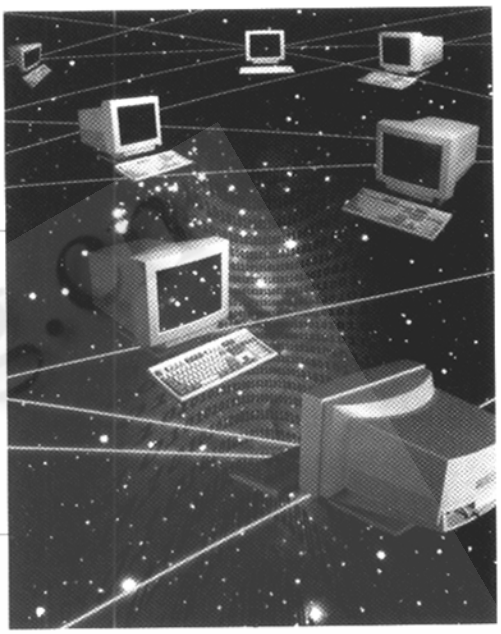
基于 WinSock 和 Berkeley Socket 的网络应用系统开发



肖祥雄 （长沙 湖南计算机股份有限公司 410007）

摘要：套接字（Socket）已成为最流行的网络通信应用程序接口。本文简要地介绍了网络应用系统的模型，讲述了Socket的通信接口原理、Windows中WinSock和UNIX中Berkeley Socket的开发环境、以及开发网络应用系统的关键流程，并简单地介绍了笔者在编写银行业务系统时的几点编程体会。

关键词：C/S模型 网络通信接口 Socket

1 引言

Socket最初是由U.C.Berkeley为UNIX操作系统开发的网络通信接口。随着UNIX操作系统的广泛使用，Berkeley Socket成为当前UNIX平台上最流行的网络通信应用程序接口(API)。Windows Socket(简称WinSock)是以Berkeley Socket为蓝本、由微软公司、SUN公司等开发的Windows平台上的网络应用系统编程接口。由于Windows操作系统的普及、界面友好、丰富的开发工具，因此成为网络应用系统的前端平台；而UNIX操作系统的稳定性、安全性、高效的处理能力，通常作为较大型网络应用系统后端平台的首选。因此，基于这两种平台上的网络应用系统开发，引起了开发者的广泛关注。

2 网络应用系统模型简介

2.1 客户/服务器（client/server）模型

客户/服务器是网络应用的标准模型。这里，服务器是一个进程，等待客户进程调用，并且为客户进程服务。Internet中的典型应用WWW、FTP、Telnet、E-mail以及时间服务、目录服务、地址解析等都采用客户/服务器模型；我们通常所讲的浏览器/服务器（Browser/Server）模型实质上也是客户/服务器模型。

在客户/服务器模型中，服务进程可进一步分为两种类型：

（1）如果服务进程可以在一个已知的、很短的时间内处理完客户的请求，那么服务进程就自己来处理这一请求，这称之为反复服务。比如时间日期服务就是以这种反复服务方式处理的。

（2）如果服务时间和请求本身有关，也就是说服务进程事前不知道要花多少时间，那么服务进程调用另一个进程来处理这个请求，而它自己又返回休眠状态，等待下一个客户请求。这称之为并发服务。显然，这种类型的服务方式要求操作系统允许多个进程或多个线程同时运行。

需要注意的是客户进程和服务器进程的作用是不对称的，也就是说这两半的编程是不相同的。

2.2 分布式模型

分布式模型比客户/服务器模型更能适应大系统中的动态变化和对信息库频繁访问的要求。这种模型在每一台主机上都有一个完整的信息库，数据拥有者周期地广播数据，其他主机被动接收数据，从而不断刷新和充实自己的信息库。分布式模型的数据收集过程是一个被动接收过程，而客户/服务器模型则是一个主动请求过程。其典型应用是IP路由系统，其中每一台机器的路由表便是一个完整的信息库，这个信息库是通过主动广播、被动接收的方式建立起来并不断刷新的。

分布式模型系统的缺点在于每台主机都必须有一个并发后台进程，它不断为信息库收集数据，而不管有无进程查询信息库。

3 WinSock 和 Berkeley Socket 的实现原理

我们知道，要实现进程之间的通信，其首要问题是进程的标识。在单机环境中，每个进程可用其进程号来唯一标识，例如：在某个 UNIX 系统中，8 号进程和 9 号进程，其含义是十分明确的。但是，在网络环境中，就不能只用进程号来唯一标识某个进程，还必须说明进程所在的主机。并且，由于进程是操作系统中的一个概念，因此必须提出一个比进程更低级、更稳定的概念，它就是端口（port），端口是网络协议软件与应用程序打交道的访问点，是协议软件的一部分，一个主机内的每个网络进程使用协议端口进行标识。这样，要唯一确定网络环境的某个进程，就同时需要主机和端口号，在Internet 环境中，就同时采用IP地址和端口号来标识。网络通信需要解决的第二个问题是多重协议的标识。网络进程间通信时，必须在众多的通信协议中指明是何种通信协议；因为不同协议的地址格式不同，端口分配相互独立，工作方式也各不相同。

综上所述，在 Internet 网络环境中，协议、IP 地址和端口构成了进程间通信的一端，将它称为套接字(Socket)，它是网络编程的一个接口。而两个进程之间的通信链路定义为连接（connection），一对套接字完全确定了构成一个连接的两个进程，称之为关联（association），自然，一个套接字指定一个连接的一半，称为半关联(half-association)。

在 TCP/IP 协议族中，Socket 接口与网络协议的关系如图 1 所示。

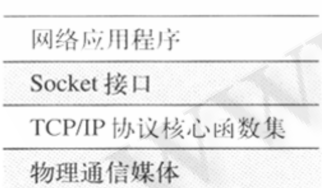


图 1　Socket 接口与协议的关系

4 开发环境和关键流程

4.1 WinSock 的开发环境

Windows 用动态链接库(DLL)的方法来支持运行 WinSock，动态链接库在需要时实时地连接到应用程序上，16 位 DLL 名为 WINSOCK.DLL，而 32 位 DLL 名为 WSOCK32.D L L 。WinSock 有两个版本：V1.1(1993 年 1 月制订)和 V2.0(1995 年 5 月制定)，目前广泛使用的 Windows 98 和 NT 4.0 本身都支持 WinSock V2.0，为 WinSock 应用程序提供了完善的运行环境。而对于 WinSock的开发环境，目前的主流是Microsoft Visual C++ 6.0，它已经具有了完善的 WinSock 开发环境，提供了开发 WinSock 所需的所有头文件和库文件。

WinSock API 提供了一个函数集（或函数库），WinSock 规范把 API 划分为以下三部分：

(1) 与 Berkeley Socket 兼容的基本 Socket 函数，用来创建、打开、关闭 Socket、设置 Socket 属性、发送与接收数据等；

(2) 网络数据信息检索函数，应用程序用它取回域名、通信服务、协议等 Internet 信息；

(3) WinSock 专有的为 Windows 专用的扩展函数。

在Socket开发早期阶段，在Visual C++中进行Socket 编程只有调用 Windows Socket API 函数的一种方法，许多开发商也开发了 Socket 使用类来包容简化这些 Socket 调用。直接在 WinSock API 上做开发比较复杂。从 Visual C++ 2.1 开始，MFC 中提供了两个新的类：CasyncSocket 和Csocket来包装Socket API 函数。在CasyncSocket 类中封装了Windows API，该类使我们可以使用面向对象的方式来进行 Socket 编程，而且可以非常方便地调用其他 MFC 对象。Csocket 类是从 CasyncSocket 类派生的高级抽象，支持同步操作，功能更为强大。通过这两个类和其他 MFC 类的配合，Visual C++ 为网络应用的开发者提供了更加方便易用、强大的开发平台。

WinSock 1.1 有其局限性，它只能处理 TCP/IP 协议。WinSock 2.0 是 WinSock 1.1 的超集，它不仅支持 TCP/IP 协议，还提供了对DECnet、IPX/SPX、OSI 等网络协议集的支持，利用它可以开发独立于网络协议的、支持服务质量、支持多点广播等特性的应用。

4.2 SCO UNIX 下 Berkeley Socket 的开发环境

SCO UNIX 下的 Berkeley Socket 支持 UNIX 、Internet、Xerox 等通信协议族，socket 常用的通信类型有 SOCK_STREAM、SOCK_DGRAM、SOCK_RAW等几种。其中SOCK_STREAM定义了一种可靠的面向连接的服务，它利用了传输层字节流协议(TCP)的可靠性，实现了无差错、无重复的顺序数据传输；应用程序可以发送任意长度的数据，将数据当作字节流。SOCK_DGRAM 定义了一种无连接的服务，数据通过相互独立的包进行传输。包的传输是无序的，并且不保证是否出错、丢失和重复，包的长度

是有限的。SOCK_RAW 提供原始网络协议访问。根据socket所选用的通信协议族和不同的通信类型,可构成不同的组合。

用socket实现网络通信,需要调用许多系统调用,而这些系统调用常常需要一个指向地址结构的指针作为参数。因此,在这里我们先介绍一下socket地址结构,这个结构定义在文件< sys/socket.h >中。

```
struct sockaddr {
  u_short sa_family;
  char sa_data [14];
};
```

其中,14字节协议专用地址内容随地址类型的不同而异。对于Internet协议族(本文只讨论Internet协议族),则在文件< netinet/in.h >中定义如下结构:

```
struct in_addr {
  ulong s_addr;
};
struct sockaddr_in {
   short sin_family;
   u_short sin_port;
   struct in_addr sin_addr;
   char sin_zero [8];
};
```

在socket地址结构中使用的数据类型与C语言定义的数据类型之间的对应关系,在文件< sys/types.h >中定义,其对应关系如下:

```
unsigned char    u_char;
unsigned short   u_short;
unsigned int     u_int;
unsigned long    u_long;
```

至于Socket中各系统调用的详细说明,请参考相关资料。

4.3 网络应用系统的关键流程

以下流程使用的系统调用均以SCO UNIX下的Socket为例。

4.3.1 面向连接的C/S应用系统的关键流程

(1) 服务器进程(Server)首先启动,通常完成以下步骤

① 打开一条通信信道,并且通知本机主机,它可以在众所周知的地址上接受客户的请求。

② 等待客户请求到达地址。

③ 对于反复式服务器来说,是处理该请求,并且发出回答信息。如果客户请求可以由服务员一次响应加以处理,通常就可采用反复式服务;对于并发式服务来说,是启动一个新进程来处理客户请求。这就需要在UNIX下使用fork(),有的还要exec()来启动。这个新进程处理该请求,不必响应其他客户请求。当它运行完成时,关闭和该客户的通信信道,然后结束。

④ 回到第2步,等待客户请求。

(2) 客户进程(Client)执行一套不同的动作

① 打开一条通信信道,接通一台指定主机(如服务器)上的一个指定的地址;

② 向服务器发服务请求,并且接收响应,必要时可以连续做这一步;

③ 关闭通信信道并且终止。

上述步骤可用流程图2表示如下。

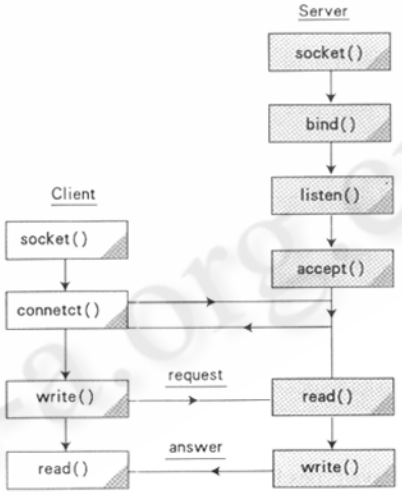


图 2 面向连接C/S应用系统的关键流程

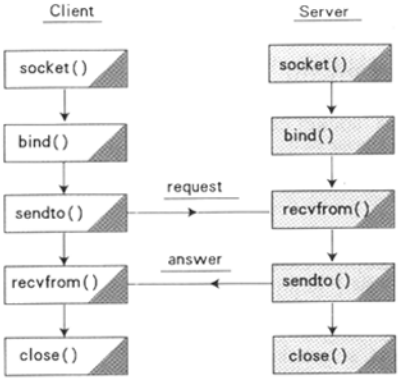


图 3 无连接C/S应用系统的关键流程

4.3.2 无连接C/S应用系统的关键流程

由socket通信过程可知,对于无连接的C/S应用系统,服务器和客户机开始都必须调用socket()系统调用,去创建一个socket,然后分别调用bind()系统调用将Socket与本地网络地址捆扎在一起,这时服务器和客户机之间不需要建立连接,而是通过sendto()和recvform()系统调用,进行网络地址和数据的传输。其流程图如图3所示。

上述无连接socket通信既可以实现点对点通信,也可以实现广播通信,进行广播通信时,只要把sendto()系统调用中发送数据的目的地址换成广播地址,即可实现数据广播。

5 几点编程体会

笔者在SCO UNIX下利用Socket,编写银行业务系统程序(采用C/S模型)时,碰到过一些问题,现简单地介绍给大家。

5.1 可重用的端口地址

在开发调试阶段、经常要将socket服务进程杀死,或者由于某种原因socket服务进程异常退出,而在重新启动该进程时,不时出现"bind error!:Address already in use"的错误信息,这是由于该进程不允许该socket端口地址重新使用的缘故,因此当进程异常退出后,"残留"在该端口地址上的TCP/IP链有时可延续较长时间,还没有全部拆除,仍然占用着端口地址,使进程无法重新启动。解决的方法是:执行socket系统调用setsockopt()来改变socket端口属性,使该端口地址可以重新使用。程序段如下:

```
#include   <sys/types.h>
#include   <sys/socket.h>
int   sockid;
.....
      if((sockid=socket(AF_INET,SOCK_STREAM,
IPPROTO_TCP))<0){
       perror("socket error!"!);
       exit(0);
    }
    if(setsockopt(sockid,SOL_SOCKET,SO_REUSEADDR,
(char *)0,0)<0){
       perror("set socket option error!");
       exit(0);
    }
.....
```

5.2 子进程数量的控制

对client端请求进行并发处理的server端进程,通常要对同时处理的子进程数目进行控制,否则当成千上万的client端请求同时到达的时候,server服务器一下子处理不过来,将会造成系统性能急剧下降,直至系统崩溃。控制子进程数目的方法有很多;最简单的可以调用系统操作命令"ps"获得子进程数,还可以通过建立一个系统临时文件保存当前活动子进程数,或者通过管道、消息队列等IPC通信机制实现。我们通过管道来实现子进程数目控制,在程序中定义了一个管道p_id,用于负责父子进程的通信,服务器进程每处理一次client端请求,都fork一个子进程,用于完成client的请求。父进程本身只判断子进程计数器是否已达到临界值MaxChild,如果没有达到,就继续接收下一个请求;如果已达到临界值,则读管道p_id,看是否有子进程完成的信息,如没有信息,则阻塞在read()系统调用上,直到有子进程完成退出为止,如果读到信息,说明已有子进程退出,则把变量childs减1,然后继续接收下一个请求。而子进程在执行完毕退出前,都要往管道p_id写一条信息,告诉父进程有子进程退出的信息自己才退出。

5.3 "僵尸"进程(Zombie)的问题

对于并发处理的server端进程,client端的每次请求都将fork一个子进程,当每个子进程处理完毕退出时,将产生一个子进程终止信号(SIGCLD),由于并发处理的server端父进程不能执行系统调用wait来处理子进程的终止信号,于是该子进程则变成僵尸进程,占用了系统资源(进程表项等),随着僵尸进程的增多,server端将不再能fork子进程,导致服务终止。解决的方法是:在程序中调用signal系统调用,并设置SIGCLD信号的处理方法,通常是调用signal(SIGCLD,SIG_IGN),父进程忽略该信号。■

参考文献

1 朱三元、杨明、薛钫,网络通信软件设计指南,清华大学出版社,1995。

2 汤毅坚,计算机实用网络编程,人民邮电出版社,1995。

3 [美]Kris Jamsa Ph.D.,Web Programming,电子工业出版社,1997。