第2章　IPC SAP：进程间通信服务访问点包装

socket、TLI、STREAM管道和FIFO为访问局部和全局IPC机制提供广泛的接口。但是，有许多问题与这些不统一的接口有关联。比如类型安全的缺乏和多维度的复杂性会导致成问题的和易错的编程。

ACE的IPC SAP类属提供了统一的层次类属，对那些麻烦而易错的接口进行封装。在保持高性能的同时，IPC SAP被设计用于改善通信软件的*正确性*、*易学性*、*可移植性*和*可复用性*。

2.1 IPC SAP类属

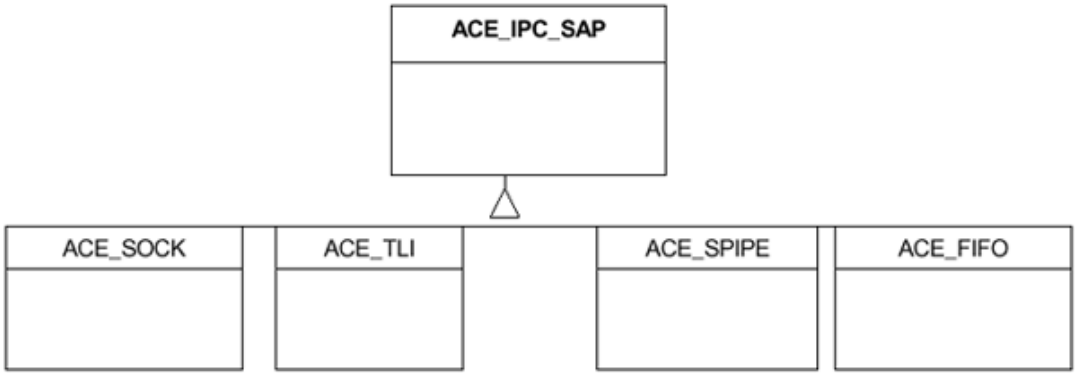

图2-1 IPC SAP类属

根据底层使用的不同IPC接口，IPC SAP类被划分为四种主要的类属，图2-1描述了这一划分。ACE_IPC_SAP类提供的一些函数是所有IPC接口公有的。有四个不同的类由此类派生而出，每个类各自代表ACE包含的一种IPC SAP包装类属。这些类封装适用于特定IPC接口的功能。例如，ACE_SOCK类包含的功能适用于BSD socket编程接口，而ACE_TLI包装TLI编程接口。

在这四个类的每一个类下面都有一整层次的包装类，它们完整地包装底层接口，并提供高度可复用、模块化、安全和易用的包装类。

2.2 socket类属（ACE_SOCK）

该类属中的类都位于ACE_SOCK之下；它提供使用BSD socket编程接口的Internet域和UNIX域协议族的接口。这个类属中的类被进一步划分为：

- 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。

表2-1详细描述了该类属中的类以及它们的职责：

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

表2-1 ACE_SOCK中的类及其职责

在下面的部分，我们将要演示怎样将IPC_SAP包装类直接用于处理进程间通信。记住这些只是ACE的冰山一角。在教程的后续章节中将会介绍其他类和组件。

2.2.1 使用ACE的流

ACE中的流包装提供面向连接的通信。流数据传输包装类包括ACE_SOCK_Stream和ACE_LSOCK_Stream，它们分别包装TCP/IP和UNIX域socket协议数据传输功能。连接建立类包括针

对 TCP/IP 的 ACE_SOCK_Connector 和 ACE_SOCK_Acceptor ， 以 及 针 对 UNIX 域 socket 的 ACE_LSOCK_Connector和ACE_LSOCK_Acceptor。

Acceptor类用于被动地接受连接（使用BSD accept()调用），而Connector类用于主动地建立连接（使用BSD connect()调用）。

下面的例子演示接收器和连接器是怎样用于建立连接的。该连接随后将用于使用流数据传输类来传输数据。

例2-1

```
#include "ace/SOCK_Acceptor.h"
#include "ace/SOCK_Stream.h"
#define SIZE_DATA 18
#define SIZE_BUF 1024
#define NO_ITERATIONS 5


class Server
{
public:
Server (int port): server_addr_(port),peer_acceptor_(server_addr_)
{
      data_buf_= new char[SIZE_BUF];
}


//Handle the connection once it has been established. Here the
//connection is handled by reading SIZE_DATA amount of data from the
//remote and then closing the connection stream down.
int handle_connection()
{
      // Read data from client
      for(int i=0;i<NO_ITERATIONS;i++)
      {
            int byte_count=0;
            if( (byte_count=new_stream_.recv_n (data_buf_, SIZE_DATA, 0))==-1)
                        ACE_ERROR ((LM_ERROR, "%p\n", "Error in recv"));
            else
            {
                        data_buf_[byte_count]=0;
                        ACE_DEBUG((LM_DEBUG,"Server received %s
                        \n",data_buf_));
            }
```

```
      }
      // Close new endpoint
      if (new_stream_.close () == -1)
            ACE_ERROR ((LM_ERROR, "%p\n", "close"));
      return 0;
}

//Use the acceptor component peer_acceptor_ to accept the connection
//into the underlying stream new_stream_. After the connection has been
//established call the handle_connection() method.
int accept_connections ()
{
      if (peer_acceptor_.get_local_addr (server_addr_) == -1)
            ACE_ERROR_RETURN ((LM_ERROR,"%p\n","Error in get_local_addr"),1);

      ACE_DEBUG ((LM_DEBUG,"Starting server at port %d\n",
      server_addr_.get_port_number ()));

      // Performs the iterative server activities.
      while(1)
      {
            ACE_Time_Value timeout (ACE_DEFAULT_TIMEOUT);
            if (peer_acceptor_.accept (new_stream_, &client_addr_, &timeout)== -1)
            {
                  ACE_ERROR ((LM_ERROR, "%p\n", "accept"));
                  continue;
            }
            else
            {
                  ACE_DEBUG((LM_DEBUG,
                              "Connection established with
                              remote %s:%d\n",
                                    client_addr_.get_host_name(),client_addr_.get_port_number()));
                  //Handle the connection
                  handle_connection();
            }
      }
}

private:
char *data_buf_;
```

```
ACE_INET_Addr server_addr_;
ACE_INET_Addr client_addr_;
ACE_SOCK_Acceptor peer_acceptor_;
ACE_SOCK_Stream new_stream_;
};

int main (int argc, char *argv[])
{
if(argc<2)
{
      ACE_ERROR((LM_ERROR,"Usage %s <port_num>", argv[0]));
      ACE_OS::exit(1);
}
Server server(ACE_OS::atoi(argv[1]));
server.accept_connections();
}
```

上面的例子创建了一个被动服务器，侦听到来的客户连接。在连接建立后，服务器接收来自客户的数据，然后关闭连接。Server类表示该服务器。

Server类包含的accept_connections()方法使用接受器（也就是ACE_SOCK_Acceptor）来将连接接受“进”ACE_SOCK_Stream new_stream_。该操作是这样来完成的：调用接受器上的accept()，并将流作为参数传入其中；我们想要接受器将连接接受进这个流。一旦连接已建立进流中，流的包装方法send()和recv()就可以用来在新建立的链路上发送和接收数据。还有一个空的ACE_INET_Addr也被传入接受器的accept()方法，并在其中被设定为发起连接的远地机器的地址。

在连接建立后，服务器调用handle_connection()方法，它开始从客户那里读取一个预先知道的单词，然后将流关闭。对于要处理多个客户的服务器来说，这也许并不是很实际的情况。在现实世界的情况中可能发生的是，连接在单独的线程或进程中被处理。在后续章节中将反复演示怎样完成这样的多线程和多进程类型的处理。

连接关闭通过调用流上的close()方法来完成，该方法会释放所有的socket资源并终止连接。

下面的例子演示怎样与前面例子中演示的接受器协同使用连接器。

例2-2

```
#include "ace/SOCK_Connector.h"
#include "ace/INET_Addr.h"
#define SIZE_BUF 128
#define NO_ITERATIONS 5

class Client
```

```
{
public:
Client(char *hostname, int port):remote_addr_(port,hostname)
{
      data_buf_="Hello from Client";
}

//Uses a connector component `connector_' to connect to a
//remote machine and pass the connection into a stream
//component client_stream_
int connect_to_server()
{
      // Initiate blocking connection with server.
      ACE_DEBUG ((LM_DEBUG, "(%P|%t) Starting connect to %s:%d\n",
                        remote_addr_.get_host_name(),remote_addr_.get_port_number()));
      if (connector_.connect (client_stream_, remote_addr_) == -1)
            ACE_ERROR_RETURN ((LM_ERROR,"(%P|%t) %p\n","connection failed"),-1);
      else
            ACE_DEBUG ((LM_DEBUG,"(%P|%t) connected to %s\n",
                              remote_addr_.get_host_name ()));
      return 0;
}

//Uses a stream component to send data to the remote host.
int send_to_server()
{
      // Send data to server
      for(int i=0;i<NO_ITERATIONS; i++)
      {
            if (client_stream_.send_n (data_buf_,
                              ACE_OS::strlen(data_buf_)+1, 0) == -1)
            {
                        ACE_ERROR_RETURN ((LM_ERROR,"(%P|%t)
                        %p\n","send_n"),0);
                        break;
            }
      }

      //Close down the connection
      close();
}
```

```
//Close down the connection properly.
int close()
{
      if (client_stream_.close () == -1)
            ACE_ERROR_RETURN ((LM_ERROR,"(%P|%t) %p\n","close"),-1);
      else
            return 0;
}

private:
ACE_SOCK_Stream client_stream_;
ACE_INET_Addr remote_addr_;
ACE_SOCK_Connector connector_;
char *data_buf_;
};

int main (int argc, char *argv[])
{
if(argc<3)
{
      ACE_DEBUG((LM_DEBUG,"Usage %s <hostname> <port_number>\n", argv[0]));
      ACE_OS::exit(1);
}
Client client(argv[1],ACE_OS::atoi(argv[2]));
client.connect_to_server();
client.send_to_server();
}
```

上面的例子演示的客户主动连接到例2-1所描述的服务器。在建立连接后，客户将单个字符串发送若干次到服务器，并关闭连接。

客户由单个Client类表示。Client含有connect_to_server()和send_to_server()方法。

Connect_to_server()方法使用类型为ACE_SOCK_Connector的连接器（connector_）来主动地建立连接。连接的设置通过调用连接器connector_上的connect()方法来完成：传入的参数为我们想要连接的机器的地址remote_addr_，以及用于在其中建立连接的空ACE_SOCK_Stream client_stream_。远地机器在例子的运行时参数中指定。一旦connect()方法成功返回，通过使用ACE_SOCK_Stream封装类中的send()和recv()方法族，流就可以用于在新建立的链路上发送和接收数据了。

在此例中，一旦连接建立好，send_to_server()方法就会被调用，以将一个字符串发送NO_ITERATIONS次到服务器。如前面所提到的，这是通过使用流包装类的send()方法来完成的。

2.2.2 使用ACE的数据报

ACE_SOCK_Dgram和ACE_LSOCK_Dgram是ACE中的数据报包装类。这些包装包含了发送和接收数据报的方法，并包装了非面向连接的UDP协议和UNIX域socket协议。与流包装不同，这些包装封装的是非面向连接的协议。这也就意味着不存在用于“设置”连接的接受器和连接器。相反，在这种情况下，通信通过一系列的发送和接收来完成。每个send()都要指定目的远地地址作为参数。下面的例子演示怎样通过ACE使用数据报。这个例子使用了ACE_SOCK_Dgram包装（也就是UDP包装）。还可以使用包装UNIX域数据报的ACE_LSOCK_Dgram。两种包装的用法非常类似，唯一的不同是ACE_LSOCK_Dgram要用 ACE_UNIX_Addr类作为地址，而不是ACE_INET_Addr。

例2-3

```
//Server
#include "ace/OS.h"
#include "ace/SOCK_Dgram.h"
#include "ace/INET_Addr.h"

#define DATA_BUFFER_SIZE 1024
#define SIZE_DATA 19

class Server
{
public:
Server(int local_port)
      :local_addr_(local_port),local_(local_addr_)
{
      data_buf = new char[DATA_BUFFER_SIZE];
}



//Expect data to arrive from the remote machine. Accept it and display
//it. After receiving data, immediately send some data back to the
//remote.
int accept_data()
{
      int byte_count=0;
      while( (byte_count=local_.recv(data_buf,SIZE_DATA,remote_addr_))!=-1)
      {
            data_buf[byte_count]=0;
            ACE_DEBUG((LM_DEBUG, "Data received from remote %s was %s \n"
                                          ,remote_addr_.get_host_name(),
                                          data_buf));
```

```
            ACE_OS::sleep(1);
            if(send_data()==-1) break;
      }
      return -1;
}

//Method used to send data to the remote using the datagram component
//local_
int send_data()
{
      ACE_DEBUG((LM_DEBUG,"Preparing to send reply to client %s:%d\n",
                        remote_addr_.get_host_name(),remote_addr_.get_port_number()));
      ACE_OS::sprintf(data_buf,"Server says hello to you too");
      if( local_.send(data_buf, ACE_OS::strlen(data_buf)+1,remote_addr_)==-1)
            return -1;
      else
            return 0;
}

private:
char *data_buf;
ACE_INET_Addr remote_addr_;
ACE_INET_Addr local_addr_;
ACE_SOCK_Dgram local_;
};

int main(int argc, char *argv[])
{
if(argc<2)
{
      ACE_DEBUG((LM_DEBUG,"Usage %s <Port Number>", argv[0]));
      ACE_OS::exit(1);
}
Server server(ACE_OS::atoi(argv[1]));
server.accept_data();
}
```

上面的代码是一个简单的服务器，它等待客户应用通过周知端口给它发送一个数据报。在该数据报中含有定长的和预定数量的数据。服务器在收到这些数据时，就发送回复给原先发送数据的客户。

Server类拥有名为local_的ACE_SOCK_Dgram私有成员，它被同时用于接收和发送数据。Server在它的构造器中通过已知的ACE_INET_Addr(本地主机以及已知端口)实例化local_，这样客户就可以对它进行定位、并发送消息给它了。

Server类包含两个方法：accept_data()，用于从客户接收数据（使用recv()调用包装）；以及send_data()，用于发送数据给远地客户（使用send()调用包装）。注意local_包装类的send()和receive()的底层调用都包装了BSD sendto()和recvfrom()调用，并具有相类似的特征。

主函数实例化Server类型的对象、并调用它的accept_data()方法，等待来自客户的数据。当它得到所需的数据后，它调用send_data()发送回复消息给客户。如此循环往复，直到客户被关闭为止。

相应的客户代码与前面的服务器例子非常类似：

例2-4

```
//Client
#include "ace/OS.h"
#include "ace/SOCK_Dgram.h"
#include "ace/INET_Addr.h"
#define DATA_BUFFER_SIZE 1024
#define SIZE_DATA 28


class Client
{
public:
Client(char * remote_host,int port)
      :remote_addr_(remote_host),
      local_addr_((u_short)0),local_(local_addr_)
{
      data_buf = new char[DATA_BUFFER_SIZE];
      remote_addr_.set_port_number(port);
}


//Receive data from the remote host using the datgram wrapper `local_'.
//The address of the remote machine is received in `remote_addr_'
//which is of type ACE_INET_Addr. Remember that there is no established
//connection.
int accept_data()
{
      if(local_.recv(data_buf,SIZE_DATA,remote_addr_)!=-1)
      {
            ACE_DEBUG((LM_DEBUG, "Data received from remote server %s was: %s \n" ,
```

```
                                        remote_addr_.get_host_name(),
                                        data_buf));
            return 0;
      }
      else
            return -1;
}

//Send data to the remote. Once data has been sent wait for a reply
//from the server.
int send_data()
{
      ACE_DEBUG((LM_DEBUG,"Preparing to send data to server %s:%d\n",
                  remote_addr_.get_host_name(),remote_addr_.get_port_number()));
      ACE_OS::sprintf(data_buf,"Client says hello");

      while(local_.send(data_buf,ACE_OS::strlen(data_buf),remote_addr_)!=-1)
      {
            ACE_OS::sleep(1);
            if(accept_data()==-1)
                  break;
      }

      return -1;
}

private:
char *data_buf;
ACE_INET_Addr remote_addr_;
ACE_INET_Addr local_addr_;
ACE_SOCK_Dgram local_;
};

int main(int argc, char *argv[])
{
if(argc<3)
{
      ACE_OS::printf("Usage: %s <hostname> <port_number> \n", argv[0]);
      ACE_OS::exit(1);
}
Client client(argv[1],ACE_OS::atoi(argv[2]));
```

```
client.send_data();
}
```

2.2.3 使用ACE的多点传送（Multicast）

你会发现，在许多场合，同样的消息必须被发送给你的分布式系统中的众多客户或服务器。例如，可能需要将时间调整更新或其他的周期性信息广播给特定的终端集。多点传送被用于处理这一问题。它允许对特定的终端子集或组、而不是所有终端进行广播。因此，你可以认为多点传送是一种受控的广播机制。大多数现代OS都提供多点传送功能。

ACE提供ACE_SOCK_Dgram_Mcast包装，封装了不可靠的多点传送。它允许程序员将数据报消息发送给被称为“多点传送组”的受控组。这样的组由唯一的多点传送地址标识。

对在此地址上接收广播有兴趣的客户和服务器必须进行预订（也被称为“多点传送组预订”）。于是，所有预订到此多点传送组的进程将会接收到所有发送给该组的数据报消息。仅仅想要给多点传送组发送消息，而不需要收听它们的应用，无需进行预订。实际上，这样的发送者可以使用原有的简单ACE_SOCK_Dgram包装给多点传送地址发送消息，整个多点传送组将随之收到发送出的消息。

在ACE中，多点传送功能被封装在ACE_SOCK_Dgram_Mcast中，其中包括在多点传送组上的预订、取消预订和接收功能。

下面的例子演示在ACE中是怎样使用多点传送的：

例2-5

```
#include "ace/SOCK_Dgram_Mcast.h"
#include "ace/OS.h"
#define DEFAULT_MULTICAST_ADDR "224.9.9.2"
#define TIMEOUT 5

//The following class is used to receive multicast messages from
//any sender.
class Receiver_Multicast
{
public:
Receiver_Multicast(int port):
      mcast_addr_(port,DEFAULT_MULTICAST_ADDR),remote_addr_((u_short)0)
{
      // Subscribe to multicast address.
      if (mcast_dgram_.subscribe (mcast_addr_) == -1)
      {
            ACE_DEBUG((LM_DEBUG,"Error in subscribing to Multicast address \n"));
            exit(-1);
      }
```

```
}

~Receiver_Multicast()
{
      if(mcast_dgram_.unsubscribe()==-1)
            ACE_DEBUG((LM_ERROR,"Error in unsubscribing from Mcast group\n"));
}

//Receive data from someone who is sending data on the multicast group
//address. To do so it must use the multicast datagram component
//mcast_dgram_.
int recv_multicast()
{
      //get ready to receive data from the sender.
      if(mcast_dgram_.recv (&mcast_info,sizeof (mcast_info),remote_addr_)==-1)
            return -1;
      else
      {
            ACE_DEBUG ((LM_DEBUG, "(%P|%t) Received multicast from %s:%d.\n",
                        remote_addr_.get_host_name(),
                        remote_addr_.get_port_number()));
            ACE_DEBUG((LM_DEBUG,"Successfully received %d\n", mcast_info));
            return 0;
      }
}

private:
ACE_INET_Addr mcast_addr_;
ACE_INET_Addr remote_addr_;
ACE_SOCK_Dgram_Mcast mcast_dgram_;
int mcast_info;
};

int main(int argc, char*argv[])
{
Receiver_Multicast m(2000);

//Will run forever
while(m.recv_multicast()!=-1)
{
      ACE_DEBUG((LM_DEBUG,"Multicaster successful \n"));
```

```
}

ACE_DEBUG((LM_ERROR,"Multicaster failed \n"));
exit(-1);
}
```

上面的例子说明应用怎样使用ACE_SOCK_Dgram_Mcast预订多点传送组，以及从多点传送组接收消息。

Receiver_Multicast类的构造器将对象预订到多点传送组，析构器取消预订。一旦预订之后，应用无限期地等待任何发往此多点传送地址的数据。

下一个例子说明应用怎样使用ACE_SOCK_Dgram包装类将数据报消息发送到多点传送地址或组。

例2-6

```
#include "ace/SOCK_Dgram_Mcast.h"
#include "ace/OS.h"
#define DEFAULT_MULTICAST_ADDR "224.9.9.2"
#define TIMEOUT 5

class Sender_Multicast
{
public:
Sender_Multicast(int port):
      local_addr_((u_short)0),dgram_(local_addr_),
      multicast_addr_(port,DEFAULT_MULTICAST_ADDR)
{
}

//Method which uses a simple datagram component to send data to the //multicast group.
int send_to_multicast_group()
{
      //Convert the information we wish to send into network byte order
      mcast_info= htons (1000);

      // Send multicast
      if(dgram_.send (&mcast_info, sizeof (mcast_info), multicast_addr_)==-1)
            return -1;

      ACE_DEBUG ((LM_DEBUG,
```

```
            "%s; Sent multicast to group. Number sent is %d.\n",
            __FILE__,
            mcast_info));

      return 0;
}

private:
ACE_INET_Addr multicast_addr_;
ACE_INET_Addr local_addr_;
ACE_SOCK_Dgram dgram_;
int mcast_info;
};

int main(int argc, char*argv[])
{
Sender_Multicast m(2000);
if(m.send_to_multicast_group()==-1)
{
      ACE_DEBUG((LM_ERROR,"Send to Multicast group failed \n"));
      exit(-1);
}
else
      ACE_DEBUG((LM_DEBUG,"Send to Multicast group successful \n"));
}
```

在此例中，客户使用数据报包装给多点传送组发送数据。Sender_Multicast类含有一个简单的send_to_multicast_group()方法。该方法使用数据报包装组件dgram_发送单个消息给多点传送组，消息中仅包含一个整数。当接收者接收到此消息时，它把该整数打印到标准输出。