据时出现网络错误，那么 recv( )函数将返回 -1；如果 s 的发送缓冲区中没有数据或者数据被协议成功发送完毕后，recv( )调用将进一步检查套接字 s 的接收缓冲区，如果 s 的接收缓冲区中没有数据或者协议正在接收数据，则 recv( )函数将一直阻塞，等待协议把数据接收完毕；当协议把数据接收完毕，recv( )函数将把 s 的接收缓冲区中的数据拷贝到 buf 中（注意：由于协议接收到的数据可能大于 buf 的长度，因此在这种情况下需要多次调用 recv( )函数才能把 s 的接收缓冲区中的数据拷贝完全）。

2.2.8　sendto( )函数

系统函数 sendto( )用于在无连接的数据报 socket 方式下进行数据的发送，由于在无连接的数据报 socket 方式下，本地 socket 并没有与远端机器之间建立连接，因此，在调用

sendto( ) 函数来发送数据时应指明目的机器的端点地址。与 send( ) 函数类似，sendto( ) 函数也返回实际发送的数据字节长度或在出现发送错误时返回 -1。sendto( ) 函数的原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
```

在上述 sendto( ) 函数的原型之中，各个参数的含义如下。

sockfd：指明用来发送数据的套接字描述符。

msg：指明一个存放应用程序要发送数据的缓冲区。

len：指明缓冲区的长度。

flags：指明调用执行方式，一般设置为 0。

to：是一个指向数据结构 sockaddr 的指针，其中包含远端机器上的对等方套接字的端点地址。在客户端，该参数中所包含的远端机器的端点地址即为服务器的 IP 地址和熟知端口号；在服务器端，该参数即为 recvfrom( ) 函数中的参数 from。

tolen：远端地址结构 serv_addr 的长度，可使用 sizeof (struct sockaddr) 来获得。

2.2.9 recvfrom( ) 函数

系统函数 recvfrom( ) 用于实现在无连接的数据报 socket 方式下进行数据的接收，与 recv( ) 函数类似，recvfrom( ) 函数也返回实际接收到的数据字节长度或在出现接收错误时返回 -1。recvfrom( ) 函数的原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);
```

在上述 recvfrom( ) 函数的原型之中，各个参数的含义如下。

sockfd：指明用来接收数据的套接字描述符。

buf：指明一个存放应用程序用于接收数据的缓冲区。

len：指明缓冲区的长度。

flags：指明函数执行方式，一般设置为 0。

from：是一个指向数据结构 sockaddr 的指针，用于存放远端机器上的对等方套接字的端点地址。其中，recvfrom( ) 函数将从远端机器通过 sendto( ) 函数发送过来的数据报中自动抽取远端机器的端点地址信息存入到 from 指针所指向的数据结构之中。

fromlen：远端地址结构 serv_addr 的长度，可使用 sizeof (struct sockaddr) 来获得。

2.2.10 close( ) 函数

系统函数 close( ) 用于在服务器与客户端之间的所有数据收发操作结束以后关闭套接字、以释放该套接字所占用的资源。close( ) 函数若调用成功将返回 0，若出错则返回 -1。close( ) 函数的原型如下：

```
#include <unistd.h> //提供通用的文件、目录、进程等操作的函数原型定义
```

```
#include <sys/socket.h>
int close(sockfd);
```

在上述 close( ) 函数的原型之中，其参数的含义如下。

sockfd：指明要关闭的套接字描述符。

注：系统中的每一个文件或套接字都有一个引用计数，表示当前打开或者正在引用该文件或套接字的进程的个数。调用 close( ) 函数终止一个连接时，它只是减少了描述符的引用计数，并不直接关闭对应的连接，只有当描述符的引用计数为 0 时，才真正关闭该连接。因此，在只有一个进程使用某个套接字时，close( ) 函数会将该套接字的读写都关闭，这就容易导致发起关闭的一方还没有读完所有数据就关闭连接了，从而使得对方发来的数据将会被丢弃；但如果有多个进程共享某个套接字时，则系统函数 close( ) 每被调用一次，都只是会使得其引用计数减 1，而只有等到其引用计数为 0 时，也就是说，只有等到所有进程都调用了 close( ) 函数来关闭该套接字时，该套接字才会被最终释放；为此，在有多个进程共享某个套接字的时候，当某个进程调用 close( ) 函数关闭了一个套接字之后，除了使得该进程将不能再访问该套接字之外，其他正在引用该套接字的进程均可继续正常使用该套接字与远端机器进行通信。

2.2.11　shutdown( ) 函数

系统函数 shutdown( ) 用于在套接字的某个方向上关闭数据传输，而一个方向上的数据传输仍可继续进行。例如：可以关闭某个套接字的写操作而允许继续在该套接字上接收数据，直至读入所有的数据。shutdown( ) 函数若调用成功将返回 0，若出错则返回 -1。shutdown( ) 函数的原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int shutdown(int sockfd, int howto);
```

在上述 shutdown( ) 函数的原型之中，各个参数的含义如下。

sockfd：指明要关闭的套接字描述符。

howto：指明关闭方向，该参数的取值包括以下三种。

- 0：仅关闭读，套接字将不再接收任何数据，且将套接字接收到的缓冲区中的现有数据全部丢弃。
- 1：仅关闭写，当前留在缓冲区中的数据将被发送完，进程将不能够再对该套接字调用任何写函数。
- 2：同时关闭读和写，与 close( ) 函数的功能类似，但不同的是，shutdown( ) 函数在关闭描述符时不考虑该套接字描述符的引用计数，而是直接关闭该套接字。

注：与 close( ) 函数不同，在有多个进程共享某个套接字的时候，如果一个进程调用了 shutdown( ) 函数关闭某个套接字之后，将会使得其他的所有进程都无法再使用该套接字进行通信。

2.2.12　getpeername( ) 函数

系统函数 getpeername( ) 用于获取所连接的远端机器上的对等方套接字的名称。getpeer-

name( )函数若执行成功将返回0，若出错则返回 -1。getpeername( )函数的原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int getpeername(int sockfd, struct sockaddr * addr, int * addrlen);
```

在上述 getpeername( )函数的原型之中，所包含的各个参数的含义如下。

sockfd：指明连接的套接字描述符。

addr：是一个指向数据结构 sockaddr 的指针，用于存放远端机器上的对等方套接字的端点地址。

addrlen：远端地址结构 serv_addr 的长度，可使用 sizeof (struct sockaddr) 来获得。

2.2.13 setsockopt( )函数

系统函数 setsockopt( )用于设置与某个套接字关联的选项。setsockopt( )函数若执行成功将返回0，若出错则返回 -1。setsockopt( )函数的原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(int sock, int level, int optname, const void * optval, socklen_t optlen);
```

在上述 setsockopt( )函数的原型之中，各个参数的含义如下。

sockfd：指明将要被设置选项的套接字描述符。

level：指明选项所在的协议层，选项可能存在于多层协议中，但它们总会出现在最上面的套接字层。因此，当操作套接字选项时，为了设置套接字层的选项，应将 level 的值指定为 SOL_SOCKET。

optname：指明需要设置的选项名，SOL_SOCKET 层所包含的选项如表2.2所示。

optval：指向包含新选项值的缓冲区，该参数的类型将根据选项名称的数据类型进行转换。

optlen：选项值的长度。

表2.2 SOL_SOCKET 层所包含的选项

选项名称	说　明	数据类型
SO_BROADCAST	允许发送广播数据	int
SO_DEBUG	允许调试	int
SO_DONTROUTE	不查找路由	int
SO_ERROR	获得套接字错误	int
SO_KEEPALIVE	保持连接	int
SO_LINGER	延迟关闭连接	struct linger
SO_OOBINLINE	带外数据放入正常数据流	int
SO_RCVBUF	接收缓冲区大小	int
SO_SNDBUF	发送缓冲区大小	int

续表

选项名称	说　明	数据类型
SO_RCVLOWAT	接收缓冲区下限	int
SO_SNDLOWAT	发送缓冲区下限	int
SO_RCVTIMEO	接收超时	struct timeval
SO_SNDTIMEO	发送超时	struct timeval
SO_REUSERADDR	允许重用本地地址和端口	int
SO_TYPE	获得套接字类型	int
SO_BSDCOMPAT	与 BSD 系统兼容	int

例如：每个套接字都有一个发送缓冲区和一个接收缓冲区，可以使用表 2.2 中的 SO_RCVBUF 和 SO_SNDBUF 这两个套接字选项来改变套接字的默认缓冲区大小。

```
int rBuf = 32 * 1024;       //设置接收缓冲区为 32 KB
setsockopt(s,SOL_SOCKET,SO_RCVBUF,(const char *)&rBuf,sizeof(int));
int sBuf = 32 * 1024;       //设置发送缓冲区为 32 KB
setsockopt(s,SOL_SOCKET,SO_SNDBUF,(const char *)&sBuf,sizeof(int));
```

注：当设置 TCP 套接口接收缓冲区的大小时，函数调用顺序是很重要的，因为 TCP 的窗口规模选项是在建立连接时用 SYN 与对方互换得到的。对于客户，SO_RCVBUF 选项必须在调用 connect( )函数之前设置；对于服务器，SO_RCVBUF 选项必须在调用 listen( )函数之前设置。

2.2.14　getsockopt( ) 函数

系统函数 getsockopt( )用于获取与某个套接字关联的选项。getsockopt( )函数若调用成功将返回 0，若出错则返回 -1。getsockopt( )函数的原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);
```

在上述 getsockopt( )函数的原型之中，各个参数的含义如下。

sockfd：指明将要被获取选项的套接字描述符。

level：指明选项所在的协议层，选项可能存在于多层协议中，但它们总会出现在最上面的套接字层。因此，当操作套接字选项时，为了获得套接字层的选项，应将 level 的值指定为 SOL_SOCKET。

optname：指明需要访问的选项名，SOL_SOCKET 层所包含的选项如表 2.2 所示。

optval：指向返回选项值的缓冲区，该参数的类型将根据选项名称的数据类型进行转换。

optlen：选项值的长度。

2.3 基于套接字 API 的 C/S 网络通信模型

2.3.1 基于 UDP 的 C/S 网络通信模型

无连接套接字是一种实现绑定到无连接协议（UDP 协议）的套接字，基于无连接套接字的 C/S 网络通信模型如图 2.1 所示。

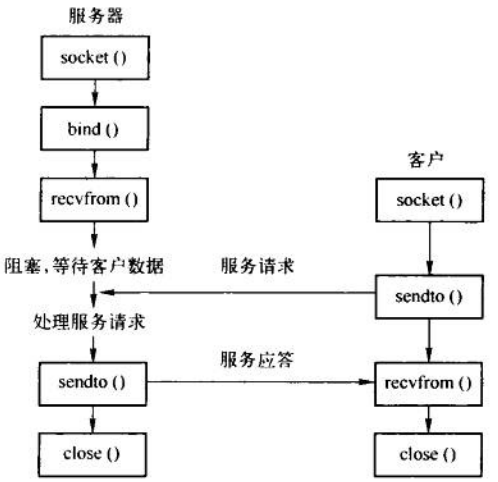


图 2.1 基于无连接套接字的 C/S 网络通信模型

由图 2.1 可知，在基于无连接套接字的 C/S 网络通信模型中，服务器和客户端算法的实现流程可分别概略描述如下。

1. UDP 服务器端算法的实现流程

（1）UDP 服务器端算法的步骤描述

步骤 1：调用 socket( ) 函数创建服务器端无连接套接字。

步骤 2：调用 bind( ) 函数将套接字绑定到本机的一个可用的端点地址。

步骤 3：调用 recvfrom( ) 函数从套接字接收来自远程客户端的数据并存入到缓冲区中，同时获得远程客户的套接字端点地址并保存。

步骤 4：基于保存的远程客户的套接字端点地址，调用 sendto( ) 函数将缓冲区中的数据从套接字发送给该远程客户。

步骤 5：与客户交互完毕，调用 close( ) 函数将套接字关闭，释放所占用的系统资源。

（2）UDP 服务器端算法的 C 语言伪代码实现

```
#define BUFSIZE 4096    //定义缓冲区大小为 4 MB
char buf[BUFSIZE];      //定义缓冲区变量 buf
struct sockaddr_in servaddr;   /*定义服务器端套接字的端点地址结构变量 servaddr*/
int serverfd;                  //定义服务器端套接字描述符变量 serverfd
serverfd = socket(AF_INET,SOCK_DGRAM,0); //步骤 1
```

```
bzero(&servaddr,sizeof(struct sockaddr_in));  /*将服务器端点地址结构变量 servaddr 清空*/
//以下代码段实现为服务器端点地址结构变量 servaddr 赋值
servaddr.sin_family = AF_INET;  //赋值协议族为 TCP/IP 协议族
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);/*赋值 IP 地址为符号常量 INADDR_ANY,需将
                                符号常量 INADDR_ANY 由本机字节顺序转换为网络字节顺序*/
servaddr.sin_port = htons(SERVER_PORT);/*赋值端口号,需将端口号由本机字节顺序转换为网
                                络字节顺序*/
//为服务器端点地址结构变量 servaddr 赋值完毕
bind(serverfd,(struct sockaddr *)&servaddr,sizeof(struct sockaddr_in));//步骤 2
recvfrom(serverfd,buf,sizeof(buf),0,(struct sockaddr*)&clientaddr,&addrlen);//步骤 3
sendto(serverfd,buf,strlen(buf),0,(struct sockaddr*)&clientaddr,sizeof(struct sockaddr_in));
//步骤 4
close(serverfd);//步骤 5
```

2. UDP 客户端算法的实现流程

(1) UDP 客户端算法的步骤描述

步骤 1：调用 socket()函数创建客户端无连接套接字。

步骤 2：找到期望与之通信的远程服务器的 IP 地址和协议端口号；然后再调用 sendto()函数将缓冲区中的数据从套接字发送给远程服务器。

步骤 3：调用 recvfrom()函数从套接字接收来自远程服务器端的数据并存入缓冲区中。

步骤 4：与服务器交互完毕，调用 close()函数将套接字关闭，释放所占用的系统资源。

(2) UDP 客户端算法的 C 语言伪代码实现

```
#define BUFSIZE 4096      //定义缓冲区大小为 4 MB
char buf[BUFSIZE];        //定义缓冲区变量 buf
char serverip[50] = "127.0.0.1";   //定义服务器 IP 地址变量
struct sockaddr_in servaddr;   /*定义服务器端套接字的端点地址结构变量 servaddr */
int clientfd; //定义客户端套接字描述符变量 clientfd
clientfd = socket (AF_INET, SOCK_DGRAM, 0); //步骤 1
bzero (&servaddr, sizeof (struct sockaddr_in)); /*将服务器端点地址结构变量 servaddr 清空*/
//以下代码段实现为服务器端点地址结构变量 servaddr 赋值
servaddr.sin_family = AF_INET;   //赋值协议族为 TCP/IP 协议族
servaddr.sin_port = htons (SERVER_PORT); /*赋值端口号，需将端口号由本机字节顺序转换为
                                网络字节顺序*/
inet_aton (serverip, &servaddr.sin_addr); /*赋值 IP 地址，需将点分十进制 IP 地址由本机字节
                                顺序转换为网络字节顺序*/
//为服务器端点地址结构变量 servaddr 赋值完毕
sendto (clientfd, buf, strlen (buf), 0, (struct sockaddr*) &servaddr, sizeof (struct sockaddr_in));
//步骤 2
recvfrom (clientfd, buf, sizeof (buf), 0, (struct sockaddr*) &servaddr, &addrlen); //步骤 3
close (clientfd); //步骤 4
```

2.3.2 基于 TCP 的 C/S 网络通信模型

面向连接的套接字用于实现面向连接的协议（TCP 协议），基于面向连接的套接字的 C/S 网络通信模型如图 2.2 所示。

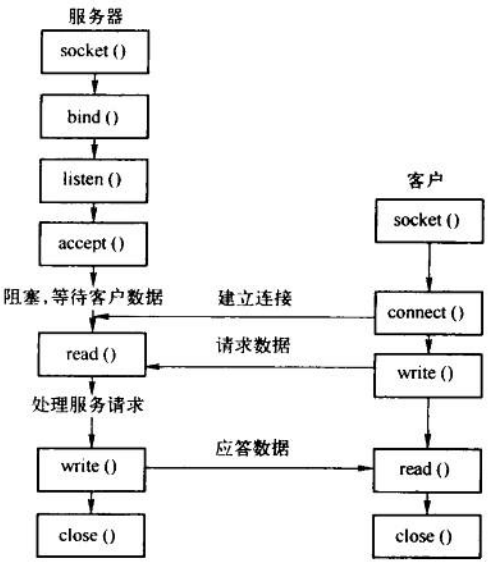


图 2.2 基于面向连接的套接字的 C/S 网络通信模型

由图 2.2 可知，在基于面向连接的套接字的 C/S 网络通信模型中，服务器和客户端算法的具体实现流程可分别概略描述如下。

1. TCP 服务器端算法的实现流程

（1）TCP 服务器端算法的步骤描述

步骤 1：调用 socket( ) 函数创建服务器端面向连接的套接字。

步骤 2：调用 bind( ) 函数将套接字绑定到本机的一个可用的端点地址。

步骤 3：调用 listen( ) 函数将套接字设为被动模式，并设置等待队列长度。

步骤 4：调用 accept( ) 函数从套接字上接受下一个客户连接请求，并在与该客户之间成功建立了连接之后，为该连接创建一个新的套接字（由该新套接字来负责与客户之间进行实际的通信）。

步骤 5：基于新创建的套接字，调用 read( )/recv( ) 函数从套接字读取客户发送过来的数据并存入缓冲区中。

步骤 6：基于新创建的套接字，调用 write( )/send( ) 函数将缓冲区中的数据从套接字发送给该远程客户。

步骤 7：与客户交互完毕，调用 close( ) 函数将套接字关闭，释放所占用的系统资源。

（2）TCP 服务器端算法的 C 语言伪代码实现

```
#define BUFSIZE 4096    //定义缓冲区大小为 4 MB
```

```
char buf[BUFSIZE];    //定义缓冲区变量 buf
struct sockaddr_in servaddr, clientaddr; /*定义服务器端和客户端套接字的端点地址结构变量 servaddr 和 clientaddr */
int serverfd, newclientfd; /*定义服务器端主套接字（负责接受来自客户的连接请求，该套接字为通过系统调用 socket()创建）和从套接字（负责与客户进行实际的交互，该套接字为通过系统调用 accept()创建）描述符变量 serverfd 和 newclientfd */
serverfd = socket (AF_INET, SOCK_STREAM, 0); //步骤 1
bzero (&servaddr, sizeof (struct sockaddr_in)); /*将服务器端点地址结构变量 servaddr 清空*/
//以下代码段实现为服务器端点地址结构变量 servaddr 赋值
servaddr.sin_family = AF_INET; //赋值协议族为 TCP/IP 协议族
servaddr.sin_addr.s_addr = htonl (INADDR_ANY); /*赋值 IP 地址为符号常量 INADDR_ANY，需将符号常量 INADDR_ANY 由本机字节顺序转换为网络字节顺序*/
servaddr.sin_port = htons (SERVER_PORT); /*赋值端口号，需将端口号由本机字节顺序转换为网络字节顺序*/
//为服务器端点地址结构变量 servaddr 赋值完毕
bind (serverfd, (struct sockaddr *) &servaddr, sizeof (struct sockaddr_in)); //步骤 2
listen (serverfd, 20); //步骤 3
newclientfd = accept (sockfd, (struct sockaddr *) &clientaddr, sizeof (clientaddr)); //步骤 4
recv (newclientfd, buf, sizeof (buf), 0); //步骤 5
send (newclientfd, buf, strlen (buf), 0); //步骤 6
close (newclientfd); //步骤 7
```

2. TCP 客户端算法的实现流程

（1）TCP 客户端算法的步骤描述

步骤 1：调用 socket()函数创建客户端面向连接的套接字。

步骤 2：找到期望与之通信的远程服务器的 IP 地址和协议端口号；然后调用 connect()函数向远程服务器发起连接建立请求。

步骤 3：在与服务器成功地建立了连接之后，调用 write()/send()函数将缓冲区中的数据从套接字发送给该远程服务器。

步骤 4：调用 read()/recv()函数从套接字读取服务器端发送过来的数据并存入缓冲区中。

步骤 5：与服务器端交互完毕，调用 close()函数将套接字关闭，释放所占用的系统资源。

（2）TCP 客户端算法的 C 语言伪代码实现

```
#define BUFSIZE 4096    //定义缓冲区大小为 4 MB
char buf[BUFSIZE];    //定义缓冲区变量 buf
char serverip[50] = "127.0.0.1";    //定义服务器 IP 地址变量
struct sockaddr_in servaddr; /*定义服务器端套接字的端点地址结构变量 servaddr */
int clientfd;    //定义客户端套接字描述符变量 clientfd
```

```
clientfd = socket (AF_INET, SOCK_DGRAM, 0);   //步骤 1
bzero (&servaddr, sizeof (struct sockaddr_in)); /* 将服务器端点地址结构变量 servaddr 清空 */
//以下代码段实现为服务器端点地址结构变量 servaddr 赋值
servaddr. sin_family = AF_INET; //赋值协议族为 TCP/IP 协议族
servaddr. sin_port = htons (SERVER_PORT); /* 赋值端口号，需将端口号由本机字节顺序转换为
                                              网络字节顺序 */
inet_aton (serverip, &servaddr. sin_addr); /* 赋值 IP 地址，需将点分十进制 IP 地址由本机字节
                                              顺序转换为网络字节顺序 */
connect (clientfd, servaddr, sizeof (servaddr)); //步骤 2
send (clientfd, data, data_len, 0); //步骤 3
recv (clientfd, data, data_len, 0); //步骤 4
close (clientfd); //步骤 5
```

2.4 本章小结

本章主要对套接字 API 中提供的主要系统函数和基于套接字 API 的 C/S 网络通信模型进行了详细介绍。通过本章的学习，需要了解套接字 API 中提供的系统函数；需要熟习套接字 API 中提供的各主要系统函数的原型、参数及其调用方法；需要掌握 TCP/IP 网络通信中的基于 TCP 和 UDP 的 C/S 网络通信模型，以及相应的服务器端和客户端算法及其 C 语言实现方法。

本章习题

1. 套接字 API 中提供的主要系统函数有哪些?
2. 简述系统调用 socket( ) 函数的功能及其原型的定义。
3. 简述系统调用 connect( ) 函数的功能及其原型的定义。
4. 简述系统调用 bind( ) 函数的功能及其原型的定义。
5. 简述系统调用 listen( ) 函数的功能及其原型的定义。
6. 简述系统调用 accept( ) 函数的功能及其原型的定义。
7. 简述系统调用 send( ) 函数的功能及其原型的定义。
8. 简述系统调用 recv( ) 函数的功能及其原型的定义。
9. 简述系统调用 close( ) 函数与 shutdown( ) 函数的区别。
10. 简述系统调用 getpeername( ) 函数的功能及其原型的定义。
11. 简述系统调用 setsockopt( ) 函数的功能及其原型的定义。
12. 简述基于无连接套接字的 C/S 网络通信模型。
13. 简述基于面向连接的套接字的 C/S 网络通信模型。

第 3 章　循环服务器例程剖析

第 2 章描述了套接字 API 的概念，以及基于套接字 API 的 C/S 网络通信模型及其用 C 语言伪代码的概略实现方法。本章将在此基础上进一步针对其中的一种最简单的服务器——循环服务器软件的设计流程进行更深入的分析和讨论。同时，为了更清晰地说明循环服务器软件的设计流程与实现方法，本章还给出了一个循环的无连接的 TIME 服务器与一个访问该 TIME 服务的无连接客户端的完整 C 语言例程，以及一个循环的面向连接的 DAYTIME 服务器与一个访问该 DAYTIME 服务的面向连接的客户端的完整 C 语言例程。

3.1　循环服务器进程结构

3.1.1　循环的 UDP 服务器进程结构

虽然循环的无连接服务器采用的是循环的方式来处理来自多个客户端的请求，每次从套接字上读取了一个新的客户请求之后，需要在将该客户请求处理完毕并将结果返回给了该客户之后才能读取下一个客户请求，但由于采用了无连接的 UDP 方式来进行通信，使得没有一个客户端可以总是占据着服务端，因此该类服务器不但设计、编程、排错及修改等工作都非常简单，而且只要处理过程没有被设计成死循环，则该类服务器就总能够满足每一个客户的请求。循环的无连接服务器的进程结构如图 3.1 所示。

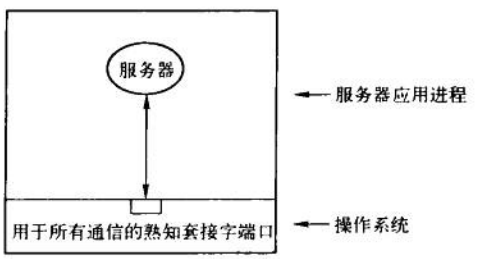


图 3.1　循环的无连接服务器的进程结构

由图 3.1 易知，循环的无连接服务器只需要一个单线程的进程即可实现，它仅使用一个被动套接字，该套接字绑定到所提供服务的熟知端口上，服务器从该套接字上循环获取新的客户请求，计算出响应，然后再通过把该客户请求中包含的源地址作为应答中的目的地址来将响应返回给该客户。

3.1.2　循环的 TCP 服务器进程结构

虽然循环的面向连接的服务器也是采用的循环的方式来处理来自多个客户端的请求，但

由于采用了面向连接的 TCP 方式来进行通信，因此每次从套接字上读取了一个新的客户连接请求之后，将首先与该客户建立一个连接，然后通过该连接与该客户进行交互，当交互结束之后再关闭该连接，然后再次等候/读取了一个新的客户连接请求。循环的面向连接的服务器的进程结构如图 3.2 所示。

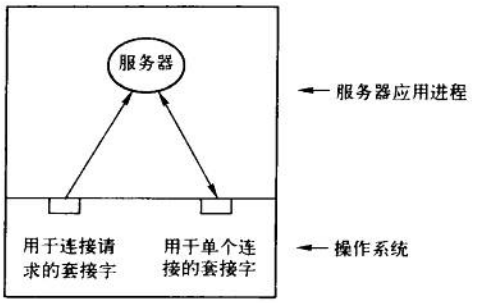


图 3.2 循环的面向连接的服务器的进程结构

由图 3.2 易知，循环的面向连接的服务器也只需要一个单线程的进程即可实现，但它使用两个套接字：其中，一个套接字用于循环接受来自客户的连接请求，该套接字绑定到所提供服务的熟知端口上，服务器从该套接字上循环获取新的客户连接请求，该套接字为永久套接字；而另一个套接字则为临时套接字，该套接字用于处理单个连接，当服务器接受一个新的客户请求之后，将建立一个与该客户的连接，并创建一个临时的套接字来负责在此连接上与该客户进行通信（计算出响应并将响应返回给该客户），当与该客户之间的交互完毕之后，该临时套接字将被关闭，然后服务器再基于永久套接字获取来自下一个客户的连接请求。

3.2 循环服务器软件设计流程

3.2.1 循环的 UDP 服务器软件设计流程

依据上述给出的循环的无连接服务器的进程结构，其算法的设计流程可大致描述如下。

步骤 1：调用 socket( ) 函数创建服务器端 UDP 套接字。

步骤 2：调用 bind( ) 函数将套接字绑定到本机的一个可用的端点地址。

步骤 3：调用 while(1) 函数设置无限循环。

步骤 4：在循环体内：

步骤 4.1：调用 recvfrom( ) 函数读取来自客户的请求。

步骤 4.2：然后构造响应。

步骤 4.3：再调用 sendto( ) 函数按照应用协议将响应发回给客户。

基于以上给出的算法流程，可以用 C 语言伪代码描述如下：

```
socket(…);                  //步骤 1
bind(…);                    //步骤 2
while(1){                   //步骤 3
        recvfrom(…);        //步骤 4.1
        process(…);         //步骤 4.2
        sendto(…);          //步骤 4.3
```

3.2.2　循环的 TCP 服务器软件设计流程

依据上述给出的循环的面向连接的服务器的进程结构，其算法的设计流程可大致描述如下。

步骤 1：调用 socket( ) 函数创建服务器端 TCP 套接字。

步骤 2：调用 bind( ) 函数将套接字绑定到本机的一个可用的端点地址。

步骤 3：调用 listen( ) 函数将套接字设置为被动模式。

步骤 4：调用 while(1) 函数设置无限循环。

步骤 5：在循环体内，

步骤 5.1：调用 accept( ) 函数接受来自客户的连接请求并创建一个用于处理该连接的临时套接字。

步骤 5.2：调用 recv/send( ) 函数基于新创建的临时套接字与客户进行交互。

步骤 5.3：与客户交互完毕，调用 close( ) 函数将临时套接字关闭。

基于以上给出的算法流程，可以用 C 语言伪代码描述如下：

```
socket(…);                    //步骤 1
bind(…);                      //步骤 2
listen(…);                    //步骤 3
while(1){                     //步骤 4
        accept( ) (…);        //步骤 5.1
        process(…);           //步骤 5.2
        close(…);             //步骤 5.3
}
```

单线程服务器模型如图 3.3 所示。

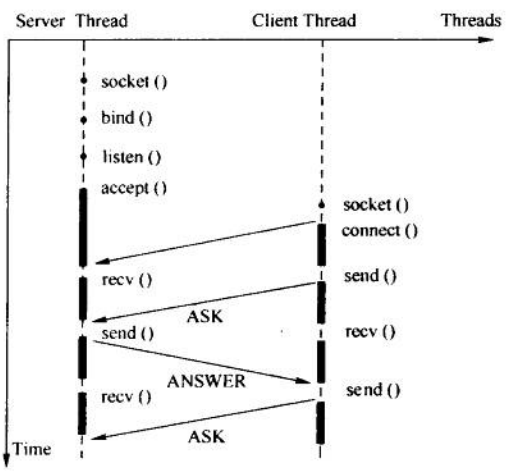


图 3.3　单线程服务器模型

3.3 循环的无连接的 TIME 服务器例程

3.3.1 相关系统函数及其调用方法简介

在实际给出循环的无连接的 TIME 服务器例程并对其进行深入剖析之前，首先详细介绍例程中在本书中首次出现的系统函数及其调用方法如下。

1. sizeof( ) 操作符

sizeof( ) 是 C/C ++ 中的一个操作符（operator），其作用是返回一个对象或者类型所占用的内存字节数。sizeof( ) 操作符的返回值类型为 size_t（即 unsigned int），该类型可以保证能够容纳实现所建立的最大对象的字节大小。sizeof( ) 操作符的调用方法如下（两种原型等价）。

方法 1：

```
size_t sizeof( object ) ; //sizeof( 对象 )
```

方法 2：

```
size_t sizeof( type_name ) ; //sizeof( 类型 )
```

例如：

```
int i;
size_t sz;
sz = sizeof (i);          //返回对象 i 所占用的内存字节数
sz = sizeof (int);        /* 返回类型 int 所占用的内存字节数，由于 i 的类型为 int，因此 sizeof (int) 等价于 sizeof (i) */
```

2. strlen( ) 函数

与 sizeof( ) 是操作符不同，strlen( ) 是函数，它用于计算不包含终止符'\0'在内的字符串长度，而 sizeof( ) 则计算包括终止符'\0'在内的缓冲区长度。strlen( ) 函数的原型如下：

```
#include <string. h >
size_t strlen (const char *s);
```

在上述 strlen( ) 函数的原型中，其参数的含义如下。

s：字符指针（指向字符串的指针）或字符串。

例如：

```
char *c = "abcdef";
char d[ ] = "abcdef";
```

则 sizeof(c) 的返回值为 4，strlen(c) 的返回值为 6；sizeof(d) 的返回值为 7，strlen(d) 的返回值为 6。其中，由于 c 是一个指向字符串"abcdef" 的指针，而指针一般分配 4 个字节，因此 sizeof(c) 的结果就是 4；再由于指针 c 指向的字符串的长度为 6 个字节，因此 strlen(c)

的结果就是 6；其次，由于 d 是一个未指定大小的字符串，其大小将根据后面初始化的内容来自动分配，而后面初始化的实际内容是一个 6 字节的字符串"abcdef"，因此 strlen(c)的结果就是 6；再由于字符串最后还包括有一个终止符'\0'，因此 sizeof(d)的结果就是 6 + 1 = 7。

3. printf( ) 函数

printf( )函数是一个可变参数函数，其主要功能是用来向标准输出设备按规定格式输出信息。printf( )函数的原型如下：

```
#include <stdio.h>
int printf(const char *format[ ,argument,…]);
```

在上述 printf( )函数的原型中，其参数的含义如下。

format："格式控制"字符串，用于指明输出的格式。其完整形式为：% - 0 m.n l 或 h 格式字符，其中：

- % ：表示格式说明的起始符号，不可缺少。
- - ：有 - 表示左对齐输出，如省略表示右对齐输出。
- 0：有 0 表示指定空位填 0，如省略表示指定空位不填。
- m.n：其中，m 表示域宽，即对应的输出项在输出设备上所占的字符数。n 指精度。用于说明输出的实型数的小数位数。未指定 n 时，隐含的精度为 n = 6 位。
- l 或 h：其中，l 用于对整型指 long 型，对实型指 double 型。h 用于将整型的格式字符修正为 short 型。
- 格式字符如下：
  - d 格式：用来输出十进制整数。
  - o 格式：以无符号八进制形式输出整数。
  - x 格式：以无符号十六进制形式输出整数。
  - u 格式：以无符号十进制形式输出整数。
  - c 格式：输出一个字符。
  - s 格式：用来输出一个串。
  - f 格式：用来输出实数（包括单、双精度），以小数形式输出。
  - e 格式：以指数形式输出实数。
  - g 格式：自动选 f 格式或 e 格式中较短的一种输出，且不输出无意义的零。

注：若想输出字符"%"，则需在"格式控制"字符串中用连续两个%表示。

argument：指向需要输出的字符串的指针。

例如：

```
printf("%.3f",12.3456);          //输出结果为 12.346
printf("%.9f",12.3456);          //输出结果为 12.345600000,不足位补 0
```

4. fprintf( ) 函数

fprintf( )函数用来将输出的内容输出到硬盘上的文件或是相当于文件的设备上。fprintf( )函数的原型如下：

```
#include <stdio.h> /*标准输入输出头文件,包含了标准输入输出函数(如 perror 和 printf 等)的
```

```
                    定义 */
int fprintf( FILE  * stream, const char  * format[ , argument, … ] );
```

调用 fprintf( ) 函数向文件指针指向的文件输出 ASCII 代码时其调用方法为：

```
fprintf( 文件指针, "输出格式", 输出项系列);
```

调用 fprintf( ) 函数向显示器输出错误信息时其调用方法为：

```
fprintf( stderr, "错误信息" );
```

在上述 fprintf( ) 函数的原型中，各参数的含义如下。

stream：指向用于接受输出的设备或文件的指针。

format：参数的输出格式，具体可参见 printf( ) 函数。

argument：指向需要输出的字符串的指针。

5. memset( ) 函数

memset( ) 函数用来对一段内存地址空间全部设置为某个值或清空（否则，可能会在测试当中出现野值），一般用在对定义的字符串进行初始化，也可用来方便地清空一个结构类型的变量或数组；memset( ) 函数的原型如下：

```
#include  < mem. h > //提供 memset( ) 函数原型的定义
void *  memset( void *  s,  int c,  size_t n );
```

在上述 memset( ) 函数的原型中，各参数的含义如下。

s：指向目标内存地址空间的起始地址的指针。

c：要赋的值。

n：要赋值的长度（字节数）。由该参数可知，memset( ) 函数是以字节为单位来进行赋值的。

例 3-1 调用 memset( ) 函数给数组赋值。

① 调用 memset( ) 函数给整型数组赋值：

```
int buf[ 50 ];
memset( buf,  0,  50 * sizeof( int ) );
```

② 调用 memset( ) 函数给字符型数组赋值：

```
char buf[ 50 ];
memset(  buf, '\0',  50);    /* 转义符'\0'为 C 语言中的字符串结束符,在数值类型里代表数字 0,
                                即 8 位的 00000000 */
```

例 3-2 调用 memset( ) 函数给结构赋值。

```
struct sample_struct {
    char csName [ 16 ];
    int iSeq;
    int iType;
};
```

```
struct sample_strcut stTest;
memset (&stTest, 0, sizeof (struct sample_struct));
```

例 3-3　调用 memset( ) 函数给结构数组赋值。

```
struct sample_struct TEST [10];
memset (TEST, 0, sizeof (struct sample_struct) * 10);
```

6. strcmp( ) 函数

strcmp( ) 函数用于比较两个字符串的大小。实际上，字符串的比较是比较字符串中各对字符的 ASCII 码：首先比较两个字符串的第一个字符，若不相等则停止比较并得出大于或小于的结果；若相等就接着比较第二个字符然后第三个字符等等；若两个字符串前面的字符均相等，像"network"和"networks"的前七个字符都相同，则比较第八个字符，字符串"network"的第八字符为字符串的结束符'\0'，而字符串"networks"的第八个字符为's'，由于'\0'的 ASCII 码小于's'的 ASCII 码，于是可得出结果字符串"network"小于字符串"networks"。因此，无论两个字符串是什么样，strcmp( ) 函数最多比较到其中一个字符串的结束符'\0'为止，就能得出结果。strcmp( ) 函数的原型如下：

```
#include <string.h>//提供字符串函数 stpcpy、strcat、strcmp 等原型的定义
int strcmp(const char *s1,const char * s2);
```

在上述 strcmp( ) 函数的原型中，各参数的含义如下。

s1：指向用于比较的第一个字符串的指针。

s2：指向用于比较的第二个字符串的指针。

当 s1 < s2 时，strcmp( ) 函数的返回值 < 0；当 s1 = s2 时，strcmp( ) 函数的返回值 = 0；当 s1 > s2 时，strcmp( ) 函数的返回值 > 0。

7. atoi( ) 函数

atoi( ) 函数用于将一个字符串转换成一个整型数值，若成功转换将返回转换后得到的整型数值，若失败则返回 0。atoi( ) 函数的原型如下：

```
#include <stdlib.h> /*C 标准库头文件,包含了 C 语言标准库函数(如 exit、atoi 等)原型的定
                      义*/
int atoi( const char *str );
```

在上述 atoi( ) 函数的原型中，其参数的含义如下。

str：待转换为整型数值的字符串。

atoi( ) 函数会扫描待转换为整型数值的字符串 str，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而等到再遇到非数字或字符串结束符'\0'时就结束转换并将结果返回。

例如：

```
char *a = " -100abc";
char *b = "456.12";
int c;
```

```
c = atoi(a) + atoi(b); /* c 的值为 356( -100 +456) */
```

8. 可变参数函数

可变参数函数的参数个数是可变的。一般情况下，所编的程序中的函数的参数个数都是固定的，但是有时候需要用到可变参数的函数。要在函数中包含可变个数的参数，首先应该在头文件中包含 <stdarg.h>，即#include <stdarg.h>。该头文件声明了一个 va_list 类型和四个操作可变参数的函数：

```
void va_start (va_list ap, argN);
void va_copy (va_list dest, va_list src);
type va_arg (va_list ap, type);
void va_end (va_list ap);
```

可变参数函数的所有操作均是主要围绕头文件中声明的上述 va_list 类型和四个宏（函数）va_start()，va_copy()，va_arg()和 va_end()。

va_list：该变量主要用来操纵整个可变参数列表；

va_start()：该函数主要用来初始化 va_list 类型的参数 ap，并且使得 ap 指向第一个可选参数；后面的参数 argN 一般指的是紧邻可变参数的前面一个固定参数（ANSI C 中要求可变参数函数在可变参数之前至少得有一个固定参数）；

va_arg()：该函数主要用于返回参数 ap 所指向的列表中的参数的下一个参数，每一次调用 va_arg()都会修改 ap 的值，这样就能正确的返回参数列表中的所有参数值；后面的 type 参数是用来存储参数 ap 所指向的参数的数据类型；

va_copy()：该函数用于复制 va_list 类型的变量；

va_end()：每次调用 va_start()函数和 v_copy()函数之后，都要调用 va_end()函数来销毁变量 ap，即将指针置为 NULL；

例 3-4 定义一个可变参数函数 errexit()用于向标准出错文件输出不同格式的出错信息。

```
//以下代码保存于文件 errexit.c
#include <stdarg.h>          /* 提供 C 标准库的 va_list 类型和 va_start 、va_arg 、va_end 宏的定义 */
#include <stdio.h>           /* 标准输入输出头文件,包含了标准输入输出函数(如 perror 和 printf 等)的定义 */
#include <stdlib.h>          /* C 标准库头文件,包含了 C 语言标准库函数(如 exit、atoi 等)原型的定义 */
int errexit(const char *format,…) { /* 定义可变参数函数 errexit(),该函数包含一个固定参数 format,可变参数用省略号"…"表示 */
va_list args;                //声明一个 va_list 类型的变量 args
va_start(args, format);      /* 对变量 args 进行初始化,使得 args 指向可变参数列表中的第一个可选参数;format 为紧邻可变参数"…"的前面一个固定参数 */
vfprintf(stderr, format, args);   /* 调用 vfprintf()函数向标准出错文件输出一个出错信息 */
va_end(args);                //调用 va_end()函数来销毁变量 args,释放其所占资源
exit(1); /* exit()函数通常是用于在子程序中来终结程序/进程的，在调用 exit()函数之后，程
```

```
        序将自动结束。其中，exit（0）表示程序是正常退出，exit（1）或 exit（-1）表
        示程序是异常（出错）退出*/
}
```

注：vfprintf( )函数的功能和 printf( )函数类似，都是用于向一个标准输出设备或标准的字符流输出格式化后的字符串，*vfprintf( )函数的原型如下*：

```
#include <stdarg.h>    /*提供C标准库的 va_list 类型和 va_start、va_arg、va_end 宏的定义*/
#include <stdio.h>     /*标准输入输出头文件，包含了标准输入输出函数（如 perror 和 printf
                         等）的定义*/
#include <stdlib.h>    /*C标准库头文件，包含了C语言标准库函数（如 exit、atoi 等）原型的
                         定义*/
int vfprintf (FILE *fp, const char *format, va_list arglist);
```

在上述 vfprintf( )函数的原型中，各参数的含义如下。

fp：指向输出文件的指针。

format：参数的输出格式，具体可参见 printf( )函数。

arglist：va_list 类型的参数列表。

9. main( )函数

C 程序是从 main( )函数开始执行的，其原型如下：

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

在上述 main( )函数的原型中，各参数的含义如下。

argc：指用命令行方式输入的命令行参数的个数。

argv：实际存储了所有输入的命令行参数。

假如编写的程序是 hello.exe，如果在命令行方式下运行该程序（首先应该在命令行方式下用 cd 命令进入到 hello.exe 文件所在目录），则运行改程序的命令为“hello.exe a b c”；此时，argc 的值将为“4”，其中，argv[0]的值是"hello.exe"，argv[1]的值是"a"，argv[2]的值是"b"，argv[3]的值是"c"。

10. switch…case 分支判断语句

```
switch(表达式) {
    case 表达式条件1:
        执行相应操作处理1;
        break;
    case 表达式条件2:
        执行相应操作处理2;
        break;
    ...
    case 表达式条件N:
        执行相应操作处理N;
        break;
    default: /*除以上三个条件外的其他条件*/
```

```
        执行相应其他操作处理;
        break;
}
```

例 3-5

```
switch(a){
    case 1: /*若变量 a 等于 1*/
            printf("1");
            break;
    case 2: /*若变量 a 等于 2*/
            printf("2");
            break;
    case 4: /*若变量 a 等于 4*/
            printf("4");
            break;
    default: /*若变量 a 不等于 1、2 或 4*/
            printf("other nums");
            break;
    }
```

11. time( )函数

time( )函数的功能是用于返回当前的日历时间，若调用该函数时发生错误则返回零。time( )函数的原型如下：

```
#include <time.h> //提供 time( )函数原型的定义
time_t time (time_t *time);
```

在上述 time( )函数的原型之中，其参数的含义如下。

time：指向用于存储当前时间的缓冲区的指针。

12. ctime( )函数

ctime( )函数返回一个字符串指针，其功能是用于将日历时间转换为字符串形式的本地时间。ctime( )函数的原型如下：

```
#include <time.h> //提供 time( )函数原型的定义
char *ctime(const time_t *timer);
```

在上述 ctime( )函数的原型之中，其参数的含义如下。

timer：指向存储有当前日历时间的缓冲区的指针，该参数一般是通过调用函数 time( )获得。

例如：

```
time_t t;
time(&t);
printf("Today's date and time: %s\n", ctime(&t));
```

13. strerror( ) 函数

strerror( ) 函数的功能是用于返回对应于某个错误编号的错误原因的描述字符串，从而得到一个可读的出错提示信息，而不再只是得到一个冷冰冰的错误编号数字。strerror( ) 函数的原型如下：

```
#include < string. h >                    //提供字符串函数原型的定义
char * strerror( int errnum) ;
```

在上述 strerror( ) 函数的原型之中，其参数的含义如下。

errnum：错误编号。

注：C 语言中在头文件 < errno. h > 中定义有一个全局变量 errno 来记录程序出错时的对应错误编号。

14. fputs( ) 函数

fputs( ) 函数的功能是用于向指定的文件写入一个字符串，如果调用成功将返回 0，否则将返回 -1。fputs( ) 函数的原型如下：

```
#include < stdio. h > / * 标准输入输出头文件,包含了标准输入输出函数(如 perror 和 printf 等)的
                      定义 * /
int fputs( char * string, FILE * stream) ;
```

在上述 fputs( ) 函数的原型之中，各参数的含义如下。

string：需送入流的字符串指针；

stream：一个 FILE 型的指针。

3.3.2　服务器例程剖析

1. 例程功能描述

该服务器例程能够反复读取来自客户的任何请求，且一旦收到了客户的请求之后，则计算出服务器的当前时间，并将该时间值作为响应回送给发送请求的那个客户。

2. 例程源码剖析

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

首先，从命令行输入参数中获取服务器的服务名（端口号）；然后调用 passiveUDP( ) 函数基于得到的服务器的服务名（端口号）建立服务器端的 UDP 被动套接字；然后再反复调用 recvfrom( ) 函数接收来自客户的请求；然后调用 time( ) 函数计算出服务器的当前时间；最后，调用 sendto( ) 函数将计算出来的服务器的当前时间发送给该客户。参数说明：argc 记录了命令行输入的参数个数，argv 则存储了输入的各个参数值。

```
//以下代码保存于文件 UDPtimed. c
#include  < sys/types. h >        //该头文件中定义了各种数据类型
#include  < sys/socket. h >       //该头文件中包含套接字 API 中系统函数的定义
```

```
#include <netinet/in.h>          //该头文件中定义了数据结构 sockaddr_in
#include <stdio.h>               //标准输入/输出函数所在的头文件
#include <time.h>                //该头文件包含了有关时间计算的各类函数
#include <string.h>              //该头文件中包含了 strerror()函数的定义
#include <errno.h>               //该头文件中包含了全局变量 errno 的定义
extern int errno;                //外部全局变量 errno 记录了系统函数出错时的错误编号
int passiveUDP(const char *service); /*声明用于建立被动 UDP 套接字的子函数 passiveUDP()*/
int errexit(const char *format,…); /*声明用于输出不同格式的出错信息的可变参数子函数 errexit()*/
#define UNIXEPOCH 2208988800UL  /*定义 UNIX 系统的时间纪元常量 UNIXEPOCH,由于 UNIX/LINUX 操作系统中的时间是从 1970 年 1 月 1 日零时开始计算的,与用因特网时间纪元(从 1900 年 1 月 1 日零时开始计算)测量的时间值相差 2 208 988 800 秒,因此,从 UNIX/LINUX 操作系统获得的时间需要在转换为用因特网纪元测量的时间值之后,才能通过网络回送给客户*/
#define MSG "what time is it ? \n"    //定义符号常量 MSG
int main(int argc, char *argv[]) {
    struct sockaddr_in fsin; //定义服务器端套接字的端点地址结构变量 fsin
    char *service = "time"; //定义用于存储服务名(端口号)的变量 service
    char buf[1]; /*定义变量 buf 以存储来自客户的请求,由于无论来自客户的请求是什么内容,有多少字节,服务器都统一回送当前时间作为响应,因此服务器无需将客户请求中的所有内容均读完并保存,而只需读取来自客户请求中的 1 个字节,即可回送当前时间作为响应,故 buf 缓冲区只需 1 个字节大小即可*/
    int sock;      //定义服务器端套接字描述符变量 sock
    time_t now;   //定义用于存储当前时间的变量 now
    unsigned int alen;   /*定义用于存储服务器端套接字的端点地址结构长度的变量 alen*/
    switch(argc) { //判断命令行输入的参数的个数
    case 1: /*若只输入了一个参数,即只输入了可执行文件名,而没有输入服务器的服务名(端口号),则直接跳出该分支判断语句*/
        break;
    case 2: /*若输入了两个参数,即除了输入了可执行文件名之外,还输入了服务器的服务名(端口号),则在将服务名(端口号)保存到变量 service 中之后,再跳出该分支判断语句*/
        service = argv[1];
        break;
    default: /*若输入了三个或以上参数,则调用 errexit()函数提示出错信息(用法提示)并退出系统*/
        errexit("usage: UDPtimed [port]\n");
    }
    sock = passiveUDP(service); //调用 passiveUDP()建立 UDP 被动套接字
    while(1) { //反复读取来自客户的请求
        alen = sizeof(fsin);
```

```
        if(recvfrom(sock,buf,sizeof(buf),0,(struct sockaddr *)&fsin, &alen) <0)
            errexit("recvfrom: %s\n", strerror(errno));
    /*若调用 recvfrom()函数接收一个新的来自客户的请求时出错,则调用 errexit()函数显示出错信息并退出系统*/
        (void) time(&now); /*若调用 recvfrom()函数接收一个新的来自客户的请求成功,则调用 time()函数获得服务器当前时间并存入到变量 now 之中,这里将 time()函数强制转换为 void 类型,表示 time()函数无需返回值*/
        now = htonl((unsigned long)(now + UNIXEPOCH)); /*将 LINUX 服务器的当前时间转换为用因特网纪元测量的时间值*/
        (void) sendto(sock,(char *)&now, sizeof(now), 0, (struct sockaddr *) &fsin, sizeof(fsin)); /*将该时间值作为响应发送给该客户,这里将 sendto()函数强制转换为 void 类型,表示 sendto()函数无需返回值*/
    }
}
```

(2) 子函数 passiveUDP (const char *service)

基于指定的服务名，建立服务端 UDP 被动套接字。参数说明：service 为服务器所提供服务的服务名。

```
//以下代码保存于文件 passiveUDP.c
int passivesock(const char *service, const char *transport, int qlen);
/*声明用于建立被动套接字的子函数 passivesock()*/
int passiveUDP(const char *service){
    return passivesock(service, "UDP", 0); /*调用子函数 passivesock(),基于指定的服务名和 UDP 协议,建议服务端 UDP 被动套接字*/
}
```

(3) 子函数 passivesock (const char *service, const char *transport, int qlen)

基于指定的服务名与协议，建立服务端被动套接字，返回值为新创建的套接字的描述符。参数说明：service 为服务器所提供服务的服务名，transport 为通信所采用的协议名，qlen 为设置的客户请求等待队列的长度。

```
//以下代码保存于文件 passivesock.c
#include <sys/types.h>    //该头文件中定义了各种数据类型
#include <sys/socket.h>   //该头文件中包含套接字 API 中系统函数的定义
#include <netinet/in.h>   //该头文件中定义了数据结构 sockaddr_in
#include <stdlib.h>       //该头文件中定义了 atoi()、exit()等 C 语言标准库函数
#include <string.h>       //该头文件中包含了 strerror()函数的定义
#include <errno.h>        //该头文件中包含了全局变量 errno 的定义
#include <netdb.h> /*该头文件中定义了与网络有关的结构、变量类型、宏、函数,如 gethostbyname()函数等*/
extern int errno; //外部全局变量 errno 中记录了程序出错时的对应错误编号
int errexit(const char *format,…); /*声明用于输出不同格式的出错信息的可变参数子函数 errexit()*/
```

```
unsigned int portbase = 0; /* 定义全局变量 portbase,采用该变量的主要目的是为了允许在同一台
                              机器上测试服务器新版本时仍可继续运行服务器工作版本,或在同一
                              台机器上同时测试多个服务器新版本。由于在同一台机器上,两个服
                              务器程序不能使用相同的协议端口号,因此采用该变量之后只需要将
                              不同版本的 portbase 变量设置为不同值即可,而无需再对程序中引用
                              端口号的每一个地方都去进行修改,从而减少程序出错率 */
int passivesock(const char *service, const char *transport, int qlen){
    struct servent *pse; /* 当调用 getservbyname()函数由服务名获得服务器熟知端口号时,定义
                            数据结构 pse 用于保存 getservbyname()函数的返回值 */
    struct protoent *ppe; /* 当调用 getprotobyname()函数由协议名获得协议号时,定义数据结构
                             ppe 用于保存 getprotobyname()函数的返回值 */
    struct sockaddr_in sin; //定义服务器端套接字的端点地址结构变量 sin
    int s, type; /* 定义服务器端套接字描述符变量 s 和套接字提供的服务类型变量 type */
    memset(&sin, 0, sizeof(sin)); /* 调用 memset()函数清空服务器端套接字的端点地址结构
                                     变量 sin */
    //以下代码段实现为服务器端点地址结构变量 sin 赋值
    sin.sin_family = AF_INET; //赋值协议族为 TCP/IP 协议族
    sin.sin_addr.s_addr = INADDR_ANY; /* 赋值 IP 地址为符号常量 INADDR_ANY,需将符号常
                                         量 INADDR_ANY 由本机字节顺序转换为网络字节
                                         顺序 */
    if(pse = getservbyname(service, transport))
        sin.sin_port = htons(ntohs((unsigned short)pse->s_port) + portbase);
        /* 若成功调用 getservbyname()函数由服务名获得对应的熟知端口号,则调用 htons()函
数将端口号由主机字节顺序转换为网络字节顺序之后,再赋值给服务器端点地址结构变量 sin 中
的 sin_port 字段 */
    else if((sin.sin_port = htons((unsigned short)atoi(service))) == 0))
     errexit("can'get \" %s \" service entry \n", service);
    /* 若调用 getservbyname()函数由服务名获得对应的熟知端口号时出错,则判断变量 service
       中存储的不是服务名,而是端口号,于是先试着调用 atoi()函数将端口号由字符串转换为
       整型数值,然后在调用 htons()函数将转换得到的整型端口号值由主机字节顺序转换为网
       络字节顺序之后,再赋值给服务器端点地址结构变量 sin 中的 sin_port 字段。若上述赋值
       过程出错,则调用 errexit()函数提示相应出错信息并退出系统 */
    if((ppe = getprotobyname(transport)) == 0)
        errexit("can'get \" %s \" protocol entry \n", transport);
        /* 若调用 getprotobyname()函数由协议名获得对应的协议号时出错,则调用 errexit()函
           数提示出相应错信息并退出系统 */
    if(strcmp(transport,"udp") == 0)
        type = SOCK_DGRAM; //若协议名为 UDP,则设置服务类型为 SOCK_DGRAM
    else
        type = SOCK_STREAM; //否则,设置服务类型为 SOCK_STREAM
    s = socket(PF_INET, type, ppe->p_proto); //调用 socket()函数创建服务器端套接字
    if(s < 0)
        errexit("can'creat socket: %s \n", strerror(errno));
```

```
        /* 若创建服务器端套接字出错,则调用 errexit()提示出相应错信息并退出系统 */
    if(bind(s, (struct sockaddr *)&sin, sizeof(sin)) <0)
        errexit("can'bind to %s port: %s \n", service, strerror(errno));
/* 调用 bind()函数将创建的端套接字与端点地址绑定;若绑定出错,则调用 errexit()函数提示出
   相应错信息并退出系统 */
    if(type == SOCK_STREAM && listen(s, qlen) <0)
        errexit("can'listen on %s port: %s \n", service, strerror(errno));
/* 若创建的端套接字是 TCP 套接字,则调用 listen()函数将该套接字设置为被动模式,且设置等
   待队列长度为 qlen;若调用 listen()函数时出错,则调用 errexit()函数提示出相应错信息并退出
   系统 */
    return s; //将创建的套接字的描述符作为子函数 passivesock()的返回值
}
```

3.4　访问 TIME 服务的无连接的客户端例程

客户使用 UDP 在远程服务器协议端口 37 上访问 TIME 服务以获取服务器的当前时间信息。显然，客户仅只需发送一个包含任意内容的数据报请求给服务器即可，服务器在收到该请求后也无需对其进行处理，而只是需要从中获得客户端的端点地址以便在应答中使用，然后再将本机的当前时间入应答数据报中并回送给客户。为此，访问 TIME 服务的无连接的客户端例程的 C 语言编码可描述如下：

```
//以下代码保存于文件 UDPtime.c
#include <sys/types.h> //该头文件中定义了各种数据类型
#include <unistd.h> /* 该头文件中包含了 Linux/Unix 的系统函数如 read(),write()等的定
                       义 */
#include <stdlib.h> //该头文件中包含了的 C 语言标准库函数如 atoi()、exit()等的定义
#include <stdio.h> //标准输入/输出函数所在的头文件
#include <string.h> //该头文件中包含了 strerror()函数的定义
#include <errno.h> //该头文件中包含了全局变量 errno 的定义
#define BUFSIZE 64 //定义缓冲区大小为 64 字节
extern int errno; //外部全局变量 errno 记录了系统函数出错时的错误编号
int connectUDP(const char *host, const char *service); /* 声明用于建立被动 UDP 套接字的子函
                                                          数 passiveUDP() */
int errexit(const char *format,…); /* 声明用于输出不同格式的出错信息的可变参数子函数 er-
                                      rexit() */
#define UNIXEPOCH 2208988800UL
int main(int argc, char *argv[]) {
    char *host = "localhost"; //定义用于存储服务器主机名(IP 地址)的变量 host
    char *service = "time"; //定义用于存储服务名(端口号)的变量 service
    time_t now; /* 定义用于存储应答数据报中所携带的服务器时间的变量 now */
    int s, n; /* 定义用于存储客户端套接字描述符的变量 s 和用于存储从套接字中每次所读取
                 到的字节的长度的变量 n */
```

```
    switch(argc) { //判断命令行输入的参数的个数
    case 1: /* 若只输入一个参数,即只输入可执行文件名,而没有输入服务器的主机名(IP 地
            址)与服务名(端口号),则将服务器的主机名设置为默认的主机名"localhost",然
            后跳出该分支判断语句 */
        host = "localhost"
        break;
    case 2: /* 若输入两个参数,即除了输入可执行文件名之外,还输入服务器的主机名(IP 地
            址),则在将服务器的主机名(IP 地址)保存到变量 host 中之后,再跳出该分支判断
            语句 */
        host = argv[1];
        break;
    case 3: /* 若输入三个参数,即除了输入可执行文件名之外,还输入服务器的主机名(IP 地
            址)与服务名(端口号),则在将服务器的主机名(IP 地址)保存到变量 host 中、同
            时将服务器的服务名(端口号)保存到变量 service 中之后,再跳出该分支判断语
            句 */
        host = argv[1];
        service = argv[2];
        break;
default: /* 若输入四个或以上参数,则调用 fprintf()函数输出用法提示,然后调用 exit(1) 使得系
          统异常(出错)退出 */
        fprintf(stderr,"usage: UDPtimed[ host [ port] ] \n");
        exit(1);
    }
    s = connectUDP(host, service); /* 调用 connectUDP()建立客户端套接字并将该套接字连接
                                      到远程服务器的端点地址 */
    (void) write(s, MSG, strlen(MSG)); /* 调用 write()通过新建的套接字将符号常量 MSG 中
                                          的内容发送到服务器 */
    n = read(s, (char *)&now, sizeof(now)); /* 调用 read()从套接字中读取服务器的应答并
                                               存储到变量 now 中 */
    if(n<0)
        errexit("read failed: %s\n", strerror(errno)); /* 若没有读取到服务器的应答,则调用 er-
                                                          rexit()提示出错信息并退出系统 */
    now = ntohl((unsigned long) now); /* 若读取到服务器的应答,则调用 ntohl()将变量 now 中
                                         保存的服务器的应答(即服务器的当前时间)值由网络
                                         字节顺序转换为本机字节顺序 */
    now -= UNIXEPOCH; /* 若读取到服务器的应答,则将用因特网纪元测量的时间值转换为
                         LINUX 服务器的当前时间 */
    printf("%s", ctime(&now)); /* 调用 ctime()将长整型的时间值转换为时间格式的时间值,
                                  然后再调用 printf()函数输出当前时间 */
    eixt(0); //调用 exit(0)正常退出系统
}
```

3.5　循环的面向连接的 DAYTIME 服务器例程

1. 例程功能描述

该服务器例程能够反复读取来自客户的任何连接请求，且一旦收到了客户的连接请求之后，则建立与该客户的一个连接和一个用于处理该连接的临时套接字，然后计算出服务器的当前日期和时间，并将该当前日期和时间值作为响应通过所建立的临时套接字回送给该客户，关闭该临时套接字，然后再读取下一个客户连接请求。

2. 例程源码剖析

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

首先，从命令行输入参数中获取服务器的服务名（端口号）；然后调用 passiveTCP( ) 函数基于得到的服务器的服务名（端口号）建立服务器端的 TCP 被动套接字；然后再反复调用 accept( ) 函数接收来自客户的连接请求、建立与客户的连接并创建一个用于处理该连接的临时套接字；然后调用 time( ) 函数计算出服务器的当前日期和时间，并调用 write( ) 函数通过所建立的临时套接字将该当前日期和时间值作为响应回送给该客户，最后，再关闭该临时套接字。

```
//以下代码保存于文件 TCPdaytimed.c
#include <sys/types.h>      //该头文件中定义了各种数据类型
#include <sys/socket.h>     //该头文件中包含套接字 API 中系统函数的定义
#include <netinet/in.h>     //该头文件中定义了数据结构 sockaddr_in
#include <unistd.h>         /*该头文件中包含 LINUX/UNIX 的系统函数,如 read( ),write( )等
                              的定义*/
#include <stdio.h>          //标准输入/输出函数所在的头文件
#include <time.h>           //该头文件包含有关时间计算的各类函数
#include <string.h>         //该头文件中包含 strerror( )函数的定义
#include <errno.h>          //该头文件中包含全局变量 errno 的定义
extern int errno; //外部全局变量 errno 记录了系统函数出错时的错误编号
int passiveTCP(const char *service); /*声明用于建立被动 TCP 套接字的子函数 passiveTCP( )*/
int errexit(const char *format,…); /*声明用于输出不同格式的出错信息的可变参数子函数 er-
                                     rexit( )*/
void TCPdaytimed(int fd); /*声明用于计算出服务器的当前时间,并调用 write( )函数通过所建立
                            的临时套接字将该时间值作为响应回送给该客户的子函数 TCPday-
                            timed( )*/
int main(int argc, char *argv[ ]) {
    struct sockaddr_in fsin;  //定义服务器端套接字的端点地址结构变量 fsin
    char *service = "daytime";   //定义用于存储服务名(端口号)的变量 service
    int msock, ssock;         /*定义服务器端永久套接字描述符变量 msock 与临时套接字描述符
                                变量 ssock*/
    unsigned int alen;        //定义用于存储服务器端套接字的端点地址结构长度的变量 alen
    switch(argc) {            //判断命令行输入的参数的个数
```

```
        case 1: /* 若只输入一个参数,即只输入可执行文件名,而没有输入服务器的服务名(端口
                   号),则直接跳出该分支判断语句 */
            break;
        case 2: /* 若输入两个参数,即除输入了可执行文件名之外,还输入服务器的服务名(端口
                   号),则在将服务名(端口号)保存到变量 service 中之后,再跳出该分支判断语
                   句 */
            service = argv[1];
            break;
        default: /* 若输入三个或以上参数,则调用 errexit()函数提示出错信息(用法提示)并退出系
                   统 */
            errexit("usage: TCPdaytimed [port]\n");
        }
        msock = passiveTCP(service); //调用 passiveTCP()建立 TCP 被动套接字
        while(1) { //反复读取来自客户的连接请求
            alen = sizeof(fsin);
            ssock = accept(msock, (struct sockaddr *) &fsin, &alen);
        /* 调用 accept()接收来自客户的连接请求,建立与客户的连接并创建一个用于处理该连接
           的临时套接字 ssock */
            if(ssock < 0) /* 若创建临时套接字出错,则调用 errexit()提示出错信息并退出系统 */
              errexit("accept failed: %s \n", strerror(errno));
            TCPdaytimed(ssock); /* 若创建临时套接字成功,则调用子函数 TCPdaytimed()通过所创
                                   建的临时套接字 ssock 与客户进行交互 */
            (void)close(ssock); //与客户交互完毕,则调用 close()关闭所创建的临时套接字
        }
    }
    void TCPdaytimed(int fd) { /* 子函数 TCPdaytimed(int fd):调用 time()函数计算出服务器的当前
                                  时间,然后调用 write()函数通过所建立的临时套接字将该时间值
                                  作为响应回送给该客户 */
        char *pts; //定义用于存放时间值的字符串指针变量 pts
        time_t now; //定义用于存放时间值的结构变量 now
        char *ctime(); //在函数 TCPdaytimed()内部声明 ctime()函数原型
        (void)time(&now); //调用 time()函数获得服务器当前时间并存入到变量 now 之中
        pts = ctime(&now); /* 调用 ctime()函数将变量 now 之中存储的日历时间转换为字符串形式
                              的本地时间 */
        (void)write(fd, pts, strlen(pts)); /* 调用 write()函数将服务器的当前日期和时间通过临时
                                              套接字回送给客户 */
    }
```

(2) 子函数 passiveTCP (const char *service)

基于指定的服务名，建立服务端 TCP 被动套接字。参数说明：service 为服务器所提供服务的服务名。

```
/* 以下代码保存于文件 passiveTCP.c */
int passivesock(const char *service, const char *transport, int qlen);
```

```
/* 声明用于建立被动套接字的子函数 passivesock() */
int passiveTCP(const char *service)|
    return passivesock(service,"TCP", qlen); /* 调用子函数 passivesock(),基于指定的服务名和
                                                TCP 协议,建议服务端 TCP 被动套接字 */
|
```

3.6　访问 DAYTIME 服务的面向连接的客户端例程

(1) 主函数 main(int argc, char *argv[])

客户使用 TCP 在远程服务器协议端口 13 上访问 DAYTIME 服务以获取服务器的当前日期和时间信息。显然，客户仅只需发送一个包含任意内容的数据报请求给服务器即可，服务器在收到该请求后也无需对其进行处理，而只是需要从中获得客户端的端点地址以便在应答中使用，然后再将本机的当前时间入应答数据报中并回送给客户。为此，访问 DAYTIME 服务的无连接的客户端例程的 C 语言编码可描述如下：

```
//以下代码保存于文件 TCPdaytime.c
#include <sys/types.h>    //该头文件中定义了各种数据类型
#include <unistd.h>       /* 该头文件中包含 LINUX/UNIX 的系统函数,如 read(),write()等的
                             定义 */
#include <stdlib.h>       //该头文件中包含 C 语言标准库函数,如 atoi()、exit()等的定义
#include <stdio.h>        //标准输入/输出函数所在的头文件
#include <string.h>       //该头文件中包含 strerror()函数的定义
#include <errno.h>        //该头文件中包含全局变量 errno 的定义
#define LINELEN 128       //定义缓冲区大小为 128 字节
extern int errno;         //外部全局变量 errno 记录了系统函数出错时的错误编号
int connectTCP(const char *host, const char *service); /* 声明用于建立被动 UDP 套接字的子函
                                                          数 passiveUDP() */
int errexit(const char *format,…); /* 声明用于输出不同格式的出错信息的可变参数子函数 er-
                                      rexit() */
int TCPdaytime(const char *host, const char *service);
/* 声明用于从套接字读出服务器回送的当前日期与时间的子函数 TCPdaytimed() */
int main(int argc,char *argv[])|
    char *host="localhost"; /* 定义用于存储服务器主机名(IP 地址)的变量 host */
    char *service="daytime"; //定义用于存储服务名(端口号)的变量 service
    switch(argc)| //判断命令行输入的参数的个数
    case 1: /* 若只输入一个参数,即只输入可执行文件名,而没有输入服务器的主机名(IP 地
               址)与服务名(端口号),则将服务器的主机名设置为默认的主机名"localhost",然
               后跳出该分支判断语句 */
        host="localhost"
        break;
    case 2: /* 若输入两个参数,即除了输入可执行文件名之外,还输入服务器的主机名(IP 地
               址),则在将服务器的主机名(IP 地址)保存到变量 host 中之后,再跳出该分支判断
```

```
                语句 */
            host = argv[1];
            break;
        case 3: /* 若输入三个参数,即除了输入可执行文件名之外,还输入服务器的主机名(IP 地址)
                与服务名(端口号),则在将服务器的主机名(IP 地址)保存到变量 host 中,同时将服
                务器的服务名(端口号)保存到变量 service 中之后,再跳出该分支判断语句 */
            host = argv[1];
            service = argv[2];
            break;
    default: /* 若输入四个或以上参数,则调用 fprintf()函数输出用法提示,然后调用 exit(1) 使得系
              统异常(出错)退出 */
            fprintf(stderr,"usage: TCPtimed[host [port]] \n");
            exit(1);
        }
        TCPdaytime(host, service);
        /* 若命令行输入正常,则调用 TCPdaytime()函数进行后续处理 */
        eixt(0); //调用 exit(0)正常退出系统
}
    TCPdaytime(const char *host, const char *service) {
    /* 子函数 TCPdaytime(const char *host, const char *service):基于指定的主机名与服务名,调
        用 connectTCP()函数建立与客户的连接,并创建一个被动 TCP 套接字来处理该连接。参数
        说明:host 为服务器的主机名(IP 地址),service 为服务器所提供服务的服务名(端口号) */
    char buf[LINELEN + 1]; //定义用于接收服务器响应信息的缓冲区 buf
    int s, n; /* 定义套接字描述符变量 s 和用于记录每次读取到的服务器响应信息的字节数的
                变量 n */
    s = connectTCP(host, service); /* 调用 connectTCP()函数向客户发送连接请求并创建一个被
                                    动 TCP 套接字用于来处理该连接 */
    while((n = read(s, buf, LINELEN)) > 0) { /* 循环调用 read()函数每次从套接字中读取
                                              LINELEN 个字节的来自服务器的响应信息,
                                              并存入到缓冲区 buf 之中 */
        buf[n] = '\0'; /* 在字符串 buf 的末尾添加字符串结束符'\0' */
        (void) fputs(buf, stdout); /* 调用 fputs()函数将 buf 中的数据在本地显示器(标准输出
                                    文件 stdout)上进行回显 */
    }
}
```

3.7 本章小结

本章主要对循环服务器的进程结构及循环服务器软件的算法设计流程进行了详细介绍，并给出了一个循环的无连接的 TIME 服务器与一个访问该 TIME 服务的无连接的客户端的完整 C 语言例程，以及一个循环的面向连接的 DAYTIME 服务器与一个访问该 DAYTIME 服务的面向连接的客户端的完整 C 语言例程。通过本章的学习，需要了解循环服务器的进程结

构；需要熟习循环服务器软件的算法设计流程；需要掌握 LINUX 环境下的客户端与循环服务器软件的 C 语言实现方法。

本章习题

1. 简述循环的无连接服务器的进程结构和循环的面向连接的服务器的进程结构，并指出两者之间的主要区别。

2. 简述循环的无连接服务器的算法流程和循环的面向连接的服务器的算法流程，并指出两者之间的主要区别。

3. 什么是可变参数函数？试设计出一个简单的可变参数函数。

4. sizeof( )与 strlen( )的主要区别是什么？

5. 试构造一个循环的无连接服务器例程，要求该服务器例程能够反复读取来自客户的任何请求，且一旦客户的请求中包含有"time"字段，则该服务器例程将计算服务器的当前时间，并将该时间值作为响应返回给发送请求的客户。

6. 试构造一个无连接客户端例程，要求该客户端例程不但能够将本机时间发送给服务器，而且同时还能够将接收到的服务器回应的应答消息在本机显示器上进行回显。

7. 试构造一个循环的面向连接的服务器例程，要求该服务器例程能够反复读取来自客户的任何连接请求，且一旦客户的请求中包含有"daytime"字段，则该服务器例程将计算服务器的当前时间，并将该时间值作为响应返回给发送请求的客户。

8. 试构造一个面向连接的客户端例程，要求该客户端例程不但能够将本机时间发送给服务器，而且同时还能够将接收到的服务器回应的应答消息在本机显示器上进行回显。

第 4 章　服务器中的并发机制

第 3 章介绍了两类循环服务器软件的设计流程及其实现的 C 语言例程。本章将在此基础上深入介绍服务器并发机制，同时，为了更清晰地说明服务器并发机制及其实现方法，本章还分别给出了一个创建并发进程与并发线程的完整 C 语言例程，并对基于多进程与基于多线程的服务器并发机制的性能进行了深入的分析比较。

4.1　服务器中的并发概念

4.1.1　循环服务器与并发服务器

由第 3 章的介绍可知，循环服务器是指服务器在同一时刻只可以响应一个客户端的请求的服务器；而在网络程序中，通常都是有多个客户端对应同一个服务器，因此，为了使服务器可以同时处理来自多个客户的请求，人们提出了并发服务器的概念。其中，所谓并发服务器就是指在同一个时刻可以处理来自多个客户端的请求的服务器。

由于用户需求、处理速率和通信能力的不同，往往在循环的和并发的服务器设计之中难以选择。循环服务器采用客户轮流等待的工作方式，具有设计、编程、调试和修改简单的优点，因此，在其响应时间可以满足需求的条件下（该时间可以在本地或全局网络中进行测试），可以采用循环服务器模式。如果构建一个响应需要大量的 I/O 操作，且各个请求所需要的处理时间差别非常大，或服务器在一台多处理器的计算机上运行，则可采用并发服务器模式来缩短响应时间。

4.1.2　基于多进程或多线程的服务器并发概念

当前计算机技术发展的突出特点是要求对广泛的信息与其他各类资源实现共享，从而促使了网络技术的普遍应用和快速发展，进而也要求操作系统必须为用户提供一个符合信息处理要求的分布式计算环境。因此现代操作系统一般均采用了微内核（Microkernel）结构。其中，微内核是指操作系统的小核心，它将各种操作系统共同需要的核心功能提炼出来，形成微内核的基本功能。这些操作系统的基本功能包括：IPC（Inter-Process Communication，进程通信），VM（Virtual Machine，虚拟机），Tasks（任务）管理、Threads（线程）管理，中断处理及与硬件相关部分等。这样一来，从功能上而言，微内核为各种操作系统打好了一个公共基础，或者说构成了基本操作系统。

微内核操作系统的模型如图 4.1 所示。由图 4.1 可见，微内核在核心态下工作，负责直接与硬件打交道；而操作系统的其他功能则由各服务器（除内核以外操作系统的其他部分被分成若干相对独立的进程，每个进程完成一组服务，称为服务器进程，简称服务器）实现，服务器处于微内核之上，在用户态下工作。各服务器同处一层，通过 SPI（Service Pro-

vider Interface，服务提供者接口）与微内核联系。各服务器之间相互独立但彼此间可以直接通信。微内核负责对整个操作系统中的各种来往消息进行验证，在各大部分之间进行消息传递，并保证它们对硬件的访问。

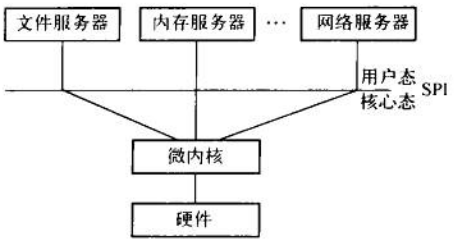


图 4.1　微内核操作系统模型

在微内核系统中，进程只是资源分配的单位，而真正可以在处理器（CPU）上独立调度运行的基本单位是线程。在多处理器系统中，每个线程在一个处理器上运行，从而实现应用程序的并发，使每个处理器都得到充分运行。因此，实际实现并发功能的是线程。进程和线程各自的优缺点可简单总结如下。

- 进程优点：编程、调试简单，可靠性较高。
- 进程缺点：创建、销毁、切换速度慢，内存、资源占用大。
- 线程优点：创建、销毁、切换速度快，内存、资源占用小。
- 线程缺点：编程、调试复杂，可靠性较差。

采用多进程或多线程的方式均可实现服务器的并发，但由以上关于进程和线程各自的优缺点的描述可知，进程和线程有着各自的特点，因此，在 C/S 通信中的服务端并发技术选型上，到底是该采用多线程还是该采用多进程的服务端并发技术呢？这样的争执由来已久。例如：在 Web 服务器技术中，Apache 是采用多进程的（每个客户连接对应一个进程，每个进程中只存在唯一一个执行线程），而 Java 的 Web 容器 Tomcat、Websphere 等则都是多线程的（每个客户连接对应一个线程，所有的线程都在同一个进程之中）。

4.1.3　并发等级

在并发服务器模式中，由于每一个访问连接都需要耗费一定的系统资源，过多的并发连接会将服务器系统资源消耗殆尽，从而导致服务器无法正常处理每一个客户络连接请求。为了避免服务器系统无法响应，有必要在服务器系统中对并发连接数量进行适当控制，以保证服务器能够有足够系统资源来处理每一个客户连接请求。其中，在某个给定时刻一个服务器中正在运行着的执行线程总数，称为该服务器的并发等级。

为了处理一个传入的客户连接请求，并发服务器均需创建一个新的从线程/进程，在处理完该请求之后，该从线程/进程再退出。因此，并发服务器的并发等级是随时间变化的。显然，服务器在任一时刻的并发等级反映了服务器已经收到、但还未处理完毕的客户请求的数目。不过，在程序设计中程序员一般无需关心某个服务器在某个给定时刻的并发等级，而只需关心服务器在整个生命周期中所展现出来的并发等级的最大值。

4.2 基于多进程的服务器并发机制

4.2.1 创建一个新进程

在 LINUX/UNIX 中一个现有进程可以调用 fork( ) 函数创建一个新进程。从本质上说，fork 函数将运行着的程序分成两个（几乎）完全一样的进程，每个进程都启动一个从代码的同一位置开始执行的线程。其中，由 fork( ) 函数所创建的新进程称为子进程（Child Process），而调用 fork( ) 函数的那个进程则称为父进程（Parent Process）。fork( ) 函数的原型如下：

```
#include <unistd.h> /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构
                       的定义 */
int fork( );
```

成功调用 fork( ) 函数之后，操作系统会复制出一个与父进程（几乎）完全相同的新进程，不过这两个进程虽说是父子关系，但是在操作系统看来，它们更像兄弟关系，这两个进程共享代码空间，但是数据空间是互相独立的，子进程数据空间中的内容是父进程的完整拷贝，指令指针也完全相同，子进程拥有父进程当前运行到的位置，也就是说子进程是从fork( ) 函数返回处开始执行的，但唯一不同的是，若 fork( ) 函数调用成功，则在子进程中其返回值为 0，而在父进程中其返回值为子进程的进程号（进程 ID）；若 fork( ) 函数调用不成功，则在父进程中其返回值为 -1。调用 fork( ) 函数创建一个新进程的基本方法主要有以下两种。

方法 1：

```
int pid;
pid = fork( );                    //调用 fork( ) 函数创建一个新进程
if( pid == -1 ) {                 //若调用 fork( ) 函数出错
    perror( "fork failed!" );
    exit(1);
}
else if( pid ==0 ) {              //以下是子进程所执行的操作
    printf( "This is the child process!" );
} else {                          //以下是父进程所执行的操作
    printf( "This is the parent process!" );
}
```

方法 2：

```
int pid;
pid = fork( );
switch( pid ) {
    case -1:                      //若调用 fork( ) 函数出错
        perror( "fork failed!" );
        exit(1);
```

```
    case 0:                    //以下是子进程所执行的操作
        printf("This is the child process!");
        break;
    default:                   //以下是父进程所执行的操作
        printf("This is the parent process!");
        break;
}
```

由以上给出的调用 fork( ) 函数的代码段可知，在调用 fork( ) 函数创建一个子进程之后，父子进程之间的关系可以这样想像：在 fork( ) 函数返回之前，这两个进程一直在同时运行，而且步调也保持着一致，但在 fork( ) 函数返回之后，它们虽然仍是在同时运行，但从此分别开始做不同的工作，也就是分岔了，这也是 fork( ) 函数为什么叫作 fork 的原因。不过在 fork( ) 函数返回之后，父子进程在同时运行时，到底是父进程先运行、还是子进程先运行，这就与操作系统的实际运行情况有关了。即：上述代码段的运行结果到底是先输出“This is the child process!”，还是先输出“This is the parent process!”是不确定的，则与操作系统的实际运行情况有关。

4.2.2　终止一个进程

在 LINUX 系统中，进程终止分为了正常终止和异常终止两种，可通过调用 exit( ) 函数来实现。exit( ) 函数在头文件 stdlib. h 中声明，其函数原型如下：

```
#include <stdlib.h>   /*C 标准库头文件,包含了 C 语言标准库函数(如 exit、atoi 等)原型的定义*/
void exit(int status);
```

上述 exit( ) 函数可用来终止当前进程的执行，并把参数 status 返回给当前进程的父进程，而当前进程所有的缓冲区数据将会被自动写回并关闭所有未关闭的文件。其中，exit(0) 表示程序正常终止，而 exit(1)/exit(-1) 则表示程序出错/异常终止。

4.2.3　获得一个进程的进程标识

进程标识也称为进程号或进程 ID，可通过 getpid( ) 函数来获得，getpid( ) 函数的原型如下：

```
#include <unistd.h>   /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的定义*/
pid_t getpid(void);
```

getpid( ) 函数的返回值即为当前进程的进程号。

4.2.4　获得一个进程的父进程的进程标识

一个进程的父进程的进程标识可通过 getppid( ) 函数来获得，getppid( ) 函数的原型如下：

```
#include <unistd.h>   /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的定义*/
```

```
pid_t getppid(void);
```

getppid()函数的返回值即为当前进程的父进程的进程号。

4.2.5 僵尸进程的清除

1. 僵尸进程的定义与清除方法

一个进程在调用 exit()函数结束自己的生命的时候，操作系统内核仍然会在进程表中为其保留一定的信息（包括进程号、退出状态、运行时间等）。由于这类进程已经放弃了几乎所有内存空间，没有任何可执行代码，也不能被调度，仅仅继续占用了系统的进程表资源，除此之外不再占有任何的内存空间，因此被称为僵尸进程（Zombie Process）。在 LINUX 中，利用命令 ps 命令可以看到有标记为 Z 的进程就是僵尸进程。

由于僵尸进程需要占用系统的进程表资源，但 Linux 系统对运行的进程数量有限制，如果产生过多的僵尸进程占用了可用的进程号，将会导致新的进程无法生成。为此，有必要对僵尸进程进行及时清除。

僵尸进程的清除工作一般是由其父进程来负责进行的，当父进程调用 fork()函数创建了子进程后，主要可通过如下几种方法来避免产生僵尸进程。

方法 1：父进程可通过调用 wait()或 waitpid()等函数来等待子进程结束，从而避免产生僵尸进程，但这会导致父进程被挂起（即父进程被阻塞，处于等待状态）。

方法 2：如果父进程很忙而不能被挂起，那么可以通过调用 signal()函数为 SIGCHLD 信号安装 handler 来避免产生僵尸进程。因为当子进程结束后，内核将会发送 SIGCHLD 信号给其父进程，而父进程在收到该信号之后，则可以在 handler 中调用 wait()函数来进行回收。

方法 3：如果父进程不关心子进程何时结束，那么可以通过调用 signal（SIGCHLD，SIG_IGN）函数来通知内核，表明自己对子进程的结束不感兴趣，那么子进程结束后将会被内核自动回收，且不会再给父进程发送 SIGCHLD 信号，由此可以避免产生僵尸进程。

方法 4：由于当一个父进程死后，其子进程将成为“孤儿进程”，从而会被过继给 1 号进程 init，init 是系统中的一个特殊进程，其进程 ID 为 1，主要负责在系统启动时启动各种系统服务及子进程的清理，只要有子进程终止，init 就会调用 wait 函数清理它。因此，当一个父进程死后，其产生的僵尸进程也会被过继给 1 号进程 init，再由 init 进程负责自动清理，这样一来，就使得一个父进程也可通过 fork 两次来避免产生僵尸进程，具体实现步骤如下：首先，父进程 fork 一个子进程并继续工作，然后该子进程再在 fork 了一个孙进程之后退出，由于该孙进程将会被 init 进程接管，因此当该孙进程结束之后，将会被 init 进程自动回收。当然，子进程的回收工作还得由父进程来负责。

2. 清除僵尸进程的相关函数定义

(1) wait()函数

wait()函数的原型如下：

```
#include <sys/types.h>      //提供数据类型定义
#include <sys/wait.h>       //提供 wait()函数的原型定义
pid_t wait(int *status);
```

在上述 wait( ) 函数的原型之中，其参数的含义如下。

status：用来保存被收集进程退出时的一些状态，它是一个指向 int 类型的指针。可以通过调用以下宏来判别子进程的结束情况。

- WIFEXITED（status）：子进程正常结束则该宏将返回非 0 值。
- WEXITSTATUS（status）：若子进程正常结束则利用该宏可获得子进程由 exit( ) 返回的结束代码。
- WIFSIGNALED（status）：子进程因为信号而结束则该宏将返回非 0 值。
- WTERMSIG（status）：若子进程因为信号结束则利用该宏可获得子进程的中止信号代码。
- WIFSTOPPEN（status）：子进程处于暂停执行状态则该宏将返回非 0 值。
- WSTOPSIG（status）：若子进程处于暂停状态则利用该宏可获得引发子进程暂停的信号代码。

进程一旦调用了 wait( ) 函数，就立即阻塞自己，由 wait( ) 函数自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait( ) 函数就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait( ) 函数就会一直阻塞在这里，直到有这样一个子进程出现为止。如果 wait( ) 函数调用成功，将会返回被收集的子进程的进程 ID，如果调用失败则返回 -1。

（2）waitpid( ) 函数

waitpid( ) 函数的原型如下：

```
#include <sys/types.h>          //提供数据类型定义
#include <sys/wait.h>           //提供 wait( ) 函数的原型定义
pid_t waitpid(pid_t pid, int *status, int options);
```

在上述 waitpid( ) 函数的原型之中，各参数的含义如下。

pid：是指需要等待的那个子进程的进程号。当 pid 取不同的值时有不同的意义：

- pid >0 时，只等待进程 ID 等于 pid 的子进程，不管其他已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid( ) 就会一直等下去；
- pid = -1 时，等待任何一个子进程退出，没有任何限制，此时 waitpid( ) 和 wait( ) 的作用完全等同；
- pid =0 时，等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，waitpid( ) 不会对它做任何理睬；
- pid <-1 时，等待一个指定进程组中的任何子进程，这个进程组的 ID 等于 pid 的绝对值。

status：用来保存被收集进程退出时的一些状态，它是一个指向 int 类型的指针。

options：提供了一些额外的选项来控制 waitpid( )，主要包括 WNOHANG 和 WUNTRACED 等选项，这些选项可以用“|”运算符连接起来使用，若不想使用这些选项，也可将参数 options 置为 0。其中，若将参数 options 置为 WUNTRACED，则当子进程处于暂停状态，waitpid( ) 将马上返回；若将参数 options 置为 WNOHANG，则即使没有子进程退出，waitpid( ) 也将立即返回；而若将参数 options 置为 0，则 waitpid( ) 会像 wait( ) 那样阻塞父进

程，直到所等待的子进程退出。

waitpid( )的返回值比 wait 稍微复杂一些，一共有三种情况：当正常返回的时候，waitpid 返回收集到的子进程的进程 ID；如果设置了选项 WNOHANG，而调用中 waitpid 发现没有已退出的子进程可收集，则返回 0；如果调用中出错，则返回 -1，这时 errno 会被设置成相应的值以指示错误所在。

为了进一步具体阐述清楚 wait( ) 和 waitpid( ) 函数的用法，下面分别举一个例子来加以说明。

例 4-1 wait( ) 函数的用法。

```
#include <sys/types.h>          //提供数据类型定义
#include <sys/wait.h>           //提供 wait( ) 函数的原型定义
#include <unistd.h>             /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和
                                   数据结构的定义 */
int main( ) {
    int status;
    pid_t pc, pr;
    pc = fork( );               //调用 fork( ) 函数创建一个子进程
    if(pc < 0) {                //若创建子进程失败
        printf("fork failed");
        exit(1);                //退出程序
    }
    else if(pc == 0) {          //子进程中执行以下代码段
        int i;
        for(i = 3; i > 0; i --) {
            printf("This is the child\n");
            sleep(5);
        }
        exit(3);                //终止子进程,并给父进程返回终止代码 3
    } else {                    //父进程中执行以下代码段
        pr = wait(&status);     //调用 wait( ) 等待子进程终止
        if(WIFEXITED(status)) { //若子进程正常结束
            printf("The child process %d exit normally. \n",pr);
    printf("The WEXITSTATUS return code is %d \n", WEXITSTATUS(status));
            printf("The WIFEXITED return code is %d \n", WIFEXITED(status));
        }else                   //若子进程非正常结束
            printf("The child process %d exit abnormally. \n", pr);
        printf("Status is %d. \n", status);
    }
    return 0;
}
```

例 4-2 waitpid( ) 函数的用法。

```
#include <sys/types.h>          //提供数据类型定义
```

```
#include <sys/wait.h>          //提供 wait()函数的原型定义
#include <unistd.h>            /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的定义*/
#include <stdio.h>             /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf()等)的定义*/
#include <stdlib.h>            /*C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()等)原型的定义*/
int main(){
    pid_t pid;
    pid = fork();              //调用 fork()函数创建一个子进程
    if(pid<0){                 //若创建子进程失败
        printf("fork failed");
        exit(1);               //退出程序
    }
    else if(pid==0){ //子进程中执行以下代码段
        int i;
        for(i=3; i>0; i--){
            printf("This is the child\n");
            sleep(5);
        }
        exit(3);               //终止子进程,并给父进程返回终止代码 3
    }else{                     //父进程中执行以下代码段
        int stat_val;
        waitpid(pid, &stat_val, 0);          //调用 waitpid()等待子进程终止
        if(WIFEXITED(stat_val))              //若子进程正常结束
          printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
        else if(WIFSIGNALED(stat_val))  //若子进程因为信号而结束
          printf("Child terminated abnormally, signal %d\n", WTERMSIG(stat_val));
    }
    return 0;
}
```

(3) signal()函数

signal()函数的原型如下：

```
#include <signal.h>  //提供 signal 函数原型的定义
void(*signal(int signum,void(*handler)(int)))(int);
```

在上述 signal()函数的原型之中，各参数的含义如下。

signum：指明了 signal () 函数所要处理的信号编号；

handler：描述了与信号关联的动作。它可取以下三种值。

- 一个参数类型为 int 返回值类型为 void 的函数地址：该函数必须在 signal()函数被调用之前申明，handler 为该函数的名字。当接收到一个信号编号为 signum 的信号时，进程就执行 handler 所指定的函数。

- SIGIGN：这个符号表示忽略该信号，执行了相应的 signal( )调用后，进程会忽略信号编号为 signum 的信号。
- SIGDFL：这个符号表示恢复系统对信号的默认处理。

由上述 signal( )函数的原型可知，signal( )函数有两个参数，第一个是参数类型为 int，第二个是指向参数类型为 int 返回值类型为 void 的函数指针，signal( )函数的返回值类型是一个函数指针，同样指向一个参数类型为 int 返回值类型为 void 的函数。signal( )函数会依参数 signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达时就会跳转到参数 handler 所指定的函数执行。

由以上描述可知，signal( )函数的第二个参数类型是一个函数指针，而且 signal( )函数的返回值类型也是一个函数指针。这里，所谓的函数指针是指一个指向函数的指针变量，即函数指针本身首先应该是一个指针变量，只不过该指针变量指向的是一个函数，其实也与用指针变量指向整型变量、字符型、数组相类似，只不过这里是用指针变量指向函数罢了。如前所述，C 在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。有了指向函数的指针变量之后，即可用该指针变量调用函数，就如同用指针变量可引用其他类型变量一样，这在概念上都是一致的。函数指针有两个用途：调用函数和做函数的参数。函数指针的声明方法如下：

数据类型标志符（指针变量名）（形参列表）；

其中，“函数类型”说明了函数的返回类型，由于“( )”的优先级高于“*”，所以指针变量名外的括号必不可少；而后面的“形参列表”则表示指针变量指向的函数所带的参数列表。例如：

```
int func(int x);     //声明一个函数
int( *f)(int x);     //声明一个函数指针
f = func; /*将 func 函数的首地址赋给指针 f,赋值时函数 func 不带括号,也不带参数,由于 func 代
          表函数的首地址,因此经过赋值以后,指针 f 就指向函数 func(x)的代码的首地址*/
```

基于以上关于函数指针的定义，signal( )函数的原型可理解为是由如下两个步骤所组成的：

步骤 1：首先，定义一个参数类型为 int 返回值类型为 void 的函数指针 sig_t：

```
typedef void( *sig_t)(int);
```

步骤 2：然后，定义一个返回值类型为函数指针 sig_t 的函数 signal( )，该函数有两个参数，一个参数类型为 int，另一个参数类型为函数指针 sig_t：

```
sig_t signal(int signum,sig_t handler);
```

由步骤 2 可知，handler 为一个类型为 sig_t 的函数指针，因此再由步骤 1 可知，handler 所指向的那个函数只能有一个 int 类型的参数；另外，由步骤 2 也可知，signal( )函数的返回值也为一个类型为 sig_t 的函数指针，因此再由步骤 1 可知，signal( )函数返回的函数指针所指向的那个函数也只能有一个 int 类型的参数。

基于以上描述，下面给出一个简单的例子来进一步说明 signal( )函数的用法：

```
#include <unistd.h>   /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构
```

```
                                的定义 */
#include < signal. h >      //提供 signal 函数原型的定义
void handler( ) {
    printf( "hello\n" );
}
int main( ) {
    int i;
    signal(SIGALRM, handler); /* 调用 signal( ) 函数获取 SIGALRM 信号,并交由 handler 所指向
                                 的函数进行处理 */
    alarm(5); /* 调用 alarm( ) 函数设置超时时钟为 5 秒,若时钟超时,则内核将给进程发送 SI-
                 GALRM 信号 */
    for(i = 1;i < 7;i ++ ) {
        printf( "sleep %d …\n", i);
        sleep(1); //调用 sleep( ) 函数休眠 1 秒
    }
    return 0;
}
```

执行结果如下：

```
sleep 1 …
sleep 2 …
sleep 3 …
sleep 4 …
sleep 5 …
hello
sleep 6 …
```

4.3　基于多线程的服务器并发机制

4.3.1　创建一个新线程

线程的创建是通过 pthread_create( ) 函数来实现的，当创建线程成功时，该函数返回 0，若不为 0 则说明创建线程失败。若创建线程成功，则新创建的线程将运行由 pthread_create( ) 函数中第三个参数和第四个参数所确定的函数，而原来的线程则继续运行下一行代码。pthread_create( ) 函数的原型如下：

```
#include  < pthread. h >   //提供线程函数原型和数据结构的定义
int pthread_create( pthread_t  * thread, pthread_attr_t  * attr, void  * ( * start_routine) ( void  * ), void
 * arg);
```

在上述 pthread_create( ) 函数的原型之中，各参数的含义如下。

thread：所创建的线程的标识符。

attr：是 pthread_attr_t 结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

start_routine：是一个参数类型为（void *）返回值类型也为（void *）的函数指针，用于指向线程的线程体函数，该线程体函数所执行的操作即为该线程所执行的操作。

arg：是用于传递给线程体函数的参数。

1. 用单变量向线程体函数传递参数

下面的代码片段演示了如何向一个线程传递一个简单的整数：

```
#include <pthread.h>              //提供线程函数原型和数据结构的定义
#include <stdio.h>        /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和
                            printf()等)的定义*/
#define NUM_THREADS 3
void *PrintHello(void *threadargs) {
    int pid;
    pid = (int) threadargs;
    printf("Hello! I am thread #%d!\n", pid);
    pthread_exit(NULL);          //调用 pthread_exit()终止该线程
}
int main(int argc, char *argv[]) {
      pthread_t pids[NUM_THREADS];
      int *args[NUM_THREADS];
      int rc, i;
      for(i=0; i<NUM_THREADS; i++) {
          args[i] = (int *) malloc(sizeof(int));
          *args[i] = i;
          printf("Creating thread %d\n", i);
          rc = pthread_create(&threads[i], NULL, PrintHello,(void *) args[i]);
          ...
      }
    ...
}
```

2. 用结构体变量向线程体函数传递参数

下面的代码片段演示了如何向一个线程传递一个简单的结构体：

```
#include <pthread.h>              //提供线程函数原型和数据结构的定义
#include <stdio.h>        /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和
                            printf()等)的定义*/
#define NUM_THREADS 3
struct thread_data{               //定义一个结构体 thread_data
    int thread_id;
    int sum;
};
```

```
void * PrintHello(void * threadarg) {
    struct thread_data * my_data;
    int pid, sum;
    char * hello_msg;
    my_data = (struct thread_data * ) threadarg;
    pid = my_data -> thread_id;
    sum = my_data -> sum;
    printf("Hello! I am thread #%d! The sum is %d! \n", pid, sum);
    pthread_exit(NULL); //调用 pthread_exit()终止该线程
}
int main(int argc, char * argv[]) {
    pthread_t pids[NUM_THREADS];
    struct thread_data thread_data_array[NUM_THREADS];
    int rc, I,sum;
    sum = 0;
    for(i = 0; i < NUM_THREADS; i ++ ) {
        sum ++ ;
        thread_data_array[i]. thread_id = i;
        thread_data_array[i]. sum = sum;
        printf("Creating thread %d\n", i);
        rc = pthread_create(&pids[i], NULL, PrintHello,(void * ) &thread_data_array[i]);
        ...
    }
    ...
}
```

4.3.2　设置线程的运行属性

线程具有运行属性，用 pthread_attr_t 结构体表示，在对该结构体进行处理之前必须进行初始化，在使用后需要对其去除初始化，以释放该结构体所占用的资源。用于对 pthread_attr_t 结构体进行初始化的函数为 pthread_attr_init()，对其去除初始化的函数为 pthread_attr_destroy()。其中，pthread_attr_init()与 pthread_attr_destroy()函数的原型如下：

```
#include <pthread. h> //提供线程函数原型和数据结构的定义
int pthread_attr_init(pthread_attr_t * attr);
int pthread_attr_destroy(pthread_attr_t * attr);
```

在上述 pthread_attr_init()与 pthread_attr_destroy()函数的原型之中，各参数的含义如下。

attr：是 pthread_attr_t 结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。其中，pthread_attr_t 结构体的定义如下：

```
typedef struct {
    int detachstate;                    /*线程的分离状态*/
    int schedpolicy;                    /*线程调度策略*/
```

```
    struct sched_param schedparam;      /*线程的调度参数*/
    int inheritsched;                   /*线程的继承性*/
    int scope;                          /*线程的作用域*/
    size_tguardsize;                    /*线程堆栈保护区的大小*/
    int stackaddr_set;                  /*线程堆栈的地址集*/
    void * stackaddr;                   /*线程堆栈的地址*/
    size_t stacksize;                   /*线程堆栈的大小*/
} pthread_attr_t;
```

LINUX 中可通过以下函数来设置上述线程的运行属性。

1. 设置/获取线程的分离状态

线程的分离状态决定了一个线程以何种方式来终止自己。在默认情况下线程为非分离状态，此时，需要让某个原有的线程调用 pthread_join( ) 函数来等待创建的线程结束，只有当 pthread_join( ) 函数返回时，创建的线程才真正终止、并释放自己所占用的系统资源。而分离线状态的程则无需被其他线程等待，一旦自己运行结束，该线程也就自动终止，并立即释放自己所占用的系统资源。为此，若在创建线程时就知道无需关注其终止状态，则可通过设置 pthread_attr_t 结构中的 detachstate 属性来让线程以分离状态运行。

设置线程的分离状态可通过调用 pthread_attr_setdetachstate( ) 函数来实现，而获取线程的分离状态可通过调用 pthread_attr_getdetachstate( ) 函数来实现，这两个函数若调用成功将返回 0，失败则返回 -1。其函数原型分别如下：

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
int pthread_attr_getdetachstate(const pthread_attr_t * attr, int * detachstate);
int pthread_attr_setdetachstate(pthread_attr_t * attr, int detachstate);
```

在上述 pthread_attr_setdetachstate( ) 函数与 pthread_attr_getdetachstate( ) 函数的原型之中，各参数的含义如下。

attr：是 pthread_attr_t 结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

detachstate：线程的分离状态属性。

在 pthread_attr_setdetachstate( ) 函数中，若将参数 detachstate 设置为 PTHREAD_CREATE_DETACHED，则该线程将以分离状态运行；若将参数 detachstate 设置为 PTHREAD_CREATE_JOINABLE，则该线程将以默认的非分离状态运行。

以下给出一个创建分离状态线程的例子：

```
#include <pthread.h>  //提供线程函数原型和数据结构的定义
void * child_thread( ) {
    printf("this is the child thread! \n");
}
int main( ) {
    pthread_t pid;
    pthread_attr_t attr; /*定义线程属性结构体变量 attr*/
    pthread_attr_init(&attr); /*对 attr 结构体变量进行初始化*/
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
```

```
    /* 设置 attr 结构体变量中的 detachstate 字段值为 PTHREAD_CREATE_DETACHED(分离
    状态线程) */
    pthread_create(&pid,&attr, child_thread,NULL); /* 创建运行属性为 attr 的新线程 */
    pthread_attr_destroy(&attr); /* 对 attr 去除初始化,以释放该结构体所占用的资源 */
    return 0;
}
```

2. 设置/获取线程的继承性

函数 pthread_attr_setinheritsched( ) 和函数 pthread_attr_getinheritsched( ) 分别用来设置和获取线程的继承性，这两个函数若调用成功将返回 0，若调用失败则返回 -1。其函数原型分别如下：

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inheritsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
```

在上述 pthread_attr_setinheritsched( ) 函数和 pthread_attr_getinheritsched( ) 函数的原型之中，各参数的含义如下。

attr：是 pthread_attr_t 结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

inheritsched：线程的继承性。

线程的继承性决定了线程是从创建自己的父线程中自动继承调度策略与参数还是使用在 pthread_attr_t 结构体中的 schedpolicy 和 schedparam 字段中显式设置的调度策略与参数。若将 pthread_attr_setinheritsched( ) 函数中的参数 inheritsched 的值设置为 PTHREAD_INHERIT_SCHED，则表示新线程将继承创建自己的父线程的调度策略和参数；若设置为 PTHREAD_EXPLICIT_SCHED 则表示使用在 schedpolicy 和 schedparam 属性中显式设置的调度策略和参数。

3. 设置/获取线程的调度策略

函数 pthread_attr_setschedpolicy( ) 和函数 pthread_attr_getschedpolicy( ) 分别用来设置和得到线程的调度策略，函数若调用成功将返回 0，若失败则返回 -1。其函数原型分别如下：

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

在上述两个函数的原型之中，各参数的含义如下。

attr：是 pthread_attr_t 结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

policy：为线程的调度策略，主要包括先进先出（SCHED_FIFO）、轮循（SCHED_RR）或其他（SCHED_OTHER）等。其中，SCHED_FIFO 策略允许一个线程运行直到有更高优先级的线程准备好，或者直到它自愿阻塞自己。在 SCHED_FIFO 调度策略下，当有一个线程准备好时，除非有平等或更高优先级的线程已经在运行，否则它会很快开始执行。而 SCHED_RR 策略则与 SCHED_FIFO 策略稍有不同：如果有一个 SCHED_RR 策略的线程执行

了超过一个固定的时期（时间片间隔）没有阻塞，而有另外的 SCHED_RR 或 SCHBD_FIFO 策略的相同优先级的线程准备好时，运行的线程将被抢占以便准备好的线程可以执行。当有 SCHED_FIFO 或 SCHED_RR 策略的线程在一个条件变量上等待或等待加锁同一个互斥量时，它们将以优先级顺序被唤醒。

4. 设置/获取线程的调度参数

函数 pthread_attr_getschedparam( ) 和函数 pthread_attr_setschedparam( ) 分别用来设置和得到线程的调度参数，函数若调用成功将返回 0，若失败则返回 -1。其函数原型分别如下：

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param);
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
```

在上述两个函数的原型之中，各参数的含义如下。

attr：是 pthread_attr_t 结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

param：是 sched_param 结构体指针，所指向的结构中的元素 sched_priority 用于标示线程运行的优先权值。结构 sched_param 的定义如下：

```
struct sched_param{
    int sched_priority; /*线程运行的优先权值*/
};
```

系统支持的最大和最小优线程先权值可以通过调用 sched_get_priority_max( ) 函数和 sched_get_priority_min( ) 函数来分别得到，其中，大的优先权值对应高的优先权。sched_get_priority_max( ) 函数和 sched_get_priority_min( ) 函数的原型分别如下：

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

上述两个函数若调用失败将返回 -1；若调用成功则返回 0，同时将得到的系统支持的最大/最小优线程先权值保存在 policy 变量之中。

5. 设置/获取线程的作用域

函数 pthread_attr_setscope( ) 和函数 pthread_attr_getscope( ) 分别用来设置和得到线程的作用域。这两个函数若调用失败将返回 -1，若调用成功则返回 0。其函数原型如下：

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

在上述两个函数的原型之中，各参数的含义如下。

attr：是 pthread_attr_t 结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

scope：是作用域（在 pthread_attr_setscope( ) 函数中）或指向作用域的指针（在 pthread_attr_getscope( ) 函数中）。

其中，作用域用于控制线程是否在进程内或系统级上竞争资源，可能的值有 PTHREAD_SCOPE_PROCESS（进程内竞争资源），PTHREAD_SCOPE_SYSTEM（系统级上竞争资源）。例如：假设有两个进程 A 和 B，其中，A 进程包含 4 个线程，B 进程只有 1 个主线程，则假若作用域设置为 PTHREAD_SCOPE_SYSTEM，则 A 进程中的 4 个线程将和 B 进程中的那 1 个线程一起竞争 CPU 资源，但若作用域设置为 PTHREAD_SCOPE_PROCESS，则 A 进程中的 4 个线程只能竞争所属进程 A 的 CPU 时间。不过，目前 LINUX 只支持 PTHREAD_SCOPE_SYSTEM 方式。

6. 设置/获取线程堆栈保护区的大小

函数 pthread_attr_getguardsize( ) 和函数 pthread_attr_setguardsize( ) 分别用来设置和得到线程的堆栈保护区（警戒堆栈）的大小，这两个函数若调用失败将返回 -1，若调用成功则返回 0，其函数原型如下：

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
int pthread_attr_getguardsize(const pthread_attr_t *restrict attr, size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t *guardsize);
```

在上述两个函数的原型之中，各参数的含义如下。

attr：是 pthread_attr_t 结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

guardsize：控制着线程堆栈末尾之后以避免堆栈溢出的堆栈保护区（扩展内存）的大小。

其中，堆栈保护区被用来在堆栈指针越界的情况下提供保护。如果一个线程具有堆栈保护的特性，那么系统在创建线程堆栈时会在堆栈的末尾多分配一块内存，以用来防止指针访问堆栈时溢出堆栈的边界。如果一个应用程序访问堆栈时溢出到堆栈保护区时将会引发一个错误（此时，当前线程将收到一个 SIGSEGV 信号）。

提供堆栈保护区属性设置函数的主要有以下两个原因：一是堆栈保护会引起系统资源浪费，因此，如果一个应用程序创建了大量线程，而且确保这些线程均不会越界访问堆栈时，则应用程序可通过调用堆栈保护区属性设置函数来取消堆栈保护区以节省系统资源；二是当线程在堆栈中存放大的数据结构时，有可能需要一个大的堆栈保护区，此时则可通过调用堆栈保护区属性设置函数来增加堆栈保护区的大小。

如果线程属性对象的 guardsize 参数值为 0，则创建线程时将不会创建堆栈保护区，若线程属性对象的 guardsize 参数值大于 0，则使用该线程属性对象创建的线程将起码有一个 guardsize 大小的堆栈保护区。

7. 设置/获取线程堆栈的地址

函数 pthread_attr_setstackaddr( ) 和函数 pthread_attr_getstackaddr( ) 分别用来设置和得到线程堆栈的地址，这两个函数若调用失败将返回 -1，若调用成功则返回 0。其函数原型如下：

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

在上述两个函数的原型之中，各参数的含义如下。

attr：是 pthread_attr_t 结构体指针，所指向的结构中的元素分别用于标示线程的运行

属性。

stackaddr：线程的堆栈地址。

注：设置堆栈地址将降低可移植性，建议最好不要自己设置堆栈地址。

8. 设置/获取线程堆栈的大小

函数 pthread_attr_setstacksize( ) 和函数 pthread_attr_getstacksize( ) 分别用来设置和得到线程堆栈的大小，这两个函数若调用失败将返回 -1，若调用成功则返回 0。其函数原型如下：

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
int pthread_attr_getstacksize(const pthread_attr_t * restrict attr, size_t * restrict stacksize);
int pthread_attr_setstacksize(pthread_attr_t * attr , size_t * stacksize);
```

在上述两个函数的原型之中，各参数的含义如下。

attr：是 pthread_attr_t 结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

stacksize：线程堆栈的大小。

4.3.3 终止一个线程

在 LINUX 系统中，一个线程既可以通过自身调用 pthread_exit( ) 函数来实现显式地终止，也可以通过在另一个线程中调用 pthread_join( ) 函数来实现隐式地终止。其中，pthread_exit( ) 函数在头文件 pthread.h 中声明，线程在调用该函数后将自行终止并释放其所占资源。pthread_exit( ) 函数的原型如下：

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
int pthread_exit(void * value_ptr);
```

在上述 pthread_exit( ) 函数的原型之中，各参数的含义如下。

value_ptr：线程返回值指针，该返回值将被传给另一个线程，另一个线程则可通过调用 pthread_join( ) 函数来获得该值。

pthread_join( ) 函数在头文件 pthread.h 中声明，其作用是用于等待一个指定的线程结束，该函数的原型如下：

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
int pthread_join(pthread_t thread,void * * value_ptr);
```

在上述 pthread_join( ) 函数的原型之中，各参数的含义如下：

thread：等待终止的线程的线程标识符。

value_ptr：如果 value_ptr 不为 NULL，那么线程 thread 的返回值存储在该指针指向的位置。该返回值可以是由 pthread_exit( ) 给出的值，或者该线程被取消而返回 PTHREAD_CANCELED。

pthread_join( ) 函数是一个线程阻塞函数，调用它的函数将一直等到被等待的线程结束为止。调用 pthread_join( ) 函数的线程将被挂起，直到参数 thread 所代表的线程终止时为止。

当一个非分离的线程终止后，该线程的内存资源（线程描述符和栈）并不会被释放，直到有线程对它调用了 pthread_join( ) 时才会被释放。因此，必须对每个创建的非分离的线程调用一次 pthread_join( ) 以避免内存泄漏。另外，至多只能有一个线程调用 pthread_join( )

等待给定的线程终止，如果已有一个线程调用了 pthread_join( ) 以等待某个线程终止，那么其他再调用 pthread_join( ) 以等待同一线程终止的线程将返回一个错误。

4.3.4　获得一个线程的线程标识

函数 pthread_self( ) 可以用来使得调用线程获取自己的线程 ID，其函数原型如下：

```
#include <pthread.h> /*提供线程函数原型和数据结构的定义*/
pthread_t pthread_self();
```

4.3.5　多线程例程剖析

为了进一步说明上述各线程函数的具体用法，下面给出一个简单的多线程的例程。

```
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror( ) 和 printf( ) 等)的定义*/
#include <stdlib.h> /*C 标准库头文件,包含了 C 语言标准库函数(如 exit( )、atoi( ) 等)原型的定义*/
#include <pthread.h>          //提供线程函数原型和数据结构的定义
void thread( ) { /*线程体函数*/
    int i;
    for(i=0;i<3;i++)
        printf("This is a pthread.\n");
}
int main( ) {
    pthread_t pid;            //定义线程标识符变量 pid
    pthread_attr_t attr;      //定义线程运行属性变量 attr
    int i, ret;
    pthread_attr_init(&attr);   //初始化线程运行属性变量 attr
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    ret=pthread_create(&pid, &attr,(void *) thread, NULL); /*创建一个新线程执行线程体函数 thread( ),该线程的标识符为 pid*/
    if(ret!=0) {              //若创建新线程失败,则输出出错提示并退出程序
        printf("Create pthread error!\n");
        exit(1);
    }
    pthread_attr_destroy(&attr);//去初始化线程运行属性变量 attr
    for(i=0;i<3;i++)          //在主线程中输出 3 次主线程提示信息
        printf("This is the main process.\n");
    pthread_join(pid,NULL); /*在主线程中调用 pthread_join( ) 函数等待新线程结束,并回收新线程所占资源*/
    return 0; //调用 return( ) 函数返回整型值 0 作为 main( ) 的返回值
}
```

4.4 从线程/进程分配技术

4.4.1 从线程/进程预分配技术

在并发服务器中，若针对每一个到达的客户连接请求，均创建一个新的从线程/进程来进行处理，在处理完该请求之后，该从线程/进程再退出。这样在服务器负载很重的时候，将导致过多的线程/进程创建开销。为此，为了降低操作系统创建线程/进程所需的额外开销、提高服务器的吞吐率，人们提出了预分配技术。

在采用预分配技术的服务器设计之中，设计人员一般按以下方法来编写服务器程序：服务器在启动时就创建若干个并发的从线程/进程，每个从线程/进程都使用操作系统中提供的设施以等待客户请求的到达，当客户请求到达后，其中一个空闲的从线程/进程就开始执行并处理该客户请求，当完成客户请求的处理后，从线程/进程不退出，而是重新返回到等待客户请求到达的状态。

显然，在基于预分配技术的并发服务器中，由于服务器不需要在客户请求到达时创建从线程/进程，避免了在每次客户请求到达时创建从线程/进程的开销，因此可更快地处理客户请求，减低了服务器的时延。特别是，当客户请求的处理涉及的 I/O 多于计算时，由于预分配技术允许服务器系统在等待与前一个请求相关的 I/O 活动时，切换到另一个从线程/进程，并开始处理下一个请求，因此，此时采用预分配技术就显得尤为重要。

另外，在多处理器上采用预分配技术还可允许设计人员使服务器的并发等级与服务器的硬件性能相关联。如果服务器有 K 个处理器，则设计人员可预分配 K 个从线程/进程，由于多处理器操作系统给每个从线程/进程一个单独的处理器，为此，采用预分配技术可保证服务器的并发等级与硬件之间的匹配，从而服务器可获得尽可能高的处理速率。

4.4.2 延迟的从线程/进程分配技术

虽然预分配技术可提高服务器的效率，但它不能解决所有问题。例如：由前述分析可知，预先创建从线程/进程不但需要消耗时间和服务器资源，也会为操作系统管理从线程/进程带来额外的开销，另外，预分配多个试图接收传入客户请求的从线程/进程还可能会给网络代码增添额外的开销，因此，只有当预先创建额外的从线程/进程能提高系统的吞吐率或降低系统的时延时，采用预分配技术才是合理的。

但是，由于处理客户请求的时间是与客户请求直接相关的，因此，设计人员有时候可能会无法预先明确知道采用预分配技术是否合理。此时，可采用一种称为延迟的从线程/进程分配（Delayed Slave Allocation）技术。该技术的主要思想如下：服务器在启动时先循环地处理每个客户请求，只有当处理需要花费很长时间时，服务器才创建一个并发的从线程/进程来处理该请求。在 LINUX 中，采用延迟的从线程/进程分配技术并不难，只需要在主线程中设置一个计时器，并设计在计时器到期时调用 fork( )函数创建一个新的从进程，由于该从进程将从父进程处继承打开的套接字，以及执行程序和数据的副本，因此，该从进程将恰好可从父进程超时所执行代码处继续进行处理。

4.4.3　两种从线程/进程分配技术的结合

由前述分析可知，预分配技术提高了在客户请求到达前的服务器的并发等级，而延迟的从线程/进程分配技术则提高了在客户请求到达服务器后的并发等级。这两种技术均是通过把服务器的并发等级从当前活跃的客户请求数目中分离出来，从而使设计人员可获得灵活性并由此提高服务器的效率。显然，这两种技术可按以下方法结合使用：服务器在启动时不采用预分配技术而是采用延迟分配技术，此时，服务器没有预先分配的从线程/进程；当有客户请求到达时，若处理需要花费很长时间时，服务器创建一个并发的从线程/进程来处理该请求；但一旦创建了从线程/进程后，该从线程/进程在处理完客户请求之后不必立即退出，它可以认为自己是永久分配的，并继续运行，在处理完一个客户请求之后，继续等待下一个客户请求的到达。

在上述结合了两类并发技术的系统之中，需要对服务器的并发等级进行控制。常用的两种方法如下：第一种方法是设法让主线程/进程在创建一个从线程/进程时，指明其最大增长值 M，从而限定了系统最终可达到的并发等级的最大值；另一种方法就是设法让一段时期内不活跃的从线程/进程退出，从线程/进程在等待下一个客户请求前先启动一个计时器，若在下一个客户请求到达之前计时器到期，则从线程/进程就退出。

4.5　基于多进程与基于多线程的并发机制的性能比较

4.5.1　多进程与多线程的任务执行效率比较

究竟何时该采用多进程的并发模式，何时该采用多线程的并发模式？为了具体说明该问题，下面将通过一个简单的例子来进行分析比较。在该例子中包含两个例程，其中，一个是多进程的例程，而另一个是多线程的例程。这两个例程所实现的功能完全相同，都是首先创建“若干”个新进程/线程，然后再由每个新创建出的进程/线程分别负责打印出“若干”条“hello linux”字符串到控制台与日志文件，其中，这里的两个“若干”是分别由两个宏 P_NUMBER 与 COUNT 来定义的。这两个例程的具体代码实现如下。

（1）多进程并发例程的 C 语言代码（process. c）

```
#include <stdlib. h> /* C 标准库头文件,包含了 C 语言标准库函数(如 exit( )、atoi( )等)原型的
                        定义 */
#include <stdio. h> /* 标准输入/输出头文件,包含了标准输入/输出函数(如 perror( )和 printf( )
                       等)的定义 */
#include <signal. h> //提供了 signal 函数原型的定义
#define P_NUMBER 255      //定义并发进程的数量
#define COUNT 100         //定义每个进程打印字符串的次数
#define TEST_LOGFILE "logFile. log"
FILE * logFile = NULL;
char * s = "hello linux\0";
int main( ) {
```

```
    int i=0,j=0;
    logFile=fopen(TEST_LOGFILE,"a+"); //打开日志文件"logFile.log"
    for(i=0; i<P_NUMBER; i++){  /*若并发线程的个数没有超过给定的最大值 P_NUMBER*/
        if(fork()==0){  //则创建一个新的子进程,并在子进程中执行以下操作
            for(j=0; j<COUNT; j++){
                printf("[%d]%s\n",j,s); //打印输出 COUNT 次字符串 s
                fprintf(logFile,"[%d]%s\n",j,s); /*向日志文件输出 COUNT 次字符串 s*/
            }
            exit(0); //结束子进程
        }
    }
    for(i=0; i<P_NUMBER; i++){ //调用 wait 函数回收所有的子进程
       wait(0);
    }
    printf("OK\n");
    return 0;
}
```

(2) 多线程并发例程的 C 语言代码 (thread.c)

```
#include <pthread.h> //提供线程函数原型和数据结构的定义
#include <unistd.h>  /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的定义*/
#include <stdlib.h>  /*C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()等)原型的定义*/
#include <stdio.h>   /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf()等)的定义*/
#define P_NUMBER 255    //定义并发线程的最大个数
#define COUNT 100       //定义每个线程打印字符串的次数
#define Test_Log "logFIle.log"
FILE *logFile=NULL;
char *s="hello linux\0";
print_hello_linux(){  //线程执行的线程体函数
    int i=0;
    for(i=0; i<COUNT; i++){
        printf("[%d]%s\n",i,s);  //向控制台输出信息
        fprintf(logFile,"[%d]%s\n",i,s); //向日志文件输出信息
    }
    pthread_exit(0);  //结束线程
}
int main(){
    int i=0;
    pthread_t pid[P_NUMBER]; //定义用于记录线程 ID 的数组 pid[]
```

```
    logFile = fopen( Test_Log,"a + " ); //打开日志文件
    for( i = 0; i < P_NUMBER; i + + ) /* 若并发线程的个数没有超过给定的最大值 P_
                                        NUMBER */
        pthread_create( &pid[ i ],NULL,( void * )print_hello_linux,NULL); /* 则创建新的子线
                                                                          程 */
    for( i = 0; i < P_NUMBER; i + + )  //调用 pthread_join 函数隐式地终止所有子线程
        pthread_join( pid[ i ],NULL);
    printf( "OK\n" );
    return 0;
}
```

(3) 任务执行效率比较结果

基于上述两个例程，通过在每批次的实验中修改宏 P_ NUMBER 和 COUNT 的值来调整进程/线程的数量与打印次数，每批次测试五轮后计算平均值，在 LINUX2.6、单核单 CPU 的 i386 处理器环境下，所得到的结果如表 4.1 至表 4.4 所示。

表 4.1　第 1 次实验结果（进程/线程数 255/打印次数：100）

	第 1 次	第 2 次	第 3 次	第 4 次	第 5 次	平均
多进程	0m1.277s	0m1.175s	0m1.227s	0m1.245s	0m1.228s	0m1.230s
多线程	0m1.150s	0m1.192s	0m1.095s	0m1.128s	0m1.177s	0m1.148s

表 4.2　第 2 次实验结果（进程/线程数 255/打印次数：500）

	第 1 次	第 2 次	第 3 次	第 4 次	第 5 次	平均
多进程	0m6.341s	0m6.121s	0m5.966s	0m6.005s	0m6.143s	0m6.115s
多线程	0m6.082s	0m6.144s	0m6.026s	0m5.979s	0m6.012s	0m6.048s

表 4.3　第 3 次实验结果（进程/线程数 255/打印次数：1000）

	第 1 次	第 2 次	第 3 次	第 4 次	第 5 次	平均
多进程	0m12.155s	0m12.057s	0m12.433s	0m12.327s	0m11.986s	0m12.184s
多线程	0m12.241s	0m11.956s	0m11.829s	0m12.103s	0m11.928s	0m12.011s

表 4.4　第 4 次实验结果（进程/线程数 255/打印次数：5000）

	第 1 次	第 2 次	第 3 次	第 4 次	第 5 次	平均
多进程	1m2.182s	1m2.635s	1m2.683s	1m2.751s	1m2.694s	1m2.589s
多线程	1m2.622s	1m2.384s	1m2.442s	1m2.458s	1m3.263s	1m2.614s

从以上实验数据可得出以下结果：当任务量较大时（打印次数大于等于 5000 次时），多进程比多线程的效率要高；而当任务量较小时（打印次数小于 5000 次时），则多线程要比多进程快。但整体上来看，多线程比较多进程在效率上没有太大的优势。

4.5.2　多进程与多线程的创建与销毁效率比较

预先创建进程或线程可以节省进程或线程的创建、销毁时间，在实际的应用中很多程序

使用了这样的策略，例如：Apapche 采用了预先创建进程，Tomcat 采用了预先创建线程的策略，预先创建的进程或线程通常被称为进程池或线程池。为了实际比较进程或线程的创建与销毁效率，下面将通过一个简单的例子来进行分析比较。在该例子中包含两个例程，其中，一个是多进程的例程，而另一个是多线程的例程。这两个例程所实现的功能完全相同，都是计算创建与销毁 10 万个进程/线程所需的绝对用时。这两个例程的具体代码实现如下。

（1）多进程并发例程的 C 语言代码（process_time. c）

```
#include <stdlib.h> /* C 标准库头文件,包含了 C 语言标准库函数(如 exit、atoi 等)原型的定义 */
#include <signal.h> //提供 signal 函数原型的定义
#include <stdio.h> /* 标准输入输出头文件,包含了标准输入输出函数(如 perror 和 printf 等)的定义 */
#include <unistd.h> /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的定义 */
#include <sys/stat.h> //文件状态头文件,含有文件或文件系统状态结构和常量的定义
#include <fcntl.h> //文件控制头文件,含有文件及其描述符的操作控制常数符号的定义
#include <sys/types.h> //基本数据类型头文件,含有基本系统数据类型的定义
#include <sys/wait.h> //提供 wait()函数原型的定义
int count;  //定义用于记录子进程创建成功个数的全局变量 count
int fcount;  //定义用于记录子进程创建失败个数的全局变量 fcount
int scount;  //子进程回收数量
void sig_chld(int signo) {  //信号处理函数 - 子进程关闭收集
        pid_t chldpid; //子进程 id
        int stat; //子进程的终止状态
        while ((chldpid = wait(&stat)) >0) { //子进程回收,避免出现僵尸进程
            scount ++ ;
        }
}
int main() {
        signal(SIGCHLD,sig_chld); //注册子进程回收信号处理函数
        int i;
        for (i =0; i <100000; i ++ ) { //循环创建 100000 个子进程
            pid_t pid = fork(); //调用 fork()函数创建子进程
            if (pid == -1) { //若子进程创建失败
                    fcount ++ ; //则将创建失败的子进程的个数加 1
            }
            else if (pid >0) {//若子进程创建成功
                    count ++ ; //则将创建成功的子进程的个数加 1
            }
            else if (pid ==0) {//子进程的执行过程
                    exit(0); //退出子进程
            }
        }
```

```
        printf("count: %d fcount: %d scount: %d\n",count,fcount,scount); /*输出创建成功与
                                                                失败的子进程的
                                                                个数*/
}
```

（2）多线程并发例程的 C 语言代码（thread_time.c）

```
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf()
                    等)的定义*/
#include <pthread.h> //提供线程函数原型的定义
int count=0;  //成功创建线程数量
void thread(void){
    //子线程啥也不做
}
int main(void){
    pthread_t id;  //定义用于记录线程 ID 的局部变量 id
    int i,ret; /*定义用于计数记录的局部变量 i 和用于记录 pthread_create()返回值的局部变量
                 ret*/
    for(i=0; i<100000; i++){ //创建 10 万个子线程
        ret=pthread_create(&id,NULL,(void *)thread,NULL); //创建子线程
        if(ret!=0){  //若创建子线程出错则提示出错信息
            printf("Create pthread error! \n");
            return(1);
        }
        count++; //将子线程的个数加 1
        pthread_join(id,NULL);  /*在主线程中调用 pthread_join()函数来实现对子线程的
                                  隐式地终止*/
    }
    printf("count: %d\n",count); //输出子线程的个数信息
}
```

（3）创建与销毁效率比较结果

基于上述两个例程，采用测试五轮后计算平均值，在 LINUX2.6、赛扬 1.5 GB 的 CPU 环境下，所得到的结果如表 4.5 所示。

表 4.5　多进程与多线程的创建与销毁效率比较

创建销毁 10 万个进程	创建销毁 10 万个线程
0m18.201s	0m3.159s

从表 4.5 中的数据结果可以看出，多线程比多进程在创建与销毁效率上有 5～6 倍的优势。但由于平均创建销毁一个进程仅约需 0.18 毫秒（=0m18.201s/100000），且预先派生子进程/线程需要对池中进程/线程数量进行动态管理，比现场创建子进程/线程要复杂很多，因此对于当前服务器几百或几千的并发量，预先派生线程也不见得比现场创建线程快。

4.6 本章小结

本章主要对基于多进程或多线程的服务器并发概念及其并发机制进行了详细介绍，并分别给出了一个创建并发进程与并发线程的完整 C 语言例程，与此同时，还对基于多进程与基于多线程的并发机制的性能进行了比较。通过本章的学习，需要了解基于多进程或多线程的服务器并发概念；需要熟习基于多进程与基于多线程的并发机制、从线程/进程预分配技术以及延迟的从线程/进程分配技术；需要掌握 LINUX 环境下基于多进程与基于多线程的并发软件的 C 语言实现方法。

本章习题

1. 简述进程和线程各自的优缺点。
2. 简述调用 fork( ) 函数创建一个新进程的两种基本方法。
3. 简述僵尸进程的定义与清除方法。
4. 简述从线程/进程预分配技术的实现原理。
5. 简述延迟的从线程/进程分配技术的实现原理。
6. 试构造一个多进程的例程，该例程能实现以下功能：首先，创建 1 万个新进程，然后再由每个新创建出的进程分别负责打印出 100 条“hello world”字符串到控制台。
7. 试构造一个多线程的例程，该例程能实现以下功能：首先，创建 1 万个新线程，然后再由每个新创建出的线程分别负责打印出 100 条“hello world”字符串到控制台。

第 5 章　基于多进程并发的面向连接服务器例程剖析

第 4 章介绍了服务器并发机制及其实现方法，本章将进一步针对其中一种并发服务器——基于多进程并发的面向连接服务器及其实现方法进行深入介绍，并在实现原理介绍的基础上，具体给出两个创建基于多进程的并发的面向连接服务器的完整 C 语言例程及其对应客户端的完整 C 语言例程。

5.1　基于多进程并发的面向连接服务器的进程结构

图 5.1 给出了基于多进程并发的面向连接服务器的进程结构，在该服务器中使用的是单线程的进程，即：一个进程之中只包含一个线程。

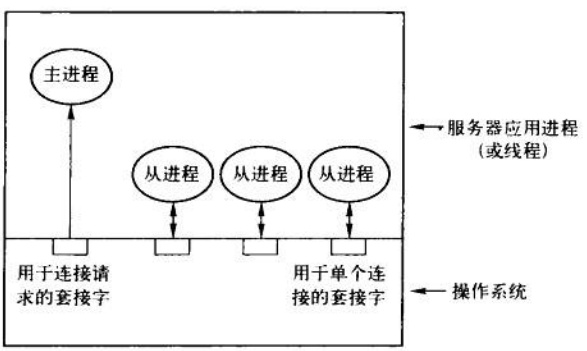


图 5.1　基于多进程并发的面向连接服务器的进程结构

由图 5.1 可知，在基于多进程并发的面向连接服务器中，主进程并不直接负责与客户进行通信，它通过调用 accept( ) 函数阻塞在熟知端口上以等待下一个客户连接请求；一旦有客户连接请求到达，accept( ) 调用就返回一个新的用于该连接的从套接字描述符，同时，主进程将调用 fork( ) 函数创建一个新的从进程来处理该连接，而主进程则重新调用 accept( ) 函数阻塞在熟知端口上以等待新客户连接请求的到来。因此在基于多进程并发的面向连接服务器中，服务器在任何时候都会包括一个主进程和零个或多个从进程，其中，主进程基于主套接字监听是否有新的客户连接请求到达，而每一个从进程则基于不同的从套接字负责处理与不同客户之间的通信和交互。

5.2 基于多进程并发的面向连接服务器软件的设计流程

5.2.1 不固定进程数的并发模型设计流程

步骤 1：主进程创建主套接字 msock 并绑定到熟知端口。

步骤 2：主进程调用 accept( ) 函数基于主套接字在熟知端口上等待客户连接请求的到达。

步骤 3：当有客户连接请求到达，主进程建立与该客户之间的通信连接，同时 accept( ) 调用返回一个新的用于该连接的从套接字描述符 ssock。

步骤 4：主进程创建一个新的从进程来处理该连接。

步骤 5：主进程关闭套接字 ssock（此时，由于从进程仍然打开着从套接字 ssock，故主进程的关闭操作仅仅只是把从套接字 ssock 的引用计数减少 1，而不会真正关闭该从套接字）。

步骤 6：主进程返回步骤 2 继续执行。

步骤 7：从进程关闭主套接字 msock（此时，由于主进程仍打开着主套接字 msock，故从进程的关闭操作仅仅只是把主套接字 msock 的引用计数减少 1，而不会真正关闭该主套接字）。

步骤 8：从进程调用 recv( ) 和 send( ) 等操作与客户进行数据交换。

步骤 9：数据交换完毕，从进程关闭从套接字 ssock，从进程结束。

5.2.2 固定进程数的并发模型设计流程

固定进程数的并发模型是一种介于单进程与多进程之间的折中方案，在固定进程数的并发模型中，主进程在创建主套接字 msock 之后将创建给定数目的从进程，由从进程来等待客户端的连接请求并完成与客户端的通信交换等工作，而主进程的功能只是用于维持从进程的数量不变。

1. 父进程的设计流程

步骤 1：主进程创建主套接字 msock 并绑定到熟知端口。

步骤 2：主进程创建给定数目的从进程。

步骤 3：主进程调用 wait( ) 函数等待从进程结束，一旦有从进程退出，则主进程立即创建一个新的从进程，以保证从进程在数量上维持不变。

2. 从进程的设计流程

步骤 1：从进程调用 accept( ) 函数等待客户连接请求的到达。

步骤 2：当有客户连接请求到达，从进程建立与该客户之间的通信连接，同时 accept( ) 调用返回一个新的用于该连接的从套接字描述符 ssock。

步骤 3：从进程调用 recv( ) 和 send( ) 等操作与客户端进行数据交换。

步骤 4：数据交换完毕，从进程关闭从套接字 ssock。

步骤 5：从进程返回步骤 1 继续执行。

5.3　基于多进程并发的面向连接服务器例程

5.3.1　例程一

1. 服务器端例程

该服务器所实现的功能为：首先，等候客户连接请求，一旦连接成功则显示客户的 IP 地址；然后，再接收该客户的名字并显示；最后，接收来自用户的其他信息，当每收到一个字符串时，则首先显示该字符串，然后再将该字符串反转并将反转后的字符串回送给该客户端。

```
//TCPserver.c
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf()
                    等)的定义*/
#include <unistd.h>  /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构
                      的定义*/
#include <sys/types.h> //基本数据类型头文件,含有基本系统数据类型的定义
#include <sys/socket.h> //提供套接字函数原型与数据结构的定义
#include <netinet/in.h> //提供数据结构 sockaddr_in 的定义
#include <arpa/inet.h> //提供 IP 地址转换函数原型的定义
#include <stdlib.h>  /*C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()等)原型的
                      定义*/
#include <string.h>  //提供字符串函数原型的定义
#define PORT 1234   //定义服务器端的熟知端口号
#define QLEN 10   //定义允许排队的连接数
#define BUFSIZE 1024   //定义缓冲区大小为 1024B
void sig_chld(int); //声明 sig_chld()函数
void process_cli(int connectfd, sockaddr_in client); /*声明用于处理与客户之间通信的子函数
                                                      process_cli()*/
int main(){
    int listenfd,connectfd;   //定义主套接字和从套接字描述符变量
    pid_t pid; //定义进程标识变量
    struct sockaddr_in server; //定义服务器端点地址结构变量
    struct sockaddr_in client; //定义客户端端点地址结构变量
    socklen_t ssize;
    if((listenfd=socket(AF_INET,SOCK_STREAM,0))==-1){ /*创建主套接字*/
        perror("Creating socket failed.");
        exit(1);
    }
    int opt=SO_REUSEADDR; /*设置与主套接字关联的选项,允许主套接字重用本地地址和端
                           口*/
```

```
        setsockopt(listenfd,SOL_SOCKET,SO_REUSEADDR,&opt,sizeof(opt));
        bzero(&server,sizeof(server)); //清空服务器端点地址结构变量
        //以下代码段用于对服务器端点地址结构变量进行赋值
        server.sin_family = AF_INET;
        server.sin_port = htons(PORT);
        server.sin_addr.s_addr = htonl (INADDR_ANY);
    //调用 bind()函数将主套接字绑定到服务器的端点地址
        if(bind(listenfd,(struct sockaddr *)&server,sizeof(struct sockaddr)) == -1){
            perror("Bind error.");
            exit(1);  //调用 bind()函数出错则退出系统
        }
        if(listen(listenfd,QLEN) == -1){ /*调用 listen()函数使主套接字处于被动的监听模式,并
                                          为主套接字建立一个输入数据队列*/
            perror("listen() error\n");
            exit(1); //调用 listen()函数出错则退出系统
        }
        ssize = sizeof(struct sockaddr_in);
        signal(SIGCHLD,sig_chld); //调用 signal()函数为 SIGCHLD 信号安装 handler
        while(1){ /*循环调用 accept()函数接受客户连接请求,建立连接,并创建新的从套接字
                    connectfd 用于处理该连接*/
            if((connectfd = accept(listenfd,(struct sockaddr *)&client,&ssize)) == -1){
                perror("accept() error\n");
                exit(1); //调用 accept()函数出错则退出系统
            }
            if((pid = fork()) >0){    //调用 fork()函数创建新的从进程
                close(connectfd);  //在父进程中关闭从套接字描述符
                continue; //父进程返回 while 循环
            }
            else if(pid ==0){
                close(listenfd);   //在子进程中关闭主套接字描述符
                process_cli(connectfd,client); /*在子进程中调用 process_cli()函数基于建新的从套
                                                 接字 connectfd 来处理与客户之间的通信*/
                exit(0); //处理完毕与客户的通信之后退出子进程
            }
            else {  //调用 fork()创建从进程出错则打印出错信息并退出系统
                printf("fork error\n");
                exit(0);
            }
        }
        close(listenfd); //主进程结束时关闭主套接字
    }
    void process_cli(int connectfd,sockaddr_in client){
        int num;
```

```
    char recvbuf[ BUFSIZE ],sendbuf[ BUFSIZE ],cli_name[ BUFSIZE ];//定义相关缓存区
    int len;
    printf( "You got a connection from %s. ",inet_ntoa( client. sin_addr ) ); /* 打印输出客户的 IP
                                                                              地址 */
    num = recv( connectfd,cli_name,BUFSIZE,0); /* 从从套接字中读取客户发送过来的数据(即
                                                 接收该客户的名字) */
    if ( num == 0) { //若从套接字中数据读取完毕则关闭从套接字并返回
        close( connectfd);
        printf( "Client disconnected. \n" );
        return;
    }
    cli_name[ num - 1 ] = '\0'; //在字符串末尾添加字符串结束符'\0'
    printf( "Client's name is %s. \n",cli_name); //打印输出客户的名字
  while ( num = recv( connectfd,recvbuf,BUFSIZE,0)) {//循环接收来自客户的其他信息
      recvbuf[ num] = '\0'; //在字符串末尾添加字符串结束符'\0'
      printf( "Received client ( %s ) message: %s",cli_name,recvbuf);
      for ( int i = 0; i < num - 1; i ++ ) {  //将接收到的客户信息进行反转
          sendbuf[ i ] = recvbuf[ num - i  - 2];
      }
      sendbuf[ num - 1 ] = '\0'; //在字符串末尾添加字符串结束符'\0'
      len = strlen( sendbuf);
      send( connectfd,sendbuf,len ,0); //将反转后的数据回送给客户端
  }
  close( connectfd); //关闭从套接字
}

void sig_chld( int signo) {
    pid_t pid;
    int stat;
  while( ( pid = waitpid( -1,&stat,WNOHANG)) >0) /* 调用 waitpid( )函数等待子进程结束,若
                                                  有子进程结束,则 waitpid( )函数将对其
                                                  回收,从而避免产生僵尸进程 */
      printf( "child %d terminated\n",pid);
    return;
}
```

2. 客户端例程

客户端首先与服务器相连，接着发送客户端名字，然后发送客户信息，接收到服务器信息并显示，之后等待用户输入 Crtl + D，就关闭连接并退出。

```
//TCPclient. c
#include  <stdio. h>   /* 标准输入/输出头文件,包含了标准输入/输出函数(如 perror( )和 printf
                        ( )等)的定义 */
```

```
#include <unistd.h>  /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构
                       的定义*/
#include <string.h>  //提供字符串函数原型的定义
#include <sys/types.h>  //基本数据类型头文件,含有基本系统数据类型的定义
#include <sys/socket.h>  //提供套接字函数原型的定义
#include <netinet/in.h>  //提供数据结构 sockaddr_in 的定义
#include <netdb.h>  //含有 hostent 结构与 gethostbyname 函数的定义
#include <stdlib.h>  /*C 标准库头文件,包含了 C 语言标准库函数(如 exit(、atoi)等)原型的定
                       义*/
#define PORT 1234  //定义服务器的熟知端口号
#define BUFSIZE 100  //定义缓冲区的大小
void process(FILE *fp,int sockfd); /*声明用于处理与服务器之间的通信的子函数 process()*/
char * getMessage(char * sendline,int len,FILE * fp); /*声明用于实现接受用户键盘输入数据的
                                                        子函数 getMessage ()*/
int main(int argc,char *argv[]) {
   int fd;  //定义文件描述符变量
   struct hostent *he;  //定义 hostent 结构变量
   struct sockaddr_in server;  //定义服务器端点地址结构变量
   if (argc!=2) { //若用户输入的命令行参数错误则提示用法并退出系统
       printf("Usage: %s <IP Address>\n",argv[0]);
       exit(1);
   }
if ((he=gethostbyname(argv[1]))==NULL) { /*调用 gethostbyname()由用户输入的远程服务器
                                           的十进制 IP 地址获得其二进制的 IP 地址*/
       printf("gethostbyname() error\n");
       exit(1); //若调用 gethostbyname()出错则退出系统
   }
   if ((fd=socket(AF_INET,SOCK_STREAM,0))==-1) {  //创建套接字
       printf("socket() error\n");
       exit(1);
   }
   bzero(&server,sizeof(server)); //清空服务器端点地址结构变量
   //以下代码段用于对服务器端点地址结构变量进行赋值
   server.sin_family=AF_INET;
   server.sin_port=htons(PORT);
   server.sin_addr=*((struct in_addr *)he->h_addr);
   if(connect(fd,(struct sockaddr *)&server,sizeof(struct sockaddr))==-1) {
   //调用 connect()函数向远程服务器发送连接请求
       printf("connect() error\n");
       exit(1);
   }
   process(stdin,fd);  /*调用子 process(process()函数基于新创建的套接字与服务器之间进行
                         交互*/
```

```
    close(fd);    //交互完毕,关闭套接字
}

void process(FILE * fp,int sockfd) {
    char sendline[BUFSIZE],recvline[BUFSIZE];
    int numbytes;
    printf("Connected to server. \n");
   printf("Input name : ");
if ( fgets(sendline,BUFSIZE,fp) == NULL) { /* 调用 fgets()函数接受用户键盘输入的客户端名
                                              字并存入缓冲区 sendline */
        printf("\nExit. \n");
        return;
    }
    send(sockfd,sendline,strlen(sendline),0); /* 将 sendline 中缓存的客户端名字发送给服务
                                                 器 */
while (getMessage(sendline,BUFSIZE,fp)!= NULL) { /* 循环调用 getMessage()函数接受用户键
                                                    盘输入的信息并存入缓冲区 sendline */
        send(sockfd,sendline,strlen(sendline),0);    /* 将缓存在 sendline 中的信息发送给服务
                                                        器 */
        if ((numbytes = recv(sockfd,recvline,BUFSIZE,0)) ==0) {/* 调用 recv()函数接收服务器
                                                                  回送的信息并存入缓冲区
                                                                  recvline */
            printf("Server terminated. \n");
            return;
        }
        recvline[numbytes] ='\0'; //在字符串末尾添加字符串结束符'\0'
        printf("Server Message: %s\n",recvline); //打印输出服务器的回送信息内容
   }
   printf("\nExit. \n");
}

char * getMessage(char * sendline,int len,FILE * fp) {
    printf("Input string to server:");
    return(fgets(sendline,BUFSIZE,fp)); /* 调用 fgets()函数接受用户键盘输入的信息并存入缓
                                           冲区 sendline */
}
```

5.3.2　例程二

1. 服务器端例程

该服务器所实现的功能为仅将它从客户端收到的所有数据原封不动地返回给该客户端。

```
/* TCPechod.c */
```

```
#define _USE_BSD
#include <sys/types.h> /*基本数据类型头文件,含有基本系统数据类型的定义*/
#include <sys/signal.h> //提供 signal 函数原型的定义
#include <sys/socket.h> //提供套接字函数原型的定义
#include <sys/time.h> //提供 time 函数原型的定义
#include <sys/resource.h> /*为资源操作提供了定义,包括在一个程序允许的尺寸,执行的优先
                            级以及文件上确定和设置限制的函数的原型定义*/
#include <sys/wait.h>  //提供 wait 函数原型的定义
#include <sys/errno.h> //提供错误号 errno 的定义,用于错误处理
#include <netinet/in.h> //提供数据结构 sockaddr_in 的定义
#include <unistd.h>  /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构
                       的定义*/
#include <stdlib.h>  /*C 标准库头文件,包含了 C 语言标准库函数(如 exit、atoi 等)原型的定
                       义*/
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf()
                     等)的定义*/
#include <string.h> //提供字符串函数原型的定义
#define QLEN 32  //定义允许排队的连接数
#define BUFSIZE 4096  //定义缓冲区大小为 1024B
extern int error;
void reaper(int);
int TCPechod(int fd);
int errexit(const char *format,…);
int passiveTCP(const char *service,int qlen);
int main(int argc,char *argv[]){
  char *service="echo"; //定义服务器提供的服务名变量
  struct sockaddr_in fsin; //定义服务器端点地址变量
  unsigned int alen;
  int msock; //定义主套接字描述符变量
  int ssock; //定义从套接字描述符变量
  switch(argc){  //判断用户命令行输入是否正确
      case 1:
          break;
      case 2:
          service=argv[1]; /*将用户命令行输入的第二个参数(即用户输入的服务器端口号)
                             存入到服务名变量*/
          break;
      default:  //若用户命令行输入出错则提示用法
          errexit("usage: TCPechod [port] \n");
  }
  msock=passiveTCP(service,QLEN);  //建立被动 TCP 套接字
  (void) signal (SIGCHLD,reaper); /*调用 signal()函数为 SIGCHLD 信号安装 handler */
  while (1){  //循环接受来自客户的连接请求
```

```
        alen = sizeof( fsin ) ;
        ssock = accept( msock , ( struct sockaddr * ) &fsin , &alen ) ;  /* 调用 accept( ) 函数建立与客户
                                                                            的连接,并创建一个新的从套
                                                                            接字 */
          if( ssock < 0 ) {   //若调用 accept( ) 函数出错
            if( errno == EINTR )
                continue ;
            errexit( " accept : % s\n" , strerror( errno ) ) ;
          }
          switch( fork( ) ) { /* 若调用 accept 函数成功,则调用 fork( ) 函数创建新的从进程来基于新
                                 创建的套接字来处理与该客户之间的连接 */
            case 0 : //在子进程中
                ( void ) close( msock ) ; //关闭主套接字
                exit( TCPechod( ssock ) ) ; /* 在调用 TCPechod( ) 函数完成与客户的交互之后,退出
                                               子进程 */
            default : //在父进程中
                ( void ) close( ssock ) ; //关闭从套接字
                break ; //跳出 switch 语句
            case  -1 : //若调用 fork( ) 函数出错,则退出系统
                 errexit( " fork : % s\n" , strerror( errno ) ) ;
          }
      }
}

int TCPechod( int fd ) {
      char buf[ BUFSIZE ] ;
      int cc ;
      while( cc = read( fd , buf , sizeof( buf ) ) ) { /* 调用 read( ) 函数从套接字中接收来自客户的信息并
                                                          存入到缓冲区 buf 之中 */
          if( cc < 0 )
              errexit( " echo read : % s\n" , strerror( errno ) ) ;
          if( write( fd , buf , cc ) < 0 ) /* 调用 write( ) 函数将缓冲区 buf 中的数据写入套接字回送给
                                              客户 */
              errexit( " echo write : % s\n" , strerror( errno ) ) ;
      }
      return 0 ;
}

void reaper( int sig ) {
      int status ;
      while( wait3( &status , WNOHANG , ( struct rusage * ) 0 ) > = 0 ) { /* 调用 wait( ) 函数等待子进程
                                                                            结束,若有子进程结束,则
                                                                            wait( ) 函数将对其回收,从
```

```
                                                    而避免产生僵尸进程,同时,
                                                    还可调用 WIFEXITED( )等
                                                    函数来获取子进程结束的有
                                                    关状态 * /
    if (WIFEXITED (status)) //若子进程正常结束
        printf ("the child proces exited normally,with exit code %d\n",WEXITSTATUS (status));
    else  //若子进程异常结束
        printf ("the child process exited abnormally\n");
    }
}
```

2. 客户端例程

客户端例程所实现的功能为：首先，向服务器发送数据，然后，再读取服务器返回的数据并将其打印出来。

(1) TCP 客户

```
#include <unistd.h> /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的
                       定义 * /
#include <stdlib.h>  /* C 标准库头文件,包含了 C 语言标准库函数(如 exit( )、atoi( )等)原型的
                       定义 * /
#include <string.h>  //提供了字符串函数原型的定义
#include <stdio.h>  /* 标准输入/输出头文件,包含了标准输入/输出函数(如 perror( )和 printf
                       ( )等)的定义 * /
#include <errno.h>  //提供错误号 errno 的定义,用于错误处理
extern int errno; //引入头文件 errno.h 中定义的用于记录出错信息的全局变量 errno
int TCPecho(const char *host,const char *service); //声明 TCPecho( )函数
int errexit(const char *format,…); //声明可变参数函数 errexit ( )
int connectTCP(const char *host,const char *service); //声明 connectTCP( )函数
#define LINELEN 128 //定义缓存区大小
int main(int argc,char *argv[]) {
    char *host = "localhost"; //定义用于存储服务器主机名(IP 地址)的变量 host
    char *service = "echo"; //定义用于存储服务名(端口号)的变量 service
    switch(argc) { //判断命令行输入的参数的个数
    case 1: /* 若只输入一个参数,即只输入可执行文件名,而没有输入服务器的主机名(IP 地
               址)与服务名(端口号),则将服务器的主机名设置为默认的主机名"localhost",然
               后跳出该分支判断语句 * /
    host = "localhost";
    break;
case 2: /* 若输入两个参数,即除了输入可执行文件名之外,还输入服务器的主机名(IP 地址),则
           在将服务器的主机名(IP 地址)保存到变量 host 中之后,再跳出该分支判断语句 * /
        host = argv[1];
        break;
    case 3: /* 若输入三个参数,即除了输入可执行文件名之外,还输入服务器的主机名(IP 地
```

```
            址)与服务名(端口号),则在将服务器的主机名(IP 地址)保存到变量 host 中、同
            时将服务器的服务名(端口号)保存到变量 service 中之后,再跳出该分支判断语
            句 */
        host = argv[1];
        service = argv[2];
        break;
    default: /* 若输入四个或以上参数,则调用 fprintf()函数输出用法提示,然后调用 exit(1) 使得系
            统异常(出错)退出 */
      fprintf(stderr,"usage: TCPecho [host [port]] \n"0;
      exit(1);
    }
    TCPecho(host,service); //调用并执行 TCPecho()函数
    exit(0); //退出系统
}

int TCPecho(const char *host,const char *service) {
    char buf[LINELEN + 1]; //定义缓存区数组 buf[]
    int s,n;
    int outchars,inchars;
    s = connectTCP(host,service); /* 调用 connectTCP()函数与服务器建立 TCP 连接并创建套接
                字 s */
    while(fgets(buf,sizeof(buf),stdin)) { /* 调用 fgets()从标准输入(键盘)中获取数据缓存到
                buf */
    buf[LINELEN] ='\0'; //在缓存区末尾添加行结束符'\0'
    outchars = strlen(buf); //调用 strlen()函数计算得到缓存的字符数
    (void)write(s,buf,outchars); /* 调用 write()函数将缓存区 buf 中的数据写入到套接字 s
                中 */
    for(inchars =0;inchars < outchars;inchars + =n) { /* 由于建立了与服务器之间的连接,因此
                这里需要调用 FOR 循环来从套接字 s
                中循环接收数据 */
        n = read(s,&buf[inchars],outchars - inchars); /* 调用 read()函数从套接字 s 中读取数
                据缓存到 buf */
        if(n <0) //若读取到的数据的长度为 0,则提示出错信息
            errexit("socket read failed"%s\n",strerror(errorno));
    }
    fputs(buf,stdout); /* 调用 fputs()函数将缓存区 buf 中的数据输出到标准输出设备(显示
                器) */
    }
}
```

(2) UDP 客户

```
#include <unistd.h> /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的
            定义 */
```

```
#include <stdlib.h>  /*C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()等)原型的
                      定义*/
#include <string.h> //提供了字符串函数原型的定义
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf()
                    等)的定义*/
#include <errno.h> //提供错误号 errno 的定义,用于错误处理
extern int errno; //引入头文件 errno.h 中定义的用于记录出错信息的全局变量 errno
int UDPecho(const char *host,const char *service); //声明 UDPecho()函数
int errexit(const char *format,…); //声明可变参数函数 errexit ()
int connectUDP(const char *host,const char *service);
#define LINELEN 128
int main(int argc,char *argv[]) {
    char *host = "localhost";
    char *service = "echo";
    switch(argc) {
    case 1: /*若只输入一个参数,即只输入可执行文件名,而没有输入服务器的主机名(IP 地
            址)与服务名(端口号),则将服务器的主机名设置为默认的主机名"localhost",然
            后跳出该分支判断语句*/
        host = "localhost";
        break;
    case 2: /*若输入两个参数,即除了输入可执行文件名之外,还输入服务器的主机名(IP 地
            址),则在将服务器的主机名(IP 地址)保存到变量 host 中之后,再跳出该分支判断
            语句*/
        host = argv[1];
        break;
    case 3: /*若输入三个参数,即除了输入可执行文件名之外,还输入服务器的主机名(IP 地
            址)与服务名(端口号),则在将服务器的主机名(IP 地址)保存到变量 host 中、同
            时将服务器的服务名(端口号)保存到变量 service 中之后,再跳出该分支判断语
            句*/
        host = argv[1];
        service = argv[2];
        break;
    default: /*若输入四个或以上参数,则调用 fprintf()函数输出用法提示,然后调用 exit(1) 使
            得系统异常(出错)退出*/
        fprintf(stderr,"usage: TCPecho [host [port]] \n"0;
        exit(1);
    }
    UDPecho(host,service); //调用并执行 UDPecho()函数
    exit(0); //退出系统
}

int UDPecho(const char *host,const char *service) {
    char buf[LINELEN+1]; //定义缓存区数组 buf[]
```

```
    int s,nchars;
    s = connectUDP(host,service); //调用 connectUDP()创建套接字 s
    while(fgets(buf,sizeof(buf),stdin)) { /* 调用 fgets()从标准输入(键盘)中获取数据缓存到
                                            buf */
        buf[LINELEN] ='\0'; //在缓存区末尾添加行结束符'\0'
        nchars = strlen(buf); //调用 strlen()函数计算得到缓存的字符数
        (void)write(s,buf,nchars); /* 调用 write()函数将缓存区 buf 中的数据写入到套接字 s
                                      中 */
        if(read(s,buf,nchars) <0)   /* 调用 read()函数从套接字 s 中读取数据缓存到 buf,由于
                                       没有建立与服务器之间的连接,因此这里不需要调用
                                       FOR 循环来从套接字 s 中循环接收数据 */
            errexit("socket read failed"%s\n",strerror(errorno));
        fputs(buf,stdout); /* 调用 fputs()函数将缓存区 buf 中的数据输出到标准输出设备(显
                              示器) */
    }
}
```

5.4　本章小结

本章主要对基于多进程并发的面向连接服务器及其实现方法进行深入介绍，并在实现原理介绍的基础上，具体给出两个创建基于多进程并发的面向连接服务器的完整 C 语言例程。通过本章的学习，需要了解基于多进程并发的面向连接服务器的进程结构；需要熟悉基于多进程并发的面向连接服务器的设计流程；需要掌握基于多进程的并发的面向连接服务器的 C 语言实现方法。

本章习题

1. 简述基于多进程并发的面向连接服务器的进程结构。
2. 简述基于多进程并发的面向连接服务器软件的设计流程。
3. 试构造一个基于多进程并发的面向连接服务器例程，该例程能实现以下功能：能同时等候来自 10 个不同客户的连接请求，一旦与某个客户连接成功则接收来自该客户的信息，当每收到一个字符串时将首先显示该字符串，然后再将该字符串反转、最后再将反转后的字符串回送给该客户。

第 6 章　基于多线程并发的面向连接服务器例程剖析

第 5 章介绍了基于多进程并发的面向连接服务器及其实现方法，本章将进一步针对另一种并发服务器——基于多线程并发的面向连接服务器及其实现方法进行深入介绍，并在实现原理介绍的基础上，具体给出一个创建基于多线程并发的面向连接服务器的完整 C 语言例程。

6.1　线程之间的协调与同步

虽然多线程能给我们带来好处，但是也有不少问题需要解决。例如，对于像磁盘驱动器这样独占性系统资源，由于线程可以执行进程的任何代码段，且线程的运行是由系统调度自动完成的，具有一定的不确定性，因此就有可能出现两个线程同时对磁盘驱动器进行操作，从而出现操作错误；又例如，对于银行系统的计算机来说，可能使用一个线程来更新其用户数据库，而用另外一个线程来读取数据库以响应储户的需要，极有可能读数据库的线程读取的是未完全更新的数据库，因为可能在读的时候只有一部分数据被更新过。为此，程序员在编写多线程的程序时，需要考虑同一进程中的多个线程的协调执行问题，其中，使得隶属于同一进程的各线程协调一致地工作的机制，就称为线程的同步机制。LINUX 提供了以下三种主要的线程同步机制：互斥（Mutex）、信号量（Semaphore）和条件变量（Condotion Variable）。

6.1.1　互斥锁

所谓的互斥也就是意味着“排他”，互斥锁提供了对共享资源的一种保护访问。其中，每个互斥锁与一个共享数据项相关，从而使得两个线程不能同时进入被互斥锁所保护的该共享数据项。互斥锁只有两种状态，即上锁和解锁，可以把互斥锁看作某种意义上的全局变量。在同一时刻，只有拥有上锁状态的线程能够对共享资源进行操作。若其他线程希望上锁一个已经上锁了的互斥锁，则该线程会被挂起，直到上锁的线程释放掉该互斥锁为止。显然，采用互斥锁可以使得共享资源能够按先后顺序在各个线程中依次操作。

以一个公用电话亭的运作为例。假定一个电话亭只有一门公用电话，一次只允许一个用户使用，因此，为了防止用户发生冲突，显然电话亭的门上就应该有这样一个标志，并用它来表示电话亭的被占用情况。例如，用一个可以变换两种颜色的牌子，用红色表示“有人”，用绿色表示“没人”。这样一来，当人们见到牌子上的颜色是绿色时就可以进去打电话；而如果是红色那么就只好等待；如果后来又陆续到了很多人，那么这些后来的人就需要排队等待。其中，这里的电话亭上的牌子就相当于是一个互斥锁。

在 LINUX 中，可以通过定义数据类型 pthread_mutex_t 的互斥锁变量来实现对于多个线

程的互斥操作，该机制的作用是对某个需要互斥的共享数据项，在进入时先得到互斥锁，如果没有得到互斥锁，表明互斥部分被其他线程拥有，此时欲获取互斥锁的线程将会被阻塞，直到拥有该互斥体的其他线程完成互斥部分的操作为止。

互斥锁可以分为快速互斥锁、递归互斥锁和检错互斥锁。这三种锁的区别主要在于其他未占有互斥锁的线程在希望得到互斥锁时的是否需要阻塞等待。其中，快速锁是指调用线程会阻塞直至拥有互斥锁的线程解锁为止。递归互斥锁能够成功地返回并且增加调用线程在互斥上加锁的次数，而检错互斥锁则为快速互斥锁的非阻塞版本，它会立即返回并返回一个错误信息。互斥锁的操作主要包括以下几个步骤。

定义互斥锁变量：

pthread_mutex_t;

定义互斥锁属性变量：

pthread_mutexattr_t;

初始化互斥锁属性变量：

pthread_mutexattr_init( );

设置互斥锁属性：

pthread_mutexattr_settype、pthread_mutexattr_setpshared、pthread_mutexattr_gettype、pthread_mutexattr_getpshared、pthread_mutexattr_destroy;

初始化互斥锁变量：

pthread_mutex_init( );

互斥锁上锁：

pthread_mutex_lock( );

互斥锁判断上锁：

pthread_mutex_trylock( );

互斥锁解锁：

pthread_mutex_unlock( );

消除互斥锁：

pthread_mutex_destroy( )。

在上述互斥锁的操作步骤中，各相关函数及其调用方法如下。

1. pthread_mutex_init( ) 函数

在 LINUX 下，线程的互斥锁变量的数据类型是 pthread_mutex_t，在使用互斥锁变量之前需要对它进行初始化。其中，对于静态分配的互斥锁变量，可以把它设置为 PTHREAD_MUTEX_INITIALIZER，也可调用 pthread_mutex_init( ) 对互斥锁变量进行动态分配。pthread_mutex_init( ) 函数若调用成功将返回 0，若出错则返回 -1。其函数原型如下：

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * mutexattr);
```

在上述 pthread_mutex_init( ) 函数的原型中，各参数的含义如下。

mutex：指向要初始化的互斥锁的指针。

mutexattr：指向互斥锁属性对象的指针，该属性对象定义要初始化的互斥锁的属性。如果该指针设置为 NULL，则表示使用默认的属性，默认属性为快速互斥锁。

2. pthread_mutexattr_init( ) 函数

互斥锁的属性类型为 pthread_mutexattr_t，声明后调用 pthread_mutexattr_init( ) 函数来初始化该互斥锁的属性对象。然后再调用 pthread_mutexattr_settype( ) 函数和 pthread_mutexattr_setpshared( ) 函数来设置其属性。其中，pthread_mutexattr_init( ) 函数的原型如下。

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_mutexattr_init(pthread_mutexattr_t * mattr);
```

在上述 pthread_mutexattr_init( ) 函数的原型中，其参数的含义如下。

mattr：指向互斥锁属性对象的指针。

3. pthread_mutexattr_getpshared( )/pthread_mutexattr_setpshared( ) 函数

pthread_mutexattr_getpshared( )/pthread_mutexattr_setpshared( ) 函数分别用于获得/设置互斥锁属性对象的共享属性，函数若调用成功将返回 0，若失败则返回错误编号。其函数原型如下：

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_mutexattrattr_setpshared(const pthread_attr_t * mattr, int pshared);
int pthread_mutexattrattr_setpshared(const pthread_attr_t * mattr, int * pshared);
```

在上述 pthread_mutexattr_getpshared( )/pthread_mutexattrattr_setpshared( ) 函数的原型中，各参数的含义如下。

mattr：指向互斥锁属性对象的指针。

pshared：互斥锁属性对象的共享属性，该参数只有两个取值，分别为 PTHREAD_PROCESS_PRIVATE 和 PTHREAD_PROCESS_SHARED。前者表示在多个进程中的线程之间共享该互斥锁，后者表示仅在那些由同一个进程所创建的线程之间共享该互斥锁。

4. pthread_mutexattr_gettype( )/pthread_mutexattr_settype( ) 函数

pthread_mutexattr_gettype( )/pthread_mutexattr_settype( ) 函数分别用于获得/设置互斥锁的类型属性。函数若调用成功将返回 0，若失败则返回错误编号。其函数原型如下。

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_mutexattr_settype(pthread_mutexattr_t * mattr, int type);
int pthread_mutexattr_gettype(pthread_mutexattr_t * mattr, int * type);
```

在上述 pthread_mutexattr_gettype( )/pthread_mutexattr_settype( ) 函数的原型中，各参数的含义如下。

mattr：指向互斥锁属性对象的指针。

type：互斥锁的类型属性。主要包括如下几种：

PTHREAD_MUTEX_NORMAL：快速互斥锁。

PTHREAD_MUTEX_RECURSIVE：递归互斥锁。

PTHREAD_MUTEX_ERRORCHECK：检错互斥锁。

5. pthread_mutexattr_destroy( ) 函数

在修改互斥锁属性对象之前需要调用 pthread_mutexattr_init( ) 函数对其进行初始化，而在使用之后还需调用 pthread_mutexattr_destroy( ) 函数来将其回收，该函数若调用若成功将返回 0，若失败则返回错误编号。其函数原型如下：

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_mutexattrattr_destroy( pthread_mutexattr_t * mattr );
```

在上述 pthread_mutexattr_destroy( ) 函数的原型中，其参数的含义如下。

mattr：指向互斥锁属性对象的指针。

6. pthread_mutex_lock( ) 函数

当前线程调用 pthread_mutex_lock( ) 函数时，如果该互斥锁尚未加锁，则当前线程将获得该互斥锁并将该互斥锁加锁；如果当前该互斥锁已经加锁，则当前线程将会被阻塞，直到该互斥锁被解锁，然后当前线程将获得该互斥锁并加锁返回（注：若有多个线程调用了该互斥锁，则每次解锁之后，将只有一个线程可以被解除阻塞恢复执行，而其他调用该互斥锁的线程都会被继续阻塞；另外，在所有被阻塞的线程之中，解除阻塞的线程是不可预知的）。pthread_mutex_lock( ) 函数若调用成功将返回 0，若失败则返回错误编号。其函数原型如下：

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_mutex_lock(pthread_mutex_t * mutex);
```

在上述 pthread_mutex_lock( ) 函数的原型中，其参数的含义如下。

mutex：指向要加锁的互斥锁的指针。

7. pthread_mutex_trylock( ) 函数

pthread_mutex_trylock( ) 函数在互斥锁已被其他线程锁住时将会立即返回，而不会阻塞当前线程；除此之外，pthread_mutex_trylock( ) 与 pthread_mutex_lock( ) 函数的功能完全一样。函数若调用成功，则在获得了互斥锁后将返回 0，若失败则将返回一个错误编号。pthread_mutex_trylock( ) 的函数原型如下。

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_mutex_trylock(pthread_mutex_t * mutex);
```

在上述 pthread_mutex_trylock( ) 函数的原型中，其参数的含义如下。

mutex：指向要加锁的互斥锁的指针。

8. pthread_mutex_unlock( ) 函数

pthread_mutex_unlock( ) 函数用于释放互斥锁。释放互斥锁后的行为取决于 mutex 的类型属性。如果有多个线程正被此互斥锁阻塞，释放此互斥锁时，互斥锁可被其他调用该互斥锁

的线程所获取，但具体哪个线程可获得该互斥锁是不可预知的，由调度策略决定（注：当互斥锁类型为 PTHREAD_MUTEX_RECURSIVE 时，只有当锁住次数为 0 且调用线程不再拥有该互斥锁时，该互斥锁才会变为可用）。pthread_mutex_unlock( ) 函数若调用成功将返回 0，若失败则将返回一个错误编号。pthread_mutex_unlock( ) 的函数原型如下：

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

在上述 pthread_mutex_unlock( ) 函数的原型中，其参数的含义如下。

mutex：指向要解锁的互斥锁的指针。

9. pthread_mutex_destroy( ) 函数

pthread_mutex_destroy( ) 函数的作用是用于释放一个互斥锁，函数若调用成功将返回 0，若失败则将返回一个错误编号。pthread_mutex_destroy( ) 函数原型如下：

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

在上述 pthread_mutex_destroy( ) 函数的原型中，其参数的含义如下。

mutex：指向待释放的互斥锁的指针。

以下给出一个基于互斥锁的多线程应用例程。

该例程使用互斥锁实现主从线程之间的同步，其中，主线程负责从标准输入设备中读取数据并保存到到全局变量 work_area 之中，而从线程则负责将全局变量 work_area 中的数据输出到标准输出设备上。具体实现过程包括以下两个步骤：①首先，主线程调用 fgets( ) 函数接收一行用户的键盘输入数据并将其保存到全局变量 work_area 之中，在用户按下回车键时，从线程将会把 work_area 中的数据输出到标准输出设备上并将 work_area 缓存区中的数据清空；②当在某次键盘输入中用户仅输入了“end”字符串时，则表示用户键盘输入数据过程结束，此时程序将结束并退出。

该例程的 C 语言源代码如下：

```
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror( ) 和 printf( ) 等)的定义*/
#include <unistd.h> /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的定义*/
#include <stdlib.h> /*C 标准库头文件,包含了 C 语言标准库函数(如 exit( )、atoi( ) 等)原型的定义*/
#include <string.h> //提供了字符串函数原型的定义
#include <pthread.h> //提供了线程函数原型和数据结构的定义
#include <semaphore.h> //提供了信号量函数原型和数据结构的定义
void * thread_function(void * arg);//声明线程体函数
pthread_mutex_t work_mutex; //定义互斥锁变量
#define WORK_SIZE 1024 //定义符号常量,用作指定缓存区大小
char work_area[WORK_SIZE]; //定义全局变量,用于缓存用户键盘输入数据
int time_to_exit = 0; //定义全局变量,用作循环结束标志符
```

```
int main( ) {
    int res; //定义局部变量,用于存储函数调用后的返回值
    pthread_t a_thread; //定义局部变量,用于存储所创建的新线程的标识符
    void * thread_result; /* 定义局部指针变量,用于存储调用 pthread_join( ) 函数等待从线程结束
                             时被等待线程的返回值 */
    res = pthread_mutex_init(&work_mutex, NULL);
                        //调用 pthread_mutex_init( ) 函数对互斥锁进行初始化
    if(res! =0) { /* 若互斥锁初始化失败则调用 perror( ) 函数提示出错信息,然后调用 exit( ) 函
                     数退出程序 */
        perror("Mutex initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
                                        //调用 pthread_create( ) 函数创建一个新线程
    if(res! =0) { /* 若新线程创建失败则调用 perror( ) 函数提示出错信息,然后调用 exit( ) 函数
                     退出程序 */
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    pthread_mutex_lock(&work_mutex);
        /* 由于需要对全局变量 work_area 进行写操作,因此主线程须先调用 pthread_mutex_lock
           ( ) 函数对互斥锁进行加锁 */
    printf("Input some text. Enter 'end' to finish\n"); /* 主线程调用 printf( ) 函数提示用户键盘输
                                                           入数据 */
    while(! time_to_exit) { /* 若 time_to_exit 等于 0,即表示用户键盘输入尚未结束,time_to_exit
                               的值由从线程修改更新 */
    fgets(work_area, WORK_SIZE, stdin);
        /* 调用 fgets( ) 函数从标准输入设备 stdin 中读入 WORK_SIZE - 1 个字符放入 work_area
           缓存区,如果在未读满 WORK_SIZE - 1 个字符之时,已读到一个换行符或一个 EOF
           (文件结束标志),则结束本次读操作,读入的字符串中最后包含读到的换行符。读入
           结束后,系统将自动在最后添加'\0'字符作为行结束符 */
    pthread_mutex_unlock(&work_mutex);
                        //调用 pthread_mutex_unlock( ) 函数对互斥锁进行解锁
    while(1) { /* 反复判断用户键盘输入数据是否已经由从线程输出到标准输出设备。由于若从
                  线程已将用户键盘输入数据输出到标准输出设备,则从线程将会把 work_area 缓
                  存区清空,故只需反复判断 work_area 缓存区是否为空,即可判定用户键盘输入
                  数据是否已由从线程输出到了标准输出设备 */
        pthread_mutex_lock(&work_mutex);
            /* 由于需要对全局变量 work_area 进行读操作,因此须先调用 pthread_mutex_lock( )
               函数对互斥锁进行加锁 */
        if(work_area[0]! ='\0') { /* 若 work_area[0]! ='\0',表示从线程尚未把 work_area 缓
                                     存区清空,即用户键盘输入数据尚未由从线程输出到标准
                                     输出设备 */
```

```
                pthread_mutex_unlock(&work_mutex);
                        //调用 pthread_mutex_unlock()函数对互斥锁进行解锁
                    sleep(1);/*调用 sleep()函数让主线程休眠 1 秒,从而让从线程获得互斥锁,
                            以便从线程将用户键盘输入数据输出到标准输出设备上*/
            }else{/*若 time_to_exit 等于 1,即表示用户键盘输入结束,调用 break 退出 while 循环*/
                break;
            }
        }
    }
    /*若 time_to_exit 等于 1,即表示用户键盘输入已经结束,此时执行以下代码段*/
    pthread_mutex_unlock(&work_mutex);/*调用 pthread_mutex_unlock()函数对互斥锁进行解
                                        锁*/
    printf("\nWaiting for thread to finish...\n");/*调用 printf()函数提示用户等待从线程结
                                                束*/
    res = pthread_join(a_thread,&thread_result);
        /*调用 pthread_join()函数让主线程等待从线程结束,主线程会一直等待直到等待的线
          程结束自己才结束;若不调用 pthread_join()函数,则主线程会很快结束从而使整个进
          程结束,此时可能从线程还未来得及将用户的键盘输入数据输出到标准输出设备*/
    if(res!=0){/*若 pthread_join()调用失败,则调用 perror()函数提示出错信息,然后调用
                exit()函数退出程序*/
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");//若从线程退出,则显示提示信息
    pthread_mutex_destroy(&work_mutex);/*调用 pthread_mutex_destroy()函数释放互斥锁*/
    exit(EXIT_SUCCESS);//调用 exit()正常退出程序
}

void *thread_function(void *arg){//从线程执行体函数
    sleep(1);//从线程先休眠,让主线程先执行,以便获取用户键盘输入
        pthread_mutex_lock(&work_mutex);/*从线程用于将用户的键盘输入数据输出到标准输
                                        出设备,由于需要对全局变量 work_area 进行读操
                                        作,因此从线程须先调用 pthread_mutex_lock()函
                                        数对互斥锁进行加锁*/
        while(strncmp("end",work_area,3)!=0){
            /*若 work_area 中包含的字符串不为"end"字符串,表示用户键盘输入尚未结束。
              显然,需要注意的是,若用户在一次输入中包含了"end"字符串,此时并不表示用
              户键盘输入已经结束;而只有在一次输入中用户仅仅输入了"end"字符串时,才
              表示用户键盘输入已经结束*/
            printf("You input %d characters\n",strlen(work_area)-1);/*调用 printf()与 strlen()函
                                                                数统计并显示用户本次输
                                                                入的字符个数*/
            work_area[0]='\0';//清空缓存区 work_area
```

```
            pthread_mutex_unlock(&work_mutex);
                        //调用 pthread_mutex_unlock()函数对互斥锁进行解锁
                sleep(1);/*从线程休眠,从而让主线程执行并获得互斥锁,以便再次获取用户的键
                        盘输入*/
                pthread_mutex_lock(&work_mutex);/*由于以下需要通过检查 work_area 是否为空
                                                来获知用户的键盘输入过程是否已经结束,
                                                因此,从线程须先调用 pthread_mutex_lock()
                                                函数对互斥锁进行加锁*/
                while(work_area[0]=='\0'){ //反复判定 work_area 是否为空
                    pthread_mutex_unlock(&work_mutex);/*若 work_area 为空,则调用 pthread_mutex_
                                                unlock()函数对互斥锁进行解锁*/
                    sleep(1);/*然后,让从线程休眠 1 秒,从而让主线程执行并获得互斥锁,以便
                            再次获取用户的键盘输入*/
                pthread_mutex_lock(&work_mutex);
                        //调用 pthread_mutex_lock()函数对互斥锁进行加锁
            }
        }
                //若 work_area 中包含的字符串等于"end"字符串,则执行以下代码段
        time_to_exit=0;/*设置 time_to_exit=0,使得主线程可跳出 while(! time_to_exit)循环*/
        work_area[0]='\0';//清空缓存区 work_area
        pthread_mutex_unlock(&work_mutex);
                        //调用 pthread_mutex_unlock()函数对互斥锁进行解锁
        pthread_exit(0);//调用 pthread_exit()让从线程正常退出
}
```

运行结果如下：

```
Input some text. Enter 'end' to finish
Wait
You input 4 characters
The Crow Road
You input 13 characters
end
Waiting for thread to finish...
Thread joined
```

6.1.2　信号量

信号量（Semaphore），有时也称为信号灯，主要用于系统中有 N 个资源可用的情况，是对互斥机制的一种推广。信号量允许 N 个线程同时执行，而不像互斥一样在某个时刻只允许一个线程执行通过临界区。

以一个公用电话亭的运作为例。假定一个电话亭可以允许多人（线程）打电话，电话亭门上的计数器在每进入一个人时自动减 1，而每出去一个人时会自动加 1。则计数器上的初值就是电话亭最多能容纳打电话的人数。如此一来，那么来人只要见到计数器的值大于

0，就可以进去打电话；否则只能等待。其中，这里的计数器就相当于用于同步线程的信号量。

抽象地讲，信号量具有如下特性：信号量是一个非负整数（电话门数），所有通过它的线程/进程（人）都会将该整数减 1，当该整数值为零时，所有试图通过它的线程都将处于等待状态。

POSIX 信号量有两种形式：有名信号量和无名信号量。其中，有名信号量的值保存在文件中，因此既可用于同一进程的不同线程之间的同步也可用于不同进程之间的同步，而无名信号量的值保存在内存中，常用于同一进程的不同线程之间的同步及相关进程（如父子进程）之间的同步。

1. 无名信号量

与互斥锁类似，无名信号量也可以动态启动：其中，函数 sem_init( )用于初始化一个信号量，它带有一个参数 N，表示可用的资源数。在初始化一个信号量之后，一个线程在使用一个资源之前必须调用函数 sem_wait( )/sem_trywait( )等待一个可用的信号量，并在用完资源之后，还需调用函数 sem_post( )来返还资源。N 个线程都可以在调用函数 sem_wait( )之后获取资源并继续执行，但此后若还有其他线程也调用了函数 sem_wait( )，则它们将会被阻塞。这些后续的线程也将一直处于阻塞状态，直至前面的 N 个线程之中有某一个调用了函数 sem_post( )将其所占资源返还之后，其中的一个阻塞线程才能得以继续运行。另外，线程还可调用 sem_getvalue( )函数来获取信号量的当前值，可调用 sem_destroy( )函数来释放信号量。

在上述无名信号量的操作步骤中，各相关函数及其调用方法如下。

（1） sem_init( )函数

信号量的数据类型为结构 sem_t，它本质上是一个长整型的数。函数 sem_init( )用来初始化一个信号量。函数若调用成功将返回 0，若出错则返回 -1，函数 sem_init( )的原型如下：

```
#include <semaphore.h> //提供了信号量函数原型和数据结构的定义
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

在上述 sem_init( )函数的原型中，各参数的含义如下。

sem：指向信号量结构的指针。

pshared：决定信号量能否在几个进程间共享。不为 0 时此信号量在进程间共享，否则只能为当前进程的所有线程共享。

value：信号量初始化值。信号量通常用来协调对资源的访问，因此，信号量初始化值通常会初始化为可用资源的数目。

（2） sem_wait( )函数

函数 sem_wait( )被用来阻塞当前线程直到信号量 sem 的值大于 0，解除阻塞后将 sem 的值减 1，表明公共资源经使用后减少。函数若调用成功将返回 0，若出错则返回 -1，函数 sem_wait( )的原型如下：

```
#include <semaphore.h> //提供了信号量函数原型和数据结构的定义
int sem_wait(sem_t *sem);
```

在上述 sem_wait( ) 函数的原型中，其参数的含义如下。

sem：指向信号量结构的指针。

(3) sem_trywait( ) 函数

函数 sem_trywait( ) 是函数 sem_wait( ) 的非阻塞版本，如果信号灯计数大于 0，则将信号量 sem 的值减 1 并返回 0，否则立即返回 -1。函数 sem_trywait( ) 的原型如下：

```
#include <semaphore.h> //提供了信号量函数原型和数据结构的定义
int sem_trywait(sem_t *sem);
```

在上述 sem_trywait( ) 函数的原型中，其参数的含义如下。

sem：指向信号量结构的指针。

(4) sem_post( ) 函数

函数 sem_post( ) 用来将信号量 sem 的值增加 1，表示增加了一个可访问的资源。当有线程阻塞在这个信号量上时，其他线程若调用这个函数会使阻塞线程中的一个不再阻塞，但选择机制是由线程的调度策略所决定的。函数若调用成功将返回 0，若出错则返回 -1，函数 sem_post( ) 的原型如下：

```
#include <semaphore.h>  //提供了信号量函数原型和数据结构的定义
int sem_ post(sem_t *sem);
```

在上述 sem_post( ) 函数的原型中，其参数的含义如下。

sem：指向信号量结构的指针。

(5) sem_getvalue( ) 函数

函数 sem_getvalue( ) 用来读取 sem 中的信号灯计数，函数若调用成功将返回 0，若出错则返回 -1。函数 sem_getvalue( ) 的原型如下：

```
#include <semaphore.h>  //提供了信号量函数原型和数据结构的定义
int sem_getvalue(sem_t *sem, int *sval);
```

在上述 sem_getvalue( ) 函数的原型中，各参数的含义如下。

sem：指向信号量结构的指针。

sval：用于存储读取到的 sem 中的信号灯计数值。

(6) sem_destroy( ) 函数

函数 sem_destroy( ) 用来释放信号量，归还自己所占用的一切资源。函数若调用成功将返回 0，若出错则返回 -1。函数 sem_destroy( ) 的原型如下：

```
#include <semaphore.h>  //提供了信号量函数原型和数据结构的定义
int sem_destroy(sem_t *sem);
```

在上述 sem_ destroy( ) 函数的原型中，其参数的含义如下。

sem：指向信号量结构的指针。

以下给出一个基于无名信号量的多线程同步应用例程。

该例程使用无名信号量实现两个从线程之间的同步，其中从线程 1 在获得无名信号量之后将对全局变量 number 执行加 1 操作，而从线程 2 在获得无名信号量之后将对全局变量 number 进行减 1 操作。

该例程的 C 语言源代码如下：

```
#include <pthread.h>   //提供了线程函数原型和数据结构的定义
#include <semaphore.h>   //提供了信号量函数原型和数据结构的定义
#include <sys/types.h> //基本数据类型头文件,含有基本系统数据类型的定义
#include <stdio.h>   /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf
                       ()等)的定义*/
#include <unistd.h>   /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构
                        的定义*/
int number; //定义全局变量
sem_t sem_id; //定义无名信号量变量

void *thread_one_fun(void *arg){ //从线程 1 执行体函数
    sem_wait(&sem_id); //阻塞线程 1 直到信号量 sem_id 的值大于 0
    printf("The thread_one has the semaphore\n"); //提示线程 1 获得信号量
    number++; //全局变量 number 加 1
    printf("number = %d\n", number); //输出全局变量 number 的值
    sem_post(&sem_id); //释放信号量 sem_id
}

void *thread_two_fun(void *arg){ //从线程 2 执行体函数
    sem_wait(&sem_id); //阻塞线程 2 直到信号量 sem_id 的值大于 0
    printf("The thread_two has the semaphore\n"); //提示线程 2 获得信号量
    number--; //全局变量 number 减 1
    printf("number = %d\n", number); //输出全局变量 number 的值
    sem_post(&sem_id); //释放信号量 sem_id
}

int main(int argc, char *argv[]){
    number = 1;
    pthread_t pid1, pid2; //定义线程描述符变量
    sem_init(&sem_id, 0,1); /*无名信号量用于多线程间的同步,设置无名信号量的初始值为 1*/
    pthread_create(&pid1, NULL,thread_one_fun, NULL); //创建从线程 1
    pthread_create(&pid2, NULL,thread_two_fun, NULL); //创建从线程 2
    pthread_join(pid1,NULL); //主线程等待从线程 1 结束
    pthread_join(pid2,NULL); //主线程等待从线程 2 结束
    printf("main…\n");
    return 0;
}
```

在上述例程中，两个线程将通过随机竞争以获取信号量资源。以下例程给出了一个无名信号量在相关进程（父子进程）中的同步应用示例。

该例程使用无名信号量实现父子进程之间的同步，其中从子进程在获得无名信号量之后将对全局变量 number 执行加 1 操作，而父进程在获得无名信号量之后将对全局变量 number

进行减 1 操作。

该例程的 C 语言源代码如下：

```
#include <pthread.h>   //提供了线程函数原型和数据结构的定义
#include <semaphore.h>   //提供了信号量函数原型和数据结构的定义
#include <sys/types.h>   //基本数据类型头文件,含有基本系统数据类型的定义
#include <stdio.h>   /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf
                     ()等)的定义*/
#include <unistd.h> /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的
                     定义*/

int main(int argc, char *argv[]){
    int i, number = 1,nloop = 10; //定义两个局部变量
    sem_t sem_id; //定义无名信号量变量
    sem_init(&sem_id, 1,1); /*无名信号量用于父子进程间的同步,设置无名信号量的初始值
                            为 1*/
    if(fork() ==0){ //在子进程中执行以下 for 循环代码段
        for(i =0, i <nloop,i ++){
            sem_wait(&sem_id); //阻塞从进程直到信号量 sem_id 的值大于 0
            printf("The son_process has the semaphore\n"); //提示从进程获得信号量
            number ++; //全局变量 number 加 1
            printf("number = %d\n", number); //输出全局变量 number 的值
            sem_post(&sem_id); //释放信号量 sem_id
        }
        exit(0); //退出子进程
    }
    //在父进程中执行以下 for 循环代码段
    for(i =0, i <nloop,i ++){
        sem_wait(&sem_id); //阻塞父进程直到信号量 sem_id 的值大于 0
        printf("The father_process has the semaphore\n"); //提示父进程获得信号量
        number --; //全局变量 number 减 1
        printf("number = %d\n", number); //输出全局变量 number 的值
        sem_post(&sem_id); //释放信号量 sem_id
    }
    exit(0); //退出父进程
}
```

2. 有名信号量

有名信号量在使用时，和无名信号量共享函数 sem_wait()、sem_trywait()和 sem_post()，但不同之处在于，有名信号量使用 sem_open()函数代替了无名信号量的初始化函数 sem_init()，另外，在结束时需要调用 sem_close()函数与 sem_unlink()函数像关闭文件一样去关闭该有名信号量。在有名信号量的操作步骤中，各相关函数及其调用方法如下。

(1) sem_open( )函数

函数 sem_open( )用于创建一个新的有名信号量或打开一个已存在的有名信号量。函数调用成功时返回指向该有名信号量的指针，出错则返回 SEM_FAILED。函数 sem_open( )的原型如下：

```
#include <semaphore.h> //提供了信号量函数原型和数据结构的定义
sem_t *sem_open(const char *name,int oflag, mode_t mode, unsigned int value);
```

在上述 sem_open( )函数的原型中，各参数的含义如下。

name：信号量的外部名字。注：由于 sem 都是创建在/dev/shm 目录之下的，因此在命名信号量的外部名字要注意不要包含路径。

oflag：有 O_CREAT | EXCL 和 O_CREAT 两个选项，选用 O_CREAT 时则当 name 指定的信号量不存在时会创建一个并要求后面的 mode 和 value 两个参数必须有效，而当 name 指定的信号量存在时则直接打开该信号量并忽略后面的 mode 和 value 两个参数；若选用 O_CREAT | EXCL 则当 name 指定的信号量不存在时与选用 O_CREAT 时功能相同，而当 name 指定的信号量存在时则将返回 error。

mode：控制新的信号量的权限。

value：信号量初始值。

(2) sem_close( )函数与 sem_unlink( )函数

函数 sem_close( )用于关闭 sem 信号量，并释放资源。函数 sem_unlink( )用于在所有进程都关闭了信号量之后删除 name 所指的信号量。函数若调用成功则返回 0，否则返回 -1。这两个函数的原型分别如下：

```
#include <semaphore.h> //提供了信号量函数原型和数据结构的定义
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

在上述 sem_close( )与 sem_unlink( )函数的原型中，各参数的含义如下。

sem：指向待关闭信号量的指针。

name：信号量的外部名字。

以下例程给出了基于有名信号量的多线程同步应用。

该例程采用循环方法建立 5 个从线程，然后让它们调用同一个线程处理函数 thread_function( )，在该函数中利用有名信号量来限制访问共享资源的线程数。共享资源用 print( )函数来代表，而在真正编程中有可能是个终端设备（如打印机）或是一段有实际意义的代码。

该例程的 C 语言源代码如下：

```
#include <pthread.h>   //提供了线程函数原型和数据结构的定义
#include <semaphore.h>   //提供了信号量函数原型和数据结构的定义
#include <sys/types.h>   //基本数据类型头文件,含有基本系统数据类型的定义
#include <stdio.h>   /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror( )和 printf
                       ( )等)的定义*/
#include <unistd.h> /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的
```

```
                        定义 */
void * thread_function( void * arg) ; //声明线程执行体函数
void print( void) ; //声明共享资源函数
sem_t bin_sem; //定义有名信号量变量
int val; //定义信号量的当前值变量
char sem_name[ ] = 'SEM_NAME' ; //定义存储有名信号量外部名字的数组变量并赋值
int main( ) {
    int n = 0; //定义用于记录循环次数的计数变量,初始值设为 0
    pthread_t a_thread[4] ; //定义线程描述符数组变量
    bin_sem = sem_open( sem_name, O_CREAT, 0644, 3) ; /* 主线程调用 sem_open( ) 函数创建有
                                                        名信号量 bin_sem,信号量初始值大
                                                        小设为 3 */
    if( bin_sem == SEM_FAILED) { /* 若创建有名信号量 bin_sem 失败,则调用 perror( ) 函数提示
                                    出错信息,并调用 sem_unlink( ) 函数删除该信号量,然后调
                                    用 exit( ) 函数退出程序 */
        perror( "unable to create semaphore") ;
        sem_unlink( sem_name) ;
        exit( -1) ;
    }
    while( n < 5) { //循环创建 5 个从线程
        n ++ ; //将循环次数计数变量加 1
        if( ( pthread_create( &a_thread[ n] , NULL, thread_function, NULL) ) ! =0) {
            /* 调用 pthread_create( ) 函数创建从线程,若失败则调用 perror( ) 函数提示出错信
               息,然后调用 exit( ) 函数退出程序 */
            perror( "Thread creation failed") ;
            exit( 1) ;
        }
        pthread_join( a_thread[ n] , NULL) ;
            //主线程调用 pthread_join( ) 函数等待子线程结束
    }
    sem_close( bin_sem) ; //主线程调用 sem_close( ) 函数释放该信号量
    sem_unlink( sem_name) ; //主线程调用 sem_unlink( ) 函数删除该信号量
}

void * thread_function( void * arg) { //从线程执行体函数
    sem_wait( &bin_sem) ; //阻塞从线程直到信号量 bin_sem 的值大于 0
    print( ) ; //从线程获得信号量并执行共享资源函数 print( )
    sleep( 1) ; //从线程调用 sleep( ) 函数休眠 1 秒,以等待其他线程执行
    sem_post( &bin_sem) ; //当执行任务完毕,从线程释放信号量
    printf( "I finished, my pid is %d\n" , pthread_self( ) ) ;//从线程输出完成提示信息
    pthread_exit( arg) ; //从线程正常结束并返回参数 arg 给主线程
}
```

```
void print( ) { //从线程的共享资源函数
    printf( "I get it,my tid is %d\n", pthread_self( ) );//从线程输出获得信号量的提示信息
    sem_getvalue( &bin_sem,&value ); //获取信号量的当前值
    printf( "Now the value have %d\n",value ); //从线程输出当前信号量的值
}
```

程序编译运行后得到的结果如下：

```
I get it,my tid is 1082330304
Now the value have 2
Iget it,my pid is 1894
Now the value have 1
Iget it,my pid is 1895
Now the value have 0
I'm finished,my pid is 1893
I'm finished,my pid is 1894
I'm finished,my pid is 1895
I get it,my pid is 1896
Now the value have 2
I get it,mypid is 1897
Now the value have 1
I'm finished,my pid is 1896
I'm finished,my pid is 1897
```

以下例程给出了一个应用有名信号量来限制访问共享代码的进程数目的实现方法。

该例程采用循环方法建立 5 个子进程，然后让它们利用有名信号量来限制访问同一段共享代码（用 print（）函数来代表）。

该例程的 C 语言源代码如下：

```
#include <pthread.h>   //提供了线程函数原型和数据结构的定义
#include <semaphore.h>   //提供了信号量函数原型和数据结构的定义
#include <sys/types.h>   //基本数据类型头文件,含有基本系统数据类型的定义
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror( )和 printf( )
                    等)的定义*/
#include <unistd.h> /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的
                     定义*/
void print(pid_t); //声明共享子函数 print( )
sem_t *bin_sem; //定义信号量变量
int val; //信号量的当前值
int main(int argc,char *argv[ ]){
    int n=0; //定义用于记录循环次数的计数变量,初始值设为 0
    if(argc!  =2){ //判断参数个数是不是等于 2,若不等于 2 则提示用户输入格式
        printf("please input a file name! \n"); //提示用户输入有名信号量的外部名字
        exit(1);
    }
```

```
    bin_sem = sem_open( argv[1], O_CREAT, 0644, 2); /* 主线程调用 sem_open( ) 函数创建有名
                                                       信号量 argv[1], 信号量初始值大小设
                                                       为 2 */
    while( n < 5) { //循环创建 5 个子进程,使它们同步运行
        n ++ ; //将循环次数计数变量加 1
        if( fork( ) == 0) { //调用 fork( ) 函数创建一个新子进程并在子进程中执行以下操作
            sem_wait( bin_sem) ; //申请信号量,若成功则将信号量的值减 1
            print( getpid( )) ; //子进程调用 print( ) 函数执行共享代码段
            sleep( 1) ; //子进程休眠 1 秒以便让其他进程执行
            sem_post( bin_sem) ; //子进程任务执行完毕,将信号量的值加 1
            printf( "I'm finished, my pid is %d\n", getpid( )) ;
                        //子进程提示结束信息
            return 0; //子进程执行完毕,返回 0
        }
    }
    wait( ) ; //父进程调用 wait( ) 函数等待所有子进程结束,以避免僵尸进程产生
    sem_close( bin_sem) ; //父进程调用 sem_close( ) 函数释放该信号量
    sem_unlink( argv[1]) ; //父进程调用 sem_unlink( ) 函数删除该信号量
    exit(0) ; //父进程调用 exit(0) 正常退出
}

void print( pid_t pid) { //子进程的共享代码段
    printf( "I get it, my pid is %d\n", pid) ; //显示子进程的进程号
    sem_getvalue( bin_sem, &val) ; //获得信号量的当前值
    printf( "Now the value have %d\n", val) ; //显示信号量的当前值
}
```

程序编译后运行得的结果如下:

```
I get it, my tid is 1082330304
Now the value have 1
I get it, my tid is 1090718784
Now the value have 0
I finished, my pid is 1082330304
I finished, my pid is 1090718784
I get it, my tid is 1099107264
Now the value have 1
I get it, my tid is 1116841120
Now the value have 0
I finished, my pid is 1099107264
I finished, my pid is 1116841120
I get it, my tid is 1125329600
Now the value have 1
I finished, my pid is 1125329600
```

注：互斥锁和信号量的区别，在于一个互斥锁只能用于对一个资源的互斥访问，它不能实现多个资源的多线程互斥问题，另外，它也无法限制访问者对资源的访问顺序，即互斥访问是无序的。但信号量可以实现多个同类资源的多线程互斥访问，且同时还能保障这些线程之间的同步，即可实现访问者对资源的有序访问。例如：信号量可以在生产者——消费者模式的程序中用于提供事件通知。在这种程序之中，消费者在消费资源（例如：队列中的数据）之前先试图获取信号量，而生产者一旦生产了资源就增加信号量的值，这里也可以理解为生产者把实例交给了信号量，而消费者则把实例从信号量中拿走了。通常，该类信号量的初值会被设置为 0，其值直到生产者生产了资源才会增加。

6.1.3 条件变量

条件变量是一种同步机制，允许线程挂起，直到共享数据上的某些条件得到满足。条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待“条件变量的条件成立”而挂起；另一个线程使“条件成立”并给出条件成立信号。条件变量的类型是 pthread_cond_t。为了防止竞争，条件变量一般需要和互斥锁结合起来使用。使用条件变量的基本过程如下。

声明一个 pthread_cond_t 条件变量，并调用 pthread_cond_init( ) 函数对其进行初始化。

声明一个 pthread_mutex_t 互斥锁变量，并调用 pthread_mutex_init( ) 函数对其进行初始化。

调用 pthread_cond_signal( ) 函数发出信号。如果此时有线程在等待该信号，那么该线程将会唤醒。如果没有，则该信号就会被忽略。如果想唤醒所有等待该信号的线程，则调用 pthread_cond_broadcast( ) 函数。

调用 pthread_cond_wait( )/pthread_cond_timedwait( ) 等待信号。如果没有信号，线程将会阻塞，直到有信号。该函数的第一个参数是条件变量，第二个参数是一个 mutex。在调用该函数之前必须先获得互斥量。如果线程阻塞，互斥量将立刻会被释放。

调用 pthread_cond_destroy( ) 销毁条件变量，释放其所占用的资源。

在上述条件变量的操作步骤中，各相关函数及其调用方法如下。

1. pthread_cond_init( ) 函数

使用条件变量之前要先进行初始化。其中，对于静态分配的条件变量，可以把它设置为 PTHREAD_COND_INITIALIZER，也可调用 pthread_cond_init( ) 对条件变量进行动态分配。pthread_cond_init( ) 函数若调用成功将返回 0，若出错则返回错误编号，其函数原型如下：

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_cond_init(pthread_cond_t * cond, const pthread_condattr_t * attr);
```

在上述 pthread_cond_init( ) 函数的原型中，各参数的含义如下。

cond：指向要初始化的条件变量的指针。

attr：条件变量属性。尽管 POSIX 标准中为条件变量定义了属性，但在 LinuxThreads 中没有实现，因此 cond_attr 值通常为 NULL，且被忽略。

2. pthread_cond_signal( ) 函数

调用 pthread_cond_signal( ) 函数将会激活在该条件变量上阻塞的所有线程之中的一个。如果有多个线程同时被阻塞在该条件变量上，则由调度策略确定具体哪个线程会被激活；如果没有在该条件变量上等待的线程，则什么也不做。函数若调用成功将返回 0，若出错则返回错误编号。pthread_cond_signal( ) 函数的原型如下：

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_cond_signal(pthread_cond_t *cond);
```

在上述 pthread_cond_signal( ) 函数的原型中，其参数的含义如下。

cond：指向条件变量的指针。

注：*调用 pthread_cond_signal( ) 函数时需要利用互斥锁来进行保护。*

3. pthread_cond_broadcast( ) 函数

调用 pthread_cond_broadcast( ) 函数将会激活在该条件变量上阻塞的所有线程；如果没有在该条件变量上等待的线程，则什么也不做。函数若调用成功将返回 0，若出错则返回错误编号。pthread_cond_broadcast( ) 函数的原型如下：

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_cond_broadcast(pthread_cond_t *cond);
```

在上述 pthread_cond_broadcast( ) 函数的原型中，其参数的含义如下。

cond：指向条件变量的指针。

注：调用 pthread_cond_broadcast( ) 函数时需要利用互斥锁来进行保护。

4. pthread_cond_wait( )/pthread_cond_timedwait( ) 函数

等待条件成立有两种方式：无条件等待 pthread_cond_wait( ) 和计时等待 pthread_cond_timedwait( )，其中，在计时等待方式下若在给定时刻前条件没有满足，则返回 ETIMEOUT 并结束等待。pthread_cond_wait( )/pthread_cond_timedwait( ) 函数若调用成功将返回 0，若出错则返回错误编号。其函数原型如下：

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *
abstime);
```

在上述 pthread_cond_wait( ) 和 pthread_cond_timedwait( ) 函数的原型中，各参数的含义如下。

cond：指向条件变量的指针。

mutex：指向互斥锁的指针。

abstime：超时时间。abstime 以与 time( ) 系统调用相同意义的绝对时间形式出现，其中，0 表示格林尼治时间 1970 年 1 月 1 日 0 时 0 分 0 秒。

注：*无论哪种等待方式，都必须和一个互斥锁配合。*

5. pthread_cond_destroy( ) 函数

调用 pthread_cond_destroy( ) 函数将会销毁指定的条件变量；pthread_cond_destroy( ) 函数

若调用成功将返回 0，若出错则返回错误编号。pthread_cond_destroy( ) 函数的原型如下：

```
#include <pthread.h> //提供了线程函数原型和数据结构的定义
int pthread_cond_destroy(pthread_cond_t *cond);
```

在上述 pthread_cond_destroy( ) 函数的原型中，其参数的含义如下。

cond：指向要销毁的条件变量的指针。

以下给出一个说明条件变量用法的简单例程。

该例程使用条件变量实现两个从线程之间的同步，其中从线程 t_b 负责打印 9 以内 3 的倍数，从线程 t_a 则负责打印其他的数，程序开始时，从线程 t_b 不满足条件等待，从线程 t_a 运行使 i 循环加 1 并打印。直到 i 为 3 的倍数时，从线程 t_a 发送信号通知从线程 t_b，这时，从线程 t_b 将满足条件并开始打印 i 的值。

该例程的 C 语言源代码如下：

```
#include <pthread.h>   //提供了线程函数原型和数据结构的定义
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror( ) 和 print( )f
                     等)的定义*/
#include <stdlib.h> /*C 标准库头文件,包含了 C 语言标准库函数(如 exit( )、atoi( ) 等)原型的
                      定义*/
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; //初始化互斥锁
pthread_cond_t cond = PTHREAD_COND_INITIALIZER; //初始化条件变量
void *thread1(void *); //声明线程 t_a 的线程体函数
void *thread2(void *); //声明线程 t_b 的线程体函数
int i = 1; //定义用于记录循环次数的计数变量,初始值设为 0
int main(void){
    pthread_t t_a; //定义线程标识符 t_a
    pthread_t t_b; //定义线程标识符 t_b
    pthread_create(&t_a,NULL,thread1,(void *)NULL); //创建从线程 t_a
    pthread_create(&t_b,NULL,thread2,(void *)NULL); //创建从线程 t_b
    pthread_join(t_a, NULL);//等待从线程 t_a 结束
    pthread_join(t_b, NULL);//等待从线程 t_b 结束
    pthread_mutex_destroy(&mutex); //释放互斥锁
    pthread_cond_destroy(&cond); //释放条件变量
    exit(0); //程序正常退出
}

void *thread1(void *junk){   //线程 t_a 的线程体函数
    for(i=1;i<=9;i++){
        pthread_mutex_lock(&mutex); //在等待条件成立之前,须先上锁
        if(i%3==0)//设置等待的条件,当 i 为 3 的倍数时执行以下语句
            pthread_cond_signal(&cond); /*若条件成立则发送信号,以激活某个在该条件变量
                                          上阻塞的线程*/
        else //当 i 不为 3 的倍数时执行以下语句
            printf("thead1:%d\n",i); //线程 t_a 执行打印任务
```

```
            pthread_mutex_unlock(&mutex); //在执行完任务之后,须解锁
            sleep(1); //线程 t_b 休眠 1 秒,以便让其他线程执行
        }
    }

    void * thread2(void * junk){    //线程 t_b 的线程体函数
        while(i<9){
            pthread_mutex_lock(&mutex); //在等待条件成立之前,须先上锁
            if(i%3! =0)//设置等待的条件,当 i 不为 3 的倍数时执行以下语句
                pthread_cond_wait(&cond,&mutex); //等待上述条件成立
            printf("thread2:%d\n",i); /* 当条件成立则执行打印任务 */
            pthread_mutex_unlock(&mutex); /* 在执行完打印任务之后需解锁 */
            sleep(1);
        }
    }
```

例程的运行结果如下：

```
thread1:1
thread1:2
thread2:3
thread1:4
thread1:5
thread2:6
thread1:7
thread1:8
thread2:9
```

6.2　基于多线程并发的面向连接服务器软件的设计流程

并发服务器面向不定长时间才能处理完的请求，对每个请求由服务器的线程处理。线程被用来建立请求驱动的服务程序，每个客户一个线程，多个线程可以并发执行。线程使得服务器的进程数目并不随客户数目的增加而线性增加，这样减少了服务器进程的压力，降低了开销。并发服务器的结构如图 6.1 所示，在某一个时刻，由同一服务器进程所产生的多个并发线程同时对多个客户的并发请求进行处理，从而解决了并发请求问题。各个线程既能独立操作，又可以协同作业，实现了简单而高效的服务器结构。

并发服务器的工作流程按照以下三个步骤建立。

① 建立套接字并在某一约定端口上等待接收客户请求。

② 当接收到来自客户端的服务请求后：建立一新线程来处理，同时主线程继续等待其他客户连接。当新线程处理完成后，关闭新线程与客户的通信链路并终止新线程。

③ 关闭服务器。采用面向连接方式的多线程并发服务器和客户端的算法流程如图 6.2

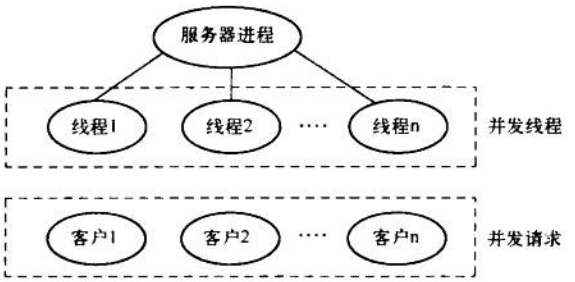


图 6.1 基于多线程的并发服务器结构

所示，多线程并发服务器模型如图 6.3 所示。

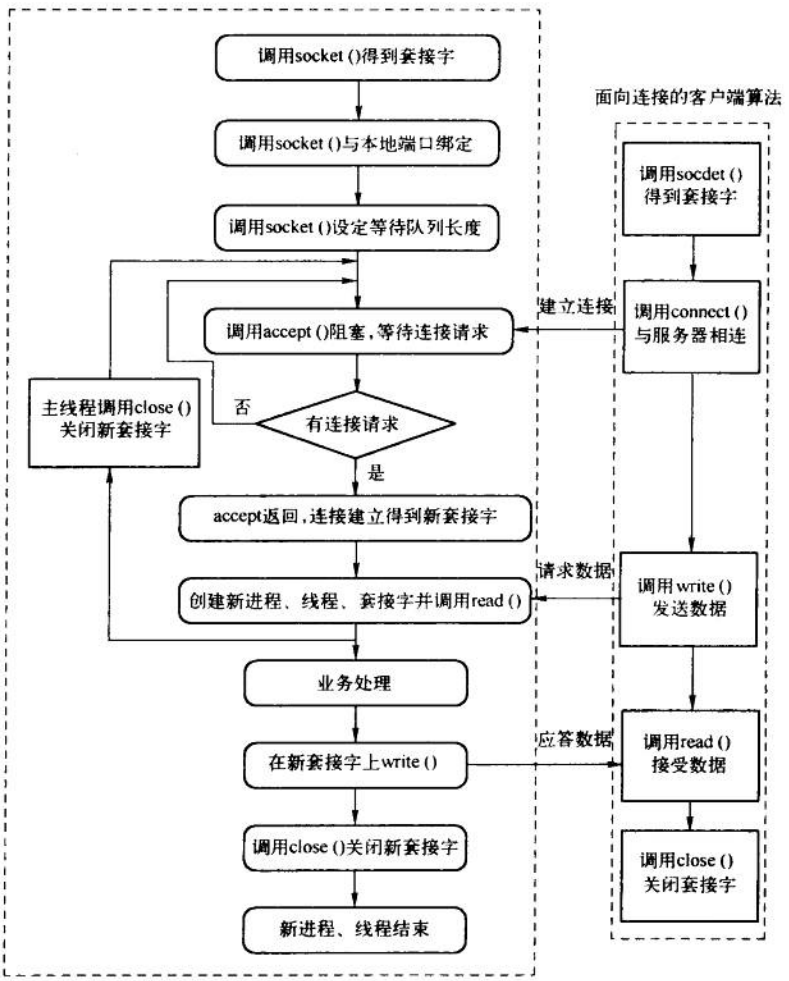


图 6.2 采用面向连接方式的多线程并发服务器和客户端的算法流程

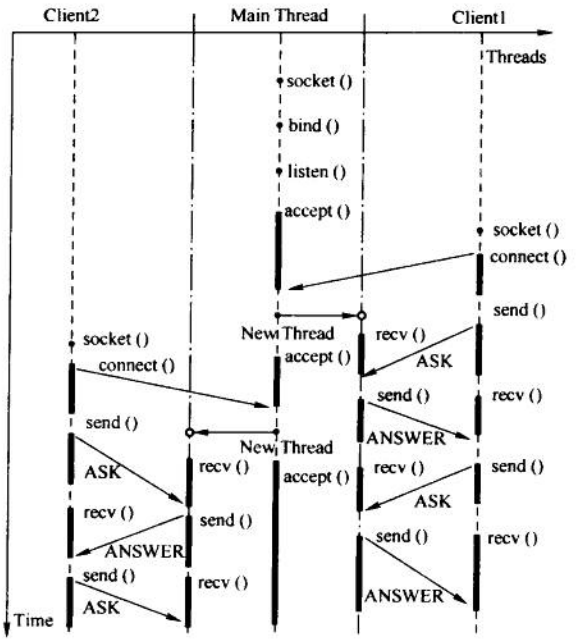


图 6.3　多线程并发服务器模型

6.3　基于多线程并发的面向连接服务器例程

该例程为一个基于多线程并发的面向连接服务器 C 语言例程，其中，主线程负责接收来自客户的连接请求，并针对每个来自不同客户的连接请求创建一个新的从线程来负责处理该连接；而在每个从线程中，从线程首先调用 recv( ) 函数接收来自客户端的名字信息，然后再调用 recv( ) 函数接收来自客户端的数据信息；然后，再将接收到的客户端数据进行反转；最后，再调用 send( ) 函数将反转后的字符串回送给客户端。

该例程的 C 语言源代码如下：

```
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror( ) 和 printf( ) 等)的定义*/
#include <string.h> //提供了字符串函数原型的定义
#include <unistd.h> /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的定义*/
#include <sys/types.h> //基本数据类型头文件,含有基本系统数据类型的定义
#include <sys/socket.h> //提供了套接字函数原型与数据结构的定义
#include <netinet/in.h> //包含了数据结构 sockaddr_in 的定义
#include <arpa/inet.h> //包含了 IP 地址转换函数原型的定义
#include <pthread.h> //提供了线程函数原型与数据结构的定义
#include <stdlib.h> /*C 标准库头文件,包含了 C 语言标准库函数(如 exit( )、atoi( ) 等)原型的定义*/
```

```
#define PORT 1234 //定义服务器端口
#define BACKLOG 5 //定义 listen 队列等待的连接数
#define MAXSIZE 1024 //定义缓冲区大小
void process_cli(int connectfd, struct sockaddr_in client);//声明客户端请求处理函数
void * start_routine(void * arg); //声明线程体函数
typedef struct ARG {//定义用于存储客户端信息的结构体
    int connfd; //用于记录处理该客户连接的套接字描述符
    struct sockaddr_in client; //用于记录客户的端点地址
}ARG;
main(){
    int listenfd, connectfd; //定义主、从套接字描述符变量
    pthread_t thread; //定义线程描述符变量
    ARG * arg; //定义用于存储客户端信息的结构体变量
    struct sockaddr_in server; //定义服务器端点地址结构体变量
    struct sockaddr_in client; //定义客户端点地址结构体变量
    int sin_size; //定义端点地址结构体长度变量
    if((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1){
                /*调用 socket()函数创建用于监听客户连接请求到达的主套接字,若失败则调
                  用 perror()函数提示出错信息,然后调用 exit()函数出错退出程序*/
        perror("Creating socket failed.");
        exit(1);
    }
    int opt = SO_REUSEADDR; //定义 socket 属性变量
    setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
                                        /*设置 socket 属性,端口可以重用*/
    //以下代码段用于初始化服务器地址结构体变量
    bzero(&server,sizeof(server)); //清空服务器端点地址结构体变量
    server.sin_family = AF_INET; //为服务器端点地址结构体变量赋值协议族
    server.sin_port = htons(PORT); //为服务器端点地址结构体变量赋值端口号
    server.sin_addr.s_addr = htonl(INADDR_ANY);
            //为服务器端点地址结构体变量赋值 IP 地址
    if(bind(listenfd,(struct sockaddr *)&server,sizeof(struct sockaddr)) == -1){
        /*调用 bind()函数绑定服务器端点地址,若失败则调用 perror()函数提示出错信息,
          然后调用 exit()函数出错退出程序*/
        perror("Bind error.");
        exit(1);
    }
    if(listen(listenfd,BACKLOG) == -1){ /*调用 listen()函数设置队列长度并开始监听客户连
                                          接请求的到达,若失败则调用 perror()函数提示出
                                          错信息,然后调用 exit()函数出错退出程序*/
        perror("listen()error\n");
        exit(1);
    }
```

```
    sin_size = sizeof( struct sockaddr_in ) ; //获得端点地址结构体的长度
    while( 1 ) {
        if( ( connectfd = accept( listenfd, ( struct sockaddr * )&client, ( socklen_t * )& sin_size ) ) == -1 ) {
            /* 调用 accept( ) 函数建立与客户端的连接并返回表示该连接的从套接字描述符,
               若失败则调用 perror( ) 函数提示出错信息,然后调用 exit( ) 函数出错退出程
               序 */
            perror( "accept( ) error\n" ) ;
            exit( 1 ) ;
        }
        arg = new ARG; //初始化用于存储客户端信息的结构体变量 arg
        arg - > connfd = connectfd; //对用于存储客户端信息的结构体变量 arg 赋值
        memcpy( &arg - > client, &client, sizeof( client ) ) ;
                        //对用于存储客户端信息的结构体变量 arg 赋值
        if( pthread_create( &thread, NULL, start_routine, ( void * ) arg ) ) {
                /* 以客户端连接为参数,start_routine 为线程体函数来创建新线程,若失败则调
                   用 perror( ) 函数提示出错信息,然后调用 exit( ) 函数出错退出程序 */
            perror( "Pthread_create( ) error" ) ;
            exit( 1 ) ;
        }
    }
    close( listenfd ) ; //关闭主套接字
}

void * start_routine( void * arg ) { //线程体函数
    ARG * info; //定义用于存储客户端信息的结构体指针变量
    info = ( ARG * ) arg; //给结构体指针变量 info 赋值
      process_cli( info - > connfd, info - > client ) ; /* 在线程体函数中调用 process_cli( ) 函数来
                                                          执行具体任务 */
    delete info; //删除结构体指针变量,释放其所占用的资源
    pthread_exit( NULL ) ; //从线程退出
}

void process_cli( int connectfd, sockaddr_in client ) {
    int num; //定义用于记录 recv( ) 函数的返回值的变量
    char recvbuf[ MAXSIZE ], sendbuf[ MAXSIZE ], cli_name[ MAXSIZE ] ;
          /* 定义三个缓存区变量,分别用于缓存接收到的数据、要发送的数据和客户端的名
             字信息 */
    printf( "You got a connection from %s. ", inet_ntoa( client. sin_addr ) ) ; //输出提示信息
    num = recv( connectfd, cli_name, MAXDATASIZE, 0 ) ; /* 调用 recv( ) 函数接收来自客户端的
                                                           名字信息 */
    if( num == 0 ) { /* 若没有从客户端接收到名字信息,则调用 close( ) 函数关闭对应的套接字,
                        终止与该客户端之间的连接 */
        close( connectfd ) ;
```

```
            printf("Client disconnected. \n");
            return;
        }
        //若从客户端接收到了名字信息,则执行以下语句
        cli_name[num - 1] ='\0'; //在 cli_name 字符串末尾加上字符串结束符'\0'
        printf("Client's name is %s. \n",cli_name); //显示客户端的名字信息
        while(num = recv(connectfd, recvbuf, MAXDATASIZE,0)){ /* 调用 recv()函数接收来自客户
                                                              端的其他数据 */
            recvbuf[num - 1] ='\0'; //在 recvbuf 字符串末尾加上字符串结束符'\0'
            printf("Received client( %s )message: %s",cli_name, recvbuf);
                                                    //显示收到的客户端的数据
            for(int i = 0; i < num - 1; i++){ /* 将 recvbuf 中的字符串反转存入到 sendbuf 之中 */
                sendbuf[i] = recvbuf[num - i - 2];
            }
            sendbuf[num - 1] ='\0'; //在 sendbuf 字符串末尾加上字符串结束符'\0'
            send(connectfd,sendbuf,strlen(sendbuf),0); /* 调用 send()函数将反转后的字符串回送
                                                    给客户端 */
        }
        close(connectfd); /* 任务结束,从线程调用 close()函数关闭对应的套接字,终止与该客户端
                            之间的连接 */
    }
```

6.4 本章小结

本章主要对基于多线程并发的面向连接服务器及其实现方法进行深入介绍，并在实现原理介绍的基础上，具体给出一个创建基于多线程并发的面向连接服务器的完整 C 语言例程。通过本章的学习，需要了解基于多线程并发的面向连接服务器的进程结构；需要熟悉基于多线程并发的面向连接服务器的设计流程；需要掌握基于多线程并发的面向连接服务器的 C 语言实现方法。

本章习题

1. 简述基于多线程并发的面向连接服务器的进程结构。
2. 简述基于多线程并发的面向连接服务器软件的设计流程。
3. 试构造一个基于多线程并发的面向连接服务器例程，该例程能实现以下功能：能同时等候来自 10 个不同客户的连接请求，一旦与某个客户连接成功则接收来自该客户的信息，当每收到一个字符串时将首先显示该字符串，然后再将该字符串反转、最后再将反转后的字符串回送给该客户。

第 7 章　基于单线程并发的面向连接服务器例程剖析

第 6 章介绍了基于多线程并发的面向连接服务器及其实现方法，本章将进一步针对另一种并发服务器——基于单线程并发的面向连接服务器及其实现方法进行深入介绍，并在实现原理介绍的基础上，具体给出两个创建基于单线程并发的面向连接服务器的完整 C 语言例程。

7.1　单线程并发服务器的线程结构

采用多线程编程的目的是为了最大限度地利用 CPU 资源，当某一线程的处理不需要占用 CPU 而只和 I/O 等资源打交道时，采用基于多线程并发模式就可以让需要占用 CPU 资源的其他线程有机会获得 CPU 资源，从而提高了 CPU 资源的利用效率。

每个程序执行时都会产生一个进程，而每一个进程至少要有一个主线程。该线程其实是进程执行的一条线索，除了主线程外，程序员还可以给进程增加其他的线程，即程序员可增加进程其他的执行线索，由此在某种程度上可以看成是给一个应用程序增加了多任务功能。当应用程序运行后就可以根据各种条件挂起或运行这些线程，尤其在多 CPU 环境中，这些线程是并发运行的。

多进程技术也可以实现应用程序的多任务功能，但由于存在创建进程的高消耗（每个进程都有独立的数据和代码空间）、进程之间通信不便（消息机制），以及进程之间切换的时间太长等不利因素，从而导致了多线程的提出。对于单 CPU 来说，由于在同一时间只能执行一个线程，所以如果想实现多任务，就只能按照某种策略让每个进程或线程获得一个时间片，而在任意一个时间片内只能有一个线程执行。由于时间片很短，这样给用户的感觉是同时有多个线程在执行。但是线程的切换是有代价的，因此如果采用多进程，那么就需要将线程所隶属的该进程所需要的内存进行切换，从而导致时间代价很高。而线程之间由于可以共享内存，因此采用多线程模型在切换上花费要比采用多进程模型少很多。但是，线程切换还是需要时间消耗的，所以采用一个拥有两个线程的进程执行所需要的时间比一个线程的进程执行两次所需要的时间要多一些。即，采用多线程不会提高反而会降低程序的执行速度，但是对于用户来说，可以减少用户的响应时间。

为此，在写服务器处理模型的程序时，除了上述多进程模型（服务器每收到一个客户请求，就创建一个新的进程来处理该请求）与多线程模型（服务器每收到一个客户请求，就创建一个新的线程来处理该请求）之外，人们针对单 CPU 环境提出了第三种模型，称为 SELECT 事件驱动模型。在该模型中，服务器每收到一个客户请求，就将其放入一个事件列表，然后让主线程通过非阻塞 I/O 方式来处理该客户请求。显然，在上述 SELECT 事件驱动模型中采用的是一种单线程的并发模型，其线程结构如图 7.1 所示。

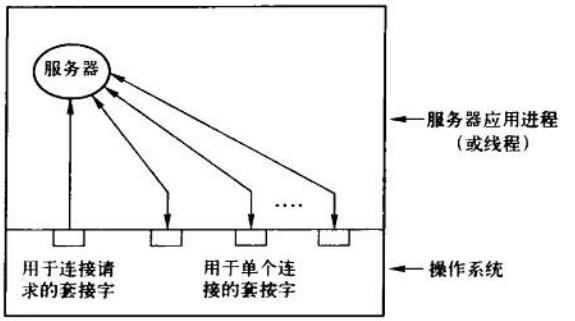


图 7.1 单线程并发服务器模型中的线程结构

由以上图 7.1 可知，单线程服务器中的单个线程需要同时完成多线程服务器中的主线程和从线程的职责。首先，它需要维护一组套接字，其中，组中有一个主套接字用于绑定到主线程需要接受客户连接请求的熟知端口上，而组中的其他每一个从套接字则都对应于一个连接；然后，当主套接字准备就绪时，服务器就接受一个新的客户连接，而当其他任意一个从套接字准备就绪时，服务器就读取一个请求并发送响应。

显然，与其他模型相比，由于 SELECT 事件驱动模型只使用了单线程（进程）执行，因此其占用的资源少，不用消耗太多的 CPU 资源，同时能够为多客户端提供服务。如果试图建立一个简单的事件驱动的服务器程序，该模型具有一定的参考价值。但该模型也有严重的缺陷，例如，如图 7.2 所示，由于该模型将事件探测和事件响应夹杂在一起，因此，一旦事件响应的执行体过于庞大，则将对整个模型造成灾难性的后果。

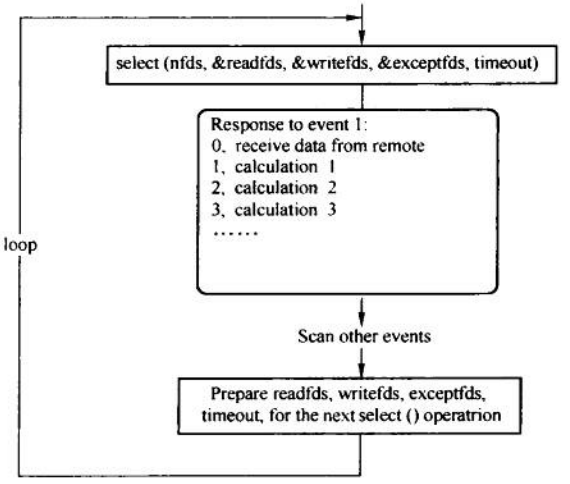


图 7.2 SELECT 事件驱动模型的缺陷

7.2 单线程并发服务器程序设计流程

如图 7.3 所示，单线程并发服务器模型的程序设计流程如下。

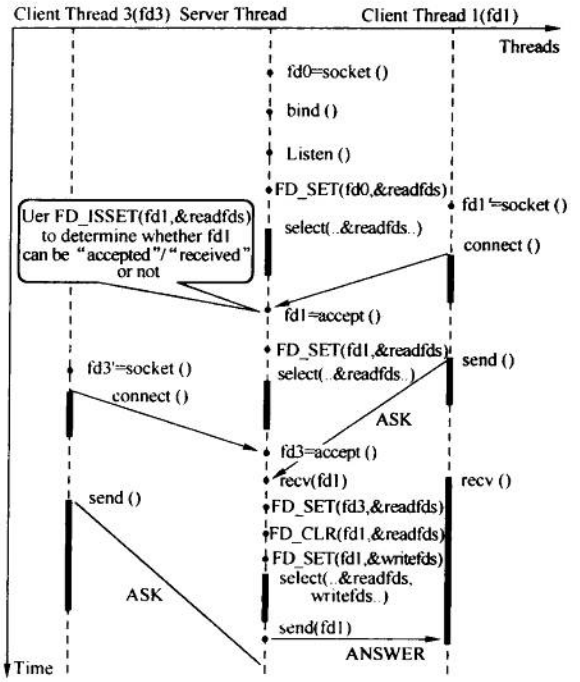


图 7.3　单线程并发服务器模型

步骤 1：创建套接字并将其绑定到这个服务的熟知端口上，将该套接字加到一个表中，该表中的项是可以进行 I/O 的描述符。

步骤 2：使用 select( ) 在已有的套接字上等待 I/O。

步骤 3：如果最初的套接字准备就绪，使用 accept( ) 获得下一个连接，并将这个新套接字加入到表中，该表中的项是可以进行 I/O 的描述符。

步骤 4：如果是最初的套接字以外的某些套接字准备就绪，就是用 recv 或 read 获得下一个请求，构造响应，用 send( ) 或 write( ) 将响应发回给客户。

步骤 5：继续按以上的步骤 2 进行处理。

在上述单线程并发服务器模型中，各函数的定义分别如下。

1. select( ) 函数

select( ) 函数提供异步 I/O，允许单进程等待指明文件描述符集合中的任一描述符最先就绪，可以使进程检测同时等待的多个 I/O 设备，当没有设备准备就绪时，select( ) 函数将阻塞，若其中有任何一个设备准备就绪时，select( ) 函数就会返回。select( ) 函数的返回值有以下四种情况：正常情况下返回就绪的文件描述符个数；经过了 timeout 时长后仍无设备准备好，将返回值为 0；如果 select 被某个信号中断，它将返回 -1；如果出错，则返回 -1。

select( ) 函数的原型如下：

```
#include <pthread.h> //提供了线程函数原型与数据结构的定义
int select(int maxfdp, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout);
```

在上述 select( ) 函数的原型中，各参数的含义如下。

maxfdp：是指集合中所有文件描述符的范围，即所有文件描述符的最大值加 1。该参数通常可通过调用系统函数 getdtablesize( ) 来进行设置。

readfds：是指向 fd_set 结构的指针，其中，fd_set 结构可以理解为一个集合，该集合用于存放可读文件描述符（File Descriptor）集合，该集合中包括普通意义的文件描述符与 socket 描述符。如果该集合中有一个文件可读，select( ) 函数就会返回一个大于 0 的值，表示有文件可读，如果没有可读的文件，则 select( ) 函数会根据 timeout 参数再判断是否超时，若超出 timeout 的时间，select( ) 函数将返回 0，若发生错误则返回负值。该参数也可以传入 NULL 值，此时，表示 select( ) 函数不关心任何文件的读变化。

fd_set 结构可通过以下宏来进行操作。

```
FD_ZERO(&fdset); /* 将 fdset 清零,以清空 fdset 与所有文件句柄之间的联系 */
FD_SET(fd, &fdset); /* 将 fd 加入 fdset,以建立文件句柄 fd 与 fdset 之间的联系 */
FD_CLR(fd, &fdset); /* 将 fd 从 fdset 中清除,以清除文件句柄 fd 与 fdset 之间的联系 */
FD_ISSET(fd, &fdset); /* 检查 fdset 联系的文件句柄 fd 是否可读/写,当 >0 则表示文件句柄 fd
                         可读/写 */
```

writefds：是指向 fd_set 结构的指针，其中，fd_set 结构可以理解为一个集合，该集合用于存放可写文件描述符（File Descriptor）集合，如果该集合中有一个文件可写，则 select( ) 函数就会返回一个大于 0 的值，表示有文件可写，如果没有可写的文件，则 select( ) 函数会根据 timeout 参数再判断是否超时，若超出 timeout 的时间，select( ) 函数将返回 0，若发生错误则返回负值。该参数也可以传入 NULL 值，此时，表示 select( ) 函数不关心任何文件的写变化。

errorfds：同上面两个参数的意图，用来监视文件错误异常。

timeout：是指向 timeval 结构的表示 select( ) 函数的超时时间指针，该参数可以使 select( ) 函数处于三种状态：

- 若将 NULL 以形参传入，即不传入时间结构，将使得 select( ) 函数置于阻塞状态，此时，select( ) 函数将一直等到监视的文件描述符集合中有某个文件描述符发生变化为止；
- 若将时间值设为 0 秒 0 毫秒，则将使得 select( ) 函数变成一个纯粹的非阻塞函数，此时，select( ) 函数将不管监视的文件描述符集合中是否有文件描述符发生变化，都会立刻返回继续执行，若文件描述符无变化时将返回 0，而有文件描述符发生变化时则返回一个正值；
- 若 timeout 的值大于 0，就是一般的定时器，此时，select( ) 函数将在 timeout 时间内阻塞，若在 timeout 设定的超时时间之内有事件到来则 select( ) 函数将立即返回，否则 select( ) 函数将在超时后返回，返回值同上所述。

timeval 结构的定义如下：

```
struct timeval{
    long tv_sec; //seconds
    long tv_usec; //microseconds
};
```

例如，可通过如下语句来设置 select( ) 函数的超时时间：

```
struct timeval tv; //申明一个 timeval 结构的时间变量来保存时间
tv. tv_sec = 1;
tv. tv_usec = 500; //设置 select( ) 函数等待的最大时间为 1 秒加 500 微秒
```

7.3　基于单线程并发的面向连接服务器例程

1. 基于单线程的并发 ECHO 服务器例程

```
#include <sys/types. h> /* 基本数据类型头文件,含有基本系统数据类型的定义 */
#include <sys/socket. h> //提供了套接字函数原型与数据结构的定义
#include <sys/time. h> //提供了 time 函数原型的定义
#include <netinet/in. h> //包含了数据结构 sockaddr_in 的定义
#include <errno. h> //提供错误号 errno 的定义,用于错误处理
#include <unistd. h> //LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的
                     定义
#include <string. h> //提供了字符串函数原型的定义
#include <stdio. h> /* 标准输入/输出头文件,包含了标准输入/输出函数(如 perror( ) 和 printf( )
                     等)的定义 */
#define QLEN 32
#define BUFSIZE 4096
extern int errno;
int errexit(const char *format, ...);
int passiveTCP(const char *service, int qlen);
int echo(int fd);
int main(int argc, char *argv[]) {
    char *service = "echo";
    struct sockaddr_in fsin;
    int msock; //主套接字描述符
    fd_set rfds; //可读文件描述符集合
    fd_set afds; //活动文件描述符集合,用于保存所有的文件描述符
    unsigned int alen;
    int fd, nfds;
    switch (argc) {
    case 1:
        break;
    case 2:
        service = argv[1];
        break;
    default:
        errexit("usage: TCPmechod [port]\n");
    }
```

```
        msock = passiveTCP( service, QLEN);
        nfds = getdtablesize( );//调用 getdtablesize( )来决定描述符的最大个数
        FD_ZERO(&afds); //初始化活动文件描述符集合
        FD_SET(msock, &afds); //将 msock 套接字加入到活动文件描述符集合之中
        while (1) {
            memcpy(&rfds, &afds, sizeof(rfds)); //将 afds 中的套接字描述符加入到 rfds
            if(select(nfds,&rfds,(fd_set *)0, (fd_set *)0,(struct timeval *)0) <0)
                                    //确定 rfds 中哪个套接字已经就绪
                errexit("select: %s\n", strerror(errno));
/*以下代码段用于测试 msock,如果主描述符在某个套接字执行程序的情况中突然出现问题,那
么之后的代码应该不得到执行,即不该接受新的连接分配新套接字*/
            if(FD_ISSET(msock, &rfds)) { //若 msock 套接字已经就绪
                int ssock;
                alen = sizeof(fsin);
                ssock = accept(msock, (struct sockaddr *)&fsin,&alen);
                /*则调用 accept( )函数来建立与客户端的连接并创建从套接字 ssock 用于负责处
                  理与该客户端的交互*/
                if(ssock <0)
                    errexit("accept: %s\n",strerror(errno));
                FD_SET(ssock, &afds); //然后, 将 ssock 加入到 afds 中
            }
            for(fd =0; fd < nfds; ++fd) { //循环判断哪些从套接字已经就绪
                if(fd! = msock && FD_ISSET(fd, &rfds)) {
                    if(echo(fd) ==0) { //调用 echo( )函数执行与客户端的交互
                        (void) close(fd);
                        FD_CLR(fd, &afds); //交互完毕,将 fd 从 afds 中清除
                    }
                }
            }
        }
    }
```

在上述例程中，首先只创建了单个线程，该单个线程负责将文件描述符加入到文件描述符集中，事实上，select( )函数会在所有可能的套接字上等待 I/O，也同时等待新连接。那么显然只要判断哪个套接字准备就绪就可以调用 echo 传输数据了。例如，假设有 10 个请求进来，那么 accept( )函数会按照这 10 个请求到来的先后顺序依次分配套接字到文件描述符中，但是之后的执行顺序就不得而知了。但是可以确定的是，在文件描述符集合中的任何一个套接字都会进入一个 FOR 循环之中，该 FOR 循环每次都从头测试文件描述符集合的套接字看是否有某个套接字已准备就绪，如果有某个套接字已准备就绪，那么就基于该套接字来传输数据。另外，该 FOR 循环也同时测试传输之后的套接字是否已经不再传输数据了，如果之前的套接字在另一个 FOR 循环中的传输数据量为 0，那么就代表之前的那个套接字已经完成数据传输，于是就关闭之前的那个套接字，并将其从文件描述符中清除。这就是并发设

计的一个想法，称为“异步关闭”。

```
Int echo(int fd) {
    char buf[BUFSIZ]; //定义缓存区数组 buf[]
    int cc;
    cc = read(fd, buf, sizeof buf); //调用 read()函数从套接字 fd 中接收数据
    if(cc <0)
        errexit("echo read: %s\n", strerror(errno));
    if(cc && write(fd, buf, cc) <0) //调用 write()函数将 buf 中的数据写入到套接字 fd
        errexit("echo write: %s\n", strerror(errno));
    return cc;
}
```

2. 基于单线程的并发服务器例程

```
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf()
                    等)的定义*/
#include <strings.h> //提供了字符串函数原型的定义
#include <unistd.h> /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构
                     的定义*/
#include <sys/types.h> /*基本数据类型头文件,含有基本系统数据类型的定义*/
#include <sys/socket.h> //提供了套接字函数原型与数据结构的定义
#include <netinet/in.h> //包含了数据结构 sockaddr_in 的定义
#include <arpa/inet.h> //包含了 IP 地址转换函数原型的定义
#include <sys/time.h> //提供了 time 函数原型的定义
#include <stdlib.h> /*C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()等)原型的
                     定义*/
#define PORT 1234 //定义服务器的端口号
#define BACKLOG 5 //定义允许排队的连接数
#define MAXDATASIZE 1000 //定义缓存区的大小
typedef struct CLIENT{ //定义用于保存客户端相关信息的数据结构
    int fd; //套接字 ID
    char * name; //客户端名字
    struct sockaddr_in addr; //客户端的端点地址
    char * data; //接收到的来自客户端的数据
};
void process_cli(CLIENT *client, char * recvbuf, int len); /*声明用于负责与客户端进行交互的
                                                            process_cli()函数*/
void savedata(char * recvbuf, int len, char * data); //声明 savedata()函数

main() {
    int i, maxi, maxfd,sockfd;
    int nready;
    ssize_t n;
```

```
fd_set rset, allset; //定义文件描述符集合变量 rset, allset
int listenfd, connectfd;
struct sockaddr_in server;
CLIENT client[FD_SETSIZE];
char recvbuf[MAXDATASIZE]; //定义缓存区数组 recvbuf[]
int sin_size;
if((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) { //调用 socket()创建主套接
                                                          字 listenfd */
    perror("Creating socket failed.");
    exit(1);
}
int opt = SO_REUSEADDR;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
                                                          //设置套接字属性
bzero(&server,sizeof(server));
server.sin_family = AF_INET;
server.sin_port = htons(PORT);
server.sin_addr.s_addr = htonl (INADDR_ANY);
if(bind(listenfd,(struct sockaddr *)&server, sizeof(struct sockaddr)) == -1){
                                        //调用 bind()函数将套接字与服务器端点地址绑定
    perror("Bind error.");
    exit(1);
}
if(listen(listenfd,BACKLOG) == -1){ /* 调用 listen()函数使得主动连接套接口变为被连
                                       接套接口,使得一个进程可以接受其他进程的请
                                       求,从而成为一个服务器进程 */
    perror("listen() error\n");
    exit(1);
}
sin_size = sizeof(struct sockaddr_in);
maxfd = listenfd;
maxi = -1;
for (i=0; i<FD_SETSIZE; i++) {
    client[i].fd = -1;
}
FD_ZERO(&allset); //初始化文件描述符集合 allset
FD_SET(listenfd, &allset); //将套接字描述符 listenfd 加入到文件描述符集合 allset
while(1) {
    struct sockaddr_in addr;
    rset = allset;
    nready = select(maxfd+1, &rset, NULL, NULL, NULL); /* 调用 select()函数判断 rset
                                                          中哪个套接字描述已经准
                                                          备就绪 */
```

```
            if(FD_ISSET(listenfd, &rset)) { //若套接字 listenfd 准备就绪
                if((connectfd = accept(listenfd, (struct sockaddr *)&addr, &sin_size)) == -1) {
                                        /* 调用 accept()函数建立与客户端之间的连接并创建
                                           从套接字 connectfd 负责处理与该客户端的交互 */
                    perror("accept() error\n");
                    continue;
                }
            }
            for (i=0;i<FD_SETSIZE;i++) { /* 将从套接字描述符 connectfd 写入到数组 client[]的末尾 */
                if(client[i].fd<0) {
                    client[i].fd = connectfd;
                    client[i].name = new char[MAXDATASIZE];
                    client[i].addr = addr;
                    client[i].data = new char[MAXDATASIZE];
                    client[i].name[0] = '\0';
                    client[i].data[0] = '\0';
                    printf("You got a connection from %s.", inet_ntoa(client[i].addr.sin_addr));
                    break;
                }
                if(i == FD_SETSIZE) //若打开的套接字个数达到上限则提示出错信息
                    printf("too many clients\n");
                FD_SET(connectfd, &allset); /* 将从套接字描述符 connectfd 添加到文件描述符集合
                                               allset 之中 */
                if(connectfd > maxfd)
                    maxfd = connectfd; //更新最大套接字描述符的值
                if(i > maxi)
                    maxi = i;
                if(--nready <=0)
                    continue;
            }
            for (i=0; i<= maxi; i++) { //循环判断到底是哪个套接字已经就绪
                if((sockfd = client[i].fd)<0)
                    continue;
                if(FD_ISSET(sockfd, &rset)) { //若套接字 sockfd 已经准备就绪
                    if((n = recv(sockfd, recvbuf, MAXDATASIZE,0))==0) {
                                                //则调用 recv()函数从该套接字 sockfd 上读取数据
                        close(sockfd); //若数据读取完毕,则关闭该套接字 sockfd
                        printf("Client( %s ) closed connection. User's data: %s\n",client[i].name,
client[i].data);
                        FD_CLR(sockfd, &allset); //将该套接字 sockfd 从 allset 中清除
                        client[i].fd = -1;
                        delete client[i].name;
                        delete client[i].data;
```

```
                    } else
                        process_cli(&client[i], recvbuf, n); /* 若读取数据尚未完毕，则调用
                                                                process_cli()函数对该次读取
                                                                到的数据进行处理 */
                    if( --nready <=0)
                        break;
                }
            }
        }
        close(listenfd); /* close listenfd */
}

void process_cli(CLIENT *client, char *recvbuf, int len) {
    char sendbuf[MAXDATASIZE]; //定义缓存区数组 sendbuf[]
    recvbuf[len-1] = '\0';
    if(strlen(client->name) ==0) {
        memcpy(client->name,recvbuf, len);
        printf("Client 's name is %s. \n",client->name);
        return;
    }
    printf("Received client( %s ) message: %s\n",client->name, recvbuf);
    savedata(recvbuf, len, client->data); /* 调用 savedata()函数将接收到的来自客户端的数据
                                             反转之后保存到 client->data 之中 */
    for (int i1 =0; i1 < len - 1; i1 ++) { /* 将接收到的来自客户端的数据反转之后保存到
                                              sendbuf 之中 */
        sendbuf[i1] = recvbuf[len - i1 -2];
    }
    sendbuf[len - 1] = '\0'; //在 sendbuf 的末尾添加行结束符'\0'
    send(client->fd,sendbuf,strlen(sendbuf),0); /* 调用 send()函数将 sendbuf 中缓存的数据
                                                    写入到套接字 client->fd */
}

void savedata(char * recvbuf, int len, char * data) {
    int start = strlen(data);
    for (int i=0; i<len; i++) {
        data[start+i] = recvbuf[i];
    }
}
```

7.4 本章小结

本章主要对基于单线程并发的面向连接服务器及其实现方法进行深入介绍，并在实现原理介绍的基础上，具体给出一个创建基于单线程的并发的面向连接服务器的完整 C 语言例

程。通过本章的学习，需要了解基于单线程并发的面向连接服务器的进程结构；需要熟悉基于单线程并发的面向连接服务器的设计流程；需要掌握基于单线程并发的面向连接服务器的 C 语言实现方法。

本章习题

1. 简述基于单线程并发的面向连接服务器的进程结构。
2. 简述基于单线程并发的面向连接服务器软件的设计流程。
3. 试构造一个基于单线程并发的面向连接服务器例程，该例程能实现以下功能：能同时等候来自 10 个不同客户的连接请求，一旦与某个客户连接成功则接收来自该客户的信息，当每收到一个字符串时将首先显示该字符串，然后再将该字符串反转、最后再将反转后的字符串回送给该客户。

第 8 章　基于线程池并发的面向连接服务器例程剖析

第 7 章介绍了基于单线程并发的面向连接服务器及其实现方法，本章将进一步针对另一种并发服务器——基于线程池并发的面向连接服务器及其实现方法进行深入介绍，并在实现原理介绍的基础上，具体给出一个创建基于线程池并发的面向连接服务器的完整 C 语言例程。

8.1　线程池简介

8.1.1　线程池定义

所谓“线程池”，就是一个用来存放“线程”的对象池。在程序中，如果某个创建某种对象所需要的代价太高，同时这个对象又可以反复使用，那么往往就可以准备一个容器来保存一批这样的对象。于是，当需要使用这种对象时，就不需要每次去创建一个，而直接从容器中挑选出一个现成的对象就可以了。由于节省了创建对象的开销，程序性能自然就上升了。这个容器就是“池”。

目前，大多数网络服务器，包括 Web 服务器、E-mail 服务器和数据库服务器等都具有一个共同点，就是单位时间内必须处理数目巨大的连接请求，但处理时间却相对较短。在传统的多线程方案中，通常采用的服务器模型是一旦接收到客户的请求之后，即创建一个新的线程，并由该线程执行任务。而等到任务执行完毕后，线程将退出，这就是“即时创建，即时销毁”的策略。

线程的生命周期包括：创建、活动和销毁，每一个步骤都占用一定的 CPU 时间，尽管与创建进程相比，创建线程的时间已经大大的缩短，但是如果提交给线程的任务是执行时间较短，而且执行次数极其频繁，那么服务器将处于不停的创建线程，销毁线程的状态。当创建和销毁占据了线程周期的总 CPU 额度的很大一部分比例之后，性能问题随之而来，即宝贵的 CPU 时间被大量地消耗在线程的创建、销毁和切换过程中。例如：假定线程的创建时间为 T1、线程的执行时间（包括线程的同步等时间）为 T2、线程的销毁时间为 T3，则可以看出，线程本身的开销所占的比例为(T1 + T3)/(T1 + T2 + T3)。显然，如果线程执行的时间很短的话，这笔开销可能占到 20% ~ 50% 左右。如果任务执行时间很频繁的话，这笔开销将是不可忽略的。

线程池的出现正是着眼于减少线程本身所带来的上述开销。线程池采用预创建的技术，在应用程序启动之后，将立即创建一定数量的线程（N_1），并放入空闲队列中。这些线程都是处于阻塞（Suspended）状态，不消耗 CPU，但占用较小的内存空间。当任务到来后，缓冲池选择一个空闲线程，把任务传入此线程中运行。当 N_1 个线程都在处理任务后，缓冲池

将自动创建一定数量的新线程，用于处理更多的任务。在任务执行完毕后线程也不退出，而是继续保持在池中等待下一次的任务。当系统比较空闲时，大部分线程都一直处于暂停状态，线程池自动销毁一部分线程，回收系统资源。

基于这种预创建技术，线程池将线程创建和销毁本身所带来的开销分摊到了各个具体的任务上，执行次数越多，每个任务所分担到的线程本身开销则越小（当然，此时需要另外考虑线程之间同步所带来的开销）。除此之外，线程池还能够减少创建的线程个数。通常，线程池所允许的并发线程是有上限的，如果同时需要并发的线程数超过上限，那么一部分线程将会等待。

8.1.2　线程池的基本工作原理

线程池的实现基本上是一个生产者消费者模型，具体就是 1 个生产者（值守线程）对应多个多个消费者（n 个任务线程）。主线程对应其中的生产者，将到达的客户请求进行封装后送到商店供消费者使用（这里的商店可以用链表或是其他容器来实现），而线程池中的多个工作线程就是这些商品（客户请求）的消费者。在实际应用中，线程池中工作线程的个数是需要动态调整的，高峰期时线程池可通过增加线程来尽可能满足任务的需要，空闲期时线程池可通过缩减线程来减小尺寸。其中，线程池的最小尺寸无需设定为某个预置的固定大小，通常可根据一定时期内任务队列的平均大小获得一个统计量来进行调整。

值守线程的功能是监视任务队列和维护线程池的尺寸，当任务队列中有任务项目时，每次摘除一个任务并将之投放到线程池的空闲线程中去，当线程池中没有空闲线程时，值守线程负责创建新的线程加入到池中。在任务空闲状态，值守线程销毁超过线程池最小尺寸的空闲线程，以释放系统资源。

任务线程是完成具体应用服务的工作线程，未被任务占据的线程称为空闲状态，当然在此状态下也不希望它只是无效地空转，因为那样同样会消耗 CPU 的时间，解决办法是将它阻塞在后台。使得任务线程的有效运行是在投放了任务之后，被投放任务的线程在任务完成之前不能再被其他任务占据，此时的线程称之为运行状态。在服务进程退出之前，当然希望任务线程都能够自然终结。鉴于以上原因，故必须定义一些状态量来控制任务线程的有序运行，可以用如下状态量：空闲态、运行态、终结态，并且为每一个任务线程定义一个任务信号。任务线程的状态转移示意图如图 8.1 所示。

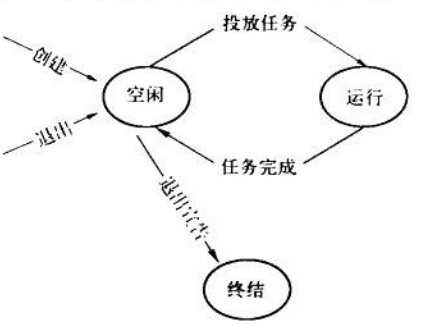


图 8.1　任务线程的状态转移示意图

由图 8.1 可知，首先，值守线程创建新的任务线程，同时，将其状态设置为空闲态，并加入到线程池中，处于空闲态的任务线程在等待任务信号的过程中阻塞。然后，当值守线程从任务队列中摘取任务项目后，从线程池中获取一个空闲状态的线程，把任务项目投放到该线程上，置其状态为运行态，同时，激活任务信号，这样，阻塞的任务线程恢复运行，执行任务。然后，当任务线程完成任务后，重置任务信号，并将自身状态置为空闲态，回归到线程池中。最后，当值守线程收到服务进程的退出宣告后，将池中的空闲线程的状态置为终结态，投放空任务（NULL），并激活任务信号，阻塞的任务

线程恢复运行，探测到自身状态为终结态后，执行退出，线程自然终结。值守线程和任务线程的流程示意图如图 8.2 所示。

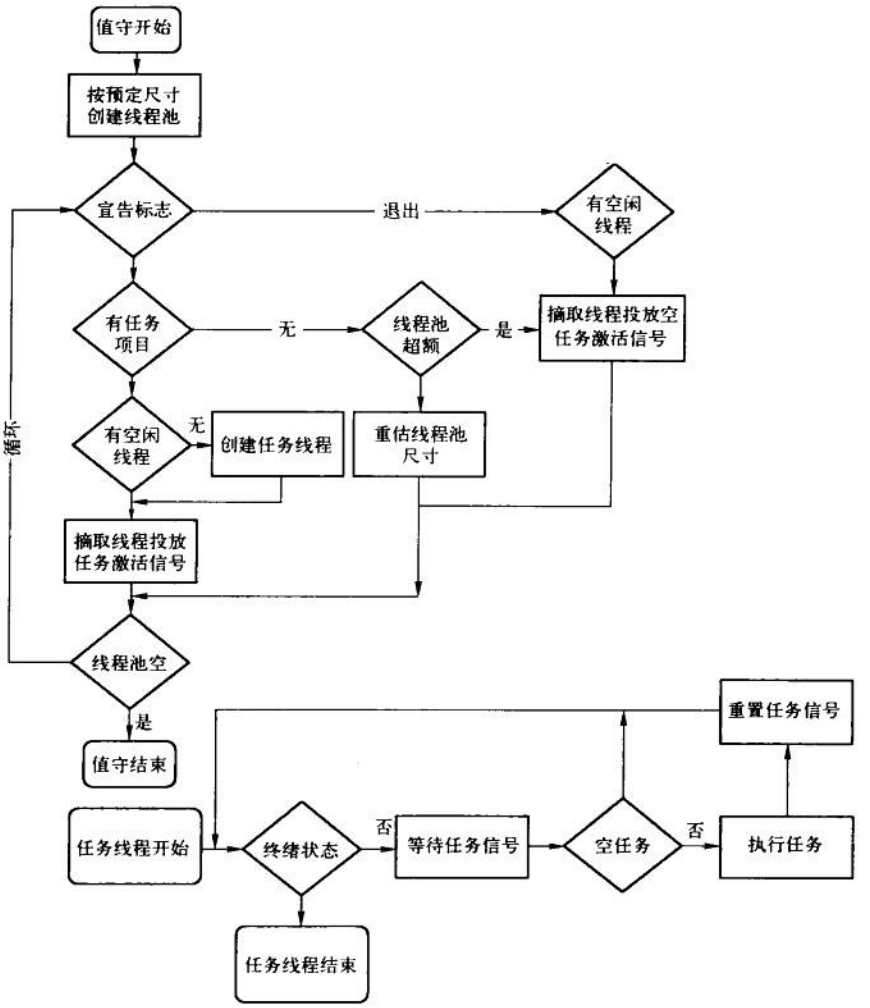


图 8.2 值守线程和任务线程的流程示意图

8.1.3 线程池的应用范围

线程池技术适用于那些需要大量的线程来完成任务，且完成任务的时间比较短的情况。像 Web 服务器处理网页请求这样的任务，使用线程池技术是非常合适的。因为单个任务小，而任务数量巨大，例如：可以想像为一个热门网站的点击次数。但对于长时间的任务，比如一个 Telnet 连接请求，线程池的优点就变得不明显了。因为 Telnet 会话时间比线程的创建时间要长得多。

线程池技术主要适用于对性能要求苛刻的应用，比如服务器要求迅速响应客户请求。另外，线程池技术还适用于接收突发性的大量请求。在没有线程池情况下，突发性大量客户请

求将产生大量线程，虽然理论上大部分操作系统线程数目最大值不是问题，但短时间内产生大量线程可能使内存到达极限。使用线程池技术可以较好地控制并发的服务数目，不至于使服务器因产生大量线程的应用而崩溃。另外，如果线程创建和销毁时间相比任务执行时间可以忽略不计，那么就没有必要使用线程池。

8.1.4　使用线程池的风险

虽然线程池技术是构建服务器应用程序的强大机制，但使用他并不是没有风险的。用线程池构建的应用程序容易遭受任何其他多线程应用程序容易遭受的所有并发风险，诸如同步错误和死锁，还容易遭受特定于线程池的其他风险，诸如与池有关的死锁、资源不足和线程泄漏等。

（1）死锁

任何多线程应用程序都有死锁风险。当一组进程或线程中的每一个都在等待一个只有该组中另一个进程才能引起的事件时，就说这组进程或线程死锁了。死锁的最简单情形是：线程 A 持有对象 X 的独占锁，并且在等待对象 Y 的锁，而线程 B 持有对象 Y 的独占锁，却在等待对象 X 的锁。除非有某种方法来打破对锁的等待，否则死锁的线程将永远等下去。虽然任何多线程程序中都有死锁的风险，但线程池却引入了另一种死锁可能。这时所有池线程都在执行已阻塞的等待队列中另一任务的执行结果的任务，但这一任务却因为没有未被占用的线程而不能运行。当线程池被用来实现涉及许多交互对象的模拟，被模拟的对象可以相互发送查询，这些查询接下来作为排队的任务执行，查询对象又同步等待着响应时，将会发生这种情况。

（2）资源不足

线程消耗包括内存和其他系统资源在内的大量资源，虽然线程之间切换的调度开销很小，但如果有很多线程，环境切换也可能严重地影响程序的性能。如果线程池太大，那么被那些线程消耗的资源可能严重地影响系统性能。在线程之间进行切换将会浪费时间，而且使用超出比实际需要的线程可能会引起资源匮乏问题，因为池线程正在消耗一些资源。除了线程自身所使用的资源以外，服务请求时所做的工作可能需要其他资源，如套接字或文件等，这些也都是有限资源。有太多的并发请求也可能引起失效，例如不能分配连接和服务线程。

（3）资源泄漏

在池的管理机制中，要求客户端在使用完资源后把这个资源放回池中，但是当一个资源长时间使用后，连接可能发生了超时，数据服务的进程可能发生了退出，网络可能中断等，都可能造成使用这个资源的客户端发生异常，使得这个资源无法回到池中供其他客户访问。如果不对这种状况进行处理，很有可能池中的所有资源都会变成不可用，从而服务器无法提供服务。发生泄漏的一种情形出现在任务抛出一个异常或一个 Error 时。如果没有捕捉到它们，那么池的大小将会永久减少一个。当这种情况发生的次数足够多时，池最终将变成空池，而且系统将停止，因为没有可用的线程来处理任务。

8.2　一个 LINUX 下线程池的 C 语言实现

一般线程池都必须具备下面几个组成部分。

① 线程池管理器：用于创建并管理线程池。

② 工作线程：线程池中实际执行的线程。

③ 任务接口：尽管线程池大多数情况下是用来支持网络服务器，但是可以将线程执行的任务抽象出来，以形成任务接口，从而使得线程池与具体的任务无关。

④ 任务队列：线程池的概念具体到实现则可能是队列，链表之类的数据结构，其中保存执行线程。

以下给出一个 LINUX 系统下用 C 语言创建的线程池实例。

该例程为一个 LINUX 系统下用 C 语言创建的线程池的例程，其中，线程池会维护一个任务链表（链表中的每个任务都用一个 CThread_worker 结构表示），首先，例程调用 pool_init( ) 函数预先创建好 max_thread_num 个线程，每个线程都执行相同的线程体函数 thread_routine( )；在该函数中，如果任务链表中没有任务，则线程将处于阻塞等待状态，否则，将从任务链表中取出一个任务并执行。pool_add_worker( ) 函数用于向线程池的任务链表中添加一个任务，加入后通过调用 pthread_cond_signal( ) 函数唤醒一个出于阻塞状态的线程（若存在的话）。pool_destroy( ) 函数用于销毁线程池，线程池销毁后，任务链表中的任务将不会再被执行，但是正在运行的线程会一直把任务运行完之后再退出。

该例程的 C 语言源代码如下：

```
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror 和 printf 等)
                     的定义*/
#include <stdlib.h> /*C 标准库头文件,包含了 C 语言标准库函数(如 exit、atoi 等)原型的定义*/
#include <unistd.h> /*LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的
                     定义*/
#include <sys/types.h> //基本数据类型头文件,含有基本系统数据类型的定义
#include <pthread.h> //提供了线程函数原型与数据结构的定义
#include <assert.h> //提供了诊断函数 assert 的原型定义

typedef struct worker{ /*线程池里所有运行和等待的任务都用一个 CThread_worker 结构表示,由
                        于所有任务都在链表中,所以构成一个任务链表结构*/
    void *(*process)(void *arg); //回调函数,任务运行时会调用该函数
    void *arg; //回调函数的参数
    struct worker *next; //指向任务链表结构中的下一个任务的指针变量
} CThread_worker;

typedef struct{ //线程池结构
    pthread_mutex_t queue_lock; //互斥锁
    pthread_cond_t queue_ready; //条件变量
    CThread_worker *queue_head; //任务链表结构,用于存储线程池中所有的等待任务
    int shutdown; //标志变量,用于表明是否销毁线程池
    pthread_t *threadid; //线程标识符
    int max_thread_num; //线程池中允许的活动线程数目
    int cur_queue_size; //当前任务链表中的任务数目
} CThread_pool;
```

```
int pool_add_worker (void *(*process) (void *arg), void *arg); /*用于向线程池的任务链表
                                                              中添加一个任务*/
void *thread_routine (void *arg); //线程体函数
static CThread_pool *pool = NULL; //定义并初始化线程池结构体变量

void pool_init (int max_thread_num) { //用于预先创建好 max_thread_num 个线程
    pool = (CThread_pool *) malloc (sizeof (CThread_pool)); /*为线程池结构体变量分配长度
                                                            内存块*/
    pthread_mutex_init (&(pool->queue_lock), NULL); //初始化互斥锁
    pthread_cond_init (&(pool->queue_ready), NULL); //初始化条件变量
    //以下语句用于初始化线程池结构体的头部
    pool->queue_head = NULL;
    pool->max_thread_num = max_thread_num;
    pool->cur_queue_size = 0;
    pool->shutdown = 0;
    pool->threadid = (pthread_t *)malloc(max_thread_num * sizeof (pthread_t));
    int i = 0;
    for (i = 0; i < max_thread_num; i++) { //创建 max_thread_num 个新线程
        pthread_create(&(pool->threadid[i]),NULL,thread_routine, NULL);
    }
}

int pool_add_worker (void *(*process) (void *arg), void *arg) { //用于向线程池中加入任务
    //以下语句用于构造一个新的任务
    CThread_worker *newworker = (CThread_worker *)malloc (sizeof (CThread_worker));
    newworker->process = process;
    newworker->arg = arg;
    newworker->next = NULL; //别忘了将该指针置空

    pthread_mutex_lock (&(pool->queue_lock)); //在操作全局变量 pool 之前,须先上锁
    CThread_worker *member = pool->queue_head; /*定义一个新任务 member 并将其初始化
为线程池中的第一个任务链表结构 pool->queue_head*/
    if(member != NULL) { //若线程池中的当前任务链表结构不为空
        while (member->next != NULL) //若当前任务不是任务链表中的最后一个任务
            member = member->next; //则将当前任务设置为下一个任务
        member->next = newworker; /*若当前任务是任务链表中的最后一个任务,则将当前任
                                    务的下一个任务指针指向新构造的任务*/
    }
    else { //若线程池中的任务链表结构为空
        pool->queue_head = newworker; /*则将任务链表结构的头部设置为所构造的新任务*/
    }
    assert (pool->queue_head != NULL); /*若线程池中的任务链表结构的头部不为空*/
    pool->cur_queue_size ++; //则将当前任务链表中的任务数目加 1
```

```
    pthread_mutex_unlock (&(pool->queue_lock)); /*在执行完操作全局变量 pool 之后,须解锁*/
    pthread_cond_signal (&(pool->queue_ready)); /*由于任务链表中已有任务了,因此可以调
用 pthread_cond_signal()函数来唤醒一个等待线程;注意:如果所有线程都在忙碌,则该语句将没
有任何作用*/
    return 0;
}

int pool_destroy() { /*销毁线程池,任务链表中的任务将不会再被执行,但是正在运行的线程会
                       一直把任务运行完之后再退出*/
    if(pool->shutdown) //若 pool->shutdown 之值等于 1
        return -1; //则直接返回,以防止两次调用 pool_destroy()
    pool->shutdown=1; //设置标志位为 1,以表示已销毁线程池
    pthread_cond_broadcast(&(pool->queue_ready));//唤醒所有等待线程以便销毁线程池
    int i;
    for (i=0; i<pool->max_thread_num; i++)
        pthread_join (pool->threadid[i], NULL); //阻塞等待所有线程退出
    free (pool->threadid); //释放线程标识符变量所占内存
    //以下语句用于销毁任务链表
    CThread_worker *head=NULL;
    while (pool->queue_head!=NULL) { /*顺序释放任务链表中的每个任务对应的 CThread_
                                       worker 结构体所占用的内存*/
        head=pool->queue_head;
        pool->queue_head=pool->queue_head->next;
        free (head);
    }
    //条件变量和互斥量也别忘了销毁
    pthread_mutex_destroy(&(pool->queue_lock));
    pthread_cond_destroy(&(pool->queue_ready));

    free (pool); //释放线程池结构体变量所占内存
    pool=NULL; //销毁后将指针置空是个好习惯
    return 0;
}

void * thread_routine (void *arg) { //线程体函数
    printf ("starting thread 0x%x\n", pthread_self ());//显示线程标识符
    while (1) {
        pthread_mutex_lock (&(pool->queue_lock)); /*将预先创建的所有线程都通过互斥锁
mutex 休眠在线程池中。这样一来,以后通过 unlock 该 mutex 即可唤醒该线程*/
        while (pool->cur_queue_size==0 && !pool->shutdown) { /*若线程池中任务链表
中的任务数目为 0 且不打算销毁线程池*/
            printf ("thread 0x%x is waiting\n", pthread_self ()); //显示线程等待提示信息
            pthread_cond_wait(&(pool->queue_ready),&(pool->queue_lock));
```

```
                                                    //线程阻塞等待条件成立
            }
            if(pool -> shutdown) { //若打算销毁线程池
                pthread_mutex_unlock (&(pool -> queue_lock)); //先解锁
                printf ("thread 0x%x will exit\n", pthread_self ()); //显示线程退出提示信息
                pthread_exit (NULL); //线程退出
            }
            printf ("thread 0x%x is starting to work\n", pthread_self ());
                                                    //显示线程工作提示信息
            assert (pool -> cur_queue_size != 0); //若线程池中任务链表中的任务数目不为 0
            assert (pool -> queue_head != NULL); //若线程池中任务链表不为空
            pool -> cur_queue_size --; //将线程池中任务链表中的任务数目减去 1
            CThread_worker * worker = pool -> queue_head; /* 取出线程池中任务链表中的头元素,
                                                    即第一个任务 */
            pool -> queue_head = worker -> next;
            pthread_mutex_unlock (&(pool -> queue_lock)); //解锁
            (*(worker -> process)) (worker -> arg); //调用回调函数,执行该任务
            free (worker); //释放任务链表结构所占的内存
            worker = NULL; //销毁后将指针置空是个好习惯
        }
        pthread_exit (NULL); //线程执行完毕之后退出
    }
```

以下是测试代码:

```
    void * myprocess (void * arg) {
        printf ("threadid is 0x%x, working on task %d\n", pthread_self(), *(int *) arg);
        sleep (1);//休息一秒,延长任务的执行时间
        return NULL;
    }

    int main (int argc, char * * argv) {
        pool_init (3); //线程池中预先生成 3 个活动线程
        //以下代码段用于连续向线程池中投入 10 个任务
        int * workingnum = (int *) malloc (sizeof (int) * 10);
        int i;
        for (i = 0; i < 10; i ++) {
            workingnum[i] = i;
            pool_add_worker (myprocess, &workingnum[i]);
        }
        sleep (5); //主线程休眠 5 秒,等待所有任务完成
        pool_destroy (); //销毁线程池
        free (workingnum);
        return 0;
```

```
}
```

将上述所有代码放入 threadpool. c 文件中，然后在 Linux 输入以下编译命令：

```
$ gcc -o threadpool threadpool. c -lpthread
```

则可得到运行结果如下：

```
starting thread 0xb7df6b90
thread 0xb7df6b90 is waiting
starting thread 0xb75f5b90
thread 0xb75f5b90 is waiting
starting thread 0xb6df4b90
thread 0xb6df4b90 is waiting
thread 0xb7df6b90 is starting to work
threadid is 0xb7df6b90, working on task 0
thread 0xb75f5b90 is starting to work
threadid is 0xb75f5b90, working on task 1
thread 0xb6df4b90 is starting to work
threadid is 0xb6df4b90, working on task 2
thread 0xb7df6b90 is starting to work
threadid is 0xb7df6b90, working on task 3
thread 0xb75f5b90 is starting to work
threadid is 0xb75f5b90, working on task 4
thread 0xb6df4b90 is starting to work
threadid is 0xb6df4b90, working on task 5
thread 0xb7df6b90 is starting to work
threadid is 0xb7df6b90, working on task 6
thread 0xb75f5b90 is starting to work
threadid is 0xb75f5b90, working on task 7
thread 0xb6df4b90 is starting to work
threadid is 0xb6df4b90, working on task 8
thread 0xb7df6b90 is starting to work
threadid is 0xb7df6b90, working on task 9
thread 0xb75f5b90 is waiting
thread 0xb6df4b90 is waiting
thread 0xb7df6b90 is waiting
thread 0xb75f5b90 will exit
thread 0xb6df4b90 will exit
thread 0xb7df6b90 will exit
```

8.3 基于线程池并发的面向连接服务器例程

该例程为一个基于线程池并发的面向连接服务器例程，其中，主线程首先生成一个包

含 10 个从线程的线程池，然后主线程负责统一接收来自客户的连接请求，在收到来自客户的连接请求之后，再指派一个空闲的从线程负责处理该客户连接请求。

服务器例程的 C 语言源代码如下：

```
#include <unistd.h> /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的定义 */
#include <syslog.h> //提供了用来改变日志功能的行为的函数原型的定义
#include <string.h> //提供了字符串函数原型的定义
#include <stdio.h> /* 标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf()等)的定义 */
#include <stdlib.h> /* C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()等)原型的定义 */
#include <errno.h> //该头文件中包含了全局变量 errno 的定义
#include <signal.h> //提供 signal 函数原型的定义
#include <sys/mman.h> //提供内存管理声明函数原型的定义
#include <sys/types.h> //基本数据类型头文件,含有基本系统数据类型的定义
#include <sys/socket.h> //提供了套接字函数原型与数据结构的定义
#include <sys/wait.h> //提供 wait 函数原型的定义
#include <netinet/in.h> //包含了数据结构 sockaddr_in 的定义
#include <arpa/inet.h> //包含了 IP 地址转换函数原型的定义
#include <netinet/tcp.h> //与 TCP 协议有关的函数库文件
#include <sys/resource.h> /* 为资源操作提供了定义,包括在一个程序允许的尺寸,执行的优先级以及文件上确定和设置限制的函数的原型定义 */
#include <fcntl.h> //文件控制头文件,含有文件及其描述符的操作控制常数符号的定义
#include <pthread.h> //提供了线程函数原型与数据结构的定义
#define PORT 6600 //定义表示端口号的符号变量
#define LISTENQ 1024 //定义表示队列长度的符号变量
#define MAXLINE 4096 //定义从套接字一次接收数据的最大缓冲区长度符号变量
#define MAXN 1024 //定义客户端要求服务端发送的最大字节数符号变量
#define HAVE_MSGHDR_MSG_CONTROL /* 定义宏 HAVE_MSGHDR_MSG_ CONTROL ,则在程序中碰到#ifdef...#else... 时,若则执行紧随#ifdef 其后的代码段,否则将执行紧随#else 其后的代码段 */
typedef struct { //定义线程池结构体
    pthread_t thread_tid; //线程 ID
    long thread_count; //线程处理的客户连接数量
} Thread;
Thread *tptr; //定义线程池结构体变量
#define MAXNCLI 32 //定义可打开的从套接字最大个数为 32
int clifd[MAXNCLI], iget, iput; /* clifd 数组用于主线程往其中存入已接受的已连接套接字描述符,并由线程池中的可用线程从中取出一个以服务响应客户。iput 是往数组里存入的下一个元素的下标,iget 是从数组里取出的下一个元素的下标 */
pthread_mutex_t clifd_mutex = PTHREAD_MUTEX_INITIALIZER; //初始化互斥锁变量
pthread_cond_t clifd_cond = PTHREAD_COND_INITIALIZER; //初始化条件变量
```

```
int listenfd, nthreads; //定义主线程描述符变量 listenfd 和线程个数变量 nthreads
socklen_t addrlen; //定义端点地址结构长度变量
struct sockaddr_in *cliaddr; //定义客户端套接字的端点地址结构变量
static pid_t *pids; //定义线程描述符变量
void pr_cpu_time(void); //声明 pr_cpu_time 函数
void web_child(int); //声明 web_child 函数
void sig_int(int); //声明 sig_int 函数
static pthread_mutex_t *mptr; //定义互斥锁变量
ssize_t read_fd(int, void *, size_t, int *); //声明 read_fd 函数
ssize_t write_fd(int, void *, size_t, int); //声明 write_fd 函数
void *thread_main(void *); //声明线程执行体函数 thread_main()
void thread_make(int i); //声明用于为线程池创建新的从线程函数 thread_make()

int main(int argc, char **argv){
    int i, navail, maxfd, nsel, connfd, rc;//定义相关局部变量
    pthread_t tid; //定义线程描述符变量
    socklen_t clilen; //定义端点地址结构长度变量
    struct sockaddr_in servaddr; //定义服务器端套接字的端点地址结构变量
    const int on = 1;
    ssize_t n;
    fd_set rset, masterset;
    listenfd = socket(AF_INET, SOCK_STREAM, 0); //创建主套接字 listenfd
    bzero(&servaddr, sizeof(servaddr)); //将服务器端套接字的端点地址结构变量清零
    //以下代码段用于对服务器端套接字的端点地址结构变量赋值
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);
    setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));/*设置主套接字
                                                                    属性*/
    bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)); /*将主套接字与服务器端
                                                                    点地址进行绑定*/
    listen(listenfd, LISTENQ);              //将主套接字设置为被动模式并设置等待队列长度
nthreads = 10;                              //初始化线程池中的线程个数为 10
tptr = calloc(nthreads,sizeof(Thread));     //为线程池结构体变量分配内存空间
iget = iput = 0;
for(i = 0;i < nthreads;i ++)
    thread_make(i);                         //为线程池中预先生成 10 个从线程
signal(SIGINT,sig_int);/*注册信号处理函数 sig_int 作为系统发送 SIGINT 信号(按下 Ctrl + C 键)
                         给本进程时的中断响应函数*/
for(;;){                                    //循环接收来自客户的连接请求
    clilen = addrlen;
    connfd = accept(listenfd,cliaddr,&clilen);/*接收来自客户的连接请求,创建新的从套接字 con-
                                                nfd 来负责处理该用户连接*/
```

```
        pthread_mutex_lock(&clifd_mutex);/* 为了操作共享变量 clifd,iput 等,先对互斥锁上锁 */
        clifd[iput] = connfd;                    //将 connfd 值保存到套接字数组变量 clifd[]
        if( ++iput == MAXNCLI){                  //若当前活动的从套接字个数超过设定的最大值
            iput = 0;
        if(iput == iget){                        /* 若当前活动的从套接字个数等于已处理完成
                                                    的套接字个数 */
            printf("iput = iget = %d\n",iput);
            exit(1);
        }
        pthread_cond_signal(&clifd_cond);/* 主线程接受一个连接后将调用 pthread_cond_signal 向
                                            条件变量 clifd_cond 发送信号,以便唤醒睡眠在其上的
                                            线程 */
        pthread_mutex_unlock(&clifd_mutex);      //共享变量操作完毕,互斥锁解锁
    }
}

void thread_make(int i){                         //为线程池创建新的从线程
    void * thread_make(void *);
    pthread_create(&tptr[i].thread_tid,NULL,&thread_main,(void *)i);
                                                 //创建新从线程
    return;
}

void * thread_main(void * arg){                  //线程执行体函数
    int connfd;                                  //定义套接字描述符局部变量
    void web_child(int);
    printf("thread %d starting\n",(int)arg);
    for(;;){
        pthread_mutex_lock(&clifd_mutex);/* 操作共享变量 iput,iget,clifd 之前,先对互斥锁上锁 */
        printf("get lock,thread = [%d]\n",(int)arg);
        while(iget == iput)/* 若当前已处理完成的套接字个数等于活动的从套接字个数,即所有
                              活动的套接字均已处理完毕 */
            pthread_cond_wait(&clifd_cond,&clifd_mutex);/* 线程休眠,等待条件变量 clifd_cond
                                                           成立,即等待新的客户连接请求到
                                                           来 */
        connfd = clifd[iget];
        if( ++iget == MAXNCLI)                   /* 若当前处理完的套接字个数达到最大值 MAX-
                                                    NCLI */
            iget = 0;
        pthread_mutex_unlock(&clifd_mutex);      //操作共享变量完毕,对互斥锁解锁
        tptr[(int)arg].thread_count ++;          //线程已处理完毕的套接字个数加 1
        web_child(connfd);                       /* 调用 web_child 函数利用套接字 connfd 进行事
                                                    务处理 */
```

```
        close(connfd);                          //线程执行完毕,关闭套接字
    }
}

void sig_int(int signo){/* 系统发送给进程的信号处理函数,当用户按下 Ctrl + C 键(即系统发送给
                           进程 SIGINT 信号)时,程序将在打印出 CPU 的处理时间之后退出 */
    int   i;
    void pr_cpu_time(void);                     //声明 CPU 处理时间打印函数 pr_cpu_time
    pr_cpu_time();                              //调用 CPU 处理时间打印函数 pr_cpu_time
    for(i=0;i<nthreads;i++)                     //打印出每个线程处理了多少个连接
        printf("thread %d,%ld connections\n",i,tptr[i].thread_count);
    exit(0);                                    //退出程序
}

void pr_cpu_time(void){                         //打印 CPU 处理时间
    double    user,sys;
    struct rusage myusage,childusage;
    if(getrusage(RUSAGE_SELF,&myusage)<0){/* 利用系统函数 getrusage()得到程序运行的 us-
                                             er time 和 sys time */
        printf("getrusage error\n");
        return;
    }
    if(getrusage(RUSAGE_CHILDREN,&childusage)<0){
        printf("getrusage error\n");
        return;
    }
    user=(double)myusage.ru_utime.tv_sec+myusage.ru_utime.tv_usec/1000000.0;
    user+=(double)childusage.ru_utime.tv_sec+childusage.ru_utime.tv_usec/1000000.0;
    sys=(double)myusage.ru_stime.tv_sec+myusage.ru_stime.tv_usec/1000000.0;
    sys+=(double)childusage.ru_stime.tv_sec+childusage.ru_stime.tv_usec/1000000.0;
    printf("\nuser time=%g,sys time=%g\n",user,sys);
}

void web_child(int sockfd){                     //事务处理函数
    intntowrite;
    ssize_tnread;
    charline[MAXLINE],result[MAXN];
    for(;;){
        if((nread=read(sockfd,line,MAXLINE))==0)
                                                //从套接字读取数据
        return;  /* connection closed by other end */
        /* 4line from client specifies #bytes to write back */
        printf("recieve from client [%s]\n",line);
```

```
            ntowrite = atol(line);                    //计算读取的字符数
            if((ntowrite <=0)||(ntowrite > MAXN)){
                printf("client request for %d bytes\n",ntowrite);
                return;
            }
            printf("send to client [%s]\n",result);
            writen(sockfd,result,ntowrite);          //发送 ntowrite 字节个任意的数据给客户程序
        }
    }

ssize_t writen(int fd,const void * vptr,size_t n){
    size_t   nleft;
    ssize_t   nwritten;
    const char * ptr;
    ptr = vptr;
    nleft = n;
    while(nleft >0){
        if((nwritten = write(fd,ptr,nleft))<=0){
                                                  //调用 write 通过套接字发送数据
            if(nwritten <0 && errno == EINTR)
                nwritten =0;   /* and call write() again */
            else
                return(-1);    /* error */
        }
        nleft -= nwritten;
        ptr   += nwritten;
    }
    return(n);
}
```

(1) 用于测试服务器程序（向服务器程序发送请求）的客户程序

在该客户程序中，父进程调用 fork()函数派生指定个数的子进程，每个子进程再与服务器建立指定次数的连接。每次连接建立后，子进程就在该连接上向服务器发送一行文本，指出需由服务器返回多少字节的数据，然后在该连接上读入这个数量的数据，最后关闭该连接。父进程只是调用 wait()函数等待所有子进程终止。

```
#include <unistd.h>/* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的定义 */
#include <stdio.h>  /* 标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf()等)的定义 */
#include <stdlib.h>  /* C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()等)原型的定义 */
#include <errno.h>       //该头文件中包含了全局变量 errno 的定义
```

```
#include < string. h >        //提供了字符串函数原型的定义
#include < sys/types. h >     //基本数据类型头文件,含有基本系统数据类型的定义
#include < sys/socket. h >    //提供了套接字函数原型与数据结构的定义
#include < sys/wait. h >      //提供 wait 函数原型的定义
#include < sys/stat. h >      //文件状态头文件,含有文件或文件系统状态结构和常量的定义
#include < signal. h >        //提供 signal 函数原型的定义
#include < netdb. h >         //含有 hostent 结构与 gethostbyname 函数的定义
#include < fcntl. h >         //文件控制头文件,含有文件及其描述符的操作控制常数符号的定义
#include < pthread. h >       //提供了线程函数原型与数据结构的定义
#define  MAXLINE  4096 //定义从套接字一次发送数据的最大缓冲区长度符号变量
#define MAXN 1024          //客户端要求服务器发送的最大字符个数
int main(int argc,char * * argv){
    int i,j,fd,nchildren,nloops,nbytes;
    pid_t pid;
    ssize_t n;
    char request[MAXLINE],reply[MAXN];
    if(argc ! =6)
        printf("usage:client <hostname or IPaddr><port><#children>"
            " <#loops/child><#bytes/request>");/* argv[0]:client ;argv[1]:hostname or IPad-
                                                  dr,主机名或 IP 地址;argv[2]:port,端口
                                                  号;argv[3]:#children,要派生的子进程个
                                                  数;argv[4]:#loops/child,每个子进程要与
                                                  服务器建立的连接的次数;argv[5]:#
                                                  bytes/request,客户端要求服务器端回送的
                                                  字符长度 */
    nchildren = atoi(argv[3]);
    nloops = atoi(argv[4]);
    nbytes = atoi(argv[5]);
    snprintf(request,sizeof(request),"%d\n",nbytes);
    for(i =0;i < nchildren;i ++){
        if((pid = fork()) ==0){          //父进程调用 fork 派生指定 nchildren 个子进程
            for(j =0;j < nloops;j ++){ //每个子进程与服务器建立指定 nloops 次连接
                fd = connectTCP(argv[1],argv[2]);/* 调用 connectTCP 函数(见 3.6 节)建立
                                                    连接 */
                write(fd,request,strlen(request));
                if((n = Readn(fd,reply,nbytes)) ! = nbytes)
                    printf("server returned %d bytes",n);
                close(fd);/* TIME_WAIT on client,not server */
            }
            printf("child %d done\n",i);
            exit(0);
        }
        /* parent loops around to fork() again */
```

```
    }
    while(wait(NULL) >0)/* now parent waits for all children */
        ;
    if(errno ! = ECHILD)
        err_sys("wait error");
    exit(0);
}

ssize_t readn(int fd,void * vptr,size_t n){
    size_t  nleft;
    ssize_t  nread;
    char  * ptr;
    ptr = vptr;
    nleft = n;
    while(nleft >0){
        if((nread = read(fd,ptr,nleft))<0){
            if(errno == EINTR)
                nread = 0;
            else
                return(-1);
        }else if(nread == 0)
            break;
        nleft -= nread;
        ptr   += nread;
    }
    return(n - nleft);
}

ssize_t Readn(int fd,void * ptr,size_t nbytes){
    ssize_t n;
    if((n = readn(fd,ptr,nbytes))<0)
        printf("readn error");
    return(n);
}
```

(2) 程序的运行结果：

① 客户端。假定客户端程序编译之后的可执行文件为 myclient：

```
$ ./myclient 173.26.100.162 12345 5 500 4000
```

5 个子进程各自发起 500 次连接，总共建立 2500 个与服务器的 TCP 连接，在每个连接上，客户向服务器发送 5 字节的数据（“4000\n”），服务器向客户返回 4000 字节的数据。

② 服务器端。假定服务器端程序编译之后的可执行文件为 myserver：

```
$ ./myserver 173.26.100.162 12345
```

运行结果如下：

```
thread 0 starting
thread 1 starting
thread 2 starting
thread 3 starting
thread 4 starting
thread 5 starting
thread 6 starting
thread 7 starting
thread 8 starting
thread 9 starting

user time = 0.012, sys time = 0.096006
thread 0,246 connections
thread 1,250 connections
thread 2,252 connections
thread 3,250 connections
thread 4,251 connections
thread 5,249 connections
thread 6,249 connections
thread 7,252 connections
thread 8,253 connections
thread 9,248 connections
```

8.4 本章小结

本章主要对基于线程池并发的面向连接服务器及其实现方法进行深入介绍，并在实现原理介绍的基础上，具体给出一个创建基于线程池并发的面向连接服务器的完整 C 语言例程。通过本章的学习，需要了解线程池的基本概念与工作原理；需要掌握基于线程池的并发的面向连接服务器的 C 语言实现方法。

本章习题

1. 简述什么是线程池。
2. 简述基于线程池并发的面向连接服务器软件的设计流程。
3. 试构造一个基于线程池并发的面向连接服务器例程，该例程能实现以下功能：能同时等候来自 10 个不同客户的连接请求，一旦与某个客户连接成功则接收来自该客户的信息，当每收到一个字符串时将首先显示该字符串，然后再将该字符串反转、最后再将反转后的字符串回送给该客户。

第9章　基于Epoll的并发的面向连接服务器例程剖析

传统select的效率会因为在线人数的线形递增而导致呈二次乃至三次方的下降，这将直接导致网络服务器可支持的人数严重受到限制，LINUX 2.6内核中提供的System Epoll的出现解决上述问题，可支持一个进程打开大数目的Socket描述符。为此，本章将对其进行深入介绍，并在介绍其实现原理的基础上，具体给出一个创建基于Epoll的并发的面向连接服务器的完整C语言例程。

9.1　Epoll简介

Epoll是LINUX下多路复用I/O接口select的增强版本，由于Epoll不会复用文件描述符集合来传递结果而迫使开发者每次等待事件之前都必须重新准备要被侦听的文件描述符集合，再加上它获取事件的时候无需遍历整个被侦听的描述符集，只要遍历那些被内核I/O事件异步唤醒而加入Ready队列的描述符集合就行了，因此，它能显著减少程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率。

与传统的select调用方法相比，select调用方法最不能让人忍受的缺点就是其支持一个进程所能打开的socket描述符的数目是有一定限制的，该值由FD_SETSIZE来进行设置，其默认大小为2048。这对于那些需要支持上万连接数目的IM（Instant Messaging，实时通信）服务器来说显然太少。此时，虽然可以选择修改FD_SETSIZE的值然后重新编译内核，不过这样会带来网络效率的下降；另外，也可以选择采用多进程的解决方案，不过在LINUX下创建进程的代价虽然是比较小，但却仍旧是不可以忽视的，再加上进程间的数据同步远比不上线程间同步的高效，所以也不是一种完美的解决方案。而采用Epoll则没有这个限制，Epoll所支持的socket描述符上限是最大可以打开文件的数目，该数字一般远大于2048。例如，在一个1GB内存的机器上可以打开文件的最大数目大约是10万左右，具体数目可以通过cat /proc/sys/fs/file - max来查看，一般来说该数目和系统的内存大小关系很大。

传统的select调用方法的另一个致命弱点就是select选择句柄的时候，是遍历所有句柄，也就是说句柄有事件响应时，select需要遍历所有句柄才能获取到哪些句柄有事件通知，因此效率非常低。而Epoll对于句柄事件的选择则不是采用遍历的方法，而是采用事件响应的方法，也就是说，一旦句柄上有事件来就马上选择出来，而不需要遍历整个句柄链表，因此Epoll的效率非常高。

9.2 Epoll 的工作原理与调用方法

9.2.1 Epoll 的基本接口函数

Epoll 的接口非常简单，用到的相关函数有以下三个，都在头文件 sys/epoll.h 中进行了声明。

(1) epoll_create( ) 函数

该函数用于创建一个 epoll 的句柄，其原型如下：

```
int epoll_create(int size);
```

在上述 strlen( ) 函数的原型中，其参数的含义如下。

size：用来告诉内核所监听的数目一共有多大，该参数不同于 select( ) 中的第一个参数，给出最大监听的 fd + 1 的值。需要注意的是，当创建好 Epoll 句柄后，它就会占用一个文件描述符 fd 值，在 LINUX 下如果查看/proc/进程 id/fd/，就能够看到该 fd，因此在使用完 epoll 之后必须调用 close( ) 关闭，否则可能导致 fd 被耗尽。

(2) epoll_ctl( ) 函数

该函数为 Epoll 的事件注册函数，与 select( ) 不同，select( ) 是在监听事件时告诉内核要监听什么类型的事件，而 Epoll 是利用该函数先注册所想要监听的事件类型，该函数的原型如下：

```
int epoll_ctl(int epfd,int op,int fd,struct epoll_event * event);
```

epfd：该参数是 epoll_create( ) 的返回值。

op：该参数表示动作，用以下三个宏来表示。

- EPOLL_CTL_ADD：注册新的 fd 到 epfd 中。
- EPOLL_CTL_MOD：修改已经注册的 fd 的监听事件。
- EPOLL_CTL_DEL：从 epfd 中删除一个 fd。

fd：该参数是需要监听的 fd。

event：该参数是告诉内核需要监听什么事件。其中，struct epoll_event 结构如下：

```
struct epoll_event {
  __uint32_t events;  /* Epoll events */
  epoll_data_t data;  /* User data variable */
};
```

其中，events 可以是以下几个宏的集合。

- EPOLLIN ：表示对应的文件描述符可以读（包括对端 SOCKET 正常关闭）。
- EPOLLOUT：表示对应的文件描述符可以写。
- EPOLLPRI：表示对应的文件描述符有紧急的数据可读（即表示有带外数据到来）。
- EPOLLERR：表示对应的文件描述符发生错误。
- EPOLLHUP：表示对应的文件描述符被挂断。

- EPOLLET：将 EPOLL 设为边缘触发（Edge Triggered）模式，这是相对于水平触发（Level Triggered）来说的。
- EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，若还需继续监听该 socket，则需要再次把该 socket 加入到 EPOLL 队列中。

（3）epoll_wait( )函数

该函数用于等待事件的产生，类似于 select( )调用，其原型如下：

```
int epoll_wait(int epfd,struct epoll_event * events,int maxevents,int timeout);
```

events：该参数用来从内核得到事件的集合。

maxevents：该参数用来告之内核 events 有多大，该 maxevents 的值不能大于 epoll_create( )的 size 参数所指定的值的大小。

timeout：该参数是超时时间（毫秒，设置为 0 表示立即返回，设置为 -1 表示不确定或永久阻塞）。epoll_wait 函数返回需要处理的事件数目，如返回 0 则表示已超时。

9.2.2　Epoll 的事件模式

Epoll 的事件有两种不同的模式：ET（Edge Triggered，边缘触发模式）和 LT（Level Triggered，水平触发模式）。

（1）ET 模式

该模式的效率非常高，在并发与大流量的情况下，会比 LT 模式要少很多 Epoll 的系统调用，因此效率高。但是对编程要求高，需要细致地处理每个请求，否则容易发生丢失事件的情况。对于 ET 而言，如果 accpet 调用有返回，除了建立当前这个连接外，不能马上就 epoll_wait 还需要继续循环 accpet，直到返回 -1，且 errno == EAGAIN，才不继续 accept。

（2）LT 模式

该模式的效率一般会低于 ET 触发模式，特别是在大并发与大流量的情况下。但是 LT 模式对代码编写的要求比较低，因此不容易出现问题。LT 模式服务编写上的表现是：只要有数据没有被获取，内核就会不断进行通知，因此不用担心事件丢失的情况发生。在采用 LT 模式时，如果 accept 调用有返回就可以马上建立当前这个连接，再调用 epoll_wait( )等待下次通知，与 select( )一样。

在采用上述两种模式时，要注意的是：如果采用 ET 模式，那么仅当状态发生变化时内核才会进行通知，而采用 LT 模式，则类似于原来的 select 操作，只要还有没有处理的事件内核就会一直进行通知。因此从本质上讲：与 LT 模式相比，ET 模式是通过减少系统调用来达到提高并行效率的目的的。

9.2.3　Epoll 的工作原理

Epoll 的工作原理为：如果想进行 I/O 操作时，先向 Epoll 查询是否可读或可写，如果处于可读或可写状态后，Epoll 会通过 epoll_wait( )函数进行通知，此时再进行进一步的 recv 或 send 操作。

Epoll 仅仅是一个异步事件的通知机制，其本身并不进行任何的 I/O 读写操作，它只负责通知是不是可以读或可以写了，而具体的读写操作，还要应用层自己来作。Epoll 仅提供这种机制也是非常好的，它保持了事件通知与 I/O 操作之间彼此的独立性，使得 Epoll 的使

用更加灵活。

9.3 基于 Epoll 线程池的 C 语言例程

以下给出一个基于 Epoll 线程池的 C 语言例程：

```
#include < iostream. h > /* iostream 是 input output stream 的简写,意思为标准的输入/输出流头文件,
                        它包含 cin >> "要输入的内容" 与 cout << "要输出的内容",若要使用这
                        两个输入/输出的方法则需要引用#include < iostream. h > 头文件来进行
                        声明 */
#include < sys/socket. h >      //包含了各种套接字函数原型与数据结构等的定义
#include < sys/epoll. h >       //包含了各种 Epoll 接口相关函数原型的定义
#include < netinet/in. h >      //包含了数据结构 sockaddr_in 的定义
#include < arpa/inet. h >       //包含了 IP 地址转换函数原型的定义
#include < fcntl. h >           //文件控制头文件,含有文件及其描述符的操作控制常数符号的
                                  定义
#include < unistd. h > /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的定
                        义 */
#include < stdio. h > /* 标准输入/输出头文件,包含了标准输入/输出函数(如 perror 和 printf 等)的
                       定义 */
#include < pthread. h >          //包含了各种线程相关函数原型和数据结构等的定义

#define MAXLINE 10
#define OPEN_MAX 100
#define LISTENQ 20
#define SERV_PORT 5555
#define INFTIM 1000

struct task{                     //线程池任务队列结构体
  int fd;                        //需要读写的文件描述符
  struct task * next;            //下一个任务
};

struct user_data{                //用于读写两个方面传递参数
  int fd;
  unsigned int n_size;
  char line[MAXLINE];
};

void * readtask(void * args);  //线程的任务函数
void * writetask(void * args);
struct epoll_event ev,events[20];/* 声明 epoll_event 结构体变量 ev 用于注册事件,数组 events[20]
                                    用于回传要处理的事件 */
```

```
int epfd;
pthread_mutex_t mutex;
pthread_cond_t cond1;
struct task * readhead = NULL, * readtail = NULL, * writehead = NULL;

void setnonblocking(int sock){
        int opts;
        opts = fcntl(sock,F_GETFL);
        if(opts < 0){
        perror("fcntl(sock,GETFL)");
        exit(1);
        }
        opts = opts|O_NONBLOCK;
        if(fcntl(sock,F_SETFL,opts)<0){
        perror("fcntl(sock,SETFL,opts)");
        exit(1);
        }
}

int main(){
      int i,maxi,listenfd,connfd,sockfd,nfds;
      pthread_t tid1,tid2;
      struct task * new_task = NULL;
      struct user_data * rdata = NULL;
      socklen_t clilen;
      pthread_mutex_init(&mutex,NULL);
      pthread_cond_init(&cond1,NULL);
                                                    //初始化用于读线程池的线程
      pthread_create(&tid1,NULL,readtask,NULL);
      pthread_create(&tid2,NULL,readtask,NULL);
                                                    /*生成用于处理 accept 的 Epoll 专用文件
                                                      描述符 epfd */
      epfd = epoll_create(256);
      struct sockaddr_in clientaddr;
      struct sockaddr_in serveraddr;
      listenfd = socket(AF_INET,SOCK_STREAM,0);
      setnonblocking(listenfd);                     //把 socket 设置为非阻塞方式
      ev.data.fd = listenfd;                        //设置与要处理的事件相关的文件描述符
      ev.events = EPOLLIN|EPOLLET;                  //设置要处理的事件类型
      epoll_ctl(epfd,EPOLL_CTL_ADD,listenfd,&ev);   //注册 epoll 事件
      bzero(&serveraddr,sizeof(serveraddr));
      serveraddr.sin_family = AF_INET;
      char * local_addr = "200.200.200.222";
```

```
inet_aton(local_addr,&(serveraddr.sin_addr));//htons(SERV_PORT);
serveraddr.sin_port=htons(SERV_PORT);
bind(listenfd,(sockaddr*)&serveraddr,sizeof(serveraddr));
listen(listenfd,LISTENQ);
maxi=0;
for(;;){
    nfds=epoll_wait(epfd,events,20,500);        //等待 epoll 事件的发生
    for(i=0;i<nfds;++i){                        //处理所发生的所有事件
        if(events.data.fd==listenfd){
            connfd=accept(listenfd,(sockaddr*)&clientaddr,&clilen);
            if(connfd<0){
                perror("connfd<0");
                exit(1);
            }
            setnonblocking(connfd);
            char*str=inet_ntoa(clientaddr.sin_addr);
            std::cout<<"connec_ from >>"<<str<<std::endl;
            ev.data.fd=connfd;                  //设置用于读操作的文件描述符
                                                //设置用于注册的读操作事件
          ev.events=EPOLLIN|EPOLLET;            //注册 ev
          epoll_ctl(epfd,EPOLL_CTL_ADD,connfd,&ev);
        }
      else if(events.events&EPOLLIN){
            printf("reading!\n");
            if((sockfd=events.data.fd)<0)continue;
            new_task=new task();
            new_task->fd=sockfd;
            new_task->next=NULL;                //添加新的读任务
            pthread_mutex_lock(&mutex);
            if(readhead==NULL){
                readhead=new_task;
                readtail=new_task;
            }
            else{
              readtail->next=new_task;
                readtail=new_task;
            }
                                                //唤醒所有等待 cond1 条件的线程
            pthread_cond_broadcast(&cond1);
            pthread_mutex_unlock(&mutex);
       }
       else if(events.events&EPOLLOUT){
         rdata=(struct user_data*)events.data.ptr;
```

```
                sockfd = rdata -> fd;
                write( sockfd, rdata -> line, rdata -> n_size );
                delete rdata;
                                                    //设置用于读操作的文件描述符
                ev. data. fd = sockfd;
                                                    //设置用于注册的读操作事件
            ev. events = EPOLLIN | EPOLLET;
                                                    //修改 sockfd 上要处理的事件为 EPOLIN
            epoll_ctl( epfd, EPOLL_CTL_MOD, sockfd, &ev );
              }
          }
      }
}

void * readtask( void * args ) {
    int fd = -1;
    unsigned int n;
                                                    //用于把读出来的数据传递出去
    struct user_data * data = NULL;
    while( 1 ) {
        pthread_mutex_lock( &mutex );
                                                    //等待到任务队列不为空
        while( readhead == NULL )
            pthread_cond_wait( &cond1, &mutex );
        fd = readhead -> fd;
                                                    //从任务队列取出一个读任务
        struct task * tmp = readhead;
        readhead = readhead -> next;
        delete tmp;
        pthread_mutex_unlock( &mutex );
        data = new user_data( );
        data -> fd = fd;
        if( ( n = read( fd, data -> line, MAXLINE ) )<0 ) {
            if( errno == ECONNRESET ) {
                close( fd );
            } else
            std::cout << " readline error" << std::endl;
        if( data! = NULL ) delete data;
    }
    else if( n == 0 ) {
        close( fd );
        printf( " Client close connect! \n" );
        if( data! = NULL ) delete data;
```

```
        }
        else{
            data->n_size = n;
                                                    //设置需要传递出去的数据
            ev.data.ptr = data;
                                                    //设置用于注测的写操作事件
            ev.events = EPOLLOUT|EPOLLET;
                                                    //修改 sockfd 上要处理的事件为 EPOLLOUT
            epoll_ctl(epfd,EPOLL_CTL_MOD,fd,&ev);
        }
    }
}
```

9.4 基于 Epoll 的并发的面向连接服务器例程

以下给出一个服务器端使用 Epoll 监听大量并发链接的 C 语言例程：

```
#include <stdio.h>/* 标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf()
                等)的定义 */
#include <stdlib.h>/* C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()等)原型的定
                义 */
#include <errno.h>          //提供错误号 errno 的定义,用于错误处理
#include <string.h>         //提供了字符串函数原型的定义
#include <sys/types.h>      //基本数据类型头文件,含有基本系统数据类型的定义
#include <netinet/in.h>     //包含了数据结构 sockaddr_in 的定义
#include <sys/socket.h>     //包含了各种套接字函数原型与数据结构等的定义
#include <sys/wait.h>       //提供了 wait 函数原型的定义
#include <unistd.h>/* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数据结构的定
                义 */
#include <arpa/inet.h>      //包含了 IP 地址转换函数原型的定义
#include <openssl/ssl.h>/* 包含了 LINUX 用于安全通信的开放库 OPENSSL API 相关安全通信函
                数原型的定义 */
#include <openssl/err.h>/* 包含了 LINUX 用于安全通信的开放库 OPENSSL API 相关错误处理函
                数原型的定义 */
#include <fcntl.h>   /* 文件控制头文件,含有文件及其描述符的操作控制常数符号的定义 */
#include <sys/epoll.h>      //包含了各种 Epoll 接口相关函数原型的定义
#include <sys/time.h>       //提供了 time 函数原型的定义
#include <sys/resource.h>/* 为资源操作提供了定义,包括在一个程序允许的尺寸,执行的优先级
                以及文件上确定和设置限制的函数的原型定义 */
#define MAXBUF 1024
#define MAXEPOLLSIZE 10000
                //setnonblocking - 设置句柄为非阻塞方式
int setnonblocking(int sockfd){
```

```
    if(fcntl(sockfd,F_SETFL,fcntl(sockfd,F_GETFD,0)|O_NONBLOCK)==-1){
        return -1;
    }
    return 0;
}
                                //handle_message - 处理每个 socket 上的消息收发
int handle_message(int new_fd){
    char buf[MAXBUF+1];
    int len;
                                //开始处理每个新连接上的数据收发
    bzero(buf,MAXBUF+1);
                                //接收客户端的消息
    len=recv(new_fd,buf,MAXBUF,0);
    if(len>0){
        printf("%d 接收消息成功:'%s',共%d 字节的数据\n",
            new_fd,buf,len);
    }
    else{
        if(len<0)
            printf("消息接收失败! 错误代码是%d,错误信息是'%s'\n",
                errno,strerror(errno));
        close(new_fd);
        return -1;
    }
                                //处理每个新连接上的数据收发结束
    return len;
}
int main(int argc,char **argv){
    int listener,new_fd,kdpfd,nfds,n,ret,curfds;
    socklen_t len;
    struct sockaddr_in my_addr,their_addr;
    unsigned int myport,lisnum;
    struct epoll_event ev;
    struct epoll_event events[MAXEPOLLSIZE];
    struct rlimit rt;
    myport=5000;
    lisnum=2;
                                //设置每个进程允许打开的最大文件数
    rt.rlim_max=rt.rlim_cur=MAXEPOLLSIZE;
    if(setrlimit(RLIMIT_NOFILE,&rt)==-1){
        perror("setrlimit");
        exit(1);
}
```

```
else {
    printf("设置系统资源参数成功!\n");
}
                                //开启 socket 监听
if((listener = socket(PF_INET,SOCK_STREAM,0)) ==-1){
    perror("socket");
    exit(1);
}
else{
    printf("socket 创建成功!\n");
}
setnonblocking(listener);
bzero(&my_addr,sizeof(my_addr));
my_addr.sin_family = PF_INET;
my_addr.sin_port = htons(myport);
my_addr.sin_addr.s_addr = INADDR_ANY;
if(bind(listener,(struct sockaddr *)&my_addr,sizeof(struct sockaddr)) ==-1){
    perror("bind");
    exit(1);
}
else{
    printf("IP 地址和端口绑定成功\n");
}
if(listen(listener,lisnum) ==-1){
    perror("listen");
    exit(1);
}
else{
    printf("开启服务成功!\n");
}
                                //创建 epoll 句柄,把监听 socket 加入到 epoll 集合里
kdpfd = epoll_create(MAXEPOLLSIZE);
len = sizeof(struct sockaddr_in);
ev.events = EPOLLIN|EPOLLET;
ev.data.fd = listener;
if(epoll_ctl(kdpfd,EPOLL_CTL_ADD,listener,&ev)<0){
    fprintf(stderr,"epoll set insertion error:fd = %d\n",listener);
    return -1;
}
else{
    printf("监听 socket 加入 epoll 成功!\n");
}
curfds = 1;
```

```
while(1){
                        //等待有事件发生
    nfds = epoll_wait(kdpfd,events,curfds, -1);
    if(nfds == -1){
        perror("epoll_wait");
        break;
    }
                        //处理所有事件
        for(n=0;n<nfds; ++n){
            if(events[n].data.fd == listener){
                new_fd = accept(listener,(struct sockaddr *)&their_addr,&len);
                if(new_fd<0){
                    perror("accept");
                    continue;
                }
                else{
                    printf("有连接来自于:%d:%d,分配的 socket 为:%d\n",
                        inet_ntoa(their_addr.sin_addr),ntohs(their_addr.sin_port),new_fd);
                }
                setnonblocking(new_fd);
                ev.events = EPOLLIN|EPOLLET;
                ev.data.fd = new_fd;
                if(epoll_ctl(kdpfd,EPOLL_CTL_ADD,new_fd,&ev)<0){
                    fprintf(stderr,"把 socket '%d'加入 epoll 失败!%s\n",
                        new_fd,strerror(errno));
                    return -1;
                }
                curfds ++;
            }
            else{
                ret = handle_message(events[n].data.fd);
                if(ret<1 && errno != 11){
                    epoll_ctl(kdpfd,EPOLL_CTL_DEL,events[n].data.fd,&ev);
                    curfds --;
                }
            }
        }
    }
    close(listener);
    return 0;
}
```

9.5 本章小结

本章主要对基于 Epoll 的并发的面向连接服务器及其实现方法进行深入介绍，并在实现原理介绍的基础上，具体给出一个创建基于 Epoll 的并发的面向连接服务器的完整 C 语言例程。通过本章的学习，需要了解基于 Epoll 的并发的面向连接服务器的进程结构；需要熟习基于 Epoll 的并发的面向连接服务器的设计流程；需要掌握基于 Epoll 的并发的面向连接服务器的 C 语言实现方法。

本章习题

1. 简述基于 Epoll 的并发的面向连接服务器的进程结构。
2. 简述基于 Epoll 的并发的面向连接服务器软件的设计流程。
3. 试构造一个基于 Epoll 的并发的面向连接服务器例程，该例程能实现以下功能：能同时等候来自 10 个不同客户的连接请求，一旦与某个客户连接成功则接收来自该客户的信息，当每收到一个字符串时将首先显示该字符串，然后再将该字符串反转、最后再将反转后的字符串回送给该客户。

第 10 章　客户进程中的并发机制

第 4 ~ 9 章介绍了服务器中的并发机制及其实现方法，本章将对客户进程中的并发机制及其实现方法进行深入介绍，并在实现原理介绍的基础上，具体给出两个实现并发客户的完整 C 语言例程。

10.1　实现并发客户的意义与进程结构

10.1.1　实现并发客户的意义

在开发客户 - 服务器体系结构的系统过程中，由于以下原因，使得设计人员往往重视服务器端的并发设计：

① 并发可改善观察到的时间，从而改善了全部客户机的总吞吐量；

② 并发可排除潜在的死锁；

③ 并发实现使得设计人员易于创建多协议的或多服务的服务器；

④ 使用多进程实现并发非常灵活，因为这样就可以在多种硬件平台上很好地运行。当把并发实现移植到具有多个处理器的计算机时，可以得到更高的工作效率，因为可以充分利用额外的处理能力而不需要改变代码。

由于客户机通常在一个时刻只进行一种活动，客户机一旦向服务器发送了一个请求，在收到响应之前一般不需进行其他活动，因而客户机似乎不能从并发中受益。此外，客户机的效率和死锁问题也不如服务器那样严重，因为如果一个客户机延缓或停止执行，它只是自己停止了，而其他的客户机将继续运行。然而，尽管表面上如此，由于以下原因，在客户机中采用并发确实有其优点：

- 由于功能已被划分为概念上能分开的一些部分，并发实现更容易编程；
- 由于代码已经模块化了，并发实现可使得维护和扩展变得更容易；
- 并发客户机可在同一时刻联系多个服务器，往往需要比较响应时间或合并服务器返回的结果；
- 并发允许用户改变参数，查询客户机状态，或动态地控制处理；
- 在客户机中使用并发的最主要的优点在于异步性。异步性允许客户机同时处理多个请求，且不严格规定其执行顺序。

由以上描述可见，并发执行提供了一个强有力的工具。并发客户机实现不但可提供更快的响应时间，而且还可避免死锁问题，帮助程序员将控制和状态处理从正常的处理中分离出来。

通常可采取两个级别的并发（进程级并发和线程级并发），来实现客户端的并发。由于采用并发进程所需消耗的系统资源相对较大，创建和管理并发进程系统开销大，而线程的创建和管理系统付出的开销小。因此，在支持多线程的环境下，应尽量采用基于多线程的方法

来实现客户端的并发，这样能进一步降低 CPU 的负荷和整个系统的开销。

10.1.2 基于多线程/多进程的并发客户的进程结构

LINUX 系统支持一个进程中有多个线程共享内存，图 10.1 给出了在 LINUX 系统中如何使用多线程/多进程的并发客户方法来支持面向连接的应用协议。

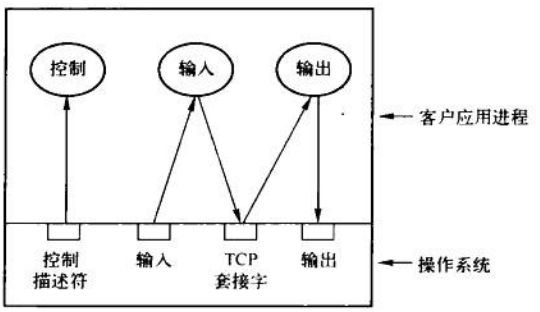


图 10.1 面向连接的基于多线程/多进程的并发客户的一种可能的进程结构

由图 10.1 可知，基于多线程/多进程的并发客户应用允许客户把输入和输出处理分开。其中，利用一个独立的线程/进程（输入线程/进程）从标准输入读入数据，以形成请求，并通过 TCP 连接发送给服务器；然后再利用另一个独立的线程/进程（输出线程/进程）从服务器接收响应，并写入到标准输出；同时还可以再利用第三个独立的线程/进程（控制线程/进程）从控制处理的用户那里接收命令。

10.1.3 基于单线程的并发客户的进程结构

基于单线程的并发客户与基于单线程的并发服务器一样，使用异步 I/O。客户为得到多个服务器的连接创建套接字描述符。同时，它还可以有一个或多个用于获得键盘或鼠标输入的描述符。客户程序的主体含有一个循环，该循环使用 select 等待其中任何一个描述符准备就绪。如果输入描述符已准备就绪，客户就读取输入，并且可以将输入存储起来以后再用，也可以立刻开始处理输入。如果 TCP 连接输出就绪，客户就在此 TCP 连接上准备和发送请求。如果 TCP 连接输入就绪，客户就读取这个服务器发出的响应并加以处理。图 10.2 给出

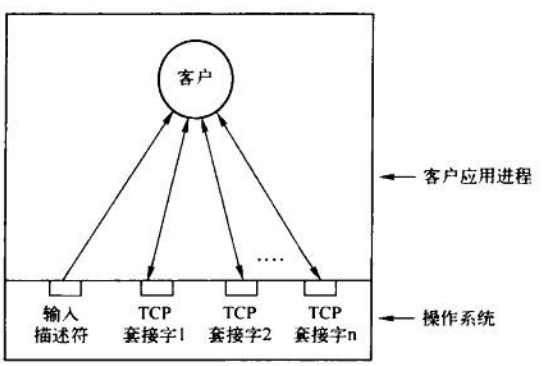


图 10.2 面向连接的基于单线程的并发客户的一种可能的进程结构

了在 LINUX 系统中如何使用单线程的并发客户方法来支持面向连接的应用协议。

如果单线程的客户调用会阻塞的系统功能，它就可能转为死锁状态。因此程序员须注意确保客户不会无限期地阻塞——在那里等待不会发生的事件。

10.2　基于多线程的并发客户例程

```
#include <netinet/in.h>        //包含了数据结构 sockaddr_in 的定义
#include <sys/types.h>         //基本数据类型头文件,含有基本系统数据类型的定义
#include <sys/socket.h>        //提供了套接字函数原型与数据结果的定义
#include <stdio.h>             /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()
                                  和 printf()等)的定义*/
#include <stdlib.h>            /*C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()等)
                                  原型的定义*/
#include <string.h>            //提供了字符串函数原型的定义
#include <pthread.h>           //提供了线程函数原型与数据结果的定义
#include <sys/errno.h>         //提供错误号 errno 的定义,用于错误处理
#define HELLO_WORLD_SERVER_PORT 6666
#define BUFFER_SIZE 1024
char * server_IP = NULL;
void * talk_to_server(void * thread_num){
    //设置一个 socket 地址结构 client_addr,代表客户机 internet 地址,端口
    struct sockaddr_in client_addr;
    bzero(&client_addr,sizeof(client_addr));    //把内存区的内容全部置为 0
    client_addr.sin_family = AF_INET;           //internet 协议族
    client_addr.sin_addr.s_addr = htons(INADDR_ANY);    //自动获取本机地址
    client_addr.sin_port = htons(0);            //表示让系统自动分配一个空闲端口
    //创建用于流协议(TCP)的 socket,用 client_socket 代表客户机 socket
    int client_socket = socket(AF_INET,SOCK_STREAM,0);
    if(client_socket < 0){
        printf("Create Socket Failed!\n");
        exit(1);
    }
    //把客户机的 socket 和客户机的 socket 地址结构联系起来
    if(bind(client_socket,(struct sockaddr *)&client_addr,sizeof(client_addr))){
        printf("Client Bind Port Failed!\n");
        exit(1);
    }
    //设置一个 socket 地址结构 server_addr,代表服务器的地址和端口
    struct sockaddr_in server_addr;
    bzero(&server_addr,sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    if(inet_aton(server_IP,&server_addr.sin_addr) == 0){
```

```
        printf("Server IP Address Error!\n");
        exit(1);
        }
        server_addr.sin_port = htons(HELLO_WORLD_SERVER_PORT);
        socklen_t server_addr_length = sizeof(server_addr);
        //向服务器发起连接
        if(connect(client_socket,(struct sockaddr *)&server_addr,server_addr_length) <0) {
            printf("Can Not Connect To %s!\n",server_IP);
            exit(1);
        }
        char buffer[BUFFER_SIZE];
        bzero(buffer,BUFFER_SIZE);
        //从服务器接收数据到 buffer 中
        int length = recv(client_socket,buffer,BUFFER_SIZE,0);
        if(length <0) {
            printf("Recieve Data From Server %s Failed!\n",server_IP);
            exit(1);
        }
        printf("From Server %s:\t%s",server_IP,buffer);
        bzero(buffer,BUFFER_SIZE);
        sprintf(buffer,"Hello,World! From Client Thread:\t%d\n",(int)thread_num);
        //向服务器发送 buffer 中的数据
        send(client_socket,buffer,BUFFER_SIZE,0);
        //关闭 socket
        close(client_socket);
        pthread_exit(NULL);
}

int main(int argc,char * * argv) {
    if (argc !=2) {
        printf("Usage: ./%s ServerIPAddress\n",argv[0]);
        exit(1);
    }
    server_IP = argv[1];
    pthread_t child_thread;
    pthread_attr_t child_thread_attr;
    pthread_attr_init(&child_thread_attr);
    pthread_attr_setdetachstate(&child_thread_attr,PTHREAD_CREATE_DETACHED);
    int i =0;
    for(i =0; i <10000; i ++) {
        if(pthread_create(&child_thread,&child_thread_attr,talk_to_server,(void *) i) <0 )
        //创建 10000 个新线程与服务器进行通信
        printf("pthread_create Failed: %s\n",strerror(errno));
```

```
    }
    return 0;
}
```

10.3　基于单线程的并发客户例程

```
#include <sys/types.h>          //基本数据类型头文件,含有基本系统数据类型的定义
#include <sys/param.h>          //定义了常数 NOFILE
#include <sys/ioctl.h>          //包含了 I/O 控制函数的定义
#include <sys/time.h>           //提供了 time 函数原型的定义
#include <sys/socket.h>         //提供了套接字函数原型与数据结果的定义
#include <unistd.h>             /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数
                                   据结构的定义 */
#include <stdlib.h>             /* C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()等)
                                   原型的定义 */
#include <stdio.h>              /* 标准输入/输出头文件,包含了标准输入/输出函数(如 perror()
                                   和 printf()等)的定义 */
#include <string.h>             //提供了字符串函数原型的定义
extern int errno;
int TCPtecho(fd_set * pafds,int nfds,int ccount,in hcount);
int reader(int fd,fd_set * pfdset);
int writer(int fd,fd_set * pfdset);
int errexit(const char * format,...);
int connectTCP(const char * host,const char * service);
long mstime(unsigned long *);
#define BUFSZIE 4096
#define CCOUNT 64 * 1024
#define USAGE "usage: TCPtecho [ -c count] host1 host2 ... \n"
char * hname[NOFILE];
int rc[NOFILE],wc[NOFILE];
char buf[BUFSZIE];
int main(int argc,char * argv[]) {
    int ccount = CCOUNT;
    int I,hcount,maxfd,fd;
    int one = 1;
    fd_set afds;
    hcount = 0;
    maxfd = -1;
    for(i = 1; i < argc; ++i) {
        if(strcmp(argv[i]," -c") == 0) {
            if( ++i < argc&&(ccount = atoi(argv[i])))
                continue;
```

```
            errexit(USAGE);
        }
        fd = connectTCP(argv[i],"echo");   /* 调用 connectTCP()建立与客户端的连接并创建
                                              主套接字 fd */
        if(ioctl(fd,FIONBIO,(char *)&one))
            errexit("can't mark socket nonblocking: %s\n",strerror(errno));
        if(fd > maxfd)              //更新最大套接字描述符的值
            maxfd = fd;
        hname[fd] = argv[i];
        ++hcount;
        FD_SET(fd,&afds);     //将套接字描述符 fd 写入到 afds
    }
    TCPtecho(&afds,maxfd + 1,ccount,hcount);
    exit(0);
}

int TCPtecho(fd_set *pafds,int nfds,int ccount,int hcount) {
    fd_set rfds,wfds;
    fd_set rcfds,wcfds;
    int fd,i;
    for (i=0; i< BUFSIZE; ++i)
        buf[i] = 'D';
    memcpy(&rcfds,pafds,sizeof(rcfds))'
    memcpy(&wcfds,pafds,sizeof(wcfds));
    for(fd=0; fd<nfds; ++fd)
        rc[fd] = wc[fd] = ccount;
    while(hcount){
        memcpy(&rfds,&rcfds,sizeof(rfds));
        memcpy(&wfds,&wcfds,sizeof(wfds));
        if(select(nfds,&rfds,&wfds,(fd_set *)0,(struct timeval *)0)<0)
            errexit("select failed: %s\n",strerror(errno));
        for(fd=0; fd<nfds; ++fd)
            if(FD_ISSET(fd,&rfds))          /* 若只读套接字描述符集合 rfds 中的套接字 fd 已
                                               经准备就绪 */
                if(reader(fd,&rcfds)==0)    /* 则调用 reader()函数从套接字 fd 中读取来自对
                                               应客户端的数据 */
                    hcount--;
            if(FD_ISSET(fd,&wfds))          /* 若只写套接字描述符集合 wfds 中的套接字 fd 已
                                               经准备就绪 */
                writer(fd,&wcfds);          //则调用 writer()函数将数据写入到套接字 fd 中
```

```
int reader(int fd,fd_set * pfdset) {
    unsigned long now;
    int cc;
    cc = read(fd,buf,sizeof(buf));      //调用 read()函数从套接字 fd 中读取数据
    if(cc <0)
        errexit("read: %s\n",strerror(errno));
    if(cc =0)
        errexit("read: premature end of file\n");
    rc[fd] - =cc;
    if(rc[fd])
        return 1;
    (void) mstime(&now);
    printf("%s: %d ms\n",hname[fd],now);
    (void) close(fd);
    FD_CLR(fd,pfdset);
    Return 0;
}

int writer(int fd,fd_set * pfdset) {
    int cc;
    cc = write(fd,buf,MIN((int)sizeof(buf),wc[fd]));   /* 调用 write()函数将缓存区 buf 中的
                                                          数据写入到套接字 fd 中 */
    if(cc <0)
        errexit("read: %s\n",strerror(errno));
    wc[fd] - =cc;
    if(wc[fd] ==0){
        (void)shutdown(fd,1);
        FD_CLR(fd,pfdset);
    }
}
```

10.4　基于多进程的并发客户例程

```
#include <sys/types.h>          //基本数据类型头文件,含有基本系统数据类型的定义
#include <sys/signal.h>         //提供了 signal 函数原型的定义
#include <sys/socket.h>         //提供了套接字函数原型与数据结构的定义
#include <sys/time.h>           //提供了 time 函数原型的定义
#include <sys/resource.h>       /* 为资源操作提供了定义,包括在一个程序允许的尺寸,执行的
                                   优先级以及文件上确定和设置限制的函数的原型定义 */
#include <sys/wait.h>           //提供了 wait 函数原型的定义
#include <sys/errno.h>          //提供错误号 errno 的定义,用于错误处理
#include <netinet/in.h>         //包含了数据结构 sockaddr_in 的定义
#include <unistd.h>             /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和
```

```
                                数据结构的定义 */
#include <stdlib.h>             /* C 标准库头文件,包含了 C 语言标准库函数(如 exit()、atoi()
                                等)原型的定义 */
#include <stdio.h>              /* 标准输入/输出头文件,包含了标准输入/输出函数(如 perror()
                                和 printf()等)的定义 */
#include <string.h>             //提供了字符串函数原型的定义
#define PORT 3490
#define MAXDATASIZE 100
int main(){
    int sockfd,numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in their_addr;
    char argv[15];
    int n;
    printf("请输入服务器 IP 地址:\n");
    scanf("%s",argv);
    if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1){
        perror("socket 创建失败");
        exit(1);
    }
    their_addr.sin_family=AF_INET;
    their_addr.sin_port=htons(PORT);
    if(inet_aton(argv,&their_addr.sin_addr)==0){
        perror("输入 ip 格式错误!");
        exit(1);
    }
    bzero(&(their_addr.sin_zero),8);
    if(connect(sockfd,(struct sockaddr*)&their_addr,sizeof(struct sockaddr))==-1){
        perror("连接服务器失败");
        exit(1);
    }

    if((numbytes=recv(sockfd,buf,MAXDATASIZE,0))==-1){
        perror("与服务器连接失败");
        exit(1);
    }
    buf[numbytes]='\0';
    printf("%s,你已经可以向服务器发送信息了!\n",buf);
    printf("请输入信息后按回车或空格键即可发送,退出请按 Ctrl+C!\n");
    while(1){
        scanf("%s",buf);            //主进程负责获得用户的键盘输入
        n=strlen(buf);
```

```
        if (!fork()) {                    //创建新进程负责处理与服务器之间的通信
            if (send(sockfd,buf,n+1,0) == -1) {
                perror("发送失败");
                exit(1);
            }
            printf("发送成功,信息已发送到服务器!\n");
        }
    }
    close(sockfd);                        //关闭套接字描述符 sockfd
    return 0;
}
```

10.5　本章小结

本章主要对客户进程中的并发机制及其实现方法进行深入介绍，并在实现原理介绍的基础上，具体给出两个实现并发客户的完整 C 语言例程。通过本章的学习，需要了解基于多线程的并发客户的进程结构；需要熟悉基于多线程的并发客户的设计流程；需要掌握基于单线程/多线程的并发客户的 C 语言实现方法。

本章习题

1. 简述基于多线程的并发客户的进程结构。

2. 简述基于多线程的并发客户软件的设计流程。

3. 试构造一个基于多线程的并发客户例程，该例程能实现以下功能：能在利用一个独立的线程（输入线程）从标准输入读入数据，以形成请求，并通过 TCP 连接发送给服务器；同时，还可利用另一个独立的线程（输出线程）从服务器接收响应，并写入到标准输出。

4. 试构造一个基于单线程的并发客户例程，该例程能实现以下功能：能从标准输入读入数据，以形成请求，并通过 TCP 连接发送给服务器；同时，还可从服务器接收响应，并写入到标准输出。

第 11 章　客户 - 服务器系统中的死锁问题

在前面的章节中详细介绍了客户与服务器进程的并发机制及其实现方法，本章将对客户 - 服务器系统中的死锁问题进行深入介绍，并在分析导致死锁问题的相关原理基础上，具体给出两个完整的 C 语言例程以进一步说明客户 - 服务器系统中所存在的死锁问题。

11.1　死锁的定义

所谓死锁，就是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去，此时，称系统处于死锁（Dead Lock）状态或系统产生了死锁，而这些永远在互相等待的进程则称之为死锁进程。由于资源占用是互斥的，当某个进程提出申请资源后，使得有关进程在无外力协助下，将会因为永远分配不到所必需的资源而无法继续运行下去，这就产生了死锁这一特殊现象。

在实际生活中也存在死锁的例子，例如：假定在一条河上有一座桥，桥面较窄，只能容纳一个人通过，无法让两个人并行；此时，如果有两个人 A 和 B 分别由桥的两端走上该桥，对于 A 来说，它走过了桥面左端的一段路（即占有了桥的一部分资源），要想过桥则必须等待 B 让出右端的桥面，此时 A 不能继续前进；而对于 B 来说，他也走过了桥面右端的一段路（即占有了桥的一部分资源），要想过桥则还须等待 A 让出左端的桥面，此时 B 也不能继续前进。若两端的人都不倒退，结果将造成互相等待对方让出桥面的局面，此时，若双方谁也不让路，则都将无休止地等下去。这种现象就是死锁。

如果把人比做进程，桥面作为资源，那么上述问题就可描述为：进程 A 占有了系统资源 R_1，等待进程 B 占有的系统资源 R_2；进程 B 占有了资源 R_2，等待进程 A 占有的资源 R_1；且资源 R_1 和 R_2 均在同一时间内只允许被一个进程占用，即，不允许被两个进程同时占用。这样一来，结果将使得两个进程都不能继续执行，且若不采取其它措施，这种循环等待状况将会无限期持续下去，从而导致了进程死锁的发生。

11.2　产生死锁的原因

11.2.1　竞争资源引起进程死锁

在两个或多个任务中，如果每个任务都锁定了其他任务试图锁定的资源，则此时将会造成这些任务的永久阻塞，从而出现死锁。例如：事务 A 获取了对资源 1 的共享锁。事务 B 获取了对资源 2 的共享锁。此时，若事务 A 请求对资源 2 的排他锁，则显然将会在事务 B 完成并释放其对资源 2 持有的共享锁之前被阻塞。与此同时，若事务 B 亦请求对资源 1 的排

他锁，同理，也将在事务 A 完成并释放其对资源 1 持有的共享锁之前被阻塞。现在的情形就变成了事务 A 需要在事务 B 完成之后才能完成，而事务 B 则需要在事务 A 完成之后才能完成。即事务 A 与事务 B 之间变成了一种循环依赖关系：事务 A 依赖于事务 B，事务 B 依赖于事务 A。显然，除非某个外部进程断开死锁，否则死锁中的两个事务都将无限期等待下去。

11.2.2　进程推进顺序不当引起死锁

由于进程在运行中具有异步性特征，这可能使 P1、P2、P3 三个进程在获取 S1，S2，S3 三个资源时会按下述两种顺序向前推进。

（1）合法的进程推进顺序：P1：Relese（S1）；Request（S3）；⇒P2：Relese（S2）；Request（S1）；⇒P3：Relese（S3）；Request（S2）。此时，这三个进程的推进顺序是合法的，不会引起进程死锁。

（2）非法的进程推进顺序：P1：Request（S3）；Relese（S1）；⇒P2：Request（S1）；Relese（S2）；⇒P3：Request（S2）；Relese（S3）。此时，由于 P1 保持了资源 S1，P2 保持了资源 S2，P3 保持了资源 S3，则当上述三个进程再向前推进时，即：当 P1 运行到 P1：Request（S3）时，将会因为资源 S3 已被 P3 占用而阻塞；当 P2 运行到 P2：Request（S1）时，将会因为资源 S1 已被 P1 占用而阻塞；而当 P3 运行到 P3：Request（S2）时，则将会因为资源 S2 已被 P2 占用而阻塞；于是发生进程死锁。

11.3　产生死锁的必要条件

虽然进程在运行的过程之中可能会发生死锁，但死锁的发生也必须具备以下四个必要条件。

① 互斥条件：指进程对所分配到的资源进行排他性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其他进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。如独木桥就是一种独占资源，两方的人不能同时过桥。

② 请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其他进程占有，此时请求进程阻塞，但又对自己已获得的其他资源保持不放。下面仍以过独木桥为例。A、B 在桥上相遇，A 走过一段桥面（即占有了一些资源），还需要走其余的桥面（申请新的资源），但那部分桥面被 B 占有（B 走过一段桥面）。A 过不去，既不能前进又不后退；而 B 也处于同样的状况。

③ 不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。例如：过独木桥的人不能强迫对方后退，也不能非法地将对方推下桥，必须是桥上的人自己过桥后空出桥面（即主动释放占有资源），对方的人才能过桥。

④ 环路等待条件：指在发生死锁时，必然会存在一个进程——资源的环形链。即：进程集合 $\{P_0, P_1, P_2, \cdots, P_n\}$ 中的 P_0 正在等待一个 P_1 所占用的资源；P_1 正在等待 P_2 所占用的资源，……，P_n 正在等待已被 P_0 所占用的资源。如前面的过独木桥问题，A 等待 B 占有的桥面，而 B 又等待 A 占有的桥面，从而彼此循环等待。

11.4 处理死锁的基本方法

在系统中已经出现死锁后，应该及时检测到死锁的发生，并采取适当的措施来解除死锁。目前处理死锁的方法可归结为以下四种。

（1）预防死锁

这是一种较简单和直观的事先预防的方法。方法是通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或者几个，来预防发生死锁。预防死锁是一种较易实现的方法，已被广泛使用。常用的死锁预防方法有如下几种。

① 打破互斥条件。即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机，等等，这是由资源本身的属性所决定的。所以，这种办法并无实用价值。

② 打破不可抢占条件。即允许进程强行从占有者那里夺取某些资源。也就是说，当一个进程已经占有了某些资源，当它又申请新的资源，但不能立即被满足时，它必须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其他进程。这就相当于该进程占有的资源被隐蔽地强占了。这种预防死锁的方法实现起来困难，会降低系统性能。

③ 打破占有且申请条件。可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。但是，这种策略也有一些缺点，例如：在许多情况下，一个进程在执行之前不可能知道它所需要的全部资源。这是由于进程在执行时是动态的，不可预测的；另外，该策略的资源利用率低，无论所分资源何时用到，一个进程只有在占有所需的全部资源后才能执行。即使有些资源最后才被该进程用到一次，但该进程在生存期间却一直占有它们，造成长期占着不用的状况，这显然是一种极大的资源浪费；此外，该策略也降低了进程的并发性，因为资源有限，又加上存在浪费，因此使得能分配到所需全部资源的进程个数就必然少了。

④ 打破循环等待条件，实行资源有序分配策略。采用这种策略，即把资源事先分类编号，按号分配，使进程在申请、占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁。这种策略与前面的策略相比，资源的利用率和系统吞吐量都有很大提高，但是也存在一些缺点，例如：限制了进程对资源的请求，同时给系统中所有资源合理编号也是件困难事，并增加了系统开销；其次，为了遵循按编号申请的次序，暂不使用的资源也需要提前申请，从而增加了进程对资源的占用时间。

（2）避免死锁

该方法同样是属于事先预防的策略，但它并不须事先采取各种限制措施去破坏产生死锁的四个必要条件，而是在资源的动态分配过程中，用某种方法去避免发生死锁。

代表性的死锁避免方法有“有序资源分配法”：在该算法中，首先按某种规则将系统中的所有资源统一编号（例如：打印机为1、磁带机为2、磁盘为3，……），然后，进程必须以上升的次序来申请这些资源。即，系统要求申请进程按照以下规则来进行资源申请。

① 进程对它所必须使用的且属于同一类的所有资源，必须一次申请完毕。

② 进程在申请不同类资源时，必须按各类资源的编号来依次申请。例如，进程 P_A 使用资源的顺序是 R_1，R_2；进程 P_B 使用资源的顺序是 R_2，R_1；若采用动态分配的方法则有可能形成环路条件，造成死锁。但采用有序资源分配法：R_1 的编号为 1，R_2 的编号为 2；则 P_A 的申请次序应是 R_1，R_2；P_B 的申请次序也应是 R_1，R_2；这样一来就破坏了环路产生的条件，从而避免了死锁的发生。

(3) 检测死锁

这种方法并不须事先采取任何限制性措施，也不必检查系统是否已经进入不安全区，此方法允许系统在运行过程中发生死锁。但可通过系统所设置的检测机构，及时地检测出死锁的发生，并精确地确定与死锁有关的进程和资源，然后采取适当措施，从系统中将已发生的死锁清除掉。

(4) 解除死锁

这是与检测死锁相配套的一种措施。当检测到系统中已发生死锁时，须将进程从死锁状态中解脱出来。常用的实施方法是撤销或挂起一些进程，以便回收一些资源，再将这些资源分配给已处于阻塞状态的进程，使之转为就绪状态，以继续运行。死锁的检测和解除措施，有可能使系统获得较好的资源利用率和吞吐量，但在实现上难度也最大。常用的死锁解除方法有如下几种。

① 撤销陷于死锁的全部进程；

② 逐个撤销陷于死锁的进程，直到死锁不存在；

③ 从陷于死锁的进程中逐个强迫放弃所占用的资源，直至死锁消失；

④ 从另外一些进程那里强行剥夺足够数量的资源分配给死锁进程，以解除死锁状态。

11.5　存在死锁问题的多线程例程

```
#include <stdio.h>              /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()
                                  和 printf()等)的定义*/
#include <sys/types.h>          //基本数据类型头文件,含有基本系统数据类型的定义
#include <unistd.h>             /* LINUX 标准头文件,包含了各种 LINUX 系统服务函数原型和数
                                  据结构的定义*/
#include <ctype.h>              //包含了字符串测试函数和字符大小转化函数原型的定义
#include <pthread.h>            //包含了各种线程相关函数原型和数据结构等的定义
#define LOOP_TIMES 10000  //定义程序循环执行的次数为 10 000 次
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
                                //初始化互斥锁 pthread_mutex_t mutex1
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
                                //初始化互斥锁 pthread_mutex_t mutex2
void * thread_worker(void *); //声明函数 thread_worker()
void critical_section(int thread_num,int i); //声明函数 critical_section()
int main(void){
    int rtn,i;
```

```
    pthread_t pthread_id = 0;      //定义并初始化用于存放子线程 ID 的变量 pthread_id
    rtn = pthread_create(&pthread_id, NULL, thread_worker, NULL);   //创建子线程
    if(rtn != 0) {                 //若创建子线程出错,则提示出错信息
        printf("pthread_create ERROR!\n");
        return -1;
    }
    for(i = 0; i < LOOP_TIMES; i++) {     //主线程循环执行 critical_section()函数功能
        pthread_mutex_lock(&mutex1);      //对互斥锁 pthread_mutex_t mutex1 上锁
        pthread_mutex_lock(&mutex2);      //对互斥锁 pthread_mutex_t mutex2 上锁
        critical_section(1, i);           //执行 critical_section()函数功能
        pthread_mutex_unlock(&mutex2);    //对互斥锁 pthread_mutex_t mutex1 解锁
        pthread_mutex_unlock(&mutex1);    //对互斥锁 pthread_mutex_t mutex2 解锁
    }
    pthread_mutex_destroy(&mutex1);       //销毁互斥锁 pthread_mutex_t mutex1
    pthread_mutex_destroy(&mutex2);       //销毁互斥锁 pthread_mutex_t mutex2
    return 0;
}

void * thread_worker(void * p) {          //子线程的线程体函数
    int i;
    for(i = 0; i < LOOP_TIMES; i++) {     //子线程循环执行 critical_section()函数功能
        pthread_mutex_lock(&mutex2);      //对互斥锁 pthread_mutex_t mutex2 上锁
        pthread_mutex_lock(&mutex1);      //对互斥锁 pthread_mutex_t mutex1 上锁
        critical_section(2, i);
        pthread_mutex_unlock(&mutex2);    //对互斥锁 pthread_mutex_t mutex2 解锁
        pthread_mutex_unlock(&mutex1);    //对互斥锁 pthread_mutex_t mutex1 解锁
    }
}

void critical_section(int thread_num, int i) {
    printf("thread%d: %d\n", thread_num, i);
}
```

显然，依据 11.2 节的描述与分析可知，上述例程将会在运行过程中导致死锁现象的发生。

11.6 本章小结

本章主要对客户 - 服务器系统中的死锁问题进行深入介绍，并在死锁发生原因介绍的基础上，具体给出处理死锁的基本方法以及两个将导致死锁发生的的完整 C 语言例程。通过本章的学习，需要了解客户 - 服务器系统中发生死锁问题的原因；需要掌握处理死锁的基本方法。

本章习题

1. 什么是客户 – 服务器系统中的死锁问题?
2. 简述产生客户 – 服务器系统中的死锁问题的主要原因。
3. 简述处理死锁的基本方法。
4. 请修改 11.5 节中给出的例程，以消除其中所存在的死锁问题。

第 12 章　GCC 编译器简介

GCC 是一个交叉平台的编译器，它不仅仅可以支持 C 语言，还可以支持 Ada 语言、C ++ 语言、Java 语言、Objective C 语言，Pascal 语言、COBOL 语言，以及支持函数式编程和逻辑编程的 Mercury 语言等，是目前 LINUX 下最重要的编译工具之一。本章将详细介绍 GCC 编译器的安装和使用方法。

12.1　GCC 编译器所支持的源程序格式

GCC 是一组编译工具的总称，其软件包里包含众多的工具，按其类型，主要有以下的分类：

- C 编译器 cc，cc1，cc1 plus，gcc；
- C ++ 编译器 c ++，cc1 plus，g ++；
- 源代码预处理程序 cpp，cpp0；
- 库文件 libgcc. a，libgcc_eh. a，libgcc_s. so，libiberty. a，libstdc ++. [a，so]，libsupc ++. a。

用 GCC 编译程序生成可执行文件有时候看起来似乎仅通过编译一步就完成了，但事实上，使用 GCC 编译工具由 C 语言源程序生成可执行文件的过程并不单单是一个编译的过程，而要经过四个过程：预处理（Pre - Processing），编译（Compiling），汇编（Assembling），链接（Linking）。

目前，GCC 编译器所支持的源程序的格式如表 12.1 所示。

表 12.1　GCC 编译器所支持的源程序格式

后缀格式	说　明	后缀格式	说　明
.c	C 语言程序	.ii	经过预处理的 C ++ 程序
.a	由目标文件构成的档案文件	.m	Objective - C 源程序
.C、cc、cxx	C ++ 源程序	.o	编译后的目标文件
.h	源程序所包含的头文件	.s	汇编语言源程序
.i	经过预处理的 C 程序	.S	经过预编译的汇编程序

12.2　GCC 编译选项解析

12.2.1　GCC 编译选项分类

GCC 是 LINUX 下基于命令行的 C 语言编译器，其基本的使用语法如下：

```
gcc [option |filename]...
```

其中，option 为 GCC 使用时的选项，filename 为需要 GCC 做编译的处理的文件名。就 GCC 来说，其本身是一个十分复杂的命令，合理地使用其命令选项可以有效地提高程序的编译效率、优化代码。GCC 拥有为数众多的命令选项，有超过 100 个的编译选项可用，具体可分类如下。

1. 常用编译选项

（1）－c 选项

该选项告诉 GCC 编译器仅把源程序编译为目标代码而不做链接工作，所以采用该选项的编译指令不会生成最终的可执行程序，而是生成一个与源程序文件名相同的以 .o 为后缀的目标文件。例如，一个 Test. c 的源程序经过下面的编译之后会生成一个 Test. o 文件：

```
# gcc -c Test. h
```

（2）－S 选项

使用该选项会生成一个后缀名为 .s 的汇编语言文件，但是同样不会生成可执行程序。

（3）－e 选项

该选项只对文件进行预处理，预处理的输出结果被送到标准输出（比如显示器）。

（4）－v 选项

在 Shell 的提示符号下输入 gcc －v 命令，屏幕上就会显示出目前正在使用的 GCC 版本的信息。

（5）－x language

该选项强制编译器用指定的语言编译器来编译某个源程序。例如下面的指令：

```
# gcc -x c++ p1. c
```

该指令表示强制 GCC 编译器采用 C ++ 编译器来编译 C 程序 P1. c。

（6）－I <DIR > 选项

该选项用于指定头文件所在的路径。在 LINUX 下开发程序的时候，通常都需要借助一个或多个函数库的支持才能够完成相应的功能。一般情况下，LINUX 下的大多数函数都将头文件放到系统/usr/include 目录下，而库文件则放到/usr/lib 目录下。但在有些情况下并不是这样的，在这些情况下，使用 GCC 编译时必须指定所需要的头文件和库文件所在的路径。－I 选项可以向 GCC 的头文件搜索路径中添加新的目录 <DIR >。例如，一个源程序所依赖的头文件在用户/home/include/目录下，此时就应该使用－I 选项来指定。

```
# gcc -I /home/include -o test test. c
```

（7）－L <DIR >

与前述－I 选项类似，该选项用于指定函数库所在的路径。如果程序使用了不在标准位置的函数库，那么可以通过－L 选项向 GCC 的库文件搜索路径中添加新的目录。例如，一个程序要用到的库 libapp. so 在/home/zxq/lib/目录下，为了能让 GCC 能够顺利地链接该库，可以使用下面的指令：

```
# gcc -Test. c -L /home/zxq/lib/ -lapp -o Test
```

这里的 -L 选项表示 GCC 链接库文件 libapp. so。在 LINUX 下的库文件在命名时遵循了一个约定，那就是应该以 lib 三个字母开头，由于所有的库文件都遵循了同样的规范，因此在使用 -L 选项指定链接的库文件名时可以省去 lib 三个字母，也就是说 GCC 在对 -lapp 进行处理的时候，会自动去链接名为 libapp. so 的文件。

（8）-static 选项

GCC 在默认情况下链接的是动态库，有时为了把一些函数静态编译到程序中，而无需链接动态库就采用 -static 选项，它会强制程序链接静态库。

（9）-o 选项

在默认的状态下，如果 GCC 指令没有指定编译选项的情况下会在当前目录下生成一个名为 a. out 的可执行程序，例如：执行# gcc Test. c 命令后会生成一个名为 a. out 的可执行程序。因此，为了指定生成的可执行程序的文件名，就可以采用 -o 选项，比如下面的指令：

```
# gcc -o Test Test. c
```

执行该指令会在当前目录下生成一个名为 Test 的可执行文件。

2. 出错检查和警告提示选项

GCC 编译器包含完整的出错检查和警告提示功能，比如 GCC 提供了 30 多条警示信息和 3 个警告级别，使用这些选项有助于增强程序的稳定性和更加完善程序代码的设计，常用的此类选项如下。

（1）-pedantic

用于以 ANSI/ISO C 标准列出的所有警告。当 GCC 在编译不符合 ANSI/ISO C 语言标准的源代码时，如果在编译指令中加上了 -pedantic 选项，那么源程序中使用了扩展语法的地方将产生相应的警告信息。

（2）-w

用于禁止输出警告信息。

（3）-Werror

用于将所有警告转换为错误。Werror 选项要求 GCC 将所有的警告当成错误进行处理，这在使用自动编译工具（如 Make 等）时非常有用。如果编译时带上 -Werror 选项，那么 GCC 会在所有产生警告的地方停止编译，只有程序员对源代码进行修改并起相应的警告信息消除时，才能够继续完成后续的编译工作。

（4）-Wall

用于显示所有的警告信息。该选项可以打开所有类型的语法警告，以便于确定程序源代码是否是正确的，并且尽可能实现可移植性。

对 LINUX 开发人员来讲，GCC 给出的警告信息是很有价值的，它们不仅可以帮助程序员写出更加健壮的程序，而且还是跟踪和调试程序的有力工具。建议在用 GCC 编译源代码时始终带上 -Wall 选项，养成良好的习惯。

3. 代码优化选项

代码优化是指编译器通过分析源代码找出其中尚未达到最优的部分，然后对其重新进行组合，进而改善代码的执行性能。GCC 通过提供编译选项 -On 来控制优化代码的生成，对

于大型程序来说，使用代码优化选项可以大幅度提高代码的运行速度。

（1）－O 选项

编译时使用选项 －O 可以告诉 GCC 同时减小代码的长度和执行时间，其效果等价于 －O1。

（2）－O2 选项

选项 －O2 告诉 GCC 除了完成所有 －O1 级别的优化之外，同时还要进行一些额外的调整工作，如处理器指令调度。

4. 调试分析选项

（1）－g 选项

用于生成调试信息，GNU 调试器可以利用该信息。GCC 编译器使用该选项进行编译时，将调试信息加入到目标文件中，这样 gdb 调试器就可以根据这些调试信息来跟踪程序的执行状态。

（2）－pg 选项

用于编译完成后，额外产生一个性能分析所需信息。

注： 由于使用调试选项都会使最终生成的二进制文件的大小急剧增加，同时增加程序在执行时的开销，因此，调试选项通常推荐仅在程序开发和调试阶段中使用。

12.2.2　GCC 编译过程解析

下面举一个简单的例子来说明 GCC 的编译过程。首先用 VI 编辑器来编辑一个简单的 C 程序 test.c，程序清单如下。

```
#include <stdio.h> /*标准输入/输出头文件,包含了标准输入/输出函数(如 perror()和 printf()
                    等)的定义*/
int main(){
    printf("Hello,this is a test!\n");
    return 0;
}
```

根据上面的内容，使用 gcc 命令来编译该程序。

```
[root@localhost]# gcc -o test test.c
[root@localhost]# ./test
Hello,this is a test!
```

可以从上面的编译过程看到，编译一个这样的程序非常简单，一条指令即可完成，而事实上，如 12.1 节所述，上述编译过程是分为预处理、编译、汇编和连接四个阶段进行的。

① 预处理：GCC 通过调用 －E 参数让编译器在预处理后停止，并输出预处理结果。

```
# gcc -E test.c -o test.i
```

编译器在这一步调用 cpp 工具来对源程序进行预处理，此时会生成 test.i 文件。其中，test.i 文件中存放着 test.c 经预处理之后的代码。在本例中，预处理结果就是将 stdio.h 文件

中的内容插入到 test. c 中。

② 编译为汇编代码：预处理之后，GCC 直接对生成的 test. i 文件编译，生成汇编代码：

```
gcc -S test. i -o test. s
```

gcc 的 -S 选项表示在程序编译期间，在生成汇编代码后停止，-o 选项表示输出汇编代码文件。

③ 汇编：使用 GAS 汇编器，GCC 将汇编语言翻译为机器代码。对于经过前述步骤生成的汇编代码文件 test. s，GAS 汇编器负责将其编译为目标文件：

```
gcc -c test. s -o test. o
```

④ 连接：GCC 连接器是 GAS 提供的，负责将程序的目标文件与所需的所有附加的目标文件连接起来，最终生成可执行文件。附加的目标文件包括静态连接库和动态连接库。对于经过前述步骤生成的 test. o，将其与 C 标准输入/输出库进行连接，最终生成可执行文件 test：

```
gcc test. o -o test
```

在命令行窗口中运行可执行文件 test：

```
[root@ localhost]# ./test
```

即可显示 Hello，this is a test!

12.2.3 多个程序文件的编译

通常整个程序是由多个源文件组成的，相应地也就形成了多个编译单元，使用 GCC 能够很好地管理这些编译单元。假设有一个由 test1. c 和 test2. c 两个源文件组成的程序，为了对它们进行编译，并最终生成可执行程序 test，可以使用下面这条命令：

```
# gcc test1. c test2. c -o test
```

如果同时处理的文件不止一个，GCC 仍然会按照预处理、编译和链接的过程依次进行。深究起来，上面这条命令大致相当于依次执行如下三条命令：

```
# gcc -c test1. c -o test1. o
# gcc -c test2. c -o test2. o
# gcc test1. o test2. o -o test
```

12.3 GCC 编译器的安装

1. 下载

在 GCC 网站上(http://gcc. gnu. org/)或通过网上搜索可查找到 GCC 编译器的下载资源。以版本 4.5.0 为例，可供下载的文件一般有两种形式：gcc - 4.5.0. tar. gz 和 gcc -4. 5. 0. tar. bz2，只是压缩格式不一样，内容完全一致，下载其中一种即可。

2. 解压缩

根据压缩格式，选择下面相应的一种方式解包（以下的“%”表示命令行提示符）：

```
% tar xzvf gcc -4.5.0.tar.gz
```

或

```
% bzcat gcc -4.5.0.tar.bz2 | tar xvf -
```

新生成的 gcc -4.5.0 目录被称为源目录，用 ${srcdir} 来表示。以后在出现 ${srcdir} 的地方，应该用真实的路径来替换它。用 pwd 命令可以查看当前路径。在 ${srcdir} /INSTALL 目录下有详细的 GCC 安装说明，可用浏览器打开 index.html 阅读。

3. 建立目标目录

目标目录（用 ${objdir} 来表示）用来存放编译结果。GCC 建议编译后的文件不要放在源目录 ${srcdir] 中（虽然这样做也可以），最好单独存放在另外一个目录中，而且不能是 ${srcdir} 的子目录。例如，可以这样建立一个称为 gcc -build 的目标目录（与源目录 ${srcdir} 是同级目录）：

```
% mkdir gcc -build
% cd gcc -build
```

以下的操作主要是在目标目录 ${objdir} 下进行。

4. 配置

配置的目的是决定将 GCC 编译器安装到什么地方（${destdir}），支持什么语言以及指定其他一些选项等。其中，${destdir} 不能与 ${objdir} 或 ${srcdir} 目录相同。配置是通过执行 ${srcdir} 下的 configure 来完成的。其命令格式为（记得用实际的真实路径替换 ${destdir}）：

```
% ${srcdir}/configure --prefix = ${destdir} [其它选项]
```

例如，如果想将 GCC 4.5.0 安装到/usr/local/gcc -4.5.0 目录下，则 ${destdir} 就表示这个路径。具体配置方法如下：

```
% ../gcc -4.5.0/configure --prefix =/usr/local/gcc -4.5.0 --enable -threads = posix --
disable -checking --enable --long -long --host = i386 -redhat -linux --with -system -zlib
--enable -languages = c,c ++ ,java
```

将 GCC 安装在/usr/local/gcc -4.5.0 目录下，支持 C/C ++ 和 Java 语言，其他选项参见 GCC 提供的帮助说明。

5. 编译

编译的命令如下：

```
% make
```

6. 安装

执行下面的命令将编译好的库文件等复制到 ${destdir} 目录中（根据设定的路径，可

能需要管理员的权限）：

```
% make install
```

至此，GCC 4.5.0 安装过程就完成了。

6. 其他设置

GCC 4.5.0 的所有文件，包括命令文件（如 gcc、g ++）、库文件等都在 ${destdir} 目录下分别存放，如命令文件放在 bin 目录下、库文件在 lib 下、头文件在 include 下等。由于命令文件和库文件所在的目录还没有包含在相应的搜索路径内，所以必须要作适当的设置之后编译器才能顺利地找到并使用它们。

（1）gcc、g ++、gcj 的设置

要想使用 GCC 4.5.0 的 gcc 等命令，简单的方法就是把它的路径 ${destdir} /bin 放在环境变量 PATH 中。若不采用这种方式而是采用符号连接的方式实现，这样做的好处是仍然可以使用系统上原来的旧版本的 GCC 编译器。首先，查看原来的 gcc 所在的路径：

```
% which gcc
```

上述命令将显示：/usr/bin/gcc。因此，原来的 gcc 命令在/usr/bin 目录下。我们可以把 GCC 4.5.0 中的 gcc、g ++、gcj 等命令在/usr/bin 目录下分别做一个符号连接：

```
% cd /usr/bin
% ln -s ${destdir}/bin/gcc gcc45
% ln -s ${destdir}/bin/g++ g++45
% ln -s ${destdir}/bin/gcj gcj45
```

这样，就可以分别使用 gcc45、g ++45、gcj45 来调用 GCC 4.5.0 的 gcc、g ++、gcj 完成对 C、C ++、Java 程序的编译了。同时，仍然能够使用旧版本的 GCC 编译器中的 gcc、g ++ 等命令。

（2）库路径的设置

将 ${destdir} /lib 路径添加到环境变量 LD_ LIBRARY_ PATH 中，最好添加到系统的配置文件中，这样就不必要每次都设置这个环境变量了。例如，若 GCC 4.5.0 安装在/usr/local/gcc -4.5.0 目录下，在 RH Linux 下可以直接在命令行上执行或者在文件/etc/profile 中添加下面一句：

```
setenv LD_LIBRARY_PATH /usr/local/gcc -4.5.0/lib: $LD_LIBRARY_PATH
```

7. 测试

用新的编译命令（gcc45、g ++45 等）编译以前的 C、C ++程序，检验新安装的 GCC 编译器是否能正常工作。另外，还可以根据需要删除或者保留 ${srcdir} 和 ${objdir} 目录。

12.4 本章小结

本章主要对 GCC 编译器进行了深入介绍，通过本章的学习，需要了解 GCC 编译器的安装过程；需要掌握 GCC 编译器的使用方法。

本章习题

1. 简述 GCC 编译器所支持的源程序的格式主要有哪些。
2. 简述 GCC 编译器的常用编译选项主要有哪些。
3. 简述 GCC 编译器的出错检查和警告提示选项主要有哪些。
4. 简述 GCC 编译器的代码优化选项主要有哪些。
5. 简述 GCC 编译器的调试分析选项主要有哪些。

参考文献

[1] COMER D E, STEVENS D L. 用 TCP/IP 进行网际互连：第 3 卷　客户 - 服务器编程与应用（Linux/POSIX 套接字版）. 赵刚，译. 北京：电子工业出版社. 2005.

[2] 周丽，焦程波，兰巨龙. Linux 系统下多线程与多进程性能分析. 微计算机信息，2005，21（9 - 3）：118 - 120.

[3] DIAO Y F. 多线程 or 多进程? http：//programmerdigest. cn/2010/08/1096. html# more - 1096.

[4] 张志佳. 基于多线程的 LINUX 下并发服务器的实现研究. 微计算机应用，2007，28（4）：368 - 371.

[5] 罗泽，车文刚. 多线程环境下邮件检索代理客户端实现的优化. 昆明理工大学学报，2001，26（6）：26 - 28.

[6] 李昊，刘志镜. 线程池技术的研究. 现代电子技术，2004，170（3）：77 - 80.

[7] 叶树华. 网络编程实用教程. 北京：人民邮电出版社，2010.

[8] 崔武子，林志英，和青芳. C 程序设计课程教案及题解. 北京：清华大学出版社，2010.

[9] 王雷，冯湘. 高等计算机网络与安全. 北京：北京交通大学出版社，2010.

[10] 吴文虎. 程序设计基础. 北京：清华大学出版社，2004.

[11] 谭浩强. C 程序设计教程. 北京：清华大学出版社，2007.

[12] 刘艳飞，迟剑，房健. C 语言范例开发大全. 北京：清华大学出版社，2010.

[13] STEVENS W R. TCP/IP 详解：卷 1　协议. 范建华，译. 北京：机械工业出版社，2007.

[14] COMER D E, STEVENS D L. 用 TCP/IP 进行网际互连：第 1 卷　原理、协议与结构. 林瑶，译. 北京：电子工业出版社，2007.

[15] 蔡建平. 软件综合开发案例教程：Linux、GCC、MySQL、Socket、Gtk + 与开源案例. 北京：清华大学出版社，2011.

[16] 林锐，韩永泉. 高质量程序设计指南：C ++/C 语言. 北京：电子工业出版社，2007.

[17] 杨宗德，邓玉春. Linux 高级程序设计. 北京：人民邮电出版社，2009.

[18] 宋敬彬，孙海滨. Linux 网络编程. 北京：清华大学出版社，2010.

[19] 徐千洋. Linux C 函数库参考手册. 北京：中国青年出版社，2002.

[20] 杜华. Linux 编程技术详解. 北京：人民邮电出版社，2007.

[21] 朱云翔，胡平. 精通 UNIX 下 C 语言编程与项目实践. 北京：电子工业出版社，2007.

致　　谢

首先，在本书的编写过程中，得到了湘潭大学信息工程学院博士生导师郑金华教授、刘任任教授、裴廷睿教授，湖南大学信息科学与工程学院硕士生导师李军义副教授，以及北京交通大学出版社的谭文芳编辑等领导和专家们的大力支持与热心帮助，在此表示衷心感谢。

其次，本书的出版还得到湘潭大学人才引进启动基金项目、湖南省普通高等学校教学改革研究项目重点项目（项目文号：湘教通［2009］321 号）、福建省工程学院预研与发展基金项目（No. GY－Z10070，No. GY－Z10050）、福建省教育厅科技项目（No. JA11188）等部分资助；本书的部分内容参考了国内外有关单位和个人的研究成果，均已经在参考文献中列出，在此也一并表示感谢。

另外，由于本书的编写目的定位于 LINUX 环境下的 C 语言 Socket 编程的基础知识与案例分析相结合，试图让本科生与研究生在深入了解 LINUX 环境下的 C 语言 Socket 编程的相关概念与关键技术的基础上，能尝试开展 LINUX 环境下的 C 语言 Socket 编程的一些初步编程工作，因此，在本书的内容编写与结构组织上具有一定的难度，加之编者水平有限，虽然几经修改，书中仍然会难免存在一些疏漏与不足之处，敬请读者、专家以及同行朋友们批评指正，在此先行表示感谢。

编者

2011 年 12 月

- TCP/IP网络编程技术基础
- C#程序设计
- 多媒体技术应用
- 计算机网络安全教程（第2版）
- VB.NET 程序设计
- 面向对象程序设计（C#实现）
- 高等计算机网络与安全
- 电子支付与网络银行
- SQL Server 2005实用教程
- Java 程序设计与应用
- ASP .NET Web应用程序设计教程
- 操作系统实验教程
- Web程序设计
- Java精解案例教程
- 计算机硬件维修
- 软件工程基础
- 计算机系统组装与维护
- 数据结构实例教程
- 程序设计导论——Java编程
- 网络工程与组网技术
- Visual Basic程序设计实验与习题指导
- 数据库应用系统开发实例
- 多媒体技术基础
- ASP动态网页设计教程
- Delphi数据库编程
- Java EE编程技术
- SQL Server 2005数据库原理及应用教程
- 基于Web的远程监控系统
- ANSYS辅助分析应用基础教程上机指导
- Microsoft Project 2003项目管理与应用：上机指导
- Visual C++ 程序设计与实践：实验与指导
- Java程序设计与网络编程
- PHP精解案例教程
- Microsoft Project 2003项目管理与应用
- 软件工程
- 网站全程设计技术（修订本）
- ASP精解案例教程（修订本）
- 计算机网络与应用教程
- 计算机网络管理——Windows 2000管理基础
- 嵌入式系统的设计与开发
- 多媒体课件设计理论与实践
- 数据库系统及应用基础
- 计算机组成原理
- Linux操作系统分析教程
- 数据结构基础教程
- 微机原理与接口技术
- Visual Basic 高级图形应用程序设计
- 多媒体计算机外部设备
- Visual FoxPro 系统开发教程
- 计算机网络技术与应用
- 计算机网络基础教程（修订本）
- C语言程序设计
- C#程序设计教程
- 计算机维修教程

ISBN 978-7-5121-0903-2

9 787512 109032 >

责任编辑：谭文芳
特邀编辑：胡花蕾
封面设计：理　安

定价：23.00元