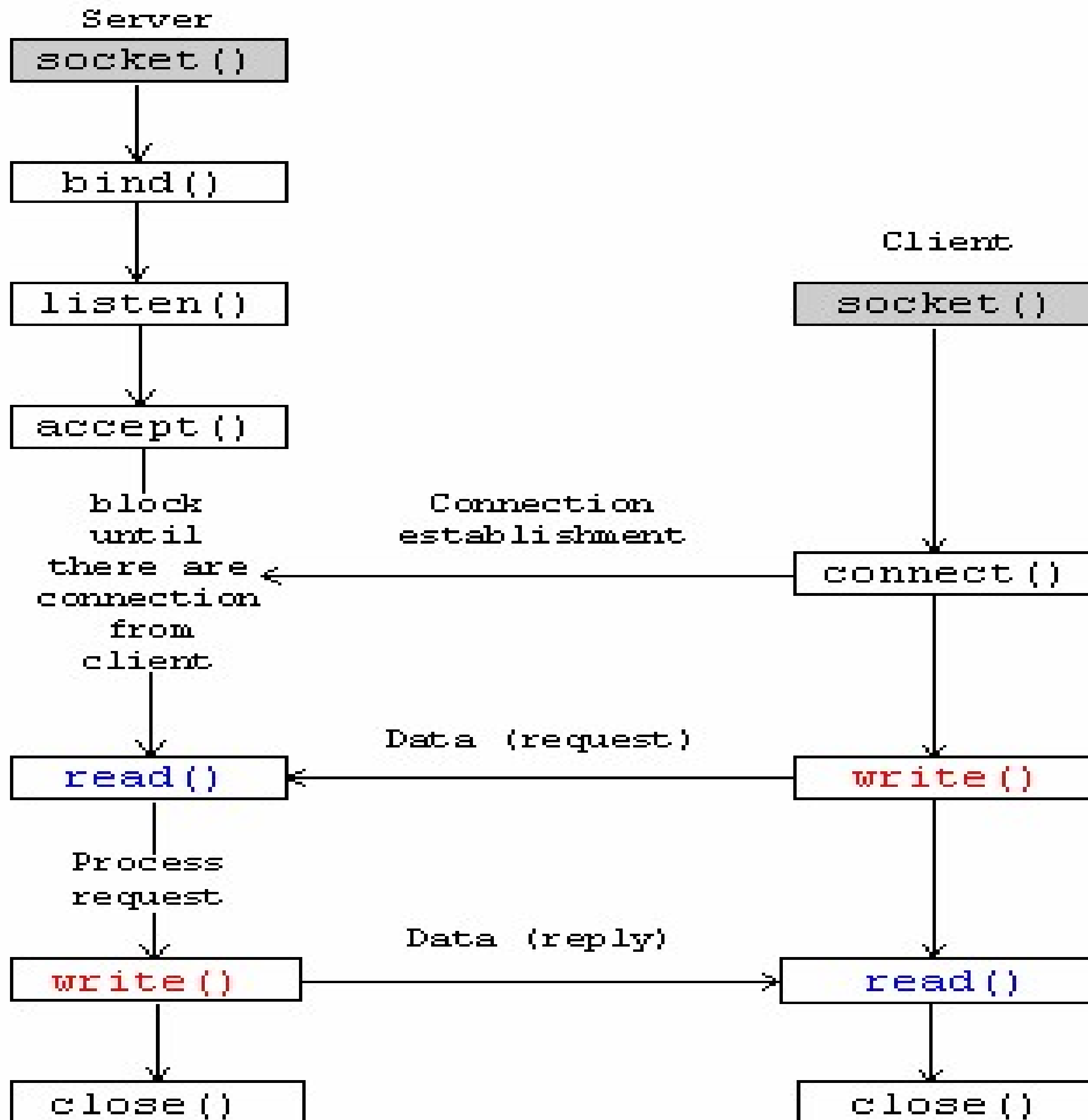
ACE Socket

Allen Long
ihuihoo@gmail.com
http://www.huihoo.com
huihoo - Enterprise Open Source

内容安排

- 如何访问OS服务
- TCP/IP Socket编程接口
- 使用ACE的UDP类进行网络编程
- 单播、广播、多播

Socket Interface



Socket API概述(1/2)

Sockets 是网络编程最普遍的接口

- Socket API最初是在 BSD Unix 中开发,用来为TCP/IP协议族提供应用级程序接口
- The Socket API 分五类有近20多个函数
- Socket通过Socket API创建通信端点,并通过句柄(handle)访问
- 每个socket都可以绑定一个本地地址和一个远程地址
- In Unix,对大多数应用来讲, socket句柄和其他I/O句柄(如:文件、管道、终端设备句柄)可以互换使用.而在windows中却不可以.

本地管理：socket接口为管理本地上下文信息提供以下函数：

- socket：分配最小的未用socket句柄；
- bind：将socket句柄与本地或远地地址相关联；
- getsockname和getpeername：分别确定socket所连接的本地或远地地址；
- close：释放socket句柄，使它可用于后面的复用。

连接建立和连接终止：socket接口为建立和终止连接提供以下函数：

- connect：客户通常使用connect来主动地与服务器建立连接；
- listen：服务器使用listen来指示它想要被动地侦听进入的客户连接请求；
- accept：服务器使用accept来创建新的通信端点，以为客户服务；
- shutdown：有选择地终止一个双向连接的读端和/或写端流。

Socket API 概述(2/2)

数据传输机制：socket接口提供以下函数来发送和接收数据：

- read/write：通过特定句柄接收和传输数据缓冲区；
- send/recv：与read/write类似，但它们提供一个额外的参数来控制特定的socket特有操作（比如交换“紧急”数据，或“偷看”接收队列中的数据，而又不把它从队列中移除）；
- sendto/recvfrom：交换无连接数据报；
- readv/writev：分别支持“分散读”和“集中写”语义（这些操作优化用户/内核模式切换并简化内存管理）；
- sendmsg/recvmsg：通用函数，包含了所有其他数据传输函数的行为。对于UNIX域的socket，sendmsg和recvmsg函数还提供在同一主机的任意进程间传递“访问权限”（比如打开文件句柄）的能力。

注意这些接口也可被用于其他类型的I/O，比如文件和终端。

选项(option)管理：socket接口定义以下函数，允许用户改变socket行为的缺省语义：

- setsockopt和getsockopt：修改或查询在协议栈不同层次中的选项。选项包括多点传送、广播，以及设置/获取发送和接收传输缓冲区的大小；
- fcntl和ioctl：是UNIX系统调用，使在socket上能够进行异步I/O、非阻塞I/O，以及紧急消息递送。

除了上面描述的socket函数，通信软件还可使用以下标准库函数和系统调用：

- gethostbyname和gethostbyaddr：处理网络寻址的多种情况，比如映射主机名到IP地址；
- getservbyname：通过服务的端口号或人类可读的名字来对它们进行标识；
- ntohl、ntohs、htonl、htons：执行网络字节序转换；
- select：在成组的打开的句柄上执行基于I/O和基于定时器的事件多路分离。

Linux Socket Server (1/2)

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define MYPORT 8000
#define BACKLOG 10

int main()
{
int sockfd, new_fd;
struct sockaddr_in my_addr;
struct sockaddr_in their_addr;
int sin_size;

sockfd = socket(AF_INET, SOCK_STREAM, 0);
if(sockfd == -1)
{
  perror("socket() error !");
  exit(1);
}
else
  printf("socket() is OK...\n");

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);
my_addr.sin_addr.s_addr = INADDR_ANY;
```

Linux Socket Server (2/2)

```
memset(&(my_addr.sin_zero), 0, 8);

if(bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
{
  perror("bind() error lol!");
  exit(1);
}
else
  printf("bind() is OK...\n");
if(listen(sockfd, BACKLOG) == -1)
{
  perror("listen() error lol!");
  exit(1);
}
else
  printf("listen() is OK...\n");

sin_size = sizeof(struct sockaddr_in);
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);

if(new_fd == -1)
  perror("accept() error !");
else
  printf("accept() is OK...\n");

close(new_fd);
close(sockfd);
return 0;
}
```

Linux Socket Client

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define DEST_IP "127.0.0.1"
#define DEST_PORT 8000

int main(int argc, char *argv[ ])
{
int sockfd;
struct sockaddr_in dest_addr;
sockfd = socket(AF_INET, SOCK_STREAM, 0);

if(sockfd == -1)
{
  perror("Client-socket() error!");
  exit(1);
}
else
  printf("Client-socket() sockfd is OK...\n");

dest_addr.sin_family = AF_INET;
dest_addr.sin_port = htons(DEST_PORT);
dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);

memset(&(dest_addr.sin_zero), 0, 8);
if(connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr)) == -1)
...
```

Windows Socket Server

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>

int __cdecl main(void)
{
    WSADATA wsaData;
    SOCKET ListenSocket = INVALID_SOCKET,
           ClientSocket = INVALID_SOCKET;
    struct addrinfo *result = NULL,
                    hints;
    char recvbuf[DEFAULT_BUFLEN];
    int iResult, iSendResult;
    int recvbuflen = DEFAULT_BUFLEN;

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }
...
```

Windows Socket Client

```
int __cdecl main(int argc, char **argv)
{
    WSADATA wsaData;
    SOCKET ConnectSocket = INVALID_SOCKET;
    struct addrinfo *result = NULL,
                    *ptr = NULL,
                    hints;
    char *sendbuf = "this is a test";
    char recvbuf[DEFAULT_BUFLEN];
    int iResult;
    int recvbuflen = DEFAULT_BUFLEN;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s server-name\n", argv[0]);
        return 1;
    }

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }
...
```

Windows Socket 例子

cl server.cpp /link "D:\Program Files\Microsoft SDKs\Windows\v6.0A\Lib\WS2_32.Lib"

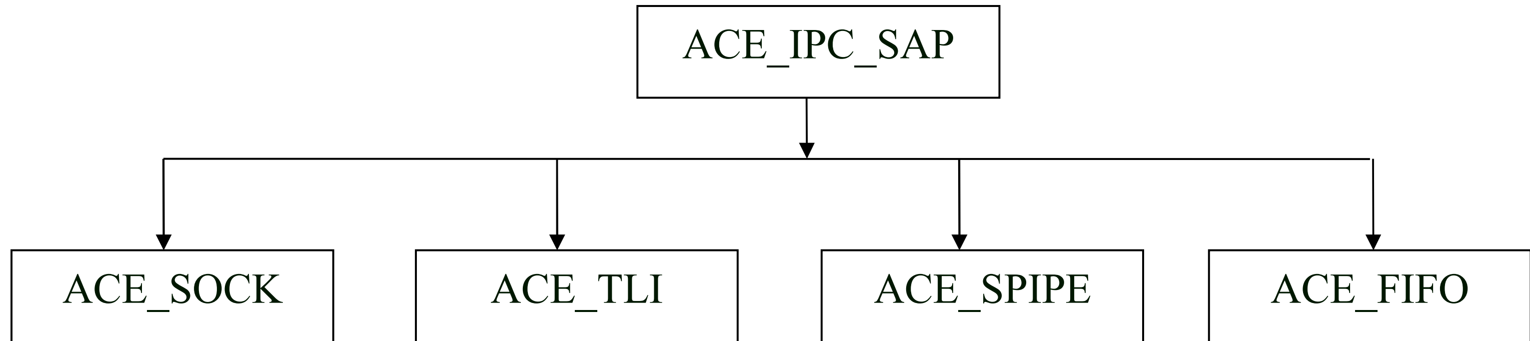
cl client.cpp /link "D:\Program Files\Microsoft SDKs\Windows\v6.0A\Lib\WS2_32.Lib"

运行:

server.exe

client.exe localhost

IPC SAP：进程间通信服务访问点包装

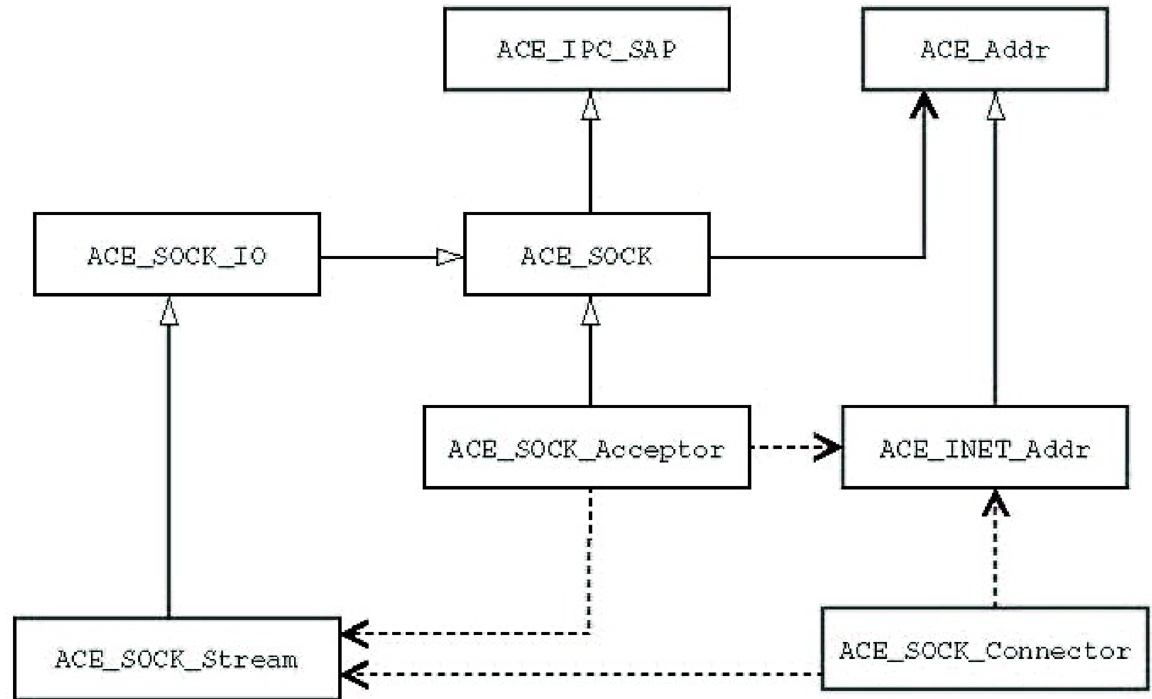


- ACE_SOCK 封装socket接口
- ACE_TLI 系统V传输层接口
- ACE_SPIPE SVR4 STREAM管道
- ACE_FIFO UNIX FIFO

除此之外，ACE还提供对ATM,DEV,FILE,SSL,UPIPE的封装

ACE Socket编程接口

- 增强的类型安全
- 确保可移植性
- 简单通用的使用
- 高层抽象的构件基础



- ACE_SOCK_Connector
- ACE_SOCK_Acceptor
- ACE_SOCK_Stream
- ACE_INET_Addr

ACE Socket编程接口

- Dgram类和Stream类：Dgram类基于UDP数据报协议,提供不可靠的无连接消息传递功能.另一方面,*Stream*类基于TCP协议,提供面向连接的消息传递。
- Acceptor、Connector类和Stream类：Acceptor和Connector类分别用于被动和主动地建立连接.Acceptor类封装BSD accept()调用,而Connector封装BSD connect()调用.Stream类用于在连接建立**之后**提供双向的数据流,并包含有发送和接收方法.

类名	**职责**
ACE_SOCK_Acceptor	用于被动的连接建立，基于BSD accept()和listen()调用。
ACE_SOCK_Connector	用于主动的连接建立，基于BSD connect()调用。
ACE_SOCK_Dgram	用于提供基于UDP（用户数据报协议）的无连接消息传递服务。封装了sendto()和receivefrom()等调用，并提供了简单的send()和recv()接口。
ACE_SOCK_IO	用于提供面向连接的消息传递服务。封装了send()、recv()和write()等调用。该类是ACE_SOCK_Stream和ACE_SOCK_CODgram类的基类。
ACE_SOCK_Stream	用于提供基于TCP（传输控制协议）的面向连接的消息传递服务。派生自ACE_SOCK_IO，并提供了更多的包装方法。
ACE_SOCK_CODgram	用于提供有连接数据报（connected datagram）抽象。派生自ACE_SOCK_IO；它包含的open()方法使用bind()来绑定到指定的本地地址，并使用UDP连接到远地地址。
ACE_SOCK_Dgram_Mcast	用于提供基于数据报的多点传送(multicast)抽象。包括预订多点传送组，以及发送和接收消息的方法
ACE_SOCK_Dgram_Bcast	用于提供基于数据报的广播(broadcast)抽象。包括在子网中向所有接口广播数据报消息的方法

TCP/IP Socket中的角色(Role)

- The ***active connection role*** (**`ACE_SOCK_Connector`**) is played by a peer application that initiates a connection to a remote peer
- The ***passive connection role*** (**`ACE_SOCK_Acceptor`**) is played by a peer application that accepts a connection from a remote peer &
- The ***communication role*** (**`ACE_SOCK_Stream`**) is played by both peer applications to exchange data after they are connected

SOCK SAP设计原则

－在编译时强制实现类型安全性: SOCK SAP类是强类型的, 非法操作在编译时、而不是运行时被拒绝;

－允许受控的类型安全性违例: 通过提供的get_handle和set_handle方法;

－为常见情况进行简化: 为常用方法参数提供缺省值(如: ACE_SOCK_Connector构造器有六个参数). 定义简洁的接口(如: 使用ACE_LSOCK* 类来传递socket句柄是非常简洁的). 将多个操作组合进单一操作(如: ACE_SOCK_Acceptor组合了socket、bind和listen);

－用层次类属替代一维的接口: 基类表示类属组件间的相似性,而派生类表示差异性;

－通过参数化类型增强可移植性: 参数化类型使应用与对特定的网络编程接口的依赖去耦合, 模板提供的类型抽象改善了支持不同网络编程接口(比如Socket或TLI)的平台间的可移植性;

－内联性能关键的方法: C++内联函数的使用以消除运行时函数调用开销;

－定义辅助类隐藏易错细节: Addr层次消除了与直接使用基于C的struct sockaddr数据结构族相关联的常见编程错误. 如: ACE_INET_Addr的构造器自动将sockaddr寻址结构清零, 并将端口号转换为网络字节序.

SOCK SAP 通信过程

ACE_SOCK 的通信过程一般为如下步骤：

1、服务器绑定端口，等待客户端连接；

2、客户端通过服务器的ip和服务器绑定的端口连接服务器；

3、服务器和客户端通过网络建立一条数据通路，通过这条数据通路进行数据交互.

1. ACE_INET_Addr类

是ACE_Addr的子类, 表示TCP/IP和UDP/IP的地址. 它通常包含机器的IP地址和端口号.

定义方法: ACE_INET_Addr addr(3000,"192.168.1.100");

2. ACE_SOCK_Acceptor类

服务期端使用, 用于绑定端口和被动地接受连接.

常用方法: open() 绑定端口, accept() 建立和客户段的连接

3. ACE_SOCK_Connector类

客户端使用, 用于主动的建立和服务器的连接.

常用方法: connect() 建立和服务器的连接

4. ACE_SOCK_Stream类

客户端和服务器都使用, 表示客户段和服务器之间的数据通路.

常用方法: send () 发送数据, recv () 接收数据, close() 关闭连接(实际上就是断开了socket连接)

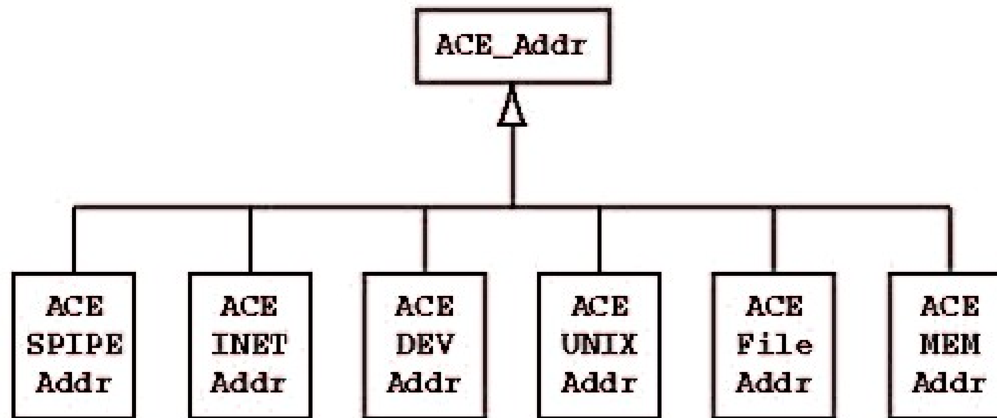
ACE Socket Server

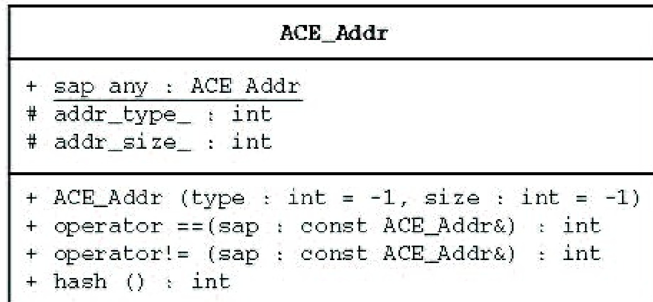
```
#include <ace/INET_Addr.h>
#include <ace/SOCK_Stream.h>
#include <ace/SOCK_Acceptor.h>

    ACE_INET_Addr port_to_listen(3000, "192.168.1.100");       // 要绑定的端口号
    ACE_SOCK_Acceptor acceptor;
    if (acceptor.open (port_to_listen, 1) == -1)     // 绑定端口
    {
        cout<<endl<<"bind port fail"<<endl;
        return -1;
    }

        ACE_SOCK_Stream peer;        // 和客户端的数据通路
        ACE_Time_Value timeout (10, 0); //

        if (acceptor.accept (peer) != -1)    // 建立和客户端的连接
        {
            cout<<endl<<endl<<"client connect. "<<endl;
            char buffer[1024];
            ssize_t bytes_received;

            ACE_INET_Addr raddr;
            peer.get_local_addr(raddr);
            cout<<endl<<"local port\t"<<raddr.get_host_name()<<"\t"<<raddr.get_port_number()<<endl; // raddr.get_ip_address() ?

            while ((bytes_received =
                peer.recv (buffer, sizeof(buffer))) != -1)    //读取客户端发送的数据
            {
                peer.send(buffer, bytes_received);    //对客户端发数据
            }
            peer.close ();        }
```

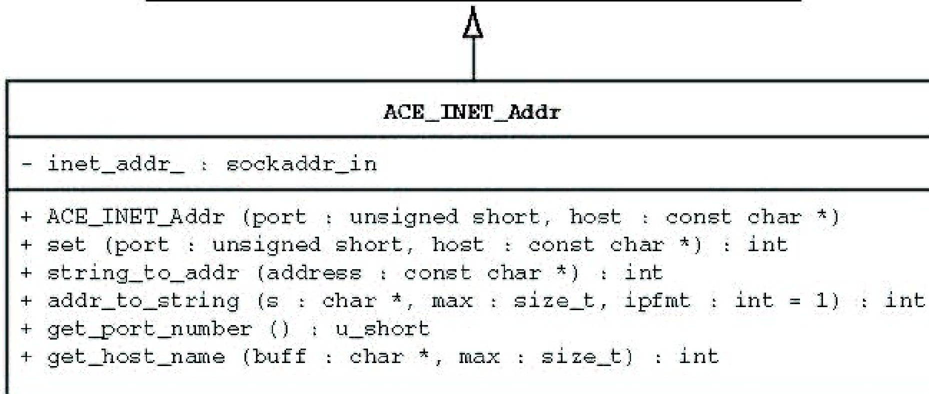
ACE Socket Client

```
ACE_INET_Addr addr(3000,"192.168.1.100");

ACE_SOCK_Connector connector;
ACE_Time_Value timeout(5,0);
ACE_SOCK_Stream peer;

if(connector.connect(peer,addr,&timeout) != 0)
{
    cout<<"connection failed !"<<endl;
    return 1;
}
cout<<"conneced !"<<endl;
string s="hello world";
peer.send(s.c_str(),s.length());    // 发送数据
cout<<endl<<"send:\t"<<s<<endl;

ssize_t bc=0;           // 接收的字节数
char buf[1024];
bc=peer.recv(buf,1024,&timeout);    // 接收数据
if(bc>=0)
{
    buf[bc]='\0';
    cout<<endl<<"rev:\t"<<buf<<endl; // 显示数据
}
peer.close();
```

ACE Socket Addressing Classes







Class Capabilities

- The **ACE_Addr** class is the root of the ACE network addressing hierarchy
- The **ACE_INET_Addr** class represents TCP/IP & UDP/IP addressing information
  - This class eliminates many subtle sources of accidental complexity

使用ACE_SOCK_Connector

```
#include <ace/INET_Addr.h>
#include <ace/SOCK_Stream.h>
#include <ace/SOCK_Connector.h>
#include <ace/Time_Value.h>

ACE_INET_Addr srvr (5000, ACE_LOCALHOST);
ACE_SOCK_Connector connector;
ACE_SOCK_Stream peer;

    if(connector.connect(peer,addr,&timeout) != 0)
    {
        cout<<"connection failed !"<<endl;
        return 1;
    }
    cout<<"conneced !"<<endl;
```

使用ACE_SOCK_Stream

```
#include <ace/INET_Addr.h>
#include <ace/SOCK_Stream.h>
#include <ace/SOCK_Acceptor.h>
#include <ace/Time_Value.h>

unsigned short portNumber = 5000;
ACE_INET_Addr myAddress(portNumber);

ACE_SOCK_Acceptor acceptor;
ACE_SOCK_Stream peer;

string s="hello world";
peer.send(s.c_str(),s.length());    //发送数据
cout<<endl<<"send:\t"<<s<<endl;

ssize_t bc=0;           //接收的字节数

char buf[1024];
bc=peer.recv(buf,1024,&timeout);    //接收数据
```

使用ACE_SOCK_Acceptor

```
#include <ace/INET_Addr.h>
#include <ace/SOCK_Stream.h>
#include <ace/SOCK_Acceptor.h>
#include <ace/Time_Value.h>

    if (acceptor.accept (peer) != -1)    //成功建立和客户端的连接
    {
       cout<<endl<<endl<<"client connect. "<<endl;
       char buffer[1024];
       ssize_t bytes_received;

       ACE_INET_Addr raddr;
       peer.get_local_addr(raddr);
           cout<<endl<<"local port\t"<<ACE_UINT32(raddr.get_ip_address())
<<"\t"<<raddr.get_port_number()<<endl; // raddr.get_host.address()

       while ((bytes_received =
          peer.recv (buffer, sizeof(buffer))) != -1)    //读取客户端发送的数据
       {
          peer.send(buffer, bytes_received);    //对客户端发数据
       }
```

使用ACE的UDP类进行网络编程

- ACE_SOCK_CODgram
- ACE_Asynch_Read_Dgram
- ACE_Asynch_Write_Dgram
- ACE_SOCK_Dgram

和TCP编程相比, UDP无需通过acceptor, connector来建立连接, 故代码相对TCP编程来说要简单许多. 另外, 由于UDP是一种无连接的通信方式, ACE_SOCK_Dgram的实例对象中无法保存远端地址信息(保存了本地地址信息), 故通信的时候需要加上远端地址信息.

UDP通信过程如下:

1、服务器端绑定一固定UDP端口, 等待接收客户端的通信;
2、客户端通过服务器的IP和地址信息直接对服务器端发送消息;
3、服务器端收到客户端发送的消息后获取客户端的IP和端口号,通过该地址信息和客户端通信.

UDP Server 例子

```
#include <ace/SOCK_Dgram.h>
#include <ace/INET_Addr.h>
#include <ace/Time_Value.h>
#include <string>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    ACE_INET_Addr port_to_listen(3000);    // 要绑定的端口
    ACE_SOCK_Dgram peer(port_to_listen);    // 通信通道
    char buf[100];
    while(true)
    {
        ACE_INET_Addr remoteAddr;    // 所连接的远程地址
        int bc = peer.recv(buf,100,remoteAddr);    // 接收消息, 并获取远程地址信息
        if( bc != -1)
        {
            string s(buf,bc);
            cout<<endl<<"rev:\t"<<s<<endl;
        }
        peer.send(buf,bc,remoteAddr);    // 和远程地址通信
    }
    return 0;
}
```

UDP Client 例子

```
#include <ace/SOCK_Dgram.h>
#include <ace/INET_Addr.h>
#include <ace/Time_Value.h>
#include <string>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    ACE_INET_Addr remoteAddr(3000,"127.0.0.1");    //所连接的远程地址
    ACE_INET_Addr localAddr;    //本地地址信息
    ACE_SOCK_Dgram peer(localAddr);    //通信通道

    peer.send("hello",5,remoteAddr);    //发送消息
    char buf[100];
    int bc = peer.recv(buf,100,remoteAddr);    //接收消息
    if( bc != -1)
    {
        string s(buf,bc);
        cout<<endl<<"rev:\t"<<s<<endl;
    }
    return 0;
}
```

单播(Unicast)

单播与TCP类似, 是一种一对一操作.

与TCP通信两点主要的不同：

- 你只需要打开使用ACE_SOCK_Dgram. 不需要使用接受器和连接器.
- 在发送数据报时, 你需要显式地指定对端的地址.

使用单播(Unicast)

```
#include "ace/OS.h"

#include "ace/Log_Msg.h"

#include "ace/INET_Addr.h"

#include "ace/SOCK_Dgram.h"

 int send_unicast (const ACE_INET_Addr &to)

{

   const char *message = "this is the message!\n";

   ACE_INET_Addr my_addr (ACE_static_cast (u_short, 10101));

   ACE_SOCK_Dgram udp (my_addr);

   ssize_t sent = udp.send (message, ACE_OS_String::strlen (message) + 1,
to);

   udp.close ();

   if (sent == -1)

       ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"), ACE_TEXT
("send")), -1);

return 0;

}
```

广播(Broadcast)

在广播模式中, 你必须为每个发送操作指定目的地址.

ACE_SOCK_Dgram_Bcast 类负责为你提供正确的IP广播地址.

你只需要指定要用于广播的UDP端口号.

因为 ACE_SOCK_Dgram_Bcast 是 ACE_SOCK_Dgram 的子类, 所以全部数据报接收操作与单播的操作都是相似的.

使用广播(Broadcast)

```
#include "ace/OS.h"
#include "ace/Log_Msg.h"
#include "ace/INET_Addr.h"
#include "ace/SOCK_Dgram_Bcast.h"
int send_broadcast (u_short to_port)
{
  const char *message = "this is the message!\n";
  ACE_INET_Addr my_addr (ACE_static_cast (u_short, 10101));
  ACE_SOCK_Dgram_Bcast udp (my_addr);
  ssize_t sent = udp.send (message,
                           ACE_OS_String::strlen (message) + 1,
                           to_port);
  udp.close ();
  if (sent == -1)
    ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                      ACE_TEXT ("send")), -1);
  return 0;
}
```

多播(Multicast)

多播模式涉及到称为多播组的一组网络节点. OS提供的底层协议软件会使用专门的协议管理多播组.

OS会根据应用是请求加入(订阅)还是离开(取消订阅)特定的多播组来指挥组的操作.

一旦应用加入了某个组, 在已加入的 socket 上发送的所有数据报都会发往该多播组, 而且不用指定每个发送操作的目的地址.

每个多播组都有一个单独的IP地址. 多播地址是D类IP地址. 如: 224.2.2.2
D类地址范围：224.0.0.1到239.255.255.255

ACE_SOCK_Dgram_Mcast 也是 ACE_SOCK_Dgram 的子类, 因此 recv() 方法也是从 ACE_SOCK_Dgram 继承来的.
如: ssize_t recv_cnt = udp.recv(buff, buflen, your_addr);

使用多播(Multicast)

```
#include "ace/OS.h"
#include "ace/Log_Msg.h"
#include "ace/INET_Addr.h"
#include "ace/SOCK_Dgram_Mcast.h"
int send_multicast (const ACE_INET_Addr &mcast_addr)
{
  const char *message = "this is the message!\n";
  ACE_SOCK_Dgram_Mcast udp;
  if (-1 == udp.join (mcast_addr))
    ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                       ACE_TEXT ("join")), -1);
  ssize_t sent = udp.send (message,
                           ACE_OS_String::strlen (message) + 1);
  udp.close ();
  if (sent == -1)
    ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                       ACE_TEXT ("send")), -1);
  return 0;
```

其他通信方式

一、Files

- ACE_FILE_IO
- ACE_FILE_Connector
- ACE_FILE_Addr

二、Pipes and FIFOs

- ACE_FIFO_Recv, ACE_FIFO_Send, ACE_FIFO_Recv_Msg, ACE_FIFO_Send_Msg
- ACE_Pipe
- ACE_SPIPE_Acceptor, ACE_SPIPE_Connector, ACE_SPIPE_Stream, ACE_SPIPE_Addr

三、Shared Memory Stream

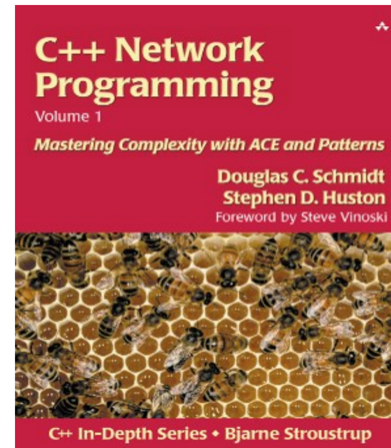
- ACE_MEM_Acceptor
- ACE_MEM_Connector
- ACE_MEM_Stream
- ACE_MEM_Addr

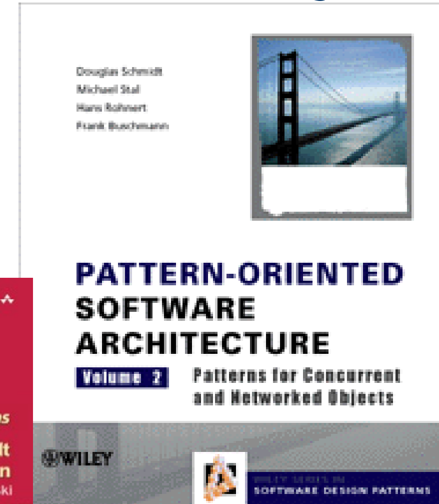
内容回顾

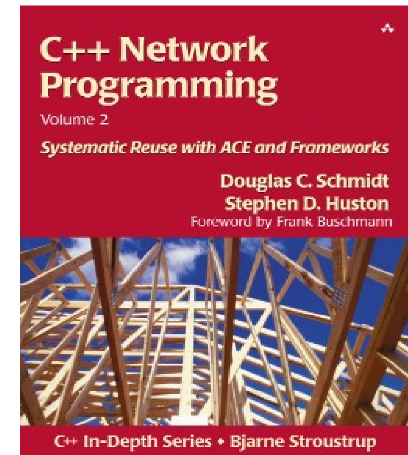
- 如何访问OS服务
- TCP/IP Socket编程接口
- 使用ACE的UDP类进行网络编程
- 单播、广播、多播

参考资料

- **Patterns & frameworks for concurrent & networked objects**
  - `www.posa.uci.edu`
- **ACE & TAO open-source middleware**
  - `www.cs.wustl.edu/~schmidt/ACE.html`
  - `www.cs.wustl.edu/~schmidt/TAO.html`







- **ACE research papers**
  - `www.cs.wustl.edu/~schmidt/ACE-papers.html`
- **Extended ACE & TAO tutorials**
  - UCLA extension, January 21-23, 2004
  - `www.cs.wustl.edu/~schmidt/UCLA.html`
- **ACE books**
  - `www.cs.wustl.edu/~schmidt/ACE/`

结束

谢谢大家！

ihuihoo@gmail.com
http://www.huihoo.com