ACE 自适配通信环境中文技术文档

中篇：ACE 程序员教程

作者：Umar Syyid usyyid@hns.com
译者：马维达 http://www.flyingdonkey.com/

致谢

谨对以下人士致以谢忱，是他们的协助使此教程成为可能。

Ambreen Ilyas ambreen@bitsmart.com
James CE Johnson jcej@lads.com
Aaron Valdivia avaldivia@hns.com

Douglas C. Schmidt	schmidt@cs.wustl.edu
Thomas Jordan	ace@programmer.net
Erik Koerber	erik.koerber@siemens.at
Martin Krumpolec	krumpo@pobox.sk
Fred Kuhns	fredk@tango.cs.wustl.edu
Susan Liebeskind	shl@cc.gatech.edu
Andy Bellafaire	amba@callisto.eci-esyst.com
Marina	marina@cs.wustl.edu
Jean-Paul Genty	jpgenty@sesinsud.com

第 1 章 ACE 自适配通信环境

ACE 自适配通信环境 (Adaptive Communication Environment)是面向对象的框架和工具包，它为通信软件实现了核心的并发和分布式模式。ACE 包含的多种组件可以帮助通信软件的开发获得更好的灵活性、效率、可靠性和可移植性。ACE 中的组件可用于以下几种目的：

- 并发和同步
- 进程间通信(IPC)
- 内存管理
- 定时器
- 信号
- 文件系统管理
- 线程管理
- 事件多路分离和处理器分派
- 连接建立和服务初始化
- 软件的静态和动态配置、重配置
- 分层协议构建和流式框架
- 分布式通信服务：名字、日志、时间同步、事件路由和网络锁定，等等。

1.1 ACE 体系结构

如图 1-1 所示，ACE 具有分层的体系结构。在 ACE 框架中有三个基本层次：

- 操作系统（OS）适配层
- C++包装层
- 框架和模式层

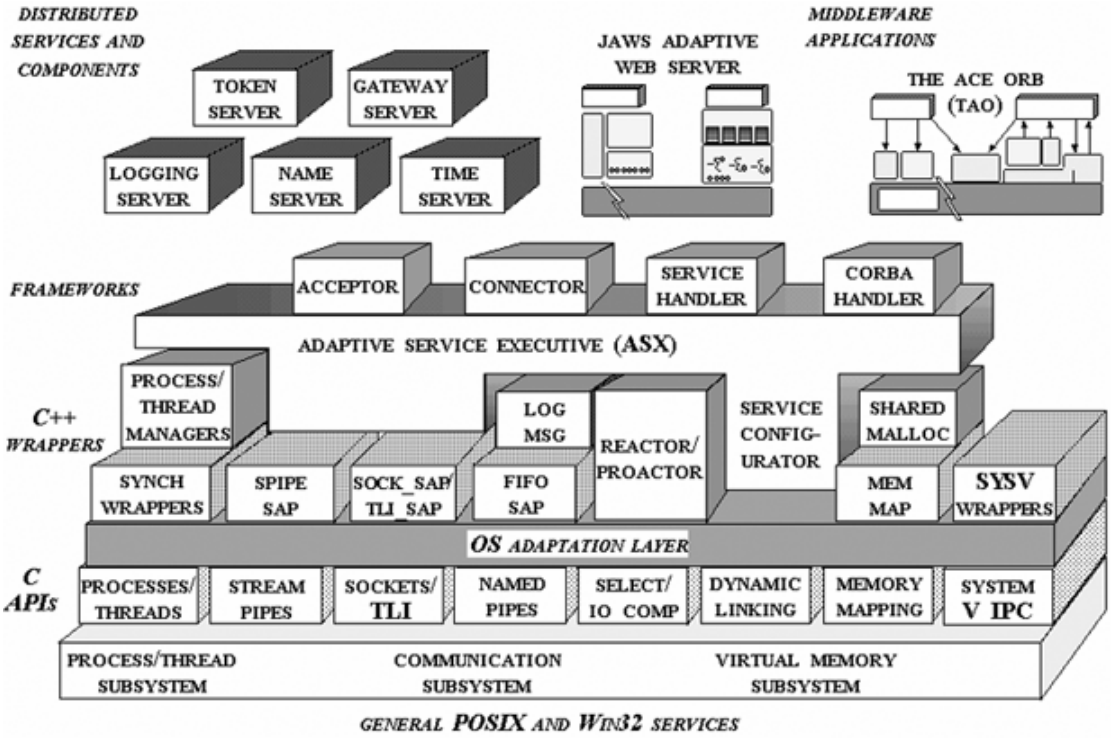

图 1-1 ACE 的体系结构

1.1.1 OS 适配层

OS 适配层是位于本地 OS API 和 ACE 之间的“瘦”代码层，它使 ACE 的较高层与平台依赖性屏蔽开来，从而使得通过 ACE 编写的代码保持了相对的平台无关性。只需要极少的努力，开发者就可以将 ACE 应用移植到任何平台上。

OS 适配层也是 ACE 框架之所以可用于如此多的平台的原因所在。目前 ACE 适用的 OS 平台包括：实时 OS（VxWorks、Chorus、LynxOS 和 pSoS）、大多数版本的 UNIX（SunOS 4.x 和 5.x; SGI IRIX 5.x 和 6.x; HP-UX 9.x, 10.x 和 11.x; DEC UNIX 3.x 和 4.x; AIX 3.x 和 4.x; DG/UX; Linux; SCO; UnixWare; NetBSD 和 FreeBSD）、Win32（使用 MSVC++和 Borland C++的 WinNT 3.5.x、4.x、Win95 和 WinCE）以及 MVS OpenEdition。

1.1.2 C++包装层

C++包装层包括一些 C++包装类，它们可用于构建高度可移植的和类型安全的 C++应用。这是 ACE 工具包最大的一部分，大约包含了总源码的 50%。C++包装类可用于：

- **并发和同步：**ACE 提供若干并发和同步包装类，对本地 OS 多线程和多进程 API 进行了抽象。这些包装类封装用于线程和进程的原语，比如信号量、锁、栅栏（Barrier）和条件变量。另外还有更高级的原语可用，比如守卫（Guard）。所有这些原语共享类似的接口，因而很容易使用和相互替换。
- **IPC：**ACE 提供若干 C++包装类，封装不同 OS 中不同的进程间通信（IPC）接口。例如，ACE 的包装类封装了以下 IPC 机制：BSD socket、TLI、UNIX FIFO、流管道、Win32 命名管道，等等。ACE 还为消息队列提供包装类，包括特定的实时 OS 的消息队列。

- **内存管理组件：**ACE 包含的一些类可用于内存动态分配和释放；其中包括允许预分配所有动态内存的类。这些预分配的内存随即通过 ACE 提供的管理类的帮助进行本地管理。在大多数实时和嵌入式系统中，这样的细粒度管理极为必要。另外还有一些类用于灵活地管理进程间共享内存。
- **定时器类：**有多种不同的类可用于处理定时器的调度和取消。ACE 中不同种类的定时器使用不同的底层机制（堆、定时器轮（timer wheel）或简单列表）来提供不同的性能特性。但是，不管底层使用何种机制，这些类的接口都是一致的，从而使得开发者很容易使用任何一种定时器类。除了这些定时器类，还有封装高分辨率定时器（在部分平台上可用，比如 VxWorks, Win32/Pentium, AIX 和 Solaris）和 Profile Timer 的包装类。
- **容器类：**ACE 还拥有若干可移植的 STL 风格的容器类，比如 Map、Hash_Map、Set、List，等等
- **信号处理：**ACE 提供对特定 OS 的信号处理接口进行封装的包装类。这些类使得开发者能够很容易地安装和移除信号处理器，并且可以为一个信号安装若干处理器。另外还有信号守卫类，可用于在看守的作用域之内禁止所有信号。
- **文件系统组件：**ACE 含有包装文件系统 API 的类。这些类包括文件 I/O、异步文件 I/O、文件加锁、文件流、文件连接包装，等等。
- **线程管理：**ACE 提供包装类来创建和管理线程。这些包装还封装了针对特定 OS 的线程 API，可被用于提供像线程专有存储这样的功能。

1.1.3 ACE 框架组件

ACE 框架组件是 ACE 中最高级的“积木”，它们的基础是若干针对特定通信软件领域的设计模式。设计者可以使用这些框架组件来帮助自己在高得多的层面上思考和构建系统。这些组件实际上为将要构建的系统提供了“袖珍体系结构”，因此这些组件不仅在开发的实现阶段、同时在设计阶段都是有用的。ACE 的这一层含有以下一些大型组件：

- **事件处理：**大多数通信软件都含有大量处理各种类型事件（比如，基于 I/O、基于定时器、基于信号和基于同步的事件）的代码。软件必须高效地多路分离、分派和处理这些事件。遗憾的是，大多数时间开发者们都在反复地编写这些代码，“重新发明轮子”。这是因为，事件多路分离、分派和处理代码全都紧密地耦合在一起，无法彼此独立地使用。ACE 提供了被称为 **Reactor**（反应器）的框架组件来解决这一问题。反应器提供用于高效地进行事件多路分离和分派的代码，并极大地降低了它们与处理代码之间的耦合，从而改善了可复用性和灵活性。
- **连接或服务初始化组件：**ACE 提供 **Connector**（连接器）和 **Acceptor**（接受器）组件，用于降低连接初始化与连接建立后应用执行的实际服务之间的耦合。在接受大量连接请求的应用服务器中，该组件十分有用。连接首先以应用特有的方式初始化，然后每一连接被相应的处理例程分别处理。这样的去耦合使得开发者能够分别去考虑连接的处理和初始化。因此，如果在随后的阶段开发者发现连接请求数多于或是少于估算，它可以选择使用不同的初始化策略集（ACE 提供了若干可供开发者挑选和选择的策略），以获得所要求的性能水平。
- **流组件：**ACE **Stream** 组件用于简化那些本质上是分层的（layered）或层次的（hierarchic）软件的开发。用户级协议栈的开发是一个好例子；这样的栈由若干互连的层次组成。这些层次或多或少可以相互独立地进行开发。当“数据”通过时，每一层都处理并改变它，并将它传递给下一层，以作进一步的处理。因为各层是相互独立的，它们很容易被复用或替换。
- **服务配置组件：**通信软件开发者面临的另一个问题是，很多时候，软件服务必须在安装时配置，或必须在运行时重配置。应用中特定服务的实现可能需要进行*改动*，因而应用可能必须用新改动的服务重

新配置。ACE **Service Configurator**（服务配置器）为应用的服务提供动态的启动、挂起和配置。

尽管计算机网络领域发展迅速，编写通信软件已经变得更为困难。大量消耗在开发通信软件上的努力不过是“ 重新发明轮子 ”的变种，已知的可以在应用间通用的组件被重写，而不是被复用。通过收集通用的组件和体系结构（它们在网络和系统编程领域一再被复用），ACE 为这一问题提供了解决方案。应用开发者可以采用 ACE，挑选和选择在他的应用中所需的组件，并开始在 ACE 工具箱的陪伴下构建应用。除了在 C++包装层中收集简单的“ 积木 ”，ACE 还包括了大的体系结构“ 积木 ”，它们采用了已被证明在软件开发领域中行之有效的“ 模式 ”和“ 软件体系结构 ”。

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

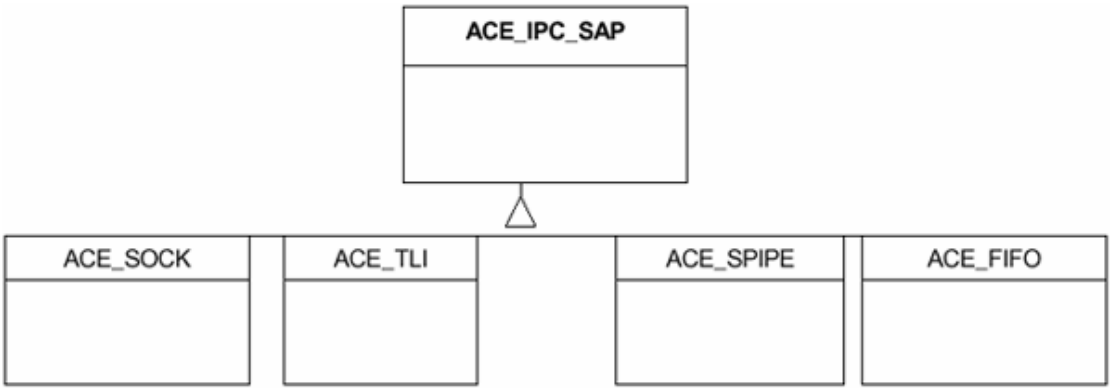

图 2-1 IPC SAP 类属

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。

第 3 章 ACE 的内存管理

ACE 框架含有一组非常丰富的内存管理类。这些类使得你能够很容易和有效地管理动态内存（从堆中申请的内存）和共享内存（在进程间共享的内存）。你可以使用若干不同的方案来管理内存。你需要决定何种方案最适合你正在开发的应用，然后采用恰当的 ACE 类来实现此方案。

ACE 含有两组不同的类用于内存管理。

第一组是那些基于 ACE_Allocator 的类。这组类使用动态绑定和策略模式来提供灵活性和可扩展性。它们只能用于局部的动态内存分配。

第二组类基于 ACE_Malloc 模板类。这组类使用 C++模板和*外部多态性*（External Polymorphism）来为内存分配机制提供灵活性。在这组类中的类不仅包括了用于局部动态内存管理的类，也包括了管理进程间共享内存的类。这些共享内存类使用底层 OS（OS）共享内存接口。

为什么使用一组类而不是另外一组呢？这是由在性能和灵活性之间所作的权衡决定的。因为实际的分配器对象可以在运行时改变，ACE_Allocator 类更为灵活。这是通过动态绑定（这在 C++里需要使用虚函数）来完成的，因此，这样的灵活性并非不需要代价。虚函数调用带来的间接性使得这一方案成了更为昂贵的选择。

另一方面，ACE_Malloc 类有着更好的性能。在编译时，malloc 类通过它将要使用的内存分配器进行配置。这样的编译时配置被称为“外部多态性”。基于 ACE_Malloc 的分配器不能在运行时进行配置。尽管 ACE_Malloc 效率更高，它不像 ACE_Allocator 那样灵活。

3.1 分配器(Allocator)

分配器用于在 ACE 中提供一种动态内存管理机制。在 ACE 中有若干使用不同策略的分配器可用。这些不同策略提供相同的功能，但是具有不同的特性。例如，在实时系统中，应用可能必须从 OS 那里预先分配所有它将要用到的动态内存，然后在内部对分配和释放进行控制。这样，分配和释放例程的性能就是高度可预测的。

所有的分配器都支持 ACE_Allocator 接口，因此无论是在运行时还是在编译时，它们都可以很容易地相互替换。这也正是灵活性之所在。所以，ACE_Allocator 可以与策略模式协同使用，以提供非常灵活的内存管理。表 3-1 给出了对 ACE 中可用的各种分配器的简要描述。这些描述规定了每种分配器所用的内存分配策略。

分配器	描述
ACE_Allocator	ACE 中的分配器类的接口类。这些类使用继承和动态绑定来提供灵活性。
ACE_Static_Allocator	该分配器管理固定大小的内存。每当收到分配内存的请求时，它就移动内部指针、以返回内存 chunk（“大块”）。它还假定内存一旦被分配，就再也不会被释放。
ACE_Cached_Allocator	该分配器预先分配内存池，其中含有特定数目和大小的内存 chunk。这些 chunk 在内部空闲表（free list）中进行维护，并在收到内存请求（malloc()）时被返回。当应用调用 free()时，chunk 被归还到内部空闲表、而不是 OS 中。
ACE_New_Allocator	为 C++ new 和 delete 操作符提供包装的分配器，也就是，它在内部使用

	new 和 delete 操作符，以满足动态内存请求。

表 3-1 ACE 中的分配器

3.1.1 使用缓存式分配器（Cached Allocator）

ACE_Cached_Allocator 预先分配内存，然后使用它自己内部的机制来管理此内存。这样的预分配发生在类的构造器中。所以，如果你使用此分配器，你的内存管理方案仅仅在开始时使用 OS 分配接口来完成预分配。在那以后，ACE_Cached_Allocator 将照管所有的内存分配和释放。

为什么要这样做呢？答案是性能和可预测性。设想一个必须高度可预测的实时系统：通过 OS 来分配内存将涉及昂贵和不可预测的 OS 内核调用；相反，ACE_Cached_Allocator 不会涉及这样的调用。每一次分配和释放都将花费固定数量的时间。

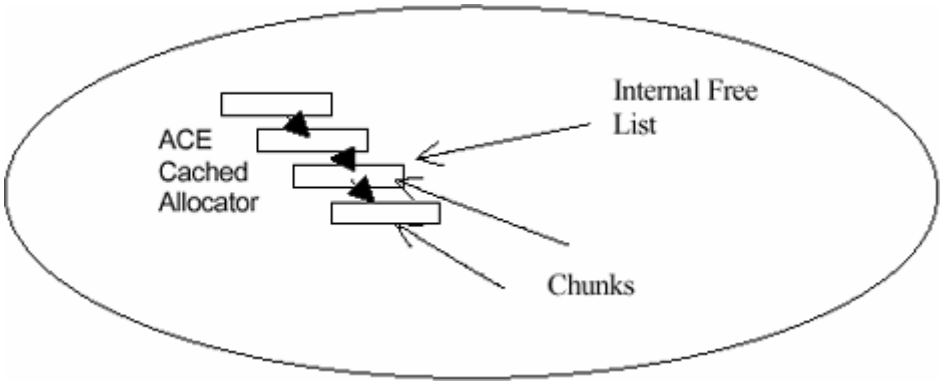

图 3-1 缓存式分配器

图 3-1 演示缓存式分配器。在构造器中预分配的内存在空闲表中进行内部管理。该表将若干内存 chunk 作为它的节点。这些 chunk 可以是任何复杂的数据类型，你可以按你所希望的那样规定它们的实际类型。后面的例子会说明怎样去做。

在此系统中分配和释放涉及固定数量的空闲表指针操作。当用户请求内存 chunk 时，他将获得一个指针，而空闲表被调整。当用户释放内存时，它将回到空闲表中。如此循环往复，直到 ACE_Cached_Allocator 被销毁，所有的内存随之被归还给 OS。在内存被用于实时系统时，需要考虑 chunk 的内部碎片。

下面的例子演示 ACE_Cached_Allocator 是怎样被用于预分配内存，然后处理内存请求的。

例 3-1

```
#include "ace/Malloc.h"
//A chunk of size 1K is created. In our case we decided to use a simple array
//as the type for the chunk. Instead of this we could use any struct or class
//that we think is appropriate.
typedef char MEMORY_BLOCK[1024];


//Create an ACE_Cached_Allocator which is passed in the type of the
```

```
//“chunk” that it must pre-allocate and assign on the free list.
// Since the Cached_Allocator is a template class we can pretty much
//pass it ANY type we think is appropriate to be a memory block.
typedef ACE_Cached_Allocator<MEMORY_BLOCK,ACE_SYNCH_MUTEX> Allocator;
class MessageManager
{
public:
    //The constructor is passed the number of chunks that the allocator
    //should pre-allocate and maintain on its free list.
    MessageManager(int n_blocks):
        allocator_(n_blocks),message_count_(0)
    {
        mesg_array_=new char*[n_blocks];
    }

    //Allocate memory for a message using the Allocator. Remember the message
    //in an array and then increase the message count of valid messages
    //on the message array.
    void allocate_msg(const char *msg)
    {
        mesg_array_[message_count_]=allocator_.malloc(ACE_OS::strlen(msg)+1);
        ACE_OS::strcpy(mesg_array_[message_count_],msg);
        message_count_++;
    }

    //Free all the memory that was allocated. This will cause the chunks
    //to be returned to the allocator’s internal free list
    //and NOT to the OS.
    void free_all_msg()
    {
        for(int i=0;i<message_count_;i++)
            allocator_.free(mesg_array_[i]);

        message_count_=0;
    }

    //Just show all the currently allocated messages in the message array.
    void display_all_msg()
    {
        for(int i=0;i<message_count_;i++)
            ACE_OS::printf("%s\n",mesg_array_[i]);
    }

private:
```

```
    char **mesg_array_;
    Allocator allocator_;
    int message_count_;
};

int main(int argc, char* argv[])
{
    if(argc<2)
    {
        ACE_DEBUG((LM_DEBUG, "Usage: %s <Number of blocks>\n", argv[0]));
        exit(1);
    }

    int n_blocks=ACE_OS::atoi(argv[1]);

    //Instantiate the Memory Manager class and pass in the number of blocks
    //you want on the internal free list.
    MessageManager mm(n_blocks);

    //Use the Memory Manager class to assign messages and free them.
    //Run this in your favorite debug environment and you will notice that the
    //amount of memory your program uses after Memory Manager has been
    //instantiated remains the same. That means the Cached Allocator
    //controls or manages all the memory for the application.
    //Do forever.
    while(1)
    {
        //allocate the messages somewhere
        ACE_DEBUG((LM_DEBUG,"\n\n\nAllocating Messages\n"));
        for(int i=0; i<n_blocks;i++){
            ACE_OS::sprintf(message,"Message %d: Hi There",i);
            mm.allocate_msg(message);
        }

        //show the messages
        ACE_DEBUG((LM_DEBUG,"Displaying the messages\n"));
        ACE_OS::sleep(2);
        mm.display_all_msg();

        //free up the memory for the messages.
        ACE_DEBUG((LM_DEBUG,"Releasing Messages\n"));
        ACE_OS::sleep(2);
        mm.free_all_msg();
    }
```

```
    return 0;
}
```

这个简单的例子包含了一个消息管理器，它对缓存式分配器进行实例化。该分配器随即用于无休止地分配、显示和释放消息。但是，该应用的内存使用并没有变化。你可以通过你喜欢的调试工具来检查这一点。

3.2 ACE_Malloc

如前面所提到的，Malloc 类集使用模板类 ACE_Malloc 来提供内存管理。如图 3-2 所示，ACE_Malloc 模板需要两个参数（一个是内存池，一个是池锁），以产生我们的分配器类。

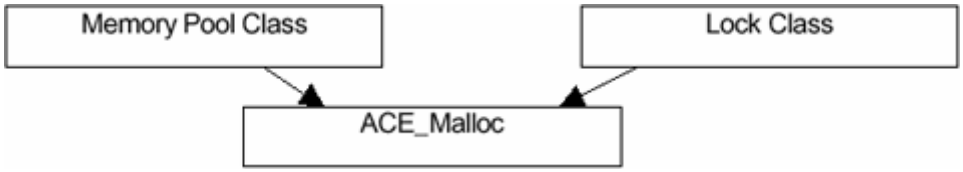

图 3-2 ACE_Malloc 模板参数示意图

3.2.1 ACE_Malloc 工作原理

ACE_Malloc 从传入的内存池中“获取”内存，应用随即通过 ACE_Malloc 类接口来分配（malloc()）内存。由底层内存池返回的内存又在 ACE 的“chunk”（大块）中被返回给 ACE_Malloc 类。ACE_Malloc 类使用这些内存 chunk 来给应用开发者分配较小的内存 block（块）。如图 3-3 所示：

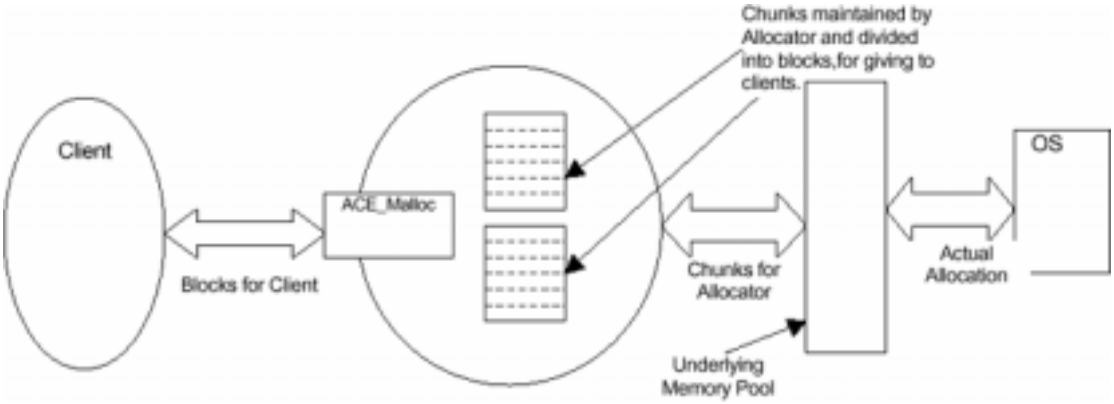

图 3-3 ACE_Malloc 的工作原理

当应用请求内存 block 时，ACE_Malloc 类会检查在它从内存池中获取的 chunk 中，是否有足够的空间来分配所需的 block。如果未能发现有足够空间的 chunk，它就会要求底层内存池返回一个更大的 chunk，以满足应用对内存 block 的请求。当应用发出 free()调用时，ACE_Malloc 不会把所释放的内存返还给内存

池，而是由它自己的空闲表进行管理。当 ACE_Malloc 收到后续的内存请求时，它会使用空闲表来查找可返回的空 block。因而，在使用 ACE_Malloc 时，如果只发出简单的 malloc()和 free()调用，从 OS 分配的内存数量将只会增加，不会减少。ACE_Malloc 类还含有一个 remove()方法，用于发出请求给内存池，将内存返还给 OS。该方法还将锁也返还给 OS。

3.2.2 使用 ACE_Malloc

ACE_Malloc 类的使用很简单。首先，用你选择的内存池和锁定机制实例化 ACE_Malloc，以创建分配器类。随后用该分配器类实例化一个对象，这也就是你的应用将要使用的分配器。当你实例化分配器对象时，传给构造器的第一个参数是一个字符串，它是你想要分配器对象使用的底层内存池的“名字”。将正确的名字传递给构造器**非常**重要，特别是如果你在使用共享内存的话。否则，分配器将会为你创建一个新的内存池。如果你在使用共享内存池，这当然不是你想要的，因为你根本没有获得共享。

为了方便底层内存池的共享（重复一次，如果你在使用共享内存的话，这是很重要的），ACE_Malloc 类还拥有一个映射（map）类型接口：可被给每个被分配的内存 block 一个名字，从而使它们可以很容易地被在内存池中查找的另一个进程找到。该接口含有 bind()和 find()调用。bind()调用用于给由 malloc()调用返回给 ACE_Malloc 的 block 命名。find()调用，如你可能想到的那样，用于查找与某个名字相关联的内存。

在实例化 ACE_Malloc 模板类时，有若干不同的内存池类可用（如表 3-2 所示）。这些类不仅可用于分配在进程内使用的内存，也可以用于分配在进程间共享的内存池。这也使得为何 ACE_Malloc 模板需要通过锁定机制来实例化显得更清楚了。当多个进程访问共享内存池时，该锁保证它们不会因此而崩溃。注意即使是多个线程在使用分配器，也同样需要提供锁定机制。

表 3-2 列出了各种可用的内存池：

池名	宏	描述
ACE_MMAP_Memory_Pool	ACE_MMAP_MEMORY_POOL	使用<mmap(2)>创建内存池。这样内存就可在进程间共享了。每次更新时，内存都被更新到后备存储(backing store)。
ACE_Lite_MMAP_Memory_Pool	ACE_LITE_MMAP_MEMORY_POOL	使用<mmap(2)>创建内存池。不像前面的映射，它不做后备存储更新。代价是较低可靠性。
ACE_Sbrk_Memory_Pool	ACE_SBRK_MEMORY_POOL	使用<sbrk(2)>调用创建内存池。
ACE_Shared_Memory_Pool	ACE_SHARED_MEMORY_POOL	使用系统 V <shmget(2)>调用

		创建内存池。
Memory_Pool		内存可在进程间共享。
ACE_Local_Memory_Pool	ACE_LOCAL_MEMORY_POOL	通过 C++的 new 和 delete 操作符创建局部内存池。该池不能在进程间共享。

表 3-2 可用的内存池

下面的例子通过共享内存池使用 ACE_Malloc 类（该例使用 ACE_SHARED_MEMORY_POOL 来做演示，但表 3-2 中的任何支持内存共享的内存池都可以被使用）。

该例创建服务器进程，该进程创建内存池，再从池中分配内存。然后服务器使用从池中分配的内存来创建它想要客户进程“拾取”的消息。其次，它将名字*绑定*（bind）到这些消息，以使客户能使用相应的 find 操作来查找服务器插入池中的消息。

客户在开始运行后创建它自己的分配器，但是使用的是**同一个**内存池。这是通过将同一个**名字**传送到分配器的构造器来完成的，然后客户使用 find()调用来查找服务器插入的消息，并将它们打印给用户看。

例 3-2

```
#include "ace/Shared_Memory_MM.h"
#include "ace/Malloc.h"
#include "ace/Malloc_T.h"
#define DATA_SIZE 100
#define MESSAGE1 "Hiya over there client process"
#define MESSAGE2 "Did you hear me the first time?"
LPCTSTR poolname="My_Pool";

typedef ACE_Malloc<ACE_SHARED_MEMORY_POOL,ACE_Null_Mutex> Malloc_Allocator;

static void server (void)
{
    //Create the memory allocator passing it the shared memory
    //pool that you want to use
    Malloc_Allocator shm_allocator(poolname);

    //Create a message, allocate memory for it and bind it with
    //a name so that the client can the find it in the memory
    //pool
    char* Message1=(char*)shm_allocator.malloc(strlen(MESSAGE1));
    ACE_OS::strcpy(Message1,MESSAGE1);
    shm_allocator.bind("FirstMessage",Message1);
    ACE_DEBUG((LM_DEBUG,"<<%s\n",Message1));
```

```
    //How about a second message
    char* Message2=(char*)shm_allocator.malloc(strlen(MESSAGE2));
    ACE_OS::strcpy(Message2,MESSAGE2);
    shm_allocator.bind("SecondMessage",Message2);
    ACE_DEBUG((LM_DEBUG,"<<%s\n",Message2));

    //Set the Server to go to sleep for a while so that the client has
    //a chance to do its stuff
    ACE_DEBUG((LM_DEBUG,
        "Server done writing.. going to sleep zzz..\n\n\n"));
    ACE_OS::sleep(2);

    //Get rid of all resources allocated by the server. In other
    //words get rid of the shared memory pool that had been
    //previously allocated
    shm_allocator.remove();
}

static void client(void)
{
    //Create a memory allocator. Be sure that the client passes
    // in the "right" name here so that both the client and the
    //server use the same memory pool. We wouldn't want them to

    // BOTH create different underlying pools.
    Malloc_Allocator shm_allocator(poolname);

    //Get that first message. Notice that the find is looking up the
    //memory based on the "name" that was bound to it by the server.
    void *Message1;
    if(shm_allocator.find("FirstMessage",Message1)==-1)
    {
        ACE_ERROR((LM_ERROR,
            "Client: Problem cant find data that server has sent\n"));
        ACE_OS::exit(1);
    }

    ACE_OS::printf(">>%s\n",(char*) Message1);
    ACE_OS::fflush(stdout);

    //Lets get that second message now.
    void *Message2;
    if(shm_allocator.find("SecondMessage",Message2)==-1)
    {
```

```
        ACE_ERROR((LM_ERROR,
            "Client: Problem cant find data that server has sent\n"));
        ACE_OS::exit(1);
    }

    ACE_OS::printf(">>%s\n",(char*)Message2);
    ACE_OS::fflush(stdout);
    ACE_DEBUG((LM_DEBUG,"Client done reading! BYE NOW\n"));
    ACE_OS::fflush(stdout);
}

int main (int, char *[])
{
    switch (ACE_OS::fork ())
    {
    case -1:
        ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "fork"), 1);

    case 0:
        // Make sure the server starts up first.
        ACE_OS::sleep (1);
        client ();
        break;

    default:
        server ();
        break;
    }

    return 0;
}
```

3.2.3 通过分配器接口使用 Malloc 类

大多数 ACE 中的容器类都可以接受分配器对象作为参数，以用于容器内的内存管理。因为某些内存分配方案只能用于 ACE_Malloc 类集，ACE 含有一个适配器模板类 ACE_Allocator_Adapter，它将 ACE_Malloc 类适配到 ACE_Allocator 接口。也就是说，在实例化这个模板之后创建的新类可用于替换任何 ACE_Allocator。例如：

```
typedef ACE_Allocator_Adapter<ACE_Malloc<ACE_SHARED_MEMORY_POOL,ACE_Null_Mutex>> Allocator;
```

这个新创建的 Allocator 类可用在任何需要分配器接口的地方，但它使用的却是采用

ACE_Shared_Memory_Pool 的 ACE_Malloc 的底层功能。这样该适配器就将 Malloc 类“适配”到了分配器（Allocator）类。

这样的适配允许我们使用与 ACE_Malloc 类集相关联的功能，同时具有 ACE_Allocator 的动态绑定灵活性。但重要的是要记住，这样的灵活性是以牺牲部分性能为代价的。

第 4 章 线程管理：ACE 的同步和线程管理机制

ACE 拥有许多不同的用于创建和管理多线程程序的类。在这一章里，我们将查看 ACE 中的一些线程管理机制。在一开始，我们将查看那些简单的线程包装类，它们的管理功能很少。但是，随着内容的进展，我们将查看 ACE_Thread_Manager 中的更为强大的管理机制。ACE 还拥有一组非常全面的处理线程同步的类。这些类也将在本章讲述。

4.1 创建和取消线程

在不同的平台上，有着若干不同的用于线程管理的接口。其中包括 POSIX pthreads 接口、Solaris 线程、Win32 线程等等。这些接口提供了相同或是相似的功能，但是它们的 API 的差别却极为悬殊。这就导致了困难、麻烦和易错的编程，因为应用程序员必须熟悉不同平台上的若干接口。而且，这样写下的程序，是不可移植和不灵活的。

ACE_Thread 提供了对 OS 的线程调用的简单包装，这些调用处理线程创建、挂起、取消和删除等问题。它提供给应用程序员一个简单易用的接口，可以在不同的线程 API 间移植。ACE_Thread 是非常“ 瘦 ”的包装，有着很少的开销。其大多数方法都是内联的，因而等价于对底层 OS 专有线程接口的直接调用。ACE_Thread 中的所有方法都是静态的，而且该类一般不进行实例化。

下面的例子演示怎样使用 ACE_Thread 包装类创建、生成和联接（join）线程。

例 4-1

```
#include "ace/Thread.h"
#include "ace/Synch.h"

static int number=0;
static int seed = 0;

static void* worker(void *arg)
{
    ACE_UNUSED_ARG(arg);
    ACE_DEBUG((LM_DEBUG,"Thread (%t) Created to do some work"));
    ::number++;
    ACE_DEBUG((LM_DEBUG," and number is %d\n",::number));

    //Let the other guy go while I fall asleep for a random period
    //of time
    ACE_OS::sleep(ACE_OS::rand()%2);

    //Exiting now
    ACE_DEBUG((LM_DEBUG,
```

```
        "\t\t Thread (%t) Done! \t The number is now: %d\n",number));

    return 0;
}

int main(int argc, char *argv[])
{
    if(argc<2)
    {
        ACE_DEBUG((LM_DEBUG,"Usage: %s <number of threads>\n", argv[0]));
        ACE_OS::exit(1);
    }

    ACE_OS::srand(::seed);

    //setup the random number generator
    int n_threads= ACE_OS::atoi(argv[1]);

    //number of threads to spawn
    ACE_thread_t *threadID = new ACE_thread_t[n_threads+1];
    ACE_hthread_t *threadHandles = new ACE_hthread_t[n_threads+1];
    if(ACE_Thread::spawn_n(threadID,       //id's for each of the threads
            n_threads,                    //number of threads to spawn
            (ACE_THR_FUNC)worker,         //entry point for new thread
            0,                            //args to worker
            THR_JOINABLE | THR_NEW_LWP,   //flags
            ACE_DEFAULT_THREAD_PRIORITY,
            0, 0, threadHandles)==-1)
        ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));

    //spawn n_threads
    for(int i=0; i<n_threads; i++)
        ACE_Thread::join(threadHandles[i]);

    //Wait for all the threads to exit before you let the main fall through
    //and have the process exit.
    return 0;
}
```

在这个简单的例子中，创建了 n_thread 个工作者线程。每个线程都执行程序中定义的 worker()函数。线程是通过使用 ACE_Thread::spawn_n()调用创建的。要作为线程的执行启动点调用的函数的指针（在此例中为 worker()函数）被作为参数传入该调用中。要注意的重点是 ACE_Thread::spawn_n()要求所有的线程启动函数（方法）必须是静态的或全局的（就如同直接使用 OS 线程 API 时所要求的一样）。

一旦工作者函数启动，它将全局变量 number 的值加 1，报告它的当前值，然后进入休眠状态，以把

处理器让给其他线程。sleep()休眠一段随机长度的时间。在线程醒来后，它将 number 的当前值通知给用户，然后退出 worker()函数。

一旦线程从它的启动函数返回，它在线程库上隐含地发出线程 exit()调用并退出。这样一旦“掉出”worker()函数，工作者线程也就退出了。负责创建工作者线程的主线程，在退出之前“等待”所有其他的线程完成它们的执行并退出。当主线程退出时（通过退出 main()函数），整个进程也将被销毁。这之所以会发生是因为每当线程退出 main()函数时，都会隐含地调用 exit(3c)函数。因此，如果主线程没有被强制等待其他线程结束，当它死掉时，进程将会被自动销毁，并在它的所有工作者线程完成工作**之前**销毁它们！

上面所说的等待是通过使用 ACE_Thread::join()调用来完成的。该方法的参数是你想要主线程与之联接的线程的句柄（ACE_hthread_t）。

在此例中有若干事实值得注意。首先，在该类中没有可供我们调用的管理功能，用以在内部记住应用所派生的线程的 ID。这使得我们难以联接（join()）、杀死（kill()）或是一般性地管理我们派生的线程。在本章后面讲述的 ACE_Thread_Manager 缓解了这些问题，一般说来，应该使用 ACE_Thread_Manager 而不是线程包装 API。

其次，在程序中没有使用同步原语来保护全局数据。在此例中，它们并不是必须的，因为所有的线程都只对全局变量执行一次加操作。但是在实际应用中，为了保护所有共享互斥数据（全局或静态变量），比如说全局的 number 变量，“锁”将会是必需的。

4.2 ACE 同步原语

ACE 有若干可用于同步目的的类。这些类可划分为以下范畴：

- ACE Lock 类属
- ACE Guard 类属
- ACE Condition 类属
- 杂项 ACE Synchronization 类

4.2.1 ACE Lock 类属

锁类属包含的类包装简单的锁定机制，比如互斥体、信号量、读／写互斥体和令牌。在这一类属中可用的类在表 4-1 中显示。每个类名后都有对用法和用途的简要描述：

名字	描述
ACE_Mutex	封装互斥机制（根据平台，可以是 mutex_t、pthread_mutex_t 等等）的包装类，用于提供简单而有效的机制来使对共享资源的访问序列化。它与二元信号量（binary semaphore）的功能相类似。可被用于线程和进程间的互斥。
ACE_Thread_Mutex	可用于替换 ACE_Mutex，专用于线程同步。
ACE_Process_Mutex	可用于替换 ACE_Mutex，专用于进程同步。
ACE_NULL_Mutex	提供了 ACE_Mutex 接口的“无为”（do-nothing）实现，可在不需要同步时用作替换。

ACE_RW_Mutex	封装读者／作者锁的包装类。它们是分别为读和写进行获取的锁，在没有作者在写的时候，多个读者可以同时进行读取。
ACE_RW_Thread_Mutex	可用于替换 ACE_RW_Mutex，专用于线程同步。
ACE_RW_Process_Mutex	可用于替换 ACE_RW_Mutex，专用于进程同步。
ACE_Semaphore	这些类实现计数信号量，在有固定数量的线程可以同时访问一个资源时很有用。在 OS 不提供这种同步机制的情况下，可通过互斥体来进行模拟。
ACE_Thread_Semaphore	应被用于替换 ACE_Semaphore，专用于线程同步。
ACE_Process_Semaphore	应被用于替换 ACE_Semaphore，专用于进程同步。
ACE_Token	提供“递归互斥体”（recursive mutex），也就是，当前持有某令牌的线程可以多次重新获取它，而不会阻塞。而且，当令牌被释放时，它确保下一个正阻塞并等待此令牌的线程就是下一个被放行的线程。
ACE_Null_Token	令牌接口的“无为”（do-nothing）实现，在你知道不会出现多个线程时使用。
ACE_Lock	定义锁定接口的接口类。一个纯虚类，如果使用的话，必须承受虚函数调用开销。
ACE_Lock_Adapter	基于模板的适配器，允许将前面提到的任意一种锁定机制适配到 ACE_Lock 接口。

表 4-1 ACE 锁类属中的类

表 4-1 中描述的类都支持同样的接口。但是，在任何继承层次中，这些类都是**互不关联**的。在 ACE 中，锁通常用模板来参数化，因为，在大多数情况下，使用虚函数调用的开销都是不可接受的。使用模板使得程序员可获得相当程度的灵活性。他可以在编译时（但不是在运行时）选择他想要使用的的锁定机制的类型。然而，在某些情形中，程序员仍可能需要使用动态绑定和替换（substitution）；对于这些情况，ACE 提供了 ACE_Lock 和 ACE_Lock_Adapter 类。

4.2.1.1 使用互斥体类

互斥体实现了“*互相排斥*”（mutual exclusion）同步的简单形式（所以名为互斥体(mutex)）。互斥体禁止多个线程同时进入受保护的代码“*临界区*”（critical section）。因此，在任意时刻，只有一个线程被允许进入这样的代码保护区。

任何线程在进入临界区之前，必须*获取*（acquire）与此区域相关联的互斥体的所有权。如果已有另一线程拥有了临界区的互斥体，其他线程就不能再进入其中。这些线程必须等待，直到当前的属主线程*释放*（release）该互斥体。

什么时候需要使用互斥体呢？互斥体用于保护共享的易变代码，也就是，全局或静态数据。这样的数据必须通过互斥体进行保护，以防止它们在多个线程同时访问时损坏。

下面的例子演示 ACE_Thread_Mutex 类的使用。注意在此处很容易用 ACE_Mutex 替换 ACE_Thread_Mutex 类，因为它们拥有同样的接口。

例 4-2

```
#include "ace/Synch.h"
#include "ace/Thread.h"

//Arguments that are to be passed to the worker thread are passed
//through this struct.
struct Args
{
public:
    Args(int iterations): mutex_(),iterations_(iterations){}
    ACE_Thread_Mutex mutex_;
    int iterations_;
};

//The starting point for the worker threads
static void* worker(void*arguments)
{
    Args *arg= (Args*) arguments;
    for(int i=0;i<arg->iterations_;i++)
    {
        ACE_DEBUG((LM_DEBUG,
            "(%t) Trying to get a hold of this iteration\n"));

        //This is our critical section
        arg->mutex_.acquire();
        ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));
        ACE_OS::sleep(2);

        //simulate critical work
        arg->mutex_.release();
    }

    return 0;
}

int main(int argc, char*argv[])
{
    if(argc<2)
    {
        ACE_OS::printf("Usage: %s <number_of_threads>
            <number_of_iterations>\n", argv[0]);
        ACE_OS::exit(1);
    }
```

```
    Args arg(ACE_OS::atoi(argv[2]));

    //Setup the arguments
    int n_threads = ACE_OS::atoi(argv[1]);

    //determine the number of threads to be spawned.
    ACE_thread_t *threadID = new ACE_thread_t[n_threads+1];
    ACE_hthread_t *threadHandles = new ACE_hthread_t[n_threads+1];
    if(ACE_Thread::spawn_n(threadID,   //id's for each of the threads
                n_threads,            //number of threads to spawn
                (ACE_THR_FUNC)worker, //entry point for new thread
                &arg,                      //args to worker
                THR_JOINABLE | THR_NEW_LWP, //flags
                ACE_DEFAULT_THREAD_PRIORITY,
                0, 0, threadHandles)==-1)
        ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));

    //spawn n_threads
    for(int i=0; i<n_threads; i++)
        ACE_Thread::join(threadHandles[i]);

    //Wait for all the threads to exit before you let the main fall through
    //and have the process exit.
    return 0;
}
```

在上面的例子中，ACE_Thread 包装类用于生成多个线程来执行 worker()函数，就如同在前面的例子里一样。Arg 对象作为参数传入各个线程，在该对象中含有循环要执行的次数，以及将要使用的互斥体。

在此例中，一开始，每个线程立即进入 for 循环。一进入循环，线程就进入了临界区。在临界区内完成的工作使用 ACE_Thread_Mutex 互斥体对象进行保护。该对象由主线程作为参数传给工作者线程。临界区控制是通过在 ACE_Thread_Mutex 对象上发出 acquire()调用，从而获取互斥体的所有权来完成的。一旦互斥体被获取，没有其他线程能够再进入这一代码区。临界区控制是通过使用 release()调用来释放的。一旦互斥体的所有权被放弃，就会唤醒所有其他在等待的线程。这些线程随即相互竞争，以获得互斥体的所有权。第一个试图获取所有权的线程会进入临界区。

4.2.1.2 将锁和锁适配器（Lock Adapter）用于动态绑定

如前面所提到的，各种互斥体锁应被直接用于你的代码，或者，如果需要灵活性，作为模板参数来使用。但是，如果你需要动态地（也就是在运行时）改变你的代码所用锁的类型，就无法使用这些锁。

为应对这个问题，ACE 拥有 ACE_Lock 和 ACE_Lock_Adapter 类，它们可用于这样的运行时替换（substitution）。

下面的例子演示 ACE_Lock 类和 ACE_Lock_Adapter 怎样为应用程序员提供方便，和锁定机制一起使

用动态绑定和替换。

例 4-3

```
#include "ace/Synch.h"
#include "ace/Thread.h"

//Arguments that are to be passed to the worker thread are passed
//through this class.
struct Args
{
public:
    Args(ACE_Lock* lock,int iterations):
        mutex_(lock),iterations_(iterations){}
    ACE_Lock* mutex_;
    int iterations_;
};

//The starting point for the worker threads
static void* worker(void*arguments)
{
    Args *arg= (Args*) arguments;
    for(int i=0;i<arg->iterations_;i++)
    {
        ACE_DEBUG((LM_DEBUG,
            "(%t) Trying to get a hold of this iteration\n"));

        //This is our critical section
        arg->mutex_->acquire();
        ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));
        ACE_OS::sleep(2);

        //simulate critical work
        arg->mutex_->release();
    }

    return 0;
}

int main(int argc, char*argv[])
{
    if(argc<4)
    {
        ACE_OS::printf("Usage: %s <number_of_threads>
            <number_of_iterations> <lock_type>\n", argv[0]);
```

```
        ACE_OS::exit(1);
    }

    //Polymorphic lock that will be used by the application
    ACE_Lock *lock;

    //Decide which lock you want to use at run time,
    //recursive or non-recursive.
    if(ACE_OS::strcmp(argv[3],"Recursive"))
        lock=new ACE_Lock_Adapter<ACE_Recursive_Thread_Mutex>;
    else
        lock=new ACE_Lock_Adapter<ACE_Thread_Mutex>

    //Setup the arguments
    Args arg(lock,ACE_OS::atoi(argv[2]));

    //spawn threads and wait as in previous examples..
}
```

在此例中，和前面的例子唯一的不同是 ACE_Lock 类是和 ACE_Lock_Adapter 一起使用的，以便能提供动态绑定。底层的锁定机制使用递归还是非递归互斥体，是在程序运行时由命令行参数决定的。使用动态绑定的好处是实际的锁定机制可以在运行时被替换。缺点是现在对锁的每次调用都需要负担额外的经由虚函数表的间接层次。

4.2.1.3 使用令牌（Token）

如表 4-1 中所提到的，ACE_Token 类提供所谓的“ 递归互斥体 ”，它可以被最初获得它的同一线程进行多次重新获取。ACE_Token 类还确保所有试图获取它的线程按严格的 FIFO（先进先出）顺序排序。

递归锁允许同一线程多次获取同一个锁。线程不会因为试图获取它已经拥有的锁而死锁。这些类型的锁能在各种不同的情况下派上用场。例如，如果你用一个锁来维护跟踪流的一致性，你可能希望这个锁是递归的，因为某个方法可以调用一个跟踪例程，获取锁，被信号中断，然后再尝试获取这个跟踪锁。如果锁是非递归的，线程将会在这里锁住它自己。你会发现很多其他需要递归锁的有趣应用。重要的是要记住，你获取递归锁多少次，就**必须释放**它多少次。

在 SunOS 5.x 上运行例 4-3，释放锁的线程常常也是重新获得它的线程（大约 90%的情况是这样）。但是如果你采用 ACE_Token 类作为锁定机制来运行这个例子，每个线程都会轮流获得令牌，然后有序地把机会让给下一个线程。

尽管 ACE_Token 作为所谓的递归锁非常有用，它们实际上是更大的“ 令牌管理 ”框架的一部分。该框架允许你维护数据存储中数据的一致性。遗憾的是，这已经超出了此教程的范围。

例 4-4

```
#include "ace/Token.h"
#include "ace/Thread.h"
```

```
//Arguments that are to be passed to the worker thread are passed
//through this struct.
struct Args
{
public:
    Args(int iterations):
        token_(“myToken”),iterations_(iterations){}
    ACE_Token token_;
    int iterations_;
};

//The starting point for the worker threads
static void* worker(void*arguments)
{
    Args *arg= (Args*) arguments;
    for(int i=0;i<arg->iterations_;i++)
    {
        ACE_DEBUG((LM_DEBUG,"(%t) Trying to get a hold of this iteration\n"));

        //This is our critical section
        arg->token_.acquire();
        ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));

        //work
        ACE_OS::sleep(2);
        arg->token_.release();
    }

    return 0;
}

int main(int argc, char*argv[])
{
    //same as previous examples..
}
```

4.2.2 ACE 守卫（Guard）类属

ACE 中的守卫用于自动获取和释放锁。守卫类的对象定义一个代码块，在其上获取一个锁。在退出此代码块时，锁被自动释放。

ACE 中的守卫类是一种模板，它通过所需锁定机制的类型来参数化。底层的锁可以是 ACE Lock 类属中的任何类，也就是，任何互斥体或锁类。它是这样工作的：对象的构造器获取锁，析构器释放锁。

表 4-2 列出了 ACE 中可用的守卫：

名字	描述
ACE_Guard	自动在底层锁上调用 acquire()和 release()。任何 ACE Lock 类属中的锁都可以作为它的模板参数传入。
ACE_Read_Guard	自动在底层锁上调用 acquire()和 release()。
ACE_Write_Guard	自动在底层锁上调用 acquire()和 release()。

表 4-2 ACE 中的守卫

下面的例子演示怎样使用看守：

例 4-5

```
#include "ace/Synch.h"
#include "ace/Thread.h"

//Arguments that are to be passed to the worker thread are passed
//through this class.
class Args
{
public:
    Args(int iterations):
        mutex_(),iterations_(iterations){}
    ACE_Thread_Mutex mutex_;
    int iterations_;
};

//The starting point for the worker threads
static void* worker(void*arguments)
{
    Args *arg= (Args*) arguments;
    for(int i=0;i<arg->iterations_;i++)
    {
        ACE_DEBUG((LM_DEBUG,"(%t) Trying to get a hold of this iteration\n"));
        ACE_Guard<ACE_Thread_Mutex> guard(arg->mutex_);
        {
            //This is our critical section
            ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));

            //work
            ACE_OS::sleep(2);
        }//end critical section
    }
```

```
    return 0;
}

int main(int argc, char*argv[])
{
    //same as previous example
}
```

在上面的例子中，守卫在工作者线程中管理临界区。守卫对象从 ACE_Guard 模板类创建而来。守卫应使用的锁的类型被传给模板。通过将所需获取的实际锁对象经由守卫对象的构造器传入，程序创建守卫对象。该锁由 ACE_Guard 在内部自动获取，所以 for 循环中的区域就是受保护的临界区。一旦出了作用域，看守对象就会被自动删除，锁也随之被释放。

守卫是很有用的，因为它们保证一旦你获取了一个锁，你总是会释放它（当然，除非你的线程因为无法预料的情况而死掉）。在有许多不同的返回路径的复杂方法中，这被证明为是极其有用的。

4.2.3 ACE 条件（Condition）类属

ACE_Condition 类是针对 OS 条件变量原语的包装类。那么，到底什么是条件变量呢？

线程常常需要特定条件被满足才能继续它的操作。例如，设想线程需要在全局消息队列里插入消息。在插入任何消息之前，它必须检查在消息队列里是否有空闲空间。如果消息队列在“满”状态，它就什么也不能做，而必须进行休眠，过一会再重试。就是说，在访问全局资源之前，某个*条件*必须为真。然后，当另外的线程空出消息队列时，应该有方法通知或发信号给原来的线程：在消息队列里有位置了，现在应该再次尝试插入消息。这可以使用条件变量来完成。条件变量不是被用作互斥原语，而是用作特定条件已经满足的指示器。

在使用条件变量时，你的程序应该完成以下步骤：

- 获取全局资源（例如，消息队列）的锁（互斥体）。
- 检查条件（例如，消息队列里有空间吗？）。
- 如果条件失败，调用条件变量的 wait()方法。等待在未来条件变为真。
- 当另一线程在全局资源上执行操作时，它发信号（signal()）给所有其他在此资源上测试条件的线程（例如，另一线程从消息队列中取出一个消息，然后通过条件变量发送信号，以使阻塞在 wait()上的线程能够再尝试将它们的消息插入队列）。
- 在醒来之后，重新检查条件现在是否为真。如为真，则在全局资源上执行操作（例如，将消息插入全局消息队列）

需要特别注意的是，在阻塞在 wait 调用中之前，条件变量机制（也就是 ACE_Cond）负责释放全局资源上的互斥体。如果没有进行此操作，将没有其他的线程能够在此资源上工作（该资源是条件改变的原因）。同样，一旦阻塞线程收到信号、重又醒来，它在检查条件之前会在内部重新获取锁。

下面的例子是对本章第一个例子的改写。如果你还记得，我们曾说过使用 ACE_Thread::join()调用可以使主线程等待其他的线程结束。另一种达到同样目的的方法是使用条件变量，它使主线程在退出之前等待“所有线程已经结束”条件为真。最后一个线程可以通过条件变量发信号给等待中的主线程，通知它所有线程已经结束、而它是最后一个。随后主线程继续执行，退出应用并销毁进程。如下如示：

例 4-6

```
#include "ace/Thread.h"
#include "ace/Synch.h"

static int number=0;
static int seed=0;

class Args
{
public:
    Args(ACE_Condition<ACE_Thread_Mutex> *cond, int threads):
        cond_(cond), threads_(threads){}
    ACE_Condition<ACE_Thread_Mutex> *cond_;
    int threads_;
};

static void* worker(void *arguments)
{
    Args *arg= (Args*)arguments;
    ACE_DEBUG((LM_DEBUG,"Thread (%t) Created to do some work\n"));
    ::number++;

    //Work
    ACE_OS::sleep(ACE_OS::rand()%2);

    //Exiting now
    ACE_DEBUG((LM_DEBUG,
        "\tThread (%t) Done! \n\tThe number is now: %d\n",number));

    //If all threads are done signal main thread that
    //program can now exit
    if(number==arg->threads_)
    {
        ACE_DEBUG((LM_DEBUG,
            "(%t) Last Thread!\n All threads have done their job!
            Signal main thread\n"));
        arg->cond_->signal();
    }

    return 0;
}

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

```
    if(argc<2)
    {
        ACE_DEBUG((LM_DEBUG,
            "Usage: %s <number of threads>\n", argv[0]));
        ACE_OS::exit(1);
    }

    int n_threads=ACE_OS::atoi(argv[1]);

    //Setup the random number generator
    ACE_OS::srand(::seed);

    //Setup arguments for threads
    ACE_Thread_Mutex mutex;
    ACE_Condition<ACE_Thread_Mutex> cond(mutex);
    Args arg(&cond,n_threads);

    //Spawn off n_threads number of threads
    for(int i=0; i<n_threads; i++)
    {
        if(ACE_Thread::spawn((ACE_THR_FUNC)worker,(void*)&arg,
                             THR_DETACHED|THR_NEW_LWP)==-1)
            ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));
    }

    //Wait for signal indicating that all threads are done and program
    //can exit. The global resource here is “number” and the condition
    //that the condition variable is waiting for is number==n_threads.
    mutex.acquire();

    while(number!=n_threads)
        cond.wait();
    ACE_DEBUG((LM_DEBUG,"(%t) Main Thread got signal. Program
        exiting..\n"));

    mutex.release();

    ACE_OS::exit(0);
}
```

注意主线程首先获取一个互斥体，然后对条件进行测试。如果条件不为真，主线程就等待在此条件变量上。条件变量随即*自动*释放互斥体，并使主线程进入睡眠。条件变量总是像这样与互斥体一起使用。这是一种可如下描述的一般模式[1]：

while(expression NOT TRUE) wait on condition variable;

记住条件变量不是用于互斥，而是用于我们所描述的发送信号功能。

除了 ACE_Condition 类，ACE 还包括 ACE_Condition_Thread_Mutex 类，它使用 ACE_Thread_Mutex 作为全局资源的底层锁定机制。

4.2.4 杂项同步类

除了上面描述的同步类，ACE 还包括其他一些同步类，比如 ACE_Barrier 和 ACE_Atomic_Op。

4.2.4.1 ACE 中的栅栏（Barrier）

栅栏有一个好名字，因为它恰切地描述了栅栏应做的事情。一组线程可以使用栅栏来进行共同的相互同步。组中的每个线程各自执行，直到到达栅栏，就阻塞在那里。在所有相关线程到达栅栏后，它们就全部继续它们的执行。就是说，它们一个接一个地阻塞，等待其他的线程到达栅栏；一旦所有线程都到达了它们的执行路径中的“ 栅栏点 ”，它们就一起重新启动。

在 ACE 中，栅栏在 ACE_Barrier 类中实现。在栅栏对象被实例化时，它将要等待的线程的数目会作为参数传入。一旦到达执行路径中的“ 栅栏点 ”，每个线程都在栅栏对象上发出 wait()调用。它们在这里阻塞，直到其他线程到达它们各自的“ 栅栏点 ”，然后再一起继续执行。当栅栏从相关线程那里接收了适当数目的 wait()调用时，它就同时唤醒所有阻塞的线程。

下面的例子演示怎样通过 ACE 使用栅栏：

例 4-7

```
#include "ace/Thread.h"
#include "ace/Synch.h"

static int number=0;
static int seed=0;

class Args
{
public:
    Args(ACE_Barrier *barrier): barrier_(barrier){}
    ACE_Barrier *barrier_;
};

static void*
worker(void *arguments)
{
    Args *arg= (Args*)arguments;
    ACE_DEBUG((LM_DEBUG,"Thread (%t) Created to do some work\n"));
```

```
    ::number++;

    //Work
    ACE_OS::sleep(ACE_OS::rand()%2);

    //Exiting now
    ACE_DEBUG((LM_DEBUG,
        "\tThread (%t) Done! \n\tThe number is now: %d\n",number));

    //Let the barrier know we are done.
    arg->barrier_->wait();
    ACE_DEBUG((LM_DEBUG,"Thread (%t) is exiting \n"));

    return 0;
}

int main(int argc, char *argv[])
{
    if(argc<2)
    {
        ACE_DEBUG((LM_DEBUG,"Usage: %s <number of threads>\n", argv[0]));
        ACE_OS::exit(1);
    }

    int n_threads=ACE_OS::atoi(argv[1]);
    ACE_DEBUG((LM_DEBUG,"Preparing to spawn %d threads",n_threads));

    //Setup the random number generator
    ACE_OS::srand(::seed);

    //Setup arguments for threads
    ACE_Barrier barrier(n_threads);
    Args arg(&barrier);

    //Spawn off n_threads number of threads
    for(int i=0; i<n_threads; i++)
    {
        if(ACE_Thread::spawn((ACE_THR_FUNC)worker,
                (void*)&arg,THR_DETACHED|THR_NEW_LWP)==-1)
            ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));
    }

    //Wait for all the other threads to let the main thread
    // know that they are done using the barrier
```

```
    barrier.wait();
    ACE_DEBUG((LM_DEBUG,"(%t)Other threads are finished. Program exiting..\n"));
    ACE_OS::sleep(2);
}
```

在此例中，主线程创建一个栅栏，并将其传递给工作者线程。每个工作者线程都在就要退出前在栅栏上调用 wait()，从而使它们在完成工作后和就要退出前阻塞住。主线程也在就要退出前阻塞。一旦所有线程（包括主线程）执行结束，它们就会一起退出。

4.2.4.2 原子操作（Atomic Op）

ACE_Atomic_Op 类用于将同步透明地参数化进基本的算术运算中。ACE_Atomic_Op 是一种模板类，锁定机制和需要参数化的类型被作为参数传入其中。ACE 是这样来实现此机制的：重载所有算术操作符，并确保在操作前获取锁，在操作后释放它。运算本身被委托给通过模板传入的的类。

下面的例子演示此类的用法：

例 4-8

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

//Global mutable and shared data on which we will perform simple
//arithmetic operations which will be protected.
ACE_Atomic_Op<ACE_Thread_Mutex,int> foo;

//The worker threads will start from here.
static void* worker(void *arg)
{
    ACE_UNUSED_ARG(arg);

    foo=5;
    ACE_ASSERT (foo == 5);

    ++foo;
    ACE_ASSERT (foo == 6);

    --foo;
    ACE_ASSERT (foo == 5);

    foo += 10;
    ACE_ASSERT (foo == 15);

    foo -= 10;
    ACE_ASSERT (foo == 5);
```

```
    foo = 5L;
    ACE_ASSERT (foo == 5);

    return 0;
}

int main(int argc, char *argv[])
{
    //spawn threads as in previous examples
}
```

在上面的程序中，在 foo 变量上执行了若干简单的算术运算。在运算之后，执行了断言检查来保证变量的值是它“应该是”的值。

你也许想知道为什么这样的算术原语（比如像上例中那样）需要同步。你一定认为增减运算本来就是原子的。

但是，这些运算通常**不是**原子的。CPU 有可能将指令划分为三个步骤：读变量、增加或减少变量的值，以及回写。在这样的情况下，如果没有使用原子操作，就可能发生下面的情况：

- 线程一读变量，增加它的值，还未及将新值写回，就被 OS 调换出去。
- 线程二读取变量的旧值，增加它并写回新的增加了的值。
- 线程一用它自己的值覆盖了线程二的增量。

即使没有使用同步原语，上面的例程也*可能*并不会出错。原因是这种情况下的线程是计算绑定的，OS 不会先占（pre-empt）这样的线程。但是，这样编写的代码是不安全的，因为你不能依赖 OS 调度器的工作方式。在大多数环境中，任何情况下同步关系都是非确定性的（因为实时效应，像页面错误或定时器的使用；或是因为实际上有多个物理处理器）。

4.3 使用 ACE_THREAD_MANAGER 进行线程管理

在前面所有的例子中，我们一直使用 ACE_Thread 包装类来创建和销毁线程。但是，该包装类的功能比较有限。ACE_Thread_Manager 提供了 ACE_Thread 中的功能的超集。特别地，它增加了管理功能，以使启动、取消、挂起和恢复一组相关线程变得更为容易。它用于创建和销毁成组的线程和任务（ACE_Task 是一种比线程更高级的构造，可在 ACE 中用于进行多线程编程。我们将在后面再来讨论任务）。它还提供了这样的功能：发送信号给一组线程，或是在一组线程上等待，而不是像我们在前面的例子中所看到的那样，以一种不可移植的方式来调用 join()。

下面的例子演示怎样使用 ACE_Thread_Manager 创建一组线程，然后等待它们的完成。

例 4-9

```
#include "ace/Thread_Manager.h"
#include "ace/Get_Opt.h"

static void* taskone(void*)
{
```

```
    ACE_DEBUG((LM_DEBUG,"Thread:(%t)started Task one! \n"));
    ACE_OS::sleep(2);
    ACE_DEBUG((LM_DEBUG,"Thread:(%t)finished Task one!\n"));
    return 0;
}

static void* tasktwo(void*)
{
    ACE_DEBUG((LM_DEBUG,"Thread:(%t)started Task two!\n"));
    ACE_OS::sleep(1);
    ACE_DEBUG((LM_DEBUG,"Thread:(%t)finished Task two!\n"));
    return 0;
}

static void print_usage_and_die()
{
    ACE_DEBUG((LM_DEBUG,"Usage program_name
        -a<num threads for pool1> -b<num threads for pool2>"));
    ACE_OS::exit(1);
}

int main(int argc, char* argv[])
{
    int num_task_1;
    int num_task_2;

    if(argc<3)
        print_usage_and_die();

    ACE_Get_Opt get_opt(argc,argv,"a:b:");

    char c;
    while( (c=get_opt())!=EOF)
    {
        switch(c)
        {
        case 'a':
            num_task_1=ACE_OS::atoi(get_opt.optarg);
            break;
        case 'b':
            num_task_2=ACE_OS::atoi(get_opt.optarg);
            break;
        default:
            ACE_ERROR((LM_ERROR,"Unknown option\n"));
```

```
            ACE_OS::exit(1);
        }
    }

    //Spawn the first set of threads that work on task 1.
    if(ACE_Thread_Manager::instance()->spawn_n(num_task_1,
            (ACE_THR_FUNC)taskone,//Execute task one
            0, //No arguments
            THR_NEW_LWP, //New Light Weight Process
            ACE_DEFAULT_THREAD_PRIORITY,
            1)==-1) //Group ID is 1
        ACE_ERROR((LM_ERROR,
            "Failure to spawn first group of threads: %p \n"));

    //Spawn second set of threads that work on task 2.
    if(ACE_Thread_Manager::instance()->spawn_n(num_task_2,
                (ACE_THR_FUNC)tasktwo,//Execute task one
                0, //No arguments
                THR_NEW_LWP, //New Light Weight Process
                ACE_DEFAULT_THREAD_PRIORITY,
                2)==-1)//Group ID is 2
        ACE_ERROR((LM_ERROR,
            "Failure to spawn second group of threads: %p \n"));

    //Wait for all tasks in grp 1 to exit
    ACE_Thread_Manager::instance()->wait_grp(1);
    ACE_DEBUG((LM_DEBUG,"Tasks in group 1 have exited! Continuing \n"));

    //Wait for all tasks in grp 2 to exit
    ACE_Thread_Manager::instance()->wait_grp(2);

    ACE_DEBUG((LM_DEBUG,"Tasks in group 2 have exited! Continuing \n"));
}
```

下一个例子演示ACE_Thread_Manager中的挂起、恢复和协作式取消机制：

例 4-10

```
// Test out the group management mechanisms provided by the
// ACE_Thread_Manager, including the group suspension and resumption,
//and cooperative thread cancellation mechanisms.
#include "ace/Thread_Manager.h"
static const int DEFAULT_THREADS = ACE_DEFAULT_THREADS;
static const int DEFAULT_ITERATIONS = 100000;
```

```
static void *
worker (int iterations)
{
     for (int i = 0; i < iterations; i++)
     {
          if ((i % 1000) == 0)
          {
               ACE_DEBUG ((LM_DEBUG,
                    "(%t) checking cancellation before iteration %d!\n", i));

               //Before doing work check if you have been canceled. If so don't
               //do any more work.
               if (ACE_Thread_Manager::instance ()->testcancel
                         (ACE_Thread::self ()) != 0)
               {
                    ACE_DEBUG ((LM_DEBUG,
                         "(%t) has been canceled before iteration %d!\n",i));
                    break;
               }
          }
     }

     return 0;
}

int main (int argc, char *argv[])
{
     int n_threads = argc > 1 ? ACE_OS::atoi (argv[1]) : DEFAULT_THREADS;
     int n_iterations = argc > 2 ? ACE_OS::atoi (argv[2]) : DEFAULT_ITERATIONS;

     ACE_Thread_Manager *thr_mgr = ACE_Thread_Manager::instance ();

     //Create a group of threads n_threads that will execute the worker
     //function the spawn_n method returns the group ID for the group of
     //threads that are spawned. The argument n_iterations is passed back
     //to the worker. Notice that all threads are created detached.
     int grp_id = thr_mgr->spawn_n (n_threads, ACE_THR_FUNC (worker),
               (void *) n_iterations,
               THR_NEW_LWP | THR_DETACHED);

     // Wait for 1 second and then suspend every thread in the group.
     ACE_OS::sleep (1);
     ACE_DEBUG ((LM_DEBUG, "(%t) suspending group\n"));
     if (thr_mgr->suspend_grp (grp_id) == -1)
```

```
        ACE_ERROR ((LM_DEBUG, "(%t) %p\n", "Could not suspend_grp"));

    // Wait for 1 more second and then resume every thread in the
    // group.
    ACE_OS::sleep (1);
    ACE_DEBUG ((LM_DEBUG, "(%t) resuming group\n"));
    if (thr_mgr->resume_grp (grp_id) == -1)
        ACE_ERROR ((LM_DEBUG, "(%t) %p\n", "resume_grp"));

    // Wait for 1 more second and then cancel all the threads.
    ACE_OS::sleep (ACE_Time_Value (1));
    ACE_DEBUG ((LM_DEBUG, "(%t) canceling group\n"));
    if (thr_mgr->cancel_grp (grp_id) == -1)
        ACE_ERROR ((LM_DEBUG, "(%t) %p\n", "cancel_grp"));

    // Perform a barrier wait until all the threads have shut down.
    thr_mgr->wait ();

    return 0;
}
```

在此例中创建了 n_threads 个线程来执行 worker 函数。每个线程在 worker 函数中执行 n_iterations 次循环。当这些线程在 worker 函数中执行循环时，主线程将挂起（suspend()）、恢复（resume()）、并最终取消它们。Worker 函数中的每个线程将使用 ACE_Thread_Manager 的 testcancel()来检查取消情况。

4.4 线程专有存储（Thread Specific Storage）

如果单线程程序希望创建一个变量，其值能在多个函数调用间持续，它可以静态或全局地分配该数据。如果这是个多线程程序，这个全局或静态数据对于所有线程来说都是一样的。这可能是，也可能不是我们所期望的。例如，伪随机数生成器可能需要静态的或全局的整型种子变量，它不会因为多个线程同时改变它的值而受到影响。但是，在另外的情形中，对于各个线程来说，可能需要不同的全局或静态数据。例如，设想一个多线程的 GUI 应用，在其中各个窗口都运行在独立的线程中，并各拥有一个输入端口，窗口从中接收事件输入。这样的输入端口必须在窗口的各个函数调用之间保持“ 持续 ”，并且还必须是窗口专有的或私有的。可使用线程专有存储来满足此需求。像输入端口这样的结构可放在线程专有存储中，并可像逻辑上的静态或全局变量一样被访问；而实际上它对线程来说是私有的。

传统上，线程专有存储通过让人迷惑的底层操作系统 API 来实现。在 ACE 中，TSS 通过使用 ACE_TSS 模板类来实现。需要成为线程专有的类被传入 ACE_TSS 模板，然后可以使用 C++的->操作符来调用它的全部公共方法。

下面的例子演示在 ACE 中使用线程专有存储是多么简单：

例 4-11

```
#include "ace/Synch.h"
#include "ace/Thread_Manager.h"
```

```
class DataType
{
public:
     DataType():data(0){}
     void increment(){ data++;}
     void set(int new_data){ data=new_data;}
     void decrement(){ data--;}
     int get(){return data;}

private:
     int data;
};

ACE_TSS<DataType> data;

static void* thread1(void*)
{
     data->set(10);
     ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
     for(int i=0;i<5;i++)
          data->increment();
     ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
     return 0;
}

static void * thread2(void*)
{
     data->set(100);
     ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
     for(int i=0; i<5;i++)
          data->increment();
     ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
     return 0;
}

int main(int argc, char*argv[])
{
     //Spawn off the first thread
     ACE_Thread_Manager::instance()->spawn((ACE_THR_FUNC)thread1,0,THR_NEW_LWP|
          THR_DETACHED);

     //Spawn off the second thread
     ACE_Thread_Manager::instance()->spawn((ACE_THR_FUNC)thread2,0,THR_NEW_LWP| THR_DETACHED);
```

```
    //Wait for all threads in the manager to complete.
    ACE_Thread_Manager::instance()->wait();
    ACE_DEBUG((LM_DEBUG,"Both threads done.Exiting.. \n"));
}
```

在上面的例子中，DataType 类是在线程专有存储中创建的。随后程序使用->操作符来从函数 thread1 和 thread2 访问此类的方法，这两个函数分别在不同的线程中执行。第一个线程将私有数据变量设置为 10，然后增加 5，将它变为 15。第二个线程使用它自己的私有数据变量，将它的值设为 100，并增加 5，变成 105。尽管数据**看起来**是全局的，它实际上是线程专有的，而且两个线程分别打印出 15 和 105，也说明了这一点。

尽可能使用线程专有存储有若干好处。如果全局或静态数据可放在线程专有存储中，就可将同步所导致的开销降到最低。这是使用 TSS 的主要好处。

第 5 章 任务和主动对象（Active Object）：并发编程模式

这一章介绍前面提到过的 ACE_Task 类，另外还介绍了主动对象模式。基本上这一章将涵盖两个主题。首先，它将讲述怎样将 ACE_Task 构造作为高级面向对象机制使用，用以编写多线程程序。其次，它将讨论怎样在主动对象模式[II]中使用 ACE_Task。

5.1 主动对象

那么到底什么是主动对象呢？传统上，所有的对象都是被动的代码段，对象中的代码是在对它发出方法调用的线程中执行的。也就是，调用线程（calling threads）被“借出”，以执行被动对象的方法。

而主动对象却不一样。这些对象持有它们自己的线程（甚或多个线程），并将这个线程用于执行对它们的任何方法的调用。因而，如果你想象一个传统对象，在里面封装了一个线程（或多个线程），你就得到了一个主动对象。

例如，设想对象“A”已在你的程序的 main()函数中被实例化。当你的程序启动时，OS 创建一个线程，以从 main()函数开始执行。如果你调用对象 A 的任何方法，该线程将“流过”那个方法，并执行其中的代码。一旦执行完成，该线程返回调用该方法的点并继续它的执行。但是，如果”A”是主动对象，事情就不是这样了。在这种情况下，主线程不会被主动对象借用。相反，当”A”的方法被调用时，方法的执行发生在主动对象持有的线程中。另一种思考方法：如果调用的是被动对象的方法（常规对象），调用会阻塞（同步的）；而另一方面，如果调用的是主动对象的方法，调用不会阻塞（异步的）。

5.2 ACE_Task

ACE_Task 是 ACE 中的任务或主动对象“处理结构”的基类。在 ACE 中使用了此类来实现主动对象模式。所有希望成为“主动对象”的对象都必须从此类派生。你也可以把 ACE_TASK 看作是更高级的、更为面向对象的线程类。

当我们在前一章中使用 ACE_Thread 包装时，你一定已经注意到了一些“不好”之处。那一章中的大多数程序都被分解为函数、而不是对象。这是因为 ACE_Thread 包装需要一个全局函数名、或是静态方法作为参数。随后该函数（静态方法）就被用作所派生的线程的“启动点”。这自然就使得程序员要为每个线程写一个函数。如我们已经看到的，这可能会导致非面向对象的程序分解。

相反，ACE_Task 处理的是对象，因而在构造 OO 程序时更便于思考。因此，在大多数情况下，当你需要构建多线程程序时，较好的选择是使用 ACE_Task 的子类。这样做有若干好处。首要的是刚刚所提到的，这可以产生更好的 OO 软件。其次，你不必操心你的线程入口是否是静态的，因为 ACE_Task 的入口是一个常规的成员函数。而且，我们会看到 ACE_Task 还包括了一种用于与其他任务进行通信的易于使用的机制。

重申刚才所说的，ACE_Task 可用作：

- 更高级的线程（我们称之为任务）。
- 主动对象模式中的主动对象。

5.2.1 任务的结构

ACE_Task 的结构在本质上与基于 Actor 的系统[III]中的“ Actor ”的结构相类似。该结构如下所示：

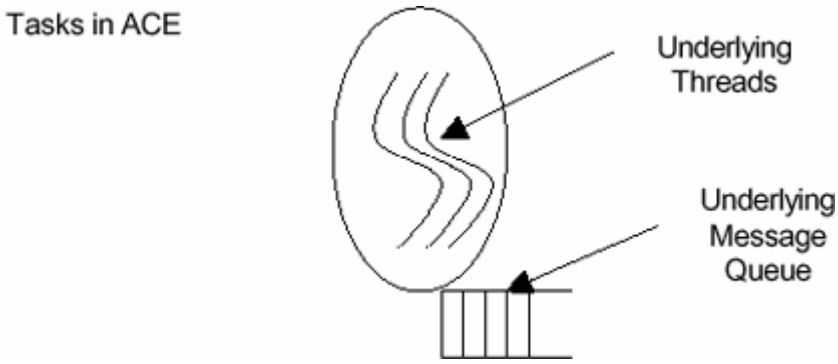

图 5-1 任务结构示意图

图 5-1 说明每个任务都含有一或多个线程，以及一个底层消息队列。各个任务通过这些消息队列进行通信。但是，消息队列并非是程序员需要关注的对象。发送任务可以使用 putq()调用来将消息插入到另一任务的消息队列中。随后接收任务就可以通过使用 getq()调用来从它自己的消息队列里将消息提取出来。

因而，你可以设想一个系统，由多个自治的任务（或主动对象）构成，这些任务通过它们的消息队列相互通信。这样的体系结构有助于大大简化多线程程序的编程模型。

5.2.2 创建和使用任务

如上面所提到的，要创建任务或主动对象，你必须从 ACE_Task 类派生子类。在子类派生之后，必须采取以下步骤：

- **实现服务初始化和终止方法：**open()方法应该包含所有专属于任务的初始化代码。其中可能包括诸如连接控制块、锁和内存这样的资源。close()方法是相应的终止方法。
- **调用启用（Activation）方法：**在主动对象实例化后，你必须通过调用 activate()启用它。要在主动对象中创建的线程的数目，以及其他一些参数，被传递给 activate()方法。activate()方法会使 svc()方法成为所有它生成的线程的启动点。
- **实现服务专有的处理方法：**如上面所提到的，在主动对象被启用后，各个新线程在 svc()方法中启动。应用开发者必须在子类中定义此方法。

下面的例子演示怎样去创建任务：

例 5-1

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

class TaskOne: public ACE_Task<ACE_MT_SYNCH>
```

```
{
public:
    //Implement the Service Initialization and Termination methods
    int open(void*)
    {
        ACE_DEBUG((LM_DEBUG,"(%t) Active Object opened \n"));

        //Activate the object with a thread in it.
        activate();

        return 0;
    }

    int close(u_long)
    {
        ACE_DEBUG((LM_DEBUG, "(%t) Active Object being closed down \n"));
        return 0;
    }

    int svc(void)
    {
        ACE_DEBUG((LM_DEBUG,
            "(%t) This is being done in a separate thread \n"));

        // do thread specific work here
        //.......
        //.......
        return 0;
    }
};

int main(int argc, char *argv[])
{
    //Create the task
    TaskOne *one=new TaskOne;

    //Start up the task
    one->open(0);

    //wait for all the tasks to exit
    ACE_Thread_Manager::instance()->wait();
    ACE_DEBUG((LM_DEBUG,"(%t) Main Task ends \n"));
}
```

上面的例子演示怎样把 ACE_Task 当作更高级的线程来使用。在例子中，TaskOne 类派生自 ACE_Task，并实现了 open()、close()和 svc()方法。在此任务对象实例化后，程序就调用它的 open()方法。该方法依次调用 activate()方法，致使一个新线程被派生和启动。该线程的入口是 svc()方法。主线程等待主动对象线程终止，然后就退出进程。

5.2.3 任务间通信

如前面所提到的，ACE 中的每个任务都有一个底层消息队列（参见上面的图示）。这个消息队列被用作任务间通信的一种方法。当一个任务想要与另一任务“谈话”时，它创建一个消息，并将此消息放入它想要与之谈话的任务的消息队列。接收任务通常用 getq()从消息队列里获取消息。如果队列中没有数据可用，它就进入休眠状态。如果有其他任务将消息插入它的队列，它就会苏醒过来，从队列中拾取数据并处理它。因而，在这种情况下，接收任务将从发送任务那里接收消息，并以应用特定的方式作出反馈。

下一个例子演示两个任务怎样使用它们的底层消息队列进行通信。这个例子包含了经典的生产者－消费者问题的实现。生产者任务生成数据，将它发送给消费者任务。消费者任务随后消费这个数据。使用 ACE_Task 构造，我们可将生产者和消费者看作是不同的 ACE_Task 类型的对象。这两种任务使用底层消息队列进行通信。

例 5-2

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

//The Consumer Task.
class Consumer:
    public ACE_Task<ACE_MT_SYNCH>
{
public:
    int open(void*)
    {
        ACE_DEBUG((LM_DEBUG, "(%t) Producer task opened \n"));

        //Activate the Task
        activate(THR_NEW_LWP,1);

        return 0;
    }

    //The Service Processing routine
    int svc(void)
    {
        //Get ready to receive message from Producer
        ACE_Message_Block * mb =0;
        do
```

```
        {
            mb=0;

            //Get message from underlying queue
            getq(mb);
            ACE_DEBUG((LM_DEBUG,
                "(%t)Got message: %d from remote task\n",*mb->rd_ptr()));
        }while(*mb->rd_ptr()<10);

        return 0;
    }

    int close(u_long)
    {
        ACE_DEBUG((LM_DEBUG,"Consumer closes down \n"));
        return 0;
    }
};

class Producer:
    public ACE_Task<ACE_MT_SYNCH>
{
public:
    Producer(Consumer * consumer):
        consumer_(consumer), data_(0)
    {
        mb_=new ACE_Message_Block((char*)&data_,sizeof(data_));
    }

    int open(void*)
    {
        ACE_DEBUG((LM_DEBUG, "(%t) Producer task opened \n"));

        //Activate the Task
        activate(THR_NEW_LWP,1);
        return 0;
    }

    //The Service Processing routine
    int svc(void)
    {
        while(data_<11)
        {
            //Send message to consumer
```

```
            ACE_DEBUG((LM_DEBUG,
                "(%t)Sending message: %d to remote task\n",data_));
            consumer_->putq(mb_);

            //Go to sleep for a sec.
            ACE_OS::sleep(1);
            data_++;
        }

        return 0;
    }

    int close(u_long)
    {
        ACE_DEBUG((LM_DEBUG,"Producer closes down \n"));
        return 0;
    }

private:
    char data_;
    Consumer * consumer_;
    ACE_Message_Block * mb_;
};

int main(int argc, char * argv[])
{
    Consumer *consumer = new Consumer;
    Producer * producer = new Producer(consumer);

    producer->open(0);
    consumer->open(0);

    //Wait for all the tasks to exit. ACE_Thread_Manager::instance()->wait();
}
```

在此例中，生产者和消费者任务非常相似。它们都没有任何服务初始化或是终止代码。但两个类的 svc()方法是不同的。生产者在 open()方法中被启用后，svc()方法会被调用。在此方法中，生产者生成一个消息，将它插入消费者的队列。消息是使用 ACE_Message_Block 类来生成的（要更多地了解如何使用 ACE_Message_Block，请阅读此教程及在线 ACE 指南中有关消息队列的章节）。生产者维护指向消费者任务（对象）的指针。它通过该指针来将消息放入消费者的消息队列中。该指针在 main()函数中通过生产者的构造器设置。

消费者驻留在它的 svc()方法的循环中，等待数据到达它的消息队列。如果队列中没有数据，消费者就会阻塞并休眠（这是由 ACE_Task 类魔术般地自动完成的）。一旦数据到达消费者的队列，它就会苏醒并消费此数据。

在此例中，生产者发送的数据由一个整数组成。生产者每次将这个整数加一，然后发送给消费者。

如你所看到的，生产者－消费者问题的解决方案十分简单，并且是面向对象的。在编写面向对象的多线程程序时，使用 ACE_Task 是比使用低级线程 API 更好的方法。

5.3 主动对象模式（Active Object Pattern）

主动对象模式用于降低方法执行和方法调用之间的耦合。该模式描述了另外一种更为透明的任务间通信方法。

该模式使用 ACE_Task 类作为主动对象。在这个对象上调用方法时，它就像是常规对象一样。就是说，方法调用是通过同样的->操作符来完成的，其不同在于这些方法的**执行**发生于封装在 ACE_Task 中的线程内。在使用被动或主动对象进行编程时，客户程序看不到什么区别，或仅仅是很小的区别。对于框架开发者来说，这是非常有用的，因为开发者需要使框架客户与框架的内部结构屏蔽开来。这样框架**用户**就不必去担心线程、同步、会合点（rendezvous），等等。

5.3.1 主动对象模式工作原理

主动对象模式是 ACE 实现的较为复杂的模式中的一个。该模式有如下参与者：

1. 主动对象（基于 ACE_Task）。
2. ACE_Activation_Queue。
3. 若干 ACE_Method_Object（主动对象的每个方法都需要有一个方法对象）。
4. 若干 ACE_Future 对象（每个要返回结果的方法都需要这样一个对象）。

我们已经看到，ACE_Task 是怎样创建和封装线程的。要使 ACE_Task 成为主动对象，需要完成一些额外的工作：

必须为所有要从客户异步调用的方法编写方法对象。每个方法对象都派生自 ACE_Method_Object，并会实现它的 call()方法。每个方法对象还维护上下文信息（比如执行方法所需的参数，以及用于获取返回值的 ACE_Future 对象。这些值作为私有属性维护）。你可以把方法对象看作是方法调用的“罩子”（closure）。客户发出方法调用，使得相应的方法对象被实例化，并被放入启用队列（activation queue）中。方法对象是**命令**（Command）模式的一种形式（参见有关设计模式的参考文献）。

ACE_Activation_Queue 是一个队列，方法对象在等待执行时被放入其中。因而启用队列中含有所有等待调用的方法（以方法对象的形式）。封装在 ACE_Task 中的线程保持阻塞，等待任何方法对象被放入启用队列。一旦有方法对象被放入，任务就将该方法对象取出，并调用它的 call()方法。call()方法应该随即调用该方法在 ACE_Task 中的相应实现。在方法实现返回后，call()方法在 ACE_Future 对象中设置（set()）所获得的结果。

客户使用 ACE_Future 对象获取它在主动对象上发出的任何异步操作的结果。一旦客户发出异步调用，立即就会返回一个 ACE_Future 对象。于是客户就可以在任何它喜欢的时候去尝试从“期货”（future）对象中获取结果。如果客户试图在结果被设置之前从期货对象中提取结果，客户将会阻塞。如果客户不希望阻塞，它可以通过使用 ready()调用来轮询（poll）期货对象。如果结果已被设置，该方法返回 1；否则就返回 0。ACE_Future 对象基于“多态期货”（polymorphic futures）的概念。

call()方法的实现应该将返回的 ACE_Future 对象的内部值设置为从调用实际的方法实现所获得的结

果（这个实际的方法实现在 ACE_Task 中编写）。

下面的例子演示主动对象模式是怎样实现的。在此例中，主动对象是一个“ Logger ”（日志记录器）对象。Logger 使用慢速的 I/O 系统来记录发送给它的消息。因为此 I/O 系统很慢，我们不希望主应用任务的执行因为相对来说并非紧急的日志记录而减慢。为了防止此情况的发生，并且允许程序员像发出普通的方法调用那样发出日志调用，我们使用了主动对象模式。

Logger 类的声明如下所示：

例 5-3a

```
//The worker thread with which the client will interact
class Logger: public ACE_Task<ACE_MT_SYNCH>
{
public:
    //Initialization and termination methods
    Logger();
    virtual ~Logger(void);
    virtual int open (void *);
    virtual int close (u_long flags = 0);

    //The entry point for all threads created in the Logger
    virtual int svc (void);

    ////////////////////////////////////////////////////
    //Methods which can be invoked by client asynchronously.
    ////////////////////////////////////////////////////

    //Log message
    ACE_Future<u_long> logMsg(const char* msg);

    //Return the name of the Task
    ACE_Future<const char*> name (void);

    ////////////////////////////////////////////////////
    //Actual implementation methods for the Logger
    ////////////////////////////////////////////////////
    u_long logMsg_i(const char *msg);
    const char * name_i();

private:
    char *name_;
    ACE_Activation_Queue activation_queue_;
};
```

如我们所看到的，Logger 主动对象派生自 ACE_Task，并含有一个 ACE_Activation_Queue。Logger 支持两个异步方法：logMsg()和 name()。这两个方法应该这样来实现：当客户调用它们时，它们实例化相

应的方法对象类型，并将它放入任务的私有启用队列。这两个方法的实际实现（也就是“真正地”完成所需工作的方法）是 logMsg_i()和 name_i()。

下面的代码段显示我们所需的两个方法对象的接口，分别针对 Logger 主动对象中的两个异步方法。

例 5-3b

```
//Method Object which implements the logMsg() method of the active
//Logger active object class
class logMsg_MO: public ACE_Method_Object
{
public:
    //Constructor which is passed a reference to the active object, the
    //parameters for the method, and a reference to the future which
    //contains the result.
    logMsg_MO(Logger * logger, const char * msg,
        ACE_Future<u_long> &future_result);
    virtual ~logMsg_MO();

    //The call() method will be called by the Logger Active Object
    //class, once this method object is dequeued from the activation
    //queue. This is implemented so that it does two things. First it
    //must execute the actual implementation method (which is specified
    //in the Logger class. Second, it must set the result it obtains from
    //that call in the future object that it has returned to the client.
    //Note that the method object always keeps a reference to the same
    //future object that it returned to the client so that it can set the
    //result value in it.
    virtual int call (void);

private:
    Logger * logger_;
    const char* msg_;
    ACE_Future<u_long> future_result_;
};

//Method Object which implements the name() method of the active Logger
//active object class
class name_MO: public ACE_Method_Object
{
public:
    //Constructor which is passed a reference to the active object, the
    //parameters for the method, and a reference to the future which
    //contains the result.
    name_MO(Logger * logger, ACE_Future<const char*> &future_result);
    virtual ~name_MO();
```

```
    //The call() method will be called by the Logger Active Object
    //class, once this method object is dequeued from the activation
    //queue. This is implemented so that it does two things. First it
    //must execute the actual implementation method (which is specified
    //in the Logger class. Second, it must set the result it obtains from
    //that call in the future object that it has returned to the client.
    //Note that the method object always keeps a reference to the same
    //future object that it returned to the client so that it can set the
    //result value in it.
    virtual int call (void);

private:
    Logger * logger_;
    ACE_Future<const char*> future_result_;
};
```

每个方法对象都有一个构造器，用于为方法调用创建“罩子”（closure）。这意味着构造器通过将调用的参数和返回值作为方法对象中的私有成员数据记录下来，来确保它们被此对象“记住”。调用方法包含的代码将对在 Logger 主动对象中定义的实际方法实现（也就是，logMsg_i()和 name_i()）进行委托。

下面的代码段含有两个方法对象的实现：

例 5-3c

```
//Implementation for the logMsg_MO method object.
//Constructor
logMsg_MO::logMsg_MO(Logger * logger, const char * msg, ACE_Future<u_long>
&future_result)
        :logger_(logger), msg_(msg), future_result_(future_result)
{
    ACE_DEBUG((LM_DEBUG, "(%t) logMsg invoked \n"));
}

//Destructor
logMsg_MO::~logMsg_MO()
{
    ACE_DEBUG ((LM_DEBUG, "(%t) logMsg object deleted.\n"));
}

//Invoke the logMsg() method
int logMsg_MO::call (void)
{
    return this->future_result_.set (
        this->logger_->logMsg_i (this->msg_));
}
```

```
//Implementation for the name_MO method object.
//Constructor
name_MO::name_MO(Logger * logger, ACE_Future<const char*> &future_result):
     logger_(logger), future_result_(future_result)
{
     ACE_DEBUG((LM_DEBUG, "(%t) name() invoked \n"));
}

//Destructor
name_MO::~name_MO()
{
     ACE_DEBUG ((LM_DEBUG, "(%t) name object deleted.\n"));
}

//Invoke the name() method
int name_MO::call (void)
{
     return this->future_result_.set (this->logger_->name_i ());
}
```

这两个方法对象的实现是相当直接的。如上面所解释的，方法对象的构造器负责创建“罩子”（捕捉输入参数和结果）。call()方法调用实际的方法实现，随后通过使用 ACE_Future::set()方法来在期货对象中设置值。

下面的代码段显示 Logger 主动对象自己的实现。大多数代码都在 svc()方法中。程序在这个方法中从启用队列里取出方法对象，并调用它们的 call()方法。

例 5-3d

```
//Constructor for the Logger
Logger::Logger()
{
     this->name_= new char[sizeof("Worker")];
     ACE_OS:strcpy(name_,"Worker");
}

//Destructor
Logger::~Logger(void)
{
     delete this->name_;
}

//The open method where the active object is activated
int Logger::open (void *)
{
```

```
    ACE_DEBUG ((LM_DEBUG, "(%t) Logger %s open\n", this->name_));
    return this->activate (THR_NEW_LWP);
}

//Called then the Logger task is destroyed.
int Logger::close (u_long flags = 0)
{
    ACE_DEBUG((LM_DEBUG, "Closing Logger \n"));
    return 0;
}

//The svc() method is the starting point for the thread created in the
//Logger active object. The thread created will run in an infinite loop
//waiting for method objects to be enqueued on the private activation
//queue. Once a method object is inserted onto the activation queue the
//thread wakes up, dequeues the method object and then invokes the
//call() method on the method object it just dequeued. If there are no
//method objects on the activation queue, the task blocks and falls
//asleep.
int Logger::svc (void)
{
    while(1)
    {
        // Dequeue the next method object (we use an auto pointer in
        // case an exception is thrown in the <call>).
        auto_ptr<ACE_Method_Object> mo
        (this->activation_queue_.dequeue ());
        ACE_DEBUG ((LM_DEBUG, "(%t) calling method object\n"));

        // Call it.
        if (mo->call () == -1)
            break;

        // Destructor automatically deletes it.
    }

    return 0;
}

//////////////////////////////////////////////////////////////
//Methods which are invoked by client and execute asynchronously.
//////////////////////////////////////////////////////////////
//Log this message
ACE_Future<u_long> Logger::logMsg(const char* msg)
```

```
{
     ACE_Future<u_long> resultant_future;

     //Create and enqueue method object onto the activation queue
     this->activation_queue_.enqueue
          (new logMsg_MO(this,msg,resultant_future));

     return resultant_future;
}

//Return the name of the Task
ACE_Future<const char*> Logger::name (void)
{
     ACE_Future<const char*> resultant_future;

     //Create and enqueue onto the activation queue
     this->activation_queue_.enqueue
          (new name_MO(this, resultant_future));

     return resultant_future;
}

//////////////////////////////////////////////////////
//Actual implementation methods for the Logger
//////////////////////////////////////////////////////
u_long Logger::logMsg_i(const char *msg)
{
     ACE_DEBUG((LM_DEBUG,"Logged: %s\n",msg));

     //Go to sleep for a while to simulate slow I/O device
     ACE_OS::sleep(2);

     return 10;
}

const char * Logger::name_i()
{
     //Go to sleep for a while to simulate slow I/O device
     ACE_OS::sleep(2);
     return name_;
}
```

最后的代码段演示应用代码，它实例化 Logger 主动对象，并用它来进行日志记录：

例 5-3e

```
//Client or application code.
int main (int, char *[])
{
    //Create a new instance of the Logger task
    Logger *logger = new Logger;

    //The Futures or IOUs for the calls that are made to the logger.
    ACE_Future<u_long> logresult;
    ACE_Future<const char *> name;

    //Activate the logger
    logger->open(0);

    //Log a few messages on the logger
    for (size_t i = 0; i < n_loops; i++)
    {
        char *msg= new char[50];
        ACE_DEBUG ((LM_DEBUG,
                 Issuing a non-blocking logging call\n"));
        ACE_OS::sprintf(msg, "This is iteration %d", i);
        logresult= logger->logMsg(msg);

        //Don’t use the log result here as it isn't that important...
    }

    ACE_DEBUG((LM_DEBUG,
        "(%t)Invoked all the log calls \
        and can now continue with other work \n"));

    //Do some work over here...
    // ...
    // ...

    //Find out the name of the logging task
    name = logger->name ();

    //Check to "see" if the result of the name() call is available
    if(name.ready())
        ACE_DEBUG((LM_DEBUG,"Name is ready! \n"));
    else
        ACE_DEBUG((LM_DEBUG,
             "Blocking till I get the result of that call \n"));
```

```
    //obtain the underlying result from the future object.
    const char* task_name;
    name.get(task_name);
    ACE_DEBUG ((LM_DEBUG,
        "(%t)==> The name of the task is: %s\n\n\n", task_name));

    //Wait for all threads to exit.
    ACE_Thread_Manager::instance()->wait();
}
```

客户代码在 Logger 主动对象上发出若干非阻塞式异步调用。注意这些调用看起来就像是在针对常规被动对象一样。事实上，调用是在另一个单独的线程控制里执行。在发出调用记录多个消息后，客户发出调用来确定任务的名字。该调用返回一个期货给客户。于是客户就开始使用 ready()方法去检查结果是否已在期货对象中设置。然后它使用 get()方法去确定期货的底层值。注意客户的代码是何等的优雅，没有用到线程、同步，等等。所以，主动对象模式可以使你更加容易地编写你的客户代码。

第 6 章 反应器（Reactor）：用于事件多路分离和分派的体系结构模式

反应器模式一直在发展之中，以为高效的事件多路分离和分派提供可扩展的面向对象框架。目前用于事件多路分离的 OS 抽象既复杂又难以使用，因而也容易出错。反应器本质上提供一组更高级的编程抽象，简化了事件驱动的分布式应用的设计和实现。除此而外，反应器还将若干不同种类的事件的多路分离集成到易于使用的 API 中。特别地，反应器对基于定时器的事件、信号事件、基于 I/O 端口监控的事件和用户定义的通知进行统一地处理。

在本章里，我们描述怎样将反应器用于对所有这些不同的事件类型进行多路分离。

6.1 反应器组件

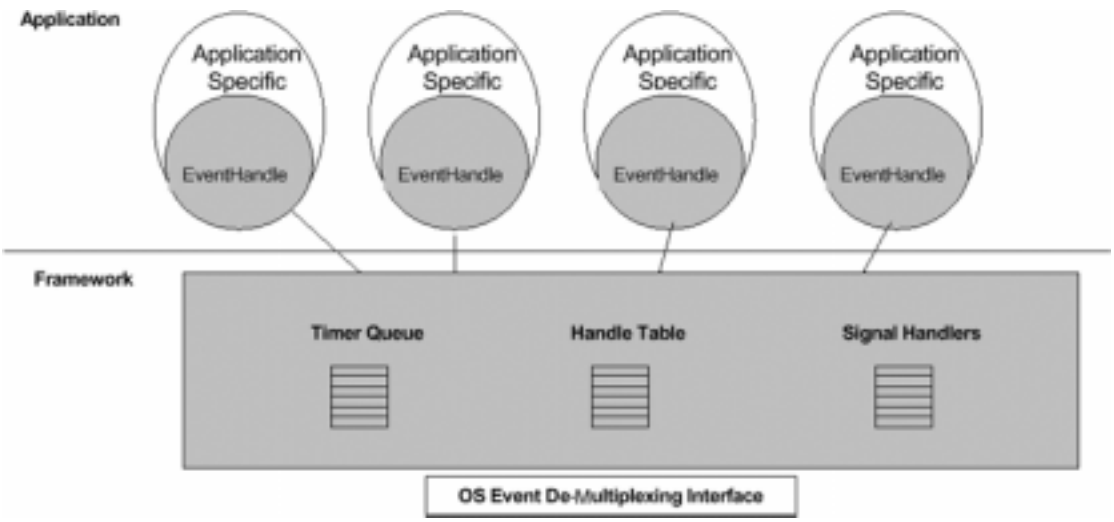

图 6-1 反应器中的内部组件和外部组件的协作

如图 6-1 所示，ACE 中的反应器与若干内部和外部组件协同工作。其基本概念是反应器框架检测事件的发生（通过在 OS 事件多路分离接口上进行侦听），并发出对预登记事件处理器（event handler）对象中的方法的“ 回调 ”（callback）。该方法由应用开发者实现，其中含有应用处理此事件的特定代码。

于是用户（也就是，应用开发者）必须：

1. 创建事件处理器，以处理他所感兴趣的某事件。
2. 在反应器上登记，通知说他有兴趣处理某事件，同时传递他想要用以处理此事件的事件处理器的指针给反应器。

随后反应器框架将自动地：

1. 在内部维护一些表，将不同的事件类型与事件处理器对象关联起来。
2. 在用户已登记的某个事件发生时，反应器发出对处理器中相应方法的回调。

6.2 事件处理器

反应器模式在 ACE 中被实现为 ACE_Reactor 类，它提供反应器框架的功能接口。

如上面所提到的，反应器将事件处理器对象作为服务提供者使用。一旦反应器成功地多路分离和分派了某事件，事件处理器对象就对它进行处理。因此，反应器会在内部记住当特定类型的事件发生时，应该回调**哪一个**事件处理器对象。当应用在反应器上登记它的处理器对象，以处理特定类型的事件时，反应器会创建这种事件和相应的事件处理器的关联。

因为反应器需要记录哪一个事件处理器将被回调，它需要知道所有事件处理器对象的类型。这是通过替换模式（Substitution Pattern）的帮助来实现的（或者换句话说，通过“是……类型”(is a type of）变种继承)。该框架提供名为 ACE_Event_Handler 的抽象接口类，所有应用特有的事件处理器都**必须**由此派生（这使得应用特有的处理器都具有相同的类型，即 ACE_Event_Handler，所以它们可以相互替换)。要了解此概念的更多细节，请阅读替换模式的参考资料[V]。

如果你留意上面的组件图，其中的事件处理器的椭圆形包括灰色的 Event_Handler 部分，对应于 ACE_Event_Handler；以及白色的部分，它对应于应用特有的部分。

图 6-2 对其进行说明：

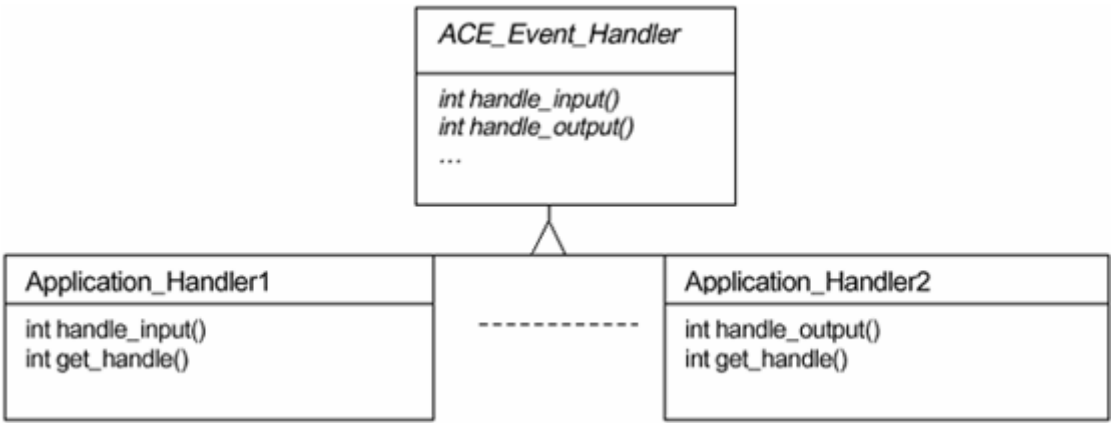

图 6-2 ACE_Event_Handler 类图

ACE_Event_Handler 类拥有若干不同的“handle”（处理）方法，每个处理方法被用于处理不同种类的事件。当应用程序员对特定事件感兴趣时，他就对 ACE_Event_Handler 类进行子类化，并实现他感兴趣的处理方法。如上面所提到的，随后他就在反应器上为特定事件“登记”他的事件处理器类。于是反应器就会保证在此事件发生时，自动回调在适当的事件处理器对象中的适当的”handle”方法。

使用 ACE_Reactor 基本上有三个步骤：

- 创建 ACE_Event_Handler 的子类，并在其中实现适当的“handle_”方法，以处理你想要此事件处理器为之服务的事件类型。(参看表 6-1 来确定你需要实现哪一个“handle_”方法。注意你可以使用同一个事件处理器对象处理多种类型的事件，因而可以重载多于一个的“handle_”方法。)

- 通过调用反应器对象的 register_handler()，将你的事件处理器登记到反应器。
- 在事件发生时，反应器将自动回调相应的事件处理器对象的适当的“ handle_”方法。

下面的简单例子可以帮助我们更好地理解这些步骤：

例 6-1

```
#include <signal.h>
#include ”ace/Reactor.h”

#include ”ace/Event_Handler.h”

//Create our subclass to handle the signal events
//that we wish to handle. Since we know that this particular
//event handler is going to be using signals we only overload the
//handle_signal method.

class MyEventHandler: public ACE_Event_Handler
{
    int handle_signal(int signum, siginfo_t*,ucontext_t*)
    {
        switch(signum)
        {
        case SIGWINCH:
            ACE_DEBUG((LM_DEBUG, ”You pressed SIGWINCH \n”));
            break;
        case SIGINT:
            ACE_DEBUG((LM_DEBUG, ”You pressed SIGINT \n”));
            break;
        }
        return 0;
    }
};

int main(int argc, char *argv[])
{
    //instantiate the handler
    MyEventHandler *eh =new MyEventHandler;

    //Register the handler asking to call back when either SIGWINCH
    //or SIGINT signals occur. Note that in both the cases we asked the
    //Reactor to callback the same Event_Handler i.e., MyEventHandler.
    //This is the reason why we had to write a switch statement in the
    //handle_signal() method above. Also note that the ACE_Reactor is
    //being used as a Singleton object (Singleton pattern)
```

```
    ACE_Reactor::instance()->register_handler(SIGWINCH,eh);
    ACE_Reactor::instance()->register_handler(SIGINT,eh);

    while(1)
        //Start the reactors event loop
        ACE_Reactor::instance()->handle_events();
}
```

在上面的例子中，我们首先创建了一个 ACE_Event_Handler 的子类，在其中我们重载了 handle_signal()方法，因为我们想要使用此处理器来处理多种类型的信号。在主函数中，我们对我们的处理器进行实例化，随后调用 ACE_Reactor 单体（Singleton）的 register_handler，指明我们希望在 SIGWINCH（终端窗口改变信号）或 SIGINT（中断信号，通常是^C）发生时，事件处理器“eh”会被回调。然后，我们通过调用在无限循环中调用 handle_events()来启动反应器的事件循环。无论是发生哪一个事件，反应器都将自动回调 eh->handle_signal()方法，将引发回调的信号号码、以及 siginfo_t 结构（有关 siginfo_t 的更多信息，参见 siginfo.h）传给它。

注意程序是怎样使用单体模式（Singleton Pattern）来获取全局反应器对象的引用的。大多数应用都只需要一个反应器，因而 ACE_Reactor::instance()会确保无论何时此方法被调用，都会返回同一个 ACE_Reactor 实例（要阅读更多有关单体模式的信息，请见“设计模式”参考文献[VI]）。

表 6-1 显示在 ACE_Event_Handler 的子类中必须重载哪些方法，以处理不同的事件类型。

ACE_Event_Handler 中的处理方法	**在子类中重载，所处理事件的类型:**
handle_signal()	信号。当任何在反应器上登记的信号发生时，反应器自动回调该方法。
handle_input()	来自 I/O 设备的输入。当 I/O 句柄（比如 UNIX 中的文件描述符）上的输入可用时，反应器自动回调该方法。
handle_exception()	异常事件。当已在反应器上登记的异常事件发生时（例如，如果收到 SIGURG（紧急信号）），反应器自动回调该方法。
handle_timeout()	定时器。当任何已登记的定时器超时的时候，反应器自动回调该方法。
handle_output()	I/O 设备输出。当 I/O 设备的输出队列有可用空间时，反应器自动回调该方法。

表 6-1 ACE_Event_Handler 中的处理方法及其对应事件

6.2.1 事件处理器登记

如我们在上面的例子中所看到的，登记事件处理器、以处理特定事件，是在反应器上调用 register_handler()方法来完成的。register_handler()方法是重载方法，就是说，实际上有若干方法可用于登记不同的事件类型，每个方法都叫做 register_handler()。但是它们有着不同的特征：它们的参数各不相同。基本上，register_handler()方法采用 handle/event_handle 元组或 signal/event_handler 元组作为参数，并将它们加入反应器的内部分派表。当有事件在 handle 上发生时，反应器在它的内部分派表中查找相应的 event_handler，并自动在它找到的 event_handler 上回调适当的方法。有关登记处理器的专用调用的更多细

节将在后面的部分进行阐释。

6.2.2 事件处理器的拆除和生存期管理

一旦所需的事件被处理后，可能就无需再让事件处理器登记在反应器上。因而，反应器提供了从它的内部分派表中拆除事件处理器的技术。一旦事件处理器被拆除，它就不再会被反应器回调。

为多个客户服务的服务器是这种情况的一个例子。客户连接到服务器，让它完成一些工作，然后从服务器断开。当有新的客户连接到服务器时，一个事件服务器对象被实例化，并登记到服务器的反应器上，以处理所有与此客户有关的 I/O。当客户断开时，服务器必须将事件处理器从反应器的分派队列中拆除，因为它将不再进行任何与此客户有关的 I/O。在此例中，客户/服务器连接可能会被关闭，使得 I/O 句柄（UNIX 中的文件描述符）变得无效。把这样的死掉的句柄从反应器里拆除是很重要的，因为，如果不这样做，反应器将会把此句柄标记为“ 读就绪 ”，并会持续不断地回调此事件处理器的 handle_input()方法。

6.2.2.1 从反应器内部分派表中隐式拆除事件处理器

隐式拆除是更为常用的从反应器中拆除事件处理器的技术。事件处理器的每个“ handle_ ”方法都会返回一个整数给反应器。如果此整数为 0，在处理器方法完成后、事件处理器将保持在反应器上的登记。但是，如果“ handle_ ”方法返回的整数<0，反应器将自动回调此事件处理器的 handle_close()方法，并将它从自己的内部分派表中拆除。handle_close()方法用于执行处理器特有的任何清除工作，它们需要在事件处理器被拆除前完成；其中可以包括像删除处理器申请的动态内存、或关闭日志文件这样的工作。

在上面所描述的例子中，必须将事件处理器从内存中实际清除。这样的清除也可以发生在具体事件处理器类的 handle_close()方法中。设想下面的具体事件处理器：

```
class MyEventHandler: public ACE_Event_Handler
{
public:
    MyEventHandler(){//construct internal data members}

    virtual int
    handle_close(ACE_HANDLE handle, ACE_Reactor_Mask mask)
    {
        delete this; //commit suicide
    }

    ~MyEventHandler(){//destroy internal data members}

private:
    //internal data members
}
```

在从反应器注销、以及 handle_close()挂钩方法被调用时，该类将自己删除。但是，**必须**保证

MyEventHandler 总是动态分配的，否则，全局内存堆可能会崩溃。确保类总是动态地创建的一种办法是将析构器移动到类的私有区域去。例如：

```
class MyEventHandler: public ACE_Event_Handler
{
public:
    MyEventHandler(){//construct internal data members}
    virtual int handle_close(ACE_HANDLE handle, ACE_Reactor_Mask mask)
    {
        dele te this; //commit suicide
    }

private:
    //Class must be allocated dynamically
    ~MyEventHandler(){//destroy internal data members}
};
```

6.2.2.2 从反应器内部分派表中显式拆除事件处理器

另一种从反应器的内部表中拆除事件处理器的方法是显式地调用反应器的 remove_handler()方法集。该方法也是重载方法，就像 register_handler()一样。它采用需要拆除的处理器的句柄或信号号码作为参数，并将该处理器从反应器的内部分派表中拆除。在 remove_handler()被调用时，反应器还自动调用事件处理器的 handle_close()方法。可以这样来对其进行控制：将 ACE_Event_Handler::DONT_CALL 掩码传给 remove_handler()，从而使得 handle_close()方法**不会**被调用。更详尽的 remove_handler()的使用例子将在下面的几个部分给出。

6.3 通过反应器进行事件处理

在下面的几个部分，我们将演示怎样将反应器用于处理各种类型的事件。

6.3.1 I/O 事件多路分离

通过在具体的事件处理器类中重载 handle_input()方法，反应器可用于处理基于 I/O 设备的输入事件。这样的 I/O 可以发生在磁盘文件、管道、FIFO 或网络 socket 上。为了进行基于 I/O 设备的事件处理，反应器在内部使用从操作系统获取的设备句柄（在基于 UNIX 的系统中，该句柄是在文件或 socket 打开时，OS 返回的文件描述符。在 Windows 中该局柄是由 Windows 返回的设备句柄）。网络应用显然是最适于这样的多路分离的应用之一。下面的例子演示反应器是怎样与具体接受器一起使用来构造一个服务器的。

例 6-2

```
#include ”ace/Reactor.h”
#include ”ace/SOCK_Acceptor.h”

#define PORT_NO 19998
typedef ACE_SOCK_Acceptor Acceptor;
//forward declaration
class My_Accept_Handler;

class My_Input_Handler: public ACE_Event_Handler
{
public:
    //Constructor
    My_Input_Handler()
    {
        ACE_DEBUG((LM_DEBUG,”Constructor\n”);
    }

    //Called back to handle any input received
    int handle_input(ACE_HANDLE)
    {
        //receive the data
        peer().recv_n(data,12);
        ACE_DEBUG((LM_DEBUG,”%s\n”,data));

        // do something with the input received.
        // ...
        //keep yourself registered with the reactor
        return 0;
    }

    //Used by the reactor to determine the underlying handle
    ACE_HANDLE get_handle()const
    {
        return this->peer_i().get_handle();
    }

    //Returns a reference to the underlying stream.
    ACE_SOCK_Stream &peer_i()
    {
        return this->peer_;
    }

private:
    ACE_SOCK_Stream peer_;
```

```
    char data [12];
};

class My_Accept_Handler: public ACE_Event_Handler
{
public:
    //Constructor
    My_Accept_Handler(ACE_Addr &addr)
    {
        this->open(addr);
    }

    //Open the peer_acceptor so it starts to ”listen”
    //for incoming clients.
    int open(ACE_Addr &addr)
    {
        peer_acceptor.open(addr);
        return 0;
    }

    //Overload the handle input method
    int handle_input(ACE_HANDLE handle)
    {
        //Client has requested connection to server.
        //Create a handler to handle the connection
        My_Input_Handler *eh= new My_Input_Handler();

        //Accept the connection ”into” the Event Handler
        if (this->peer_acceptor.accept (eh->peer (), // stream
                0, // remote address
                0, // timeout
                1) ==-1) //restart if interrupted
            ACE_DEBUG((LM_ERROR,”Error in connection\n”));

        ACE_DEBUG((LM_DEBUG,”Connection established\n”));

        //Register the input event handler for reading
        ACE_Reactor::instance()->
            register_handler(eh,ACE_Event_Handler::READ_MASK);

        //Unregister as the acceptor is not expecting new clients
        return -1;
    }
```

```
    //Used by the reactor to determine the underlying handle
    ACE_HANDLE get_handle(void) const
    {
        return this->peer_acceptor.get_handle();
    }

private:
    Acceptor peer_acceptor;
};

int main(int argc, char * argv[])
{
    //Create an address on which to receive connections
    ACE_INET_Addr addr(PORT_NO);

    //Create the Accept Handler which automatically begins to “listen”
    //for client requests for connections
    My_Accept_Handler *eh=new My_Accept_Handler(addr);

    //Register the reactor to call back when incoming client connects
    ACE_Reactor::instance()->register_handler(eh,
        ACE_Event_Handler::ACCEPT_MASK);

    //Start the event loop
    while(1)
        ACE_Reactor::instance()->handle_events();
}
```

上面的例子创建了两个具体事件处理器。第一个具体事件处理器 My_Accept_Handler 用于接受和建立从客户到来的连接。另一个事件处理器是 My_Input_Handler，它用于在连接建立后对连接进行处理。因而，My_Accept_Handler 接受连接，并将实际的处理委托给 My_Input_Handler。

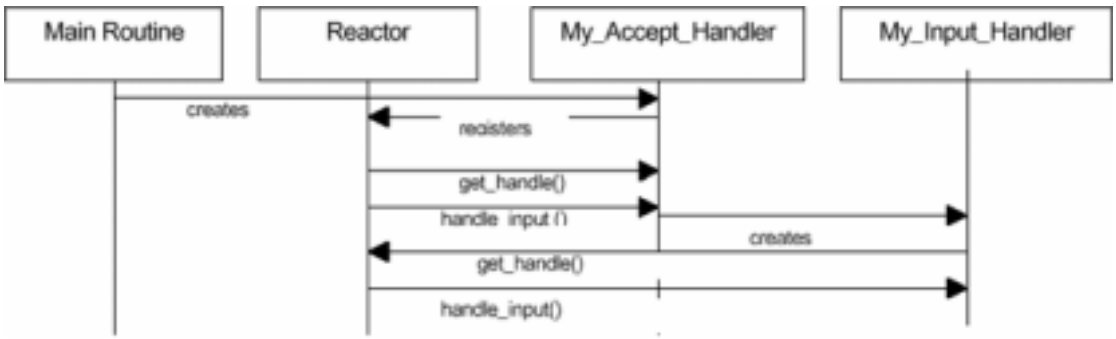

图 6-3 反应器的事件处理

在上面的例子中，我们首先创建了一个 ACE_INET_Addr 地址对象，将我们希望在其上接受连接的端口作为参数传给它。其次，实例化一个类型为 My_Accept_Handler 的对象。随后地址对象通过 My_Accept_Handler 的构造器传递给它。My_Accept_Handler 有一个用于连接建立的底层“具体接受器”（在讲述“IPC”的一章中有与具体接受器相关的内容）。My_Accept_Handler 的构造器将对新连接的“侦听”委托给该具体接受器的 open()方法。在处理器开始侦听连接后，它在反应器上登记，通知说在接收到新连接请求时，它需要被回调。为完成此操作，我们采用 ACE_Event_Handler::ACCEPT_MASK 掩码调用 register_handler()。

当反应器被告知要登记处理器时，它执行“双重分派”来确定事件处理器的底层句柄。为完成此操作，它调用 get_handler()方法。因为反应器使用 get_handle()方法来确定底层流的句柄，在 My_Accept_Handler 中必须实现 get_handle()方法。在此例中，我们简单地调用具体接受器的 get_handle()，它会将适当的句柄返回给反应器。

一旦在该句柄上接收到新的连接请求，反应器会自动地回调 My_Accept_Handler 的 handle_input()方法。随后 Accept Handler（接受处理器）实例化一个新的 Input Handler（输入处理器），并调用具体接受器的 accept()方法来实际地建立连接。注意 Input Handler 底层的流是作为 accept()调用的第一个参数传入的。这使得新实例化的 Input Handler 中的流被设置为在连接建立（由 accept()完成）后立即创建的新流。随后 Accept Handler 将 Input Handler 登记到反应器，通知它如果有任何可读的输入就进行回调（使用 ACE_Event_Handler::READ_MASK）。随后接受处理器返回-1，使自己从反应器的内部事件分派表中被拆除。

现在，如果有任何输入从客户到达，反应器将自动回调 My_Input_Handler::handle_input()。注意在 My_Input_Handler 的 handle_input()方法中，返回给反应器是 0。这指示我们希望保持它的登记；反之在 My_Accept_Handler 中我们在它的 handle_input()中返回-1，以确保它被注销。

除了在上面的例子中使用的 READ_MASK 和 ACCEPT_MASK 而外，还有若干其他的掩码，可在登记或是拆除处理器时使用。这些掩码如表 6-2 所示，它们可与 register_handler()和 remove_handler()方法一起使用。每个掩码保证反应器回调事件处理器时的不同行为方式，通常这意味着不同的“handle”方法会被回调。

掩码	回调方法	何时	和……一起使用
ACE_Event_Handler::READ_MASK	handle_input()	在句柄上有数据可读时。	register_handler()
ACE_Event_Handler::WRITE_MASK	handle_output()	在 I/O 设备输出缓冲区上有可用空间、并且新数据可以发送给它时。	register_handler()
ACE_Event_Handler::TIMER_MASK	handle_close()	传给 handle_close()以指示调用它的原因是超时。	接受器和连接器的 handle_timeout 方法。反应器**不**使用此掩码。
ACE_Event_Handler::ACCEPT_MASK	handle_input()	在 OS 内部的侦听队列上收到了客户的新连接请求时。	register_handler()
ACE_Event_Handler::CONNECT_MASK	handle_input()	在连接已经建立时。	register_handler()
ACE_Event_Handler::DONT_CALL	None.	在反应器的 remove_handler()被调用时保证事件处理器的 handle_close()方法**不**被调用。	remove_handler()

表 6-2 反应器中的掩码

6.4 定时器（Timer）

反应器还包括了调度定时器的方法，它们在超时的时候回调适当的事件处理器的 handle_timeout()方法。为调度这样的定时器，反应器拥有一个 schedule_timer()方法。该方法接收事件处理器（该事件处理器的 handle_timeout()方法将会被回调）、以及以 ACE_Time_value 对象形式出现的延迟作为参数。此外，还可以指定时间间隔，使定时器在它超时后自动被复位。

反应器在内部维护 ACE_Timer_Queue，它以定时器要被调度的顺序对它们进行维护。实际使用的用于保存定时器的数据结构可以通过反应器的 set_timer_queue()方法进行改变。反应器有若干不同的定时器结构可用，包括定时器轮（timer wheel）、定时器堆（timer heap）和哈希式定时器轮（hashed timer wheel）。这些内容将在后面的部分详细讨论。

6.4.1 ACE_Time_Value

ACE_Time_Value 是封装底层 OS 平台的日期和时间结构的包装类。它基于在大多数 UNIX 操作系统上都可用的 timeval 结构；该结构存储以秒和微秒计算的绝对时间。

其他的 OS 平台，比如 POSIX 和 Win32，使用略有不同的表示方法。该类封装这些不同，并提供了可移植的 C++接口。

ACE_Time_Value 类使用运算符重载，提供简单的算术加、减和比较。该类中的方法会对时间量进行“规范化”(normalize)。所谓规范化，是将 timeval 结构中的两个域调整为规范化的编码方式；这种编码方式可以确保精确的比较（更多内容参见附录和参考文献指南）。

6.4.2 设置和拆除定时器

下面的例子演示怎样与反应器一起使用定时器。

例 6-3

```
#include ″test_config.h″
#include ″ace/Timer_Queue.h″
#include ″ace/Reactor.h″
#define NUMBER_TIMERS 10

static int done = 0;
static int count = 0;

class Time_Handler : public ACE_Event_Handler
{
public:
    //Method which is called back by the Reactor when timeout occurs.
```

```
    virtual int handle_timeout (const ACE_Time_Value &tv,
    const void *arg)
    {
        long current_count = long (arg);
        ACE_ASSERT (current_count == count);
        ACE_DEBUG ((LM_DEBUG, "%d: Timer #%d timed out at %d!\n",
            count, current_count, tv.sec()));

        //Increment count
        count ++;

        //Make sure assertion doesn't fail for missing 5th timer.
        if (count ==5)
            count++;

        //If all timers done then set done flag
        if (current_count == NUMBER_TIMERS - 1)
            done = 1;

        //Keep yourself registered with the Reactor.
        return 0;
    }
};

int main (int, char *[])
{
    ACE_Reactor reactor;
    Time_Handler *th=new Time_Handler;
    int timer_id[NUMBER_TIMERS];
    int i;

    for (i = 0; i < NUMBER_TIMERS; i++)
        timer_id[i] = reactor.schedule_timer (th,
                        (const void *) i, // argument sent to handle_timeout()
                        ACE_Time_Value (2 * i + 1)); //set timer to go off with delay

    //Cancel the fifth timer before it goes off
    reactor.cancel_timer(timer_id[5]);//Timer ID of timer to be removed

    while (!done)
        reactor.handle_events ();

    return 0;
}
```

在上面的例子中，首先通过实现事件处理器 Time_Handler 的 handle_timeout()方法，将其设置用以处理超时。主函数实例化 Time_Handler 类型的对象，并使用反应器的 schedule_timer()方法调度多个定时器（10 个）。handle_timeout 方法需要以下参数：指向将被回调的处理器的指针、定时器超时时间，以及一个将在 handle_timeout()方法被回调时发送给它的参数。每次调用 schedule_timer()，它都返回一个唯一的定时器标识符，并随即存储在 timer_id[]数组里。这个标识符可用于在任何时候取消该定时器。在上面的例子中也演示了定时器的取消：在所有定时器被初始调度后，程序通过调用反应器的 cancel_timer()方法（使用相应的 timer_id 作为参数）取消了第五个定时器。

6.4.3 使用不同的定时器队列

不同的环境可能需要不同的调度和取消定时器的方法。在下面的任一条件为真时，实现定时器的算法的性能就会成为一个问题：

- 需要细粒度的定时器。
- 在某一时刻未完成的定时器的数目可能会非常大。
- 算法使用过于昂贵的硬件中断来实现。

ACE 允许用户从若干在 ACE 中已存在的定时器中进行选择，或是根据为定时器定义的接口开发他们自己的定时器。表 6-3 详细列出了 ACE 中可用的各种定时器：

定时器	数据结构描述	性能
ACE_Timer_Heap	定时器存储在优先级队列的堆实现中。	schedule_timer()的开销=O(lg n) cancel_timer()的开销=O(lg n) 查找当前定时器的开销=O(1)
ACE_Timer_List	定时器存储在双向链表中。	schedule_timer()的开销=O(n) cancel_timer()的开销=O(1) 查找当前定时器的开销=O(1)
ACE_Timer_Hash	在这里使用的这种结构是定时器轮算法的变种。性能高度依赖于所用的哈希函数。	schedule_timer()的开销=最坏=O(n) 最佳=O(1) cancel_timer()的开销=O(1) 查找当前定时器的开销=O(1)
ACE_Timer_Wheel	定时器存储在“数组指针”(pointers to arrays)的数组中。每个被指向的数组都已排序。	schedule_timer()的开销=最坏=O(n) cancel_timer()的开销=O(1) 查找当前定时器的开销=O(1)

更多有关定时器的信息见参考文献[VII]

表 6-3 ACE 中的定时器

6.5 处理信号（Signal）

如我们在例 6-1 中所看到的，反应器含有进行信号处理的方法。处理信号的事件处理器应重载 handle_signal()方法，因为该方法将在信号发生时被回调。要为信号登记处理器，可以使用多个 register_handler()方法中的一个，就如同例 6-1 中所演示的那样。如果对特定信号不再感兴趣，通过调用 remove_handler()，处理器可以被拆除，并恢复为先前安装的信号处理器。反应器在内部使用 sigaction()系统调用来设置和恢复信号处理器。通过使用 ACE_Sig_Handlers 类和与其相关联的方法，无需反应器也可以进行信号处理。

使用反应器进行信号处理和使用 ACE_Sig_Handlers 类的重要区别是基于反应器的机制只允许应用给每个信号关联一个事件处理器，而 ACE_Sig_Handlers 类允许在信号发生时，回调多个事件处理器。

6.6 使用通知（Notification）

反应器不仅可以在系统事件发生时发出回调，也可以在用户定义的事件发生时回调处理器。这是通过反应器的“通知”接口来完成的；该接口由两个方法组成：notify()和 max_notify_iterations()。

通过使用 notify()方法，可以明确地指示反应器对特定的事件处理器对象发出回调。在反应器与消息队列、或是协作任务协同使用时，这是十分有用的。可在 ASX 框架组件与反应器一起使用时找到这种用法的一些好例子。

max_notify_iterations()方法通知反应器，每次只完成指定次数的“迭代”(iterations)。也就是说，在一次 handle_events()调用中只处理指定数目的“通知”。因而如果使用 max_notify_iterations()将迭代的次数设置为 20，而又有 25 个通知同时到达，handle_events()方法一次将只处理这些通知中的 20 个。剩下的五个通知将在 handle_events()下一次在事件循环中被调用时再处理。

下面的例子将进一步阐释这些概念：

例 6-4

```
#include ”ace/Reactor.h”
#include ”ace/Event_Handler.h”
#include ”ace/Synch_T.h”
#include ”ace/Thread_Manager.h”
#define WAIT_TIME 1
#define SLEEP_TIME 2

class My_Handler: public ACE_Event_Handler
{
public:
    //Start the event handling process.
    My_Handler()
    {
        ACE_DEBUG((LM_DEBUG,”Event Handler created\n”));
        ACE_Reactor::instance()->max_notify_iterations(5);
        return 0;
    }
```

```
    //Perform the notifications i.e., notify the reactor 10 times
    void perform_notifications()
    {
        for(int i=0;i<10;i++)
            ACE_Reactor::instance()->
                notify(this,ACE_Event_Handler::READ_MASK);
    }

    //The actual handler which in this case will handle the notifications
    int handle_input(int)
    {
        ACE_DEBUG((LM_DEBUG,"Got notification # %d\n",no));
        no++;
        return 0;
    }

private:
    static int no;
};

//Static members
int My_Handler::no=1;

int main(int argc, char *argv[])
{
    ACE_DEBUG((LM_DEBUG,"Starting test \n"));

    //Instantiating the handler
    My_Handler handler;

    //The done flag is set to not done yet.
    int done=0;

    while(1)
    {
        //After WAIT_TIME the handle_events will fall through if no events
        //arrive.
        ACE_Reactor::instance()->handle_events(ACE_Time_Value(WAIT_TIME));
        if(!done)
        {
            handler.perform_notifications();
            done=1;
        }
```

```
        sleep(SLEEP_TIME);
    }
}
```

上面的例子和平常一样创建了具体处理器、并重载了 handle_input()方法；这和我们需要我们的处理器对来自 I/O 设备的输入数据进行处理时是一样的。处理器还含有 open()方法（用于执行处理器初始化）和实际完成通知的方法。

在 main()函数中，我们首先实例化我们的具体处理器的一个实例。通过使用反应器的 max_notify_iterations()方法，处理器的构造器保证 max_notify_iterations 被设置为 5。在此之后，反应器的事件处理循环开始了。

在这里，事件处理循环中值得注意的一个主要区别是，程序传递给 handle_events()一个 ACE_Time_Value。如果在此时间内没有事件发生，handle_events()方法就会结束。在 handle_events()结束后，perform_notification()被调用，它使用反应器的 notify()方法来请求反应器通知处理器（它是在事件发生时被作为参数传入的）。随后反应器就使用所收到的掩码来执行对处理器的适当“ handle ”方法的调用。在此例中，通过传递 ACE_Event_Handler::READ_MASK，我们使用 notify()来通知我们的事件处理器有输入，从而使得反应器回调该处理器的 handle_input()方法。

因为我们已将 max_notify_iterations 设为 5，所以在一次 handle_events()调用过程中反应器实际上只会发出 5 个通知。为说明这一点，在发出下一个 handle_events()调用前，我们使反应事件循环停止，时间为 SLEEP_TIME。

上面的例子过于简单，也非常不实际，因为通知发生的线程和反应器所在的线程是同一线程。更为实际的例子是：事件发生在另一线程中，并将这些事件通知反应器线程。下面所演示的是同一个例子，不过是由不同的线程来执行通知：

例 6-5

```
#include ″ace/Reactor.h″
#include ″ace/Event_Handler.h″
#include ″ace/Synch_T.h″
#include ″ace/Thread_Manager.h″

class My_Handler: public ACE_Event_Handler
{
public:
    //Start the event handling process.
    My_Handler()
    {
        ACE_DEBUG((LM_DEBUG,″Got open\n″));
        activate_threads();
        ACE_Reactor::instance()->max_notify_iterations(5);
        return 0;
    }

    //Spawn a separate thread so that it notifies the reactor
    void activate_threads()
```

```
    {
        ACE_Thread_Manager::instance()
            ->spawn((ACE_THR_FUNC)svc_start,(void*)this);
    }

    //Notify the Reactor 10 times.
    void svc()
    {
        for(int i=0;i<10;i++)
            ACE_Reactor::instance()->
                notify(this, ACE_Event_Handler::READ_MASK);
    }

    //The actual handler which in this case will handle the notifications
    int handle_input(int)
    {
        ACE_DEBUG((LM_DEBUG, ”Got notification # %d\n”, no));
        no++;
        return 0;
    }

    //The entry point for the new thread that is to be created.
    static int svc_start(void* arg);

private:
    static int no;
};

//Static members
int My_Handler::no=1;

int My_Handler::svc_start(void* arg)
{
    My_Handler *eh= (My_Handler*)arg;
    eh->svc();
    return -1; //de-register from the reactor
}

int main(int argc, char *argv[])
{
    ACE_DEBUG((LM_DEBUG,”Starting test \n”));
    My_Handler handler;

    while(1)
```

```
    {
        ACE_Reactor::instance()->handle_events();
        sleep(3);
    }
}
```

这个例子和前一个例子非常相像，除了增加了一些方法来派生线程，并随即在事件处理器中启用线程。特别地，具体处理器 My_Handler 的构造器调用启用方法，该方法使用 ACE_Thread_Manager::spawn()方法来派生一个分离的线程，并将 svc_start()作为它的入口。

svc_start()方法调用 perform_notifications()来将通知发送给反应器，但这一次它们是从新线程、而不是反应器所在的线程发送的。注意该线程的入口 svc_start()被定义为静态方法（它随后调用非静态的 svc()方法）。这是线程库的使用要求，也就是，线程的入口必须是文件范围内的静态函数。

第 7 章 接受器（Acceptor）和连接器（Connector）：连接建立模式

接受器/连接器模式设计用于降低连接建立与连接建立后所执行的服务之间的耦合。例如，在 WWW 浏览器中，所执行的服务或“实际工作”是解析和显示客户浏览器接收到的 HTML 页面。连接建立是次要的，可能通过 BSD socket 或其他一些等价的 IPC 机制来完成。使用这些模式允许程序员专注于“实际工作”，而最少限度地去关心怎样在服务器和客户之间建立连接。而另外一方面，程序员也可以独立于他所编写的、或将要编写的服务例程，去调谐连接建立的策略。

因为该模式降低了服务和连接建立方法之间的耦合，非常容易改动其中一个，而不影响另外一个，从而也就可以复用以前编写的连接建立机制和服务例程的代码。在同样的例子中，使用这些模式的浏览器程序员一开始可以构造他的系统、使用特定的连接建立机制来运行它和测试它；然后，如果先前的连接机制被证明不适合他所构造的系统，他可以决定说他希望将底层连接机制改变为多线程的（或许使用线程池策略）。因为此模式提供了严格的去耦合，只需要极少的努力就可以实现这样的变动。

在你能够清楚地理解这一章中的许多例子，特别是更为高级的部分之前，你必须通读有关反应器和 IPC_SAP 的章节（特别是接受器和连接器部分）。此外，你还可能需要参考线程和线程管理部分。

7.1 接受器模式

接受器通常被用在你本来会使用 BSD accept()系统调用的地方。接受器模式也适用于同样的环境，但就如我们将看到的，它提供了更多的功能。在 ACE 中，接收器模式借助名为 ACE_Acceptor 的“工厂”（Factory）实现。工厂（通常）是用于对助手对象的实例化过程进行抽象的类。在面向对象设计中，复杂的类常常会将特定功能委托给助手类。复杂的类对助手的创建和委托必须很灵活。这种灵活性是在工厂的帮助下获得的。工厂允许一个对象通过改变它所委托的对象来改变它的底层策略，而工厂提供给应用的接口却是一样的，这样，可能根本就无需对客户代码进行改动（有关工厂的更多信息，请阅读“设计模式”中的参考文献）。

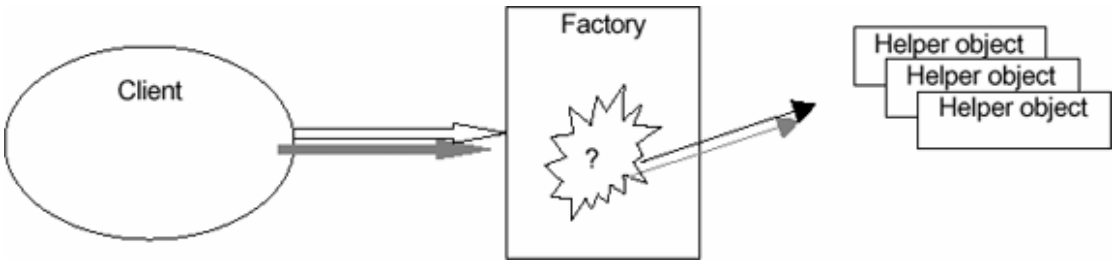

图 7-1 工厂模式示意图

ACE_Acceptor 工厂允许应用开发者改变“助手”对象，以用于：

- 被动连接建立
- 连接建立后的处理

同样地，ACE_Connector 工厂允许应用开发者改变“助手”对象，以用于：

- 主动连接建立
- 连接建立后的处理

下面的讨论同时适用于接受器和连接器，所以作者将只讨论接受器，而连接器同样具有相应的参数。

ACE_Acceptor 被实现为模板容器，通过两个类作为实参来进行实例化。第一个参数实现特定的服务（类型为 ACE_Event_Handler。因为它被用于处理 I/O 事件，所以必须来自事件处理类层次），应用在建立连接后执行该服务；第二个参数是“具体的”接受器（可以是在 IPC_SAP 一章中讨论的各种变种）。

特别要注意的是 ACE_Acceptor 工厂和底层所用的具体接受器是非常不同的。具体接受器可完全独立于 ACE_Acceptor 工厂使用，而无需涉及我们在这里讨论的接受器模式（独立使用接受器已在 IPC_SAP 一章中讨论和演示）。ACE_Acceptor 工厂内在于接受器模式，并且不能在没有底层具体接受器的情况下使用。ACE_Acceptor **使用**底层的具体接受器来建立连接。如我们已看到的，有若干 ACE 的类可被用作 ACE_Acceptor 工厂模板的第二个参数（也就是，具体接受器类）。但是服务处理类必须由应用开发者来实现，而且其类型必须是 ACE_Event_Handler。ACE_Acceptor 工厂可以这样来实例化：

```
typedef ACE_Acceptor<My_Service_Handler,ACE_SOCK_ACCEPTOR> MyAcceptor;
```

这里，名为 My_Service_Handler 的事件处理器和具体接受器 ACE_SOCK_ACCEPTOR 被传给 MyAcceptor。ACE_SOCK_ACCEPTOR 是基于 BSD socket 流家族的 TCP 接受器（各种可传给接受器工厂的不同类型的接受器，见表 7-1 和 IPC 一章）。请再次注意，在使用接受器模式时，我们总是处理两个接受器：名为 ACE_Acceptor 的工厂接受器，和 ACE 中的某种具体接受器，比如 ACE_SOCK_ACCEPTOR（你可以创建自定义的具体接受器来取代 ACE_SOCK_ACCEPTOR，但你将很可能无需改变 ACE_Acceptor 工厂类中的任何东西）。

重要提示：ACE_SOCK_ACCEPTOR 实际上是一个宏，其定义为：

```
#define ACE_SOCK_ACCEPTOR ACE_SOCK_Acceptor, ACE_INET_Addr
```

我们认为这个宏的使用是必要的，因为在类中的 typedefs 在某些平台上无法工作。如果不是这样的话，ACE_Acceptor 就可以这样来实例化：

```
typedef ACE_Acceptor<My_Service_Handler,ACE_SOCK_Acceptor>MyAcceptor;
```

在表 7-1 中对宏进行了说明。

7.1.1 组件

如上面的讨论所说明的，在接受器模式中有三个主要的参与类：

- **具体接受器**：它含有建立连接的特定策略，连接与底层的传输协议机制系在一起。下面是在 ACE 中的各种具体接受器的例子：ACE_SOCK_ACCEPTOR（使用 TCP 来建立连接）、ACE_LSOCK_ACCEPTOR（使用 UNIX 域 socket 来建立连接），等等。

- **具体服务处理器**：由应用开发者编写，它的 open()方法在连接建立后被自动回调。接受器框架假定服务处理类的类型是 ACE_Event_Handler，这是 ACE 定义的接口类（该类已在反应器一章中详细讨论过）。另一个特别为接受器和连接器模式的服务处理而创建的类是 ACE_Svc_Handler。该类不仅基于 ACE_Event_Handler 接口（这是使用反应器所必需的），同时还基于在 ASX 流框架中使用的 ACE_Task 类。ACE_Task 类提供的功能有：创建分离的线程、使用消息队列来存储到来的数据消息、并发地处理它们，以及其他一些有用的功能。如果与接受器模式一起使用的具体服务处理器派生自 ACE_Svc_Handler、而不是 ACE_Event_Handler，它就可以获得这些额外的功能。对 ACE_Svc_Handler 中的额外功能的使用，在这一章的高级课程里详细讨论。在下面的讨论中，我们将使用 ACE_Svc_Handler 作为我们的事件处理器。在简单的 ACE_Event_Handler 和 ACE_Svc_Handler 类之间的重要区别是，后者拥有一个底层通信流组件。这个流在 ACE_Svc_Handler 模板被实例化的时候设置。而在使用 ACE_Event_Handler 的情况下，我们必须自己增加 I/O 通信端点（也就是，流对象），作为事件处理器的私有数据成员。因而，在这样的情况下，应用开发者应该将他的服务处理器创建为 ACE_Svc_Handler 类的子类，并首先实现将被框架自动回调的 open()方法。此外，因为 ACE_Svc_Handler 是一个模板，通信流组件和锁定机制是作为模板参数被传入的。
- **反应器**：与 ACE_Acceptor 协同使用。如我们将看到的，在实例化接受器后，我们启动反应器的事件处理循环。反应器，如先前所解释的，是一个事件分派类；而在此情况下，它被接受器用于将连接建立事件分派到适当的服务处理例程。

7.1.2 用法

通过观察一个简单的例子，可以进一步了解接受器。这个例子是一个简单的应用，它使用接受器接受连接，随后回调服务例程。当服务例程被回调时，它就让用户知道连接已经被成功地建立。

例 7-1

```
#include ”ace/Reactor.h”
#include ”ace/Svc_Handler.h”
#include ”ace/Acceptor.h”
#include ”ace/Synch.h”
#include ”ace/SOCK_Acceptor.h”

//Create a Service Handler whose open() method will be called back
//automatically. This  class MUST derive from ACE_Svc_Handler which is an
//interface and as can be seen is a  template container class itself. The
//first parameter to ACE_Svc_Handler is the  underlying stream that it
//may use for communication. Since we are using TCP sockets the  stream
//is ACE_SOCK_STREAM. The second is the internal synchronization
//mechanism it  could use. Since we have a single threaded application we
//pass it a “null” lock which  will do nothing.
class My_Svc_Handler:
    public ACE_Svc_Handler <ACE_SOCK_STREAM,ACE_NULL_SYNCH>
{
//the open method which will be called back automatically after the
//connection has been  established.
```

```
public:
     int open(void*)
     {
          cout<<”Connection established”<<endl;
     }
};

// Create the acceptor as described above.
typedef ACE_Acceptor<My_Svc_Handler,ACE_SOCK_ACCEPTOR> MyAcceptor;

int main(int argc, char* argv[])
{
     //create the address on which we wish to connect. The constructor takes
     //the port  number on which to listen and will automatically take the
     //host’s IP address as the IP  Address for the addr object
     ACE_INET_Addr addr(PORT_NUM);

     //instantiate the appropriate acceptor object with the address on which
     //we wish to  accept and the Reactor instance we want to use. In this
     //case we just use the global  ACE_Reactor singleton. (Read more about
     //the reactor in the previous chapter)
     MyAcceptor acceptor(addr, ACE_Reactor::instance());

     while(1)
          // Start the reactor’s event loop
          ACE_Reactor::instance()->handle_events();
}
```

在上面的例子中，我们首先创建我们希望在其上接受连接的端点地址。因为我们决定使用 TCP/IP 作为底层的连接协议，我们创建一个 ACE_INET_Addr 来作为我们的端点，并将我们所要侦听的端口号传给它。我们将这个地址和反应器单体的实例传给我们要在此之后进行实例化的接受器。这个接受器在被实例化后，将自动接受任何在 PORT_NUM 端口上的连接请求，并且在为这样的请求建立连接之后回调 My_Svc_Handler 的 open()方法。注意当我们实例化 ACE_Acceptor 工厂时，我们传给它的是我们想要使用的具体接受器（也就是 ACE_SOCK_ACCEPTOR）和具体服务处理器（也就是 My_Svc_Handler）。

现在让我们尝试一下更为有趣的事情。在下一个例子中，我们将在连接请求到达、服务处理器被回调之后，将服务处理器登记回反应器。现在，如果新创建的连接上有任何数据到达，我们的服务处理例程 handle_input()方法都会被自动回调。因而在此例中，我们在同时使用反应器和接受器的特性：

例 7-2

```
#include ”ace/Reactor.h”
#include ”ace/Svc_Handler.h”
#include ”ace/Acceptor.h”
#include ”ace/Synch.h”
#include ”ace/SOCK_Acceptor.h”
```

```
#define PORT_NUM 10101
#define DATA_SIZE 12

//forward declaration
class My_Svc_Handler;

//Create the Acceptor class
typedef ACE_Acceptor<My_Svc_Handler,ACE_SOCK_ACCEPTOR>
MyAcceptor;

//Create a service handler similar to as seen in example 1. Except this
//time include the handle_input() method which will be called back
//automatically by the reactor when new data arrives on the newly
//established connection
class My_Svc_Handler:
    public ACE_Svc_Handler <ACE_SOCK_STREAM,ACE_NULL_SYNCH>
{
public:
    My_Svc_Handler()
    {
        data= new char[DATA_SIZE];
    }

    int open(void*)
    {
        cout<<"Connection established"<<endl;

        //Register the service handler with the reactor
        ACE_Reactor::instance()->register_handler(this,
            ACE_Event_Handler::READ_MASK);

        return 0;
    }

    int handle_input(ACE_HANDLE)
    {
        //After using the peer() method of ACE_Svc_Handler to obtain a
        //reference to the underlying stream of the service handler class
        //we call recv_n() on it to read the data which has been received.
        //This data is stored in the data array and then printed out
        peer().recv_n(data,DATA_SIZE);
        ACE_OS::printf("<< %s\n",data);

        //keep yourself registered with the reactor
```

```
        return 0;
    }

private:
    char* data;
};

int main(int argc, char* argv[])
{
    ACE_INET_Addr addr(PORT_NUM);

    //create the acceptor
    MyAcceptor acceptor(addr, //address to accept on
        ACE_Reactor::instance()); //the reactor to use

    while(1)
        //Start the reactor's event loop
        ACE_Reactor::instance()->handle_events();
}
```

这个例子和前一例子的唯一区别是我们在服务处理器的 open()方法中将服务处理器登记到反应器上。因此，我们必须编写 handle_input()方法；当数据在连接上到达时，这个方法会被反应器回调。在此例中我们只是将我们接收到的数据打印到屏幕上。ACE_Svc_Handler 类的 peer()方法返回对底层的对端流的引用。我们使用底层流包装类的 recv_n()方法来获取连接上接收到的数据。

该模式真正的威力在于，底层的连接建立机制完全封装在具体接受器中，从而可以很容易地改变。在下一个例子里，我们改变底层的连接建立机制，让它使用 UNIX 域 socket、而不是 TCP socket。这个例子（下划线标明少量变动）如下所示：

例 7-3

```
class My_Svc_Handler:
    public ACE_Svc_Handler <ACE_LSOCK_STREAM,ACE_NULL_SYNCH>{
public:
    int open(void*)
    {
        cout<<"Connection established"<<endl;
        ACE_Reactor::instance()
            ->register_handler(this,ACE_Event_Handler::READ_MASK);
    }

    int handle_input(ACE_HANDLE)
    {
        char* data= new char[DATA_SIZE];
        peer().recv_n(data,DATA_SIZE);
        ACE_OS::printf("<< %s\n",data);
```

```
        return 0;
    }
};

typedef ACE_Acceptor<My_Svc_Handler,ACE_LSOCK_ACCEPTOR> MyAcceptor;

int main(int argc, char* argv[])
{
    ACE_UNIX_Addr addr("/tmp/addr.ace");
    MyAcceptor acceptor(address, ACE_Reactor::instance());

    while(1) /* Start the reactor's event loop */
        ACE_Reactor::instance()->handle_events();
}
```

例 7-2 和例 7-3 不同的地方标注了下划线。正如我们所说过的，两个程序间的不同非常之少，但是它们使用的连接建立范式却极不相同。ACE 中可用的连接建立机制在表 7-1 中列出：

接受器类型	所用地址	所用流	具体接受器
TCP 流接受器	ACE_INET_Addr	ACE_SOCK_STREAM	ACE_SOCK_ACCEPTOR
UNIX 域本地流 socket 接受器	ACE_UNIX_Addr	ACE_LSOCK_STREAM	ACE_LSOCK_ACCEPTOR
管道作为底层通信机制	ACE_SPIPE_Addr	ACE_SPIPE_STREAM	ACE_SPIPE_ACCEPTOR

表 7-1 ACE 中的连接建立机制

7.2 连接器

连接器与接受器非常类似。它也是一个工厂，但却是用于*主动地*连接远程主机。在连接建立后，它将自动回调适当的服务处理对象的 open()方法。连接器通常被用在你本来会使用 BSD connect()调用的地方。在 ACE 中，连接器，就如同接受器，被实现为名为 ACE_Connector 的模板容器类。如先前所提到的，它需要两个参数，第一个是事件处理器类，它在连接建立时被调用；第二个是“具体的”连接器类。

你**必须**注意，底层的具体连接器和 ACE_Connector 工厂是非常不一样的。ACE_Connector 工厂**使用**底层的具体连接器来建立连接。随后 ACE_Connector 工厂使用适当的事件或服务处理例程（通过模板参数传入）来在具体的连接器建立起连接之后处理新连接。如我们在 IPC 一章中看到的，没有 ACE_Connector 工厂，也可以使用这个具体的连接器。但是，没有具体的连接器类，就会无法使用 ACE_Connector 工厂（因为要由具体的连接器类来实际处理连接建立）。

下面是对 ACE_Connector 类进行实例化的一个例子：

```
typedef ACE_Connector<My_Svc_Handler,ACE_SOCK_CONNECTOR> MyConnector;
```

这个例子中的第二个参数是具体连接器类 ACE_SOCK_CONNECTOR。连接器和接受器模式一样，在内部使用反应器来在连接建立后回调服务处理器的 open()方法。我们可以复用我们为前面的接受器例

子所写的服务处理例程。

一个使用连接器的例子可以进一步说明这一点：

例 7-4

```
typedef ACE_Connector<My_Svc_Handler,ACE_SOCK_CONNECTOR> MyConnector;

int main(int argc, char * argv[])
{
    ACE_INET_Addr addr(PORT_NO,HOSTNAME);
    My_Svc_Handler * handler= new My_Svc_Handler;

    //Create the connector
    MyConnector connector;

    //Connects to remote machine
    if(connector.connect(handler,addr)==-1)
        ACE_ERROR(LM_ERROR,"%P|%t, %p","Connection failed");

    //Registers with the Reactor
    while(1)
        ACE_Reactor::instance()->handle_events();
}
```

在上面的例子中，HOSTNAME 和 PORT_NO 是我们希望主动连接到的机器和端口。在实例化连接器之后，我们调用它的连接方法，将服务例程（会在连接完全建立后被回调），以及我们希望连接到的地址传递给它。

7.2.1 同时使用接受器和连接器

一般而言，接受器和连接器模式会在一起使用。在客户/服务器应用中，服务器通常含有接受器，而客户含有连接器。但是，在特定的应用中，可能需要同时使用接受器和连接器。下面给出这样的应用的一个例子：一条消息被反复发送给对端机器，而与此同时也从远端接受另一消息。因为两种功能必须同时执行，简单的解决方案就是分别在不同的线程里发送和接收消息。

这个例子同时包含接受器和连接器。用户可以在命令行上给出参数，告诉应用它想要扮演服务器还是客户角色。随后应用就相应地调用 main_accept()或 main_connect()。

例 7-5

```
#include "ace/Reactor.h"
#include "ace/Svc_Handler.h"
#include "ace/Acceptor.h"
#include "ace/Synch.h"
#include "ace/SOCK_Acceptor.h"
#include "ace/Thread.h"
```

```
//Add our own Reactor singleton
typedef ACE_Singleton<ACE_Reactor,ACE_Null_Mutex> Reactor;

//Create an Acceptor
typedef ACE_Acceptor<MyServiceHandler,ACE_SOCK_ACCEPTOR> Acceptor;

//Create a Connector
typedef ACE_Connector<MyServiceHandler,ACE_SOCK_CONNECTOR> Connector;

class MyServiceHandler:
public ACE_Svc_Handler<ACE_SOCK_STREAM,ACE_NULL_SYNCH>
{
public:
    //Used by the two threads “globally” to determine their peer stream
    static ACE_SOCK_Stream* Peer;

    //Thread ID used to identify the threads
    ACE_thread_t t_id;

    int open(void*)
    {
        cout<<”Acceptor: received new connection”<<endl;

        //Register with the reactor to remember this handle
        Reactor::instance()
            ->register_handler(this,ACE_Event_Handler::READ_MASK);

        //Determine the peer stream and record it globally
        MyServiceHandler::Peer=&peer();

        //Spawn new thread to send string every second
        ACE_Thread::spawn((ACE_THR_FUNC)send_data,0,THR_NEW_LWP,&t_id);

        //keep the service handler registered by returning 0 to the
        //reactor
        return 0;
    }

    static void* send_data(void*)
    {
        while(1)
        {
            cout<<”>>Hello World”<<endl;
```

```
            Peer->send_n(”Hello World”,sizeof(”Hello World”));

            //Go to sleep for a second before sending again
            ACE_OS::sleep(1);
        }

        return 0;
    }

    int handle_input(ACE_HANDLE)
    {
        char* data= new char[12];

        //Check if peer aborted the connection
        if(Peer.recv_n(data,12)==0)
        {
            cout<<”Peer probably aborted connection”);
            ACE_Thread::cancel(t_id); //kill sending thread ..
            return -1; //de-register from the Reactor.
        }

        //Show what you got..
        cout<<”<< %s\n”,data”<<endl;

        //keep yourself registered
        return 0;
    }
};

//Global stream identifier used by both threads
ACE_SOCK_Stream * MyServiceHandler::Peer=0;

void main_accept()
{
    ACE_INET_Addr addr(PORT_NO);
    Acceptor myacceptor(addr,Reactor::instance());
    while(1)
        Reactor::instance()->handle_events();

    return 0;
}

void main_connect()
{
```

```
    ACE_INET_Addr addr(PORT_NO,HOSTNAME);
    Connector myconnector;
    myconnector.connect(my_svc_handler,addr);
    while(1)
        Reactor::instance()->handle_events();
}

int main(int argc, char* argv[])
{
    // Use ACE_Get_Opt to parse and obtain arguments and then call the
    // appropriate function for accept or connect.
    ...
}
```

这个简单的例子演示怎样联合使用接受器和连接模式来生成服务处理例程，该例程与底层的网络连接建立方法是完全分离的。通过改变相应的设定具体连接器和接受器的模板参数，可以很容易地改用任何其他的底层网络连接建立协议。

7.3 高级课程

下面的部分更为详细地解释接受器和连接器模式实际上是如何工作的。如果你想要调谐服务处理和连接建立策略（其中包括调谐底层具体连接器将要使用的服务处理例程的创建和并发策略，以及连接建立策略），对该模式的进一步了解就是必要的。此外，还有一部分内容解释怎样使用通过 ACE_Svc_Handler 类自动获得的高级特性。最后，我们说明怎样与接受器和连接器模式一起使用简单的轻量级 ACE_Event_Handler。

7.3.1 ACE_SVC_HANDLER 类

如上面所提到的，ACE_Svc_Handler 类基于 ACE_Task（它是 ASX 流框架的一部分）和 ACE_Event_Handler 接口类。因而 ACE_Svc_Handler 既是任务，又是事件处理器。这里我们将简要介绍 ACE_Task 和 ACE_Svc_Handler 的功能。

7.3.1.1 ACE_Task

ACE_Task 被设计为与 ASX 流框架一起使用；ASX 基于 UNIX 系统 V 中的流机制。在设计上 ASX 与 Larry Peterson 构建的 X-kernel 协议工具非常类似。

ASX 的基本概念是：到来的消息会被分配给由若干*模块*（module）组成的流。每个模块在到来的消息上执行某种固定操作，然后把它传递给下一个模块作进一步处理，直到它到达流的末端为止。模块中的实际处理由*任务*来完成。每个模块通常有两个任务，一个用于处理到来的消息，一个用于处理外出的消息。在构造协议栈时，这种结构是非常有用的。因为每个模块都有固定的简单接口，所创建的模块可

以很容易地在不同的应用间复用。例如，设想一个应用，它处理来自数据链路层的消息。程序员会构造若干模块，每个模块分别处理不同层次的协议。因而，他会构造一个单独的模块，进行网络层处理；另一个进行传输层处理；还有一个进行表示层处理。在构造这些模块之后，它们可以（在 ASX 的帮助下）被“*串*”成一个流来使用。如果后来创建了一个新的（也许是更好的）传输模块，就可以在不对程序产生任何影响的情况下、在流中替换先前的传输模块。注意模块就像是容纳任务的容器。这些任务是实际的处理元件。一个模块可能需要两个任务，如同在上面的例子中；也可能只需要一个任务。如你可能会猜到的，ACE_Task 是模块中被称为任务的处理元件的实现。

7.3.1.2 任务通信的体系结构

每个 ACE_Task 都有一个内部的消息队列，用以与其他任务、模块或是外部世界通信。如果一个 ACE_Task 想要发送一条消息给另一个任务，它就将此消息放入目的任务的消息队列中。一旦目的任务收到此消息，它就会立即对它进行处理。

所有 ACE_Task 都可以作为 0 个或多个线程来运行。消息可以由多个线程放入 ACE_Task 的消息队列，或是从中取出，程序员无需担心破坏任何数据结构。因而任务可被用作由多个协作线程组成的系统的基础构建组件。各个线程控制都可封装在 ACE_Task 中，与其他任务通过发送消息到它们的消息队列来进行交互。

这种体系结构的唯一问题是，任务只能通过消息队列与在同一进程内的其他任务相互通信。ACE_Svc_Handler 解决了这一问题，它同时继承自 ACE_Task 和 ACE_Event_Handler，并且增加了一个私有数据流。这种结合使得 ACE_Svc_Handler 对象能够用作这样的任务：它能够处理事件、并与远地主机的任务间发送和接收数据。

ACE_Task 被实现为模板容器，它通过锁定机制来进行实例化。该锁用于保证内部的消息队列在多线程环境中的完整性。如先前所提到的，ACE_Svc_Handler 模板容器不仅需要锁定机制，还需要用于与远地任务通信的底层数据流来作为参数。

7.3.1.3 创建 ACE_Svc_Handler

ACE_Svc_Handler 模板通过锁定机制和底层流来实例化，以创建所需的服务处理器。如果应用只是单线程的，就不需要使用锁，可以用 ACE_NULL_SYNCH 来将其实例化。但是，如果我们想要在多线程应用中使用这个模板，可以这样来进行实例化：

```
class MySvcHandler:
    public ACE_Svc_Handler<ACE_SOCK_STREAM,ACE_MT_SYNCH>
{
    ...
}
```

7.3.1.4 在服务处理器中创建多个线程

在上面的例 7-5 中，我们使用 ACE_Thread 包装类和它的静态方法 spawn()，创建了单独的线程来发送数据给远地对端。但是，在我们完成此工作时，我们必须定义使用 C++ static 修饰符的文件范围内的静态 send_data()方法。结果当然就是，我们无法访问我们实例化的实际对象的任何数据成员。换句话说，我们被迫使 send_data()成员函数成为 class-wide 的函数，而这并**不是**我们所想要的。这样做的唯一原因是，ACE_Thread::spawn()只能使用静态成员函数来作为它所创建的线程的入口。另一个有害的副作用是到对端流的引用也必须成为静态的。简而言之，这不是编写这些代码的最好方式。

ACE_Task 提供了更好的机制来避免发生这样的问题。每个 ACE_Task 都有 activate()方法，可用于为 ACE_Task 创建线程。所创建的线程的入口是非静态成员函数 svc()。因为 svc()是非静态函数，它可以调用任何对象实例专有的数据或成员函数。ACE 对程序员隐藏了该机制的所有实现细节。activate()方法有着非常多的用途，它允许程序员创建多个线程，所有这些线程都使用 svc()方法作为它们的入口。还可以设置线程优先级、句柄、名字，等等。activate()方法的原型是：

```
// = Active object activation method.
virtual int activate (long flags = THR_NEW_LWP,
                      int n_threads = 1,
                      int force_active = 0,
                      long priority = ACE_DEFAULT_THREAD_PRIORITY,
                      int grp_id = -1,
                      ACE_Task_Base *task = 0,
                      ACE_hthread_t thread_handles[] = 0,
                      void *stack[] = 0,
                      size_t stack_size[] = 0,
                      ACE_thread_t thread_names[] = 0);
```

第一个参数 flags 描述将要创建的线程所希望具有的属性。在线程一章里有详细描述。可用的标志有：

```
THR_CANCEL_DISABLE, THR_CANCEL_ENABLE, THR_CANCEL_DEFERRED,
THR_CANCEL_ASYNCHRONOUS, THR_BOUND, THR_NEW_LWP, THR_DETACHED,
THR_SUSPENDED, THR_DAEMON, THR_JOINABLE, THR_SCHED_FIFO,
THR_SCHED_RR, THR_SCHED_DEFAULT
```

第二个参数 n_threads 指定要创建的线程的数目。第三个参数 force_active 用于指定是否应该创建新线程，即使 activate()方法已在先前被调用过、因而任务或服务处理器已经在运行多个线程。如果此参数被设为 false(0)，且如果 activate()是再次被调用，该方法就会设置失败代码，而不会生成更多的线程。

第四个参数用于设置运行线程的优先级。缺省情况下，或优先级被设为 ACE_DEFAULT_THREAD_PRIORITY，方法会使用给定的调度策略（在 flags 中指定，例如，THR_SCHED_DEFAULT）的“适当”优先级。这个值是动态计算的，并且是在给定策略的最低和最高优先级之间。如果显式地给定一个值，这个值就会被使用。注意实际的优先级值**极大地**依赖于实现，最好不要直接使用。在线程一章中，可读到更多有关线程优先级的内容。

还可以传入将要创建的线程的线程句柄、线程名和栈空间，以在线程创建过程中使用。如果它们被设置为 NULL，它们就不会被使用。但是如果要使用 activate 创建多个线程，就必须传入线程的名字或句柄，才能有效地对它们进行使用。

下面的例子可以帮助你进一步理解 activate 方法的使用：

例 7-6

```
#include ”ace/Reactor.h”
#include ”ace/Svc_Handler.h”
#include ”ace/Acceptor.h”
#include ”ace/Synch.h”
#include ”ace/SOCK_Acceptor.h”

class MyServiceHandler; //forward declaration
typedef ACE_Singleton<ACE_Reactor,ACE_Null_Mutex> Reactor;
typedef ACE_Acceptor<MyServiceHandler,ACE_SOCK_ACCEPTOR> Acceptor;

class MyServiceHandler:
public ACE_Svc_Handler<ACE_SOCK_STREAM,ACE_MT_SYNCH>
{
    // The two thread names are kept here
    ACE_thread_t thread_names[2];

public:
    int open(void*)
    {
        ACE_DEBUG((LM_DEBUG, ”Acceptor: received new connection \n”));

        //Register with the reactor to remember this handler..
        Reactor::instance()
            ->register_handler(this,ACE_Event_Handler::READ_MASK);
        ACE_DEBUG((LM_DEBUG,”Acceptor: ThreadID:(%t) open\n”));

        //Create two new threads to create and send messages to the
        //remote machine.
        activate(THR_NEW_LWP,
                 2, //2 new threads
                 0, //force active false, if already created don’t try again.
                 ACE_DEFAULT_THREAD_PRIORITY,//Use default thread priority
                 -1,
                 this,//Which ACE_Task object to create? In this case this one.
                 0,// don’t care about thread handles used
                 0,// don’t care about where stacks are created
                 0,//don’t care about stack sizes
                 thread_names); // keep identifiers in thread_names

        //keep the service handler registered with the acceptor.
        return 0;
```

```
}

void send_message1(void)
{
     //Send message type 1
     ACE_DEBUG((LM_DEBUG,”(%t)Sending message::>>”));

     //Send the data to the remote peer
     ACE_DEBUG((LM_DEBUG,”Sent message1”));
          peer().send_n(”Message1”,LENGTH_MSG_1);
} //end send_message1

int send_message2(void)
{
     //Send message type 1
     ACE_DEBUG((LM_DEBUG,”(%t)Sending message::>>”));

     //Send the data to the remote peer
     ACE_DEBUG((LM_DEBUG,”Sent Message2”));
          peer().send_n(”Message2”,LENGTH_MSG_2);
}//end send_message_2

int svc(void)
{
     ACE_DEBUG( (LM_DEBUG,”(%t) Svc thread \n”));
     if(ACE_Thread::self()== thread_names[0])
          while(1) send_message1(); //send message 1s forever
     else
          while(1) send_message2(); //send message 2s forever

     return 0; // keep the compiler happy.
}

int handle_input(ACE_HANDLE)
{
     ACE_DEBUG((LM_DEBUG,”(%t) handle_input ::”));
     char* data= new char[13];

     //Check if peer aborted the connection
     if(peer().recv_n(data,12)==0)
     {
          printf(”Peer probably aborted connection”);
          return -1; //de-register from the Reactor.
     }
```

```
        //Show what you got..
        ACE_OS::printf("<< %s\n",data);

        //keep yourself registered
        return 0;
    }
};

int main(int argc, char* argv[])
{
    ACE_INET_Addr addr(10101);
    ACE_DEBUG((LM_DEBUG,"Thread: (%t) main"));

    //Prepare to accept connections
    Acceptor myacceptor(addr,Reactor::instance());

    // wait for something to happen.
    while(1)
        Reactor::instance()->handle_events();

    return 0;
}
```

在此例中，服务处理器在它的 open()方法中被登记到反应器上，随后程序调用 activate()来创建 2 个线程。线程的名字被记录下来，以便在它们调用 svc()例程时，我们可以将它们区别开。每个线程发送一条不同类型的消息给远地对端。注意在此例中，线程的创建是完全透明的。此外，因为入口是普通的非静态成员函数，它不需要进行“丑陋的”改动来记住数据成员，比如说对端流。无论何时只要我们需要，我们就可以简单地调用成员函数 peer()来获取底层的流。

7.3.1.5 使用服务处理器中的消息队列机制

如前面所提到的，ACE_Svc_Handler 类拥有内建的消息队列。这个消息队列被用作在 ACE_Svc_Handler 和外部世界之间的主要通信接口。其他任务想要发给该服务处理器的消息被放入它的消息队列中。这些消息会在单独的线程里（通过调用 activate()方法创建）处理。随后另一个线程就可以把处理过的消息通过网络发送给另外的远地目的地（很可能是另外的 ACE_Svc_Handler）。

如先前所提到的，在这种多线程情况下，ACE_Svc_Handler 会自动地使用锁来确保消息队列的完整性。所用的锁即通过实例化 ACE_Svc_Handler 模板类创建具体服务处理器时所传递的锁。之所用通过这样的方式来传递锁，是因为这样程序员就可以对他的应用进行“调谐”。不同平台上的不同锁定机制有着不同程度的开销。如果需要，程序员可以创建他自己的优化的、遵从 ACE 的锁接口定义的锁，并将其用于服务处理器。这是程序员通过使用 ACE 可获得的灵活性的又一范例。重要的是程序员**必须**意识到，在此服务处理例程中的额外线程将带来显著的锁定开销。为将此开销降至最低，程序员必须仔细地设计他的程序，确保使这样的开销最小化。特别地，上面描述的例子有可能导致过度的开销，在大多数情况下

可能并不实用。

ACE_Task，进而是 ACE_Svc_Handler（因为服务处理器也是一种任务），具有若干可用于对底层队列进行设置、操作、入队和出队操作的方法。这里我们将只讨论这些方法中的一部分。因为在服务处理器中（通过使用 msg_queue()方法）可以获取指向消息队列自身的指针，所以也可以直接调用底层队列（也就是，ACE_Message_Queue）的所有公共方法。（有关消息队列提供的所有方法的更多细节，请参见后面的“ 消息队列 ”一章。）

如上面所提到的，服务处理器的底层消息队列是 ACE_Message_Queue 的实例，它是由服务处理器自动创建的。在大多数情况下，没有必要调用 ACE_Message_Queue 的底层方法，因为在 ACE_Svc_Handler 类中已对它们的大多数进行了包装。ACE_Message_Queue 是用于使 ACE_Message_Block 进队或出队的队列。每个 ACE_Message_Block 都含有指向“ 引用计数 ”（reference-counted）的 ACE_Data_Block 的指针，ACE_Data_Block 依次又指向存储在块中的实际数据（见“ 消息队列 ”一章）。这使得 ACE_Message_Block 可以很容易地进行数据共享。

ACE_Message_Block 的主要作用是进行高效数据操作，而不带来许多拷贝开销。每个消息块都有一个读指针和写指针。无论何时我们从块中读取时，读指针会在数据块中向前增长。类似地，当我们向块中写的时候，写指针也会向前移动，这很像在流类型系统中的情况。可以通过 ACE_Message_Block 的构造器向它传递分配器，以用于分配内存（有关 Allocator 的更多信息，参见“ 内存管理 ”一章）。例如，可以使用 ACE_Cached_Allocation_Strategy，它预先分配内存并从内存池中返回指针，而不是在需要的时候才从堆中分配内存。这样的功能在需要可预测的性能时十分有用，比如在实时系统中。

下面的例子演示怎样使用消息队列的一些功能：

例 7-7

```
#include ″ace/Reactor.h″
#include ″ace/Svc_Handler.h″
#include ″ace/Acceptor.h″
#include ″ace/Synch.h″
#include ″ace/SOCK_Acceptor.h″
#include ″ace/Thread.h″
#define NETWORK_SPEED 3
class MyServiceHandler; //forward declaration
typedef ACE_Singleton<ACE_Reactor,ACE_Null_Mutex> Reactor;
typedef ACE_Acceptor<MyServiceHandler,ACE_SOCK_ACCEPTOR> Acceptor;

class MyServiceHandler:
public ACE_Svc_Handler<ACE_SOCK_STREAM,ACE_MT_SYNCH>{
    // The message sender and creator threads are handled here.
    ACE_thread_t thread_names[2];

public:
    int open(void*)
    {
        ACE_DEBUG((LM_DEBUG, ″Acceptor: received new connection \n″));

        //Register with the reactor to remember this handler..
        Reactor::instance()
```

```
        ->register_handler(this,ACE_Event_Handler::READ_MASK);
    ACE_DEBUG((LM_DEBUG,"Acceptor: ThreadID:(%t) open\n"));

    //Create two new threads to create and send messages to the
    //remote machine.
    activate(THR_NEW_LWP,
             2, //2 new threads
             0,
             ACE_DEFAULT_THREAD_PRIORITY,
             -1,
             this,
             0,
             0,
             0,
             thread_names); // identifiers in thread_handles

    //keep the service handler registered with the acceptor.
    return 0;
}

void send_message(void)
{
    //Dequeue the message and send it off
    ACE_DEBUG((LM_DEBUG,"(%t)Sending message::>>"));

    //dequeue the message from the message queue
    ACE_Message_Block *mb;
    ACE_ASSERT(this->getq(mb)!=-1);
    int length=mb->length();
    char *data =mb->rd_ptr();

    //Send the data to the remote peer
    ACE_DEBUG((LM_DEBUG,"%s \n",data,length));
        peer().send_n(data,length);

    //Simulate very SLOW network.
    ACE_OS::sleep(NETWORK_SPEED);

    //release the message block
    mb->release();
} //end send_message

int construct_message(void)
{
```

```
    // A very fast message creation algorithm
    // would lead to the need for queuing messages..
    // here. These messages are created and then sent
    // using the SLOW send_message() routine which is
    // running in a different thread so that the message
    //construction thread isn't blocked.
    ACE_DEBUG((LM_DEBUG,"(%t)Constructing message::>> "));

    // Create a new message to send
    ACE_Message_Block *mb;
    char *data="Hello Connector";
    ACE_NEW_RETURN (mb,ACE_Message_Block (16,//Message 16 bytes long
                ACE_Message_Block::MB_DATA,//Set header to data
                0,//No continuations.
                data//The data we want to send
                ), 0);
    mb->wr_ptr(16); //Set the write pointer.

    // Enqueue the message into the message queue
    // we COULD have done a timed wait for enqueuing in case
    // someone else holds the lock to the queue so it doesn't block
    //forever..
    ACE_ASSERT(this->putq(mb)!=-1);
    ACE_DEBUG((LM_DEBUG,"Enqueued msg successfully\n"));
}

int svc(void)
{
    ACE_DEBUG( (LM_DEBUG,"(%t) Svc thread \n"));

    //call the message creator thread
    if(ACE_Thread::self()== thread_names[0])
        while(1) construct_message(); //create messages forever
    else
        while(1) send_message(); //send messages forever

    return 0; // keep the compiler happy.
}

int handle_input(ACE_HANDLE)
{
    ACE_DEBUG((LM_DEBUG,"(%t) handle_input ::"));
    char* data= new char[13];
```

```
        //Check if peer aborted the connection
        if(peer().recv_n(data,12)==0)
        {
             printf(”Peer probably aborted connection”);
             return -1; //de-register from the Reactor.
        }

        //Show what you got..
        ACE_OS::printf(”<< %s\n”,data);

        //keep yourself registered
        return 0;
    }
};

int main(int argc, char* argv[])
{
    ACE_INET_Addr addr(10101);
    ACE_DEBUG((LM_DEBUG,”Thread: (%t) main”));

    //Prepare to accept connections
    Acceptor myacceptor(addr,Reactor::instance());

    // wait for something to happen.
    while(1)
        Reactor::instance()->handle_events();

    return 0;
}
```

这个例子演示怎样使用 putq()和 getq()方法来在队列中放入或取出消息块。它还演示怎样创建消息块，随后设置它的写指针，并根据它的读指针进行读取。注意消息块中的实际数据的起始位置由消息块的读指针指示。消息块的 length()成员函数返回在消息块中存储的底层数据的长度，其中不包括 ACE_Message_Block 中用于管理目的的部分。另外，我们也显示了怎样使用 release()方法来释放消息块（mb）。

要了解更多关于如何使用消息块、数据块或是消息队列的信息，请阅读此教程中有关“消息队列”、ASX 框架和其他相关的部分。

7.4 接受器和连接器模式工作原理

接受器和连接器工厂（也就是 ACE_Connector 和 ACE_Acceptor）有着非常类似的运行结构。它们的工作可大致划分为三个阶段：

- 端点或连接初始化阶段

- 服务初始化阶段
- 服务处理阶段

7.4.1 端点或连接初始化阶段

在使用接受器的情况下，应用级程序员可以调用 ACE_Acceptor 工厂的 open()方法，或是它的缺省构造器（它实际上**会**调用 open()方法），来开始被动侦听连接。当接受器工厂的 open()方法被调用时，如果反应器单体还没有被实例化，open()方法就首先对其进行实例化。随后它调用底层具体接受器的 open()方法。于是具体接受器会完成必要的初始化来侦听连接。例如，在使用 ACE_SOCK_Acceptor 的情况中，它打开 socket，将其绑定到用户想要在其上侦听新连接的端口和地址上。在绑定端口后，它将会发出侦听调用。open 方法随后将接受器工厂登记到反应器。因而在接收到任何到来的连接请求时，反应器会自动回调接受器工厂的 handle_input()方法。注意正是因为这一原因，接受器工厂才从 ACE_Event_Handler 类层次派生；这样它才可以响应 ACCEPT 事件，并被反应器自动回调。

在使用连接器的情况中，应用程序员调用连接器工厂的 connect()方法或 connect_n()方法来发起到对端的连接。除了其他一些选项，这两个方法的参数包括我们想要连接到的远地地址，以及我们是想要同步还是异步地完成连接。我们可以同步或异步地发起 NUMBER_CONN 个连接：

```
//Synchronous
OurConnector.connect_n(NUMBER_CONN,ArrayofMySvcHandlers,Remote_Addr,0,
                       ACE_Synch_Options::synch);

//Asynchronous
OurConnector.connect_n(NUMBER_CONN,ArrayofMySvcHandlers,Remote_Addr,0,
                       ACE_Synch_Options::asynch);
```

如果连接请求是异步的，ACE_Connector 会在反应器上登记自己，等待连接被建立（ACE_Connector 也派生自 ACE_Event_Handler 类层次）。一旦连接被建立，反应器将随即自动回调连接器。但如果连接请求是同步的，connect()调用将会阻塞，直到连接被建立、或是超时到期为止。超时值可通过改变特定的 ACE_Synch_Options 来指定。详情请参见参考手册。

7.4.2 接受器的服务初始化阶段

在有连接请求在指定的地址和端口上到来时，反应器自动回调 ACE_Acceptor 工厂的 handle_input()方法。

该方法是一个“**模板方法**”（Template Method）。模板方法用于定义一个算法的若干步骤的顺序，并允许改变特定步骤的执行。这种变动是通过允许子类定义这些方法的实现来完成的。（有关模板方法的更多信息见“设计模式”参考指南）。

在我们的这个案例中，模板方法将算法定义如下：

- make_svc_handler()：创建服务处理器。
- accept_svc_handler()：将连接接受进前一步骤创建的服务处理器。

- activate_svc_handler()：启动这个新服务处理器。

这些方法都可以被重新编写，从而灵活地决定这些操作怎样来实际执行。

这样，handle_input()将首先调用 make_svc_handler()方法，创建适当类型的服务处理器（如我们在上面的例子中所看到的那样，服务处理器的类型由应用程序员在 ACE_Acceptor 模板被实例化时传入）。在缺省情况下，make_svc_handler()方法只是实例化恰当的服务处理器。但是，make_svc_handler()是一个“ 桥接 ”（bridge）方法，可被重载以提供更多复杂功能。（桥接是一种设计模式，它使类层次的接口与实现去耦合。参阅 “ 设计模式 ” 参考文献）。例如，服务处理器可创建为进程级或线程级的单体，或者从库中动态链接，从磁盘中加载，甚或通过更复杂的方式创建，如从数据库中查找并获取服务处理器，并将它装入内存。

在服务处理器被创建后，handle_input()方法调用 accept_svc_handler()。该方法将连接 “ 接受进 ” 服务处理器；缺省方式是调用底层具体接受器的 accept()方法。在 ACE_SOCK_Acceptor 被用作具体接受器的情况下，它调用 BSD accept()例程来建立连接（“ 接受 ” 连接）。在连接建立后，连接句柄在服务处理器中被自动设置（接受 “ 进 ” 服务处理器）；这个服务处理器是先前通过调用 make_svc_handler()创建的。该方法也可被重载，以提供更复杂的功能。例如，不是实际创建新连接，而是 “ 回收利用 ” 旧连接。在我们演示各种不同的接受和连接策略时，将更为详尽地讨论这一点。

7.4.3 连接器的服务初始化阶段

应用发出的 connect()方法与接受器工厂中的 handle_input()相类似，也就是，它是一个 “ 模板方法 ”。

在我们的这个案例中，模板方法 connect()定义下面一些可被重定义的步骤：

- make_svc_handler()：创建服务处理器。
- connect_svc_handler()：将连接接受进前一步骤创建的服务处理器。
- activate_svc_handler()：启动这个新服务处理器。

每一方法都可以被重新编写，从而灵活地决定这些操作怎样来实际执行。

这样，在应用发出 connect()调用后，连接器工厂通过调用 make_svc_handler()来实例化恰当的服务处理器，一如在接受器的案例中所做的那样。其缺省行为只是实例化适当的类，并且也可以通过与接受器完全相同的方式重载。进行这样的重载的原因可以与上面提到的原因非常类似。

在服务处理器被创建后，connect()调用确定连接是要成为异步的还是同步的。如果是异步的，在继续下一步骤之前，它将自己登记到反应器，随后调用 connect_svc_handler()方法。该方法的缺省行为是调用底层具体连接器的 connect()方法。在使用 ACE_SOCK_Connector 的情况下，这意味着将适当的选项设置为阻塞或非阻塞式 I/O，然后发出 BSD connect()调用。如果连接被指定为同步的，connect()调用将会阻塞、直到连接完全建立。在这种情况下，在连接建立后，它将在服务处理器中设置句柄，以与它现在连接到的对端通信（该句柄即是通过在服务处理器中调用 peer()方法获得的在流中存储的句柄，见上面的例子）。在服务处理器中设置句柄后，连接器模式将进行到最后阶段：服务处理。

如果连接被指定为异步的，在向底层的具体连接器发出非阻塞式 connect()调用后，对 connect_svc_handler()的调用将立即返回。在使用 ACE_SOCK_Connector 的情况中，这意味着发出非阻塞式 BSD connect()调用。在连接稍后被实际建立时，反应器将回调 ACE_Connector 工厂的 handle_output()方法，该方法在通过 make_svc_handler()方法创建的服务处理器中设置新句柄。然后工厂将进行到下一阶段：服务处理。

与 accept_svc_handler()情况一样，connect_svc_handler()是一个“桥接”方法，可进行重载以提供变化的功能。

7.4.4 服务处理

一旦服务处理器被创建、连接被建立，以及句柄在服务处理器中被设置，ACE_Acceptor 的 handle_input()方法（或者在使用 ACE_Connector 的情况下，是 handle_output()或 connect_svc_handler()）将调用 activate_svc_handler()方法。该方法将随即启用服务处理器。其缺省行为是调用作为服务处理器的入口的 open()方法。如我们在上面的例子中所看到的，在服务处理器开始执行时，open()方法是第一个被调用的方法。是在 open()方法中，我们调用 activate()方法来创建多个线程控制；并在反应器上登记服务处理器，这样当新的数据在连接上到达时，它会被自动回调。该方法也是一个“桥接”方法，可被重载以提供更为复杂的功能。特别地，这个重载的方法可以提供更为复杂的并发策略，比如，在另一不同的进程中运行服务处理器。

7.5 调谐接受器和连接器策略

如上面所提到的，因为使用了可以重载的桥接方法，很容易对接受器和连接器进行调谐。桥接方法允许调谐：

- **服务处理器的创建策略：**通过重载接受器或连接器的 make_svc_handler()方法来实现。例如，这可以意味着复用已有的服务处理器，或使用某种复杂的方法来获取服务处理器，如上面所讨论的那样。
- **连接策略：**连接创建策略可通过重载 connect_svc_handler()或 accept_svc_handler()方法来改变。
- **服务处理器的并发策略：**服务处理器的并发策略可通过重载 activate_svc_handler()方法来改变。例如，服务处理器可以在另外的进程中创建。

如上所示，调谐是通过重载 ACE_Acceptor 或 ACE_Connector 类的桥接方法来完成的。ACE 的设计使得程序员很容易完成这样的重载和调谐。

7.5.1 ACE_Strategy_Connector 和 ACE_Strategy_Acceptor 类

为了方便上面所提到的对接受器和连接器模式的调谐方法，ACE 提供了两种特殊的“可调谐”接受器和连接器工厂，那就是 ACE_Strategy_Acceptor 和 ACE_Strategy_Connector。它们和 ACE_Acceptor 与 ACE_Connector 非常类似，同时还使用了“策略”模式。

策略模式被用于使算法行为与类的接口去耦合。其基本概念是允许一个类（称为 Context Class，上下文类）的底层算法独立于使用该类的客户进行变动。这是通过具体策略类的帮助来完成的。具体策略类封装执行操作的算法或方法。这些具体策略类随后被上下文类用于执行各种操作（上下文类将“工作”*委托*给具体策略类）。因为上下文类不直接执行任何操作，当需要改变功能时，无需对它进行修改。对上下文类所做的唯一修改是使用另一个具体策略类来执行改变了的操作。（要阅读有关策略模式的更多信息，参见“设计模式”的附录）。

在 ACE 中，ACE_Strategy_Connector 和 ACE_Strategy_Acceptor 使用若干具体策略类来改变算法，以创建服务处理器，建立连接，以及为服务处理器设置并发方法。如你可能已经猜到的一样，ACE_Strategy_Connector 和 ACE_Strategy_Acceptor 利用了上面提到的桥接方法所提供的可调谐性。

7.5.1.1 使用策略接受器和连接器

在 ACE 中已有若干具体的策略类可用于“调谐”策略接受器和连接器。当类被实例化时，它们作为参数被传入策略接受器或连接器。表 7-2 显示了可用于调谐策略接受器和连接器类的一些类。

需要修改	具体策略类	描述
创建策略 (重定义 make_svc_handler())	ACE_NOOP_Creation_Strategy	这个具体策略并**不**实例化服务处理器，而只是一个空操作。
	ACE_Singleton_Strategy	保证服务处理器被创建为单体。也就是，所有连接将有效地使用同一个服务处理例程。
	ACE_DLL_Strategy	通过从动态链接库中动态链接服务处理器来对它进行创建。
连接策略 (重定义 connect_svc_handler())	ACE_Cached_Connect_Strategy	检查是否有已经连接到特定的远地地址的服务处理器没有在被使用。如果有这样一个服务处理器，就对它进行复用。
并发策略 (重定义 activate_svc_handler())	ACE_NOOP_Concurrency_Strategy	一个“无为”（do-nothing）的并发策略。它甚至**不**调用服务处理器的 open()方法。
	ACE_Process_Strategy	在另外的进程中创建服务处理器，并调用它的 open()方法。
	ACE_Reactive_Strategy	先在反应器上登记服务处理器，然后调用它的 open()方法。
	ACE_Thread_Strategy	先调用服务处理器的 open()方法，然后调用它的 activate()方法，以让另外的线程来启动服务处理器的 svc()方法。

表 7-2 用于调谐策略接受器和连接器类的类

下面的例子演示策略接受器和连接器类的使用。

例 7-8

```
#include ″ace/Reactor.h″
#include ″ace/Svc_Handler.h″
#include ″ace/Acceptor.h″
#include ″ace/Synch.h″
```

```
#include ”ace/SOCK_Acceptor.h”
#define PORT_NUM 10101
#define DATA_SIZE 12
//forward declaration
class My_Svc_Handler;
//instantiate a strategy acceptor
typedef ACE_Strategy_Acceptor<My_Svc_Handler,ACE_SOCK_ACCEPTOR> MyAcceptor;
//instantiate a concurrency strategy
typedef ACE_Process_Strategy<My_Svc_Handler> Concurrency_Strategy;

// Define the Service Handler
class My_Svc_Handler:
public ACE_Svc_Handler <ACE_SOCK_STREAM,ACE_NULL_SYNCH>
{
private:
    char* data;
public:
    My_Svc_Handler()
    {
        data= new char[DATA_SIZE];
    }

    My_Svc_Handler(ACE_Thread_Manager* tm)
    {
        data= new char[DATA_SIZE];
    }

    int open(void*)
    {
        cout<<”Connection established”<<endl;

        //Register with the reactor
        ACE_Event_Handler::READ_MASK);

        return 0;
    }

    int handle_input(ACE_HANDLE)
    {
        peer().recv_n(data,DATA_SIZE);
        ACE_OS::printf(”<< %s\n”,data);

        // keep yourself registered with the reactor
        return 0;
```

```
    }
};

int main(int argc, char* argv[])
{
    ACE_INET_Addr addr(PORT_NUM);

    //Concurrency Strategy
    Concurrency_Strategy my_con_strat;

    //Instantiate the acceptor
    MyAcceptor acceptor(addr,      //address to accept on
        ACE_Reactor::instance(),  //the reactor to use
        0,                        // don't care about creation strategy
        0,                        // don't care about connection estb. strategy
        &my_con_strat);           // use our new process concurrency strategy

    while(1) /* Start the reactor's event loop */
        ACE_Reactor::instance()->handle_events();
}
```

这个例子基于上面的例 7-2。唯一的不同是它使用了 ACE_Strategy_Acceptor，而不是使用 ACE_Acceptor；并且它还使用 ACE_Process_Strategy 作为服务处理器的并发策略。这种并发策略保证一旦连接建立后，服务处理器在单独的进程中被实例化。如果在特定服务上的负载变得过于繁重，使用 ACE_Process_Strategy 可能是一个好主意。但是，在大多数情况下，使用 ACE_Process_Strategy 会过于昂贵，而 ACE_Thread_Strategy 可能是更好的选择。

7.5.1.2 使用 ACE_Cached_Connect_Strategy 进行连接缓存

在许多应用中，客户会连接到服务器，然后重新连接到同一服务器若干次；每次都要建立连接，执行某些工作，然后挂断连接（比如像在 Web 客户中所做的那样）。不用说，这样做是非常低效而昂贵的，因为连接建立和挂断是非常昂贵的操作。在这样的情况下，连接者可以采用一种更好的策略：“记住”老连接并保持它，直到确定客户不会再重新建立连接为止。ACE_Cached_Connect_Strategy 就提供了这样一种缓存策略。这个策略对象被 ACE_Strategy_Connector 用于提供基于缓存的连接建立。如果一个连接已经存在，ACE_Strategy_Connector 将会复用它，而不是创建新的连接。

当客户试图重新建立连接到先前已经连接的服务器时，ACE_Cached_Connect_Strategy 确保对老的连接和服务处理器进行复用，而不是创建新的连接和服务处理器。因而，实际上，ACE_Cached_Connect_Strategy 不仅管理连接建立策略，它还管理服务处理器创建策略。因为在此例中，用户**不想**创建新的服务处理器，我们将 ACE_Null_Creation_Strategy 传递给 ACE_Strategy_Connector。如果连接先前没有建立过，ACE_Cached_Connect_Strategy 将自动使用内部的创建策略来实例化适当的服务处理器，它是在这个模板类被实例化时传入的。这个策略可被设置为用户想要使用的任何一种策略。除此而外，也可以将 ACE_Cached_Connect_Strategy 自己在其构造器中使用的创建、并发和 recycling 策略

传给它。下面的例子演示这些概念：

例 7-9

```
#include ”ace/Reactor.h”
#include ”ace/Svc_Handler.h”
#include ”ace/Connector.h”
#include ”ace/Synch.h”
#include ”ace/SOCK_Connector.h”
#include ”ace/INET_Addr.h”

#define PORT_NUM 10101
#define DATA_SIZE 16

//forward declaration
class My_Svc_Handler;
//Function prototype
static void make_connections(void *arg);

// Template specializations for the hashing function for the
// hash_map which is used by the cache. The cache is used internally by the
// Cached Connection Strategy . Here we use ACE_Hash_Addr
// as our external identifier. This utility class has already
// overloaded the == operator and the hash() method. (The
// hashing function). The hash() method delegates the work to
// hash_i() and we use the IP address and port to get a
// a unique integer hash value.
size_t ACE_Hash_Addr<ACE_INET_Addr>::hash_i (const ACE_INET_Addr &addr) const
{
    return addr.get_ip_address () + addr.get_port_number ();
}

//instantiate a strategy acceptor
typedef ACE_Strategy_Connector<My_Svc_Handler,ACE_SOCK_CONNECTOR>
STRATEGY_CONNECTOR;

//Instantiate the Creation Strategy
typedef ACE_NOOP_Creation_Strategy<My_Svc_Handler>
    NULL_CREATION_STRATEGY;
//Instantiate the Concurrency Strategy
typedef ACE_NOOP_Concurrency_Strategy<My_Svc_Handler>
    NULL_CONCURRENCY_STRATEGY;
//Instantiate the Connection Strategy
typedef ACE_Cached_Connect_Strategy<My_Svc_Handler,
                                    ACE_SOCK_CONNECTOR,
```

```
                                    ACE_SYNCH_RW_MUTEX>
    CACHED_CONNECT_STRATEGY;

class My_Svc_Handler:
public ACE_Svc_Handler <ACE_SOCK_STREAM,ACE_MT_SYNCH>
{
private:
    char* data;
public:
    My_Svc_Handler()
    {
        data= new char[DATA_SIZE];
    }

    My_Svc_Handler(ACE_Thread_Manager* tm)
    {
        data= new char[DATA_SIZE];
    }

    //Called before the service handler is recycled..
    int  recycle (void *a=0)
    {
        ACE_DEBUG ((LM_DEBUG,
            "(%P|%t) recycling Svc_Handler %d with handle %d\n",
            this, this->peer ().get_handle ()));
        return 0;
    }

    int open(void*)
    {
        ACE_DEBUG((LM_DEBUG,"(%t)Connection established \n"));

        //Register the service handler with the reactor
        ACE_Reactor::instance()
            ->register_handler(this,ACE_Event_Handler::READ_MASK);

        activate(THR_NEW_LWP|THR_DETACHED);
        return 0;
    }

    int handle_input(ACE_HANDLE)
    {
        ACE_DEBUG((LM_DEBUG,"Got input in thread: (%t) \n"));
            peer().recv_n(data,DATA_SIZE);
```

```
        ACE_DEBUG((LM_DEBUG,"<< %s\n",data));

        //keep yourself registered with the reactor
        return 0;
    }

    int svc(void)
    {
        //send a few messages and then mark connection as idle so that it can
        // be recycled  later.
        ACE_DEBUG((LM_DEBUG,"Started the service routine \n"));
        for(int i=0;i<3;i++)
        {
            ACE_DEBUG((LM_DEBUG,"(%t)>>Hello World\n"));
            ACE_OS::fflush(stdout);
            peer().send_n("Hello World",sizeof("Hello World"));
        }

        //Mark the service handler as being idle now and let the
        //other threads reuse this connection
        this->idle(1);

        //Wait for the thread to die
        this->thr_mgr()->wait();

        return 0;
    }
};

ACE_INET_Addr *addr;

int main(int argc, char* argv[])
{
    addr= new ACE_INET_Addr(PORT_NUM,argv[1]);

    //Creation Strategy
    NULL_CREATION_STRATEGY creation_strategy;

    //Concurrency Strategy
    NULL_CONCURRENCY_STRATEGY concurrency_strategy;

    //Connection Strategy
    CACHED_CONNECT_STRATEGY caching_connect_strategy;
```

```
    //instantiate the connector
    STRATEGY_CONNECTOR connector(
        ACE_Reactor::instance(), //the reactor to use
        &creation_strategy,
        &caching_connect_strategy,
        &concurrency_strategy);

    //Use the thread manager to spawn a single thread
    //to connect multiple times passing it the address
    //of the strategy connector
    if(ACE_Thread_Manager::instance()->spawn(
        (ACE_THR_FUNC) make_connections,
        (void *) &connector,
        THR_NEW_LWP) == -1)

    ACE_ERROR ((LM_ERROR, "(%P|%t) %p\n%a", "client thread spawn failed"));

    while(1) /* Start the reactor's event loop */
        ACE_Reactor::instance()->handle_events();
}

//Connection establishment function, tries to establish connections
//to the same server again and re-uses the connections from the cache
void make_connections(void *arg)
{
    ACE_DEBUG((LM_DEBUG,"(%t)Prepared to connect \n"));
    STRATEGY_CONNECTOR *connector= (STRATEGY_CONNECTOR*) arg;
    for (int i = 0; i < 10; i++)
    {
        My_Svc_Handler *svc_handler = 0;

        // Perform a blocking connect to the server using the Strategy
        // Connector with a connection caching strategy. Since we are
        // connecting to the same <server_addr> these calls will return the
        // same dynamically allocated <Svc_Handler> for each <connect> call.
        if (connector->connect (svc_handler, *addr) == -1)
        {
            ACE_ERROR ((LM_ERROR, "(%P|%t) %p\n", "connection failed\n"));
            return;
        }

        // Rest for a few seconds so that the connection has been freed up
        ACE_OS::sleep (5);
    }
```

}

在上面的例子中，缓存式连接策略被用于缓存连接。要使用这一策略，需要一点额外的工作：定义 ACE_Cached_Connect_Strategy 在内部使用的哈希映射管理器的 hash()方法。这个 hash()方法用于对服务处理器和 ACE_Cached_Connect_Strategy 内部使用的连接进行哈希运算，放入缓存映射中。它简单地使用 IP 地址和端口号的总和作为哈希函数，这也许并不是很好的哈希函数。

这个例子比至今为止我们所列举的例子都要复杂一点，所以有理由多进行一点讨论。

我们为 ACE_Strategy_Acceptor 使用空操作并发和创建策略。使用空操作创建策略**是**必须的。如上面所解释的，如果没有使用 ACE_NOOP_Creation_Strategy，ACE_Cached_Connection_Strategy 将会产生断言失败。但是，在使用 ACE_Cached_Connect_Strategy 时，任何并发策略都可以和策略接受器一起使用。如上面所提到的，ACE_Cached_Connect_Strategys 所用的底层创建策略可以由用户来设置。还可以设置 recycling 策略。这是在实例化 caching_connect_strategy 时，通过将所需的创建和 recycling 策略的对象传给它的构造器来完成的。在这里我们没有这样做，而是使用了缺省的创建和 recycling 策略。

在适当地设置连接器后，我们使用 Thread_Manager 来派生新线程，且将 make_connections()方法作为线程的入口。该方法使用我们的新的策略连接器来连接到远地站点。在连接建立后，该线程休眠 5 秒钟，然后使用我们的缓存式连接器来重新创建同样的连接。于是该线程应该在连接器缓存中找到这个连接并复用它。

和平常一样，一旦连接建立后，反应器回调我们的服务处理器（My_Svc_Handler）。随后 My_Svc_Handler 的 open()方法通过调用 My_Svc_Handler 的 activate()方法来使它成为主动对象。svc()方法随后发送三条消息给远地主机，并通过调用服务处理器的 idle()方法将该连接标记为空闲。注意 this->thr_mrg_wait()要求线程管理器等待所有在线程管理器中的线程终止。如果你不要求线程管理器等待其它线程，根据在 ACE 中设定的语义，一旦 ACE_Task（在此例中是 ACE_Task 类型的 ACE_Svc_Handler）中的线程终止，ACE_Task 对象（在此例中是 ACE_My_Svc_Handler）就会被自动删除。如果发生了这种情况，在 Cache_Connect_Strategy 查找先前缓存的连接时，它就不会如我们期望的那样找到 My_Svc_Handler，因为它已经被删除掉了。

在 My_Svc_Handler 中还重载了 ACE_Svc_Handler 中的 recycle()方法。当有旧连接被 ACE_Cache_Connect_Strategy 找到时，这个方法就会被自动回调，这样服务处理器就可以在此方法中完成回收利用所特有的操作。在我们的例子中，我们只是打印出在缓存中找到的处理器的 this 指针的地址。在程序运行时，每次连接建立后所使用的句柄的地址是相同的，从而说明缓存工作正常。

7.6 通过接受器和连接器模式使用简单事件处理器

有时，使用重量级的 ACE_Svc_Handler 作为接受器和连接器的处理器不仅没有必要，而且会导致代码臃肿。在这样情况下，用户可以使用较轻的 ACE_Event_Handler 方法来作为反应器在连接一旦建立时所回调的类。要采用这种方法，程序员需要重载 get_handle()方法，并包含将要用于事件处理器的具体底层流。下面的例子有助于演示这些变动。这里我们还编写了新的 peer()方法，它返回底层流的引用（reference），就像在 ACE_Svc_Handler 类中所做的那样。

例 7-10

```
#include ″ace/Reactor.h″
#include ″ace/Svc_Handler.h″
#include ″ace/Acceptor.h″
#include ″ace/Synch.h″
#include ″ace/SOCK_Acceptor.h″
```

```
#define PORT_NUM 10101
#define DATA_SIZE 12

//forward declaration
class My_Event_Handler;

//Create the Acceptor class
typedef ACE_Acceptor<My_Event_Handler,ACE_SOCK_ACCEPTOR>
MyAcceptor;

//Create an event handler similar to as seen in example 2. We have to
//overload the get_handle() method and write the peer() method. We also
//provide the data member peer_ as the underlying stream which is
//used.
class My_Event_Handler: public ACE_Event_Handler
{
private:
    char* data;

    //Add a new attribute for the underlying stream which will be used by
    //the Event Handler
    ACE_SOCK_Stream peer_;

public:
    My_Event_Handler()
    {
        data= new char[DATA_SIZE];
    }

    int open(void*)
    {
        cout<<”Connection established”<<endl;

        //Register the event handler with the reactor
        ACE_Reactor::instance()->register_handler(this,
        ACE_Event_Handler::READ_MASK);

        return 0;
    }

    int handle_input(ACE_HANDLE)
    {
        // After using the peer() method of our ACE_Event_Handler to obtain a
```

```
        //reference to the underlying stream of the service handler class we
        //call recv_n() on it to read the data which has been received. This
        //data is stored in the data array and then printed out
        peer().recv_n(data,DATA_SIZE);
        ACE_OS::printf("<< %s\n",data);

        // keep yourself registered with the reactor
        return 0;
    }

    // new method which returns the handle to the reactor when it
    //asks for it.
    ACE_HANDLE get_handle(void) const
    {
        return this->peer_.get_handle();
    }

    //new method which returns a reference to the peer stream
    ACE_SOCK_Stream &peer(void) const
    {
        return (ACE_SOCK_Stream &) this->peer_;
    }
};

int main(int argc, char* argv[])
{
    ACE_INET_Addr addr(PORT_NUM);

    //create the acceptor
    MyAcceptor acceptor(addr, //address to accept on
    ACE_Reactor::instance()); //the reactor to use

    while(1) /* Start the reactor's event loop */
        ACE_Reactor::instance()->handle_events();
}
```

第 8 章 服务配置器(Service Configurator)：用于服务动态配置的模式

许多分布式系统都含有一组*全局*服务。应用开发者可以调用这些服务来帮助他满足分布式开发的需求。在构造分布式应用时，需要像名字服务、远程终端访问服务、登录和时间时间服务这样的全局服务。构造这些服务的一种办法是将每个服务编写成单独的程序。随后这些服务程序就在它们自己的私有进程中执行和运行。但是，这样的方法会导致配置的噩梦。管理员需要管理每一个节点，依据当前的用户需求和策略来执行服务程序。如果需要增加新服务，或是需要移除旧服务，管理员就必须将时间花费在每台机器上、重新进行配置。此外，这样的配置是*静态的*。要进行重配置，管理员需要人工地终止服务（通过杀掉服务进程），随后重启一个替换服务。同样，在机器上运行的服务也可能不被*任何*应用所使用。显然，这样的方法是低效的和不合需要的。

如果服务可以被*动态地*启动、移除、挂起和恢复，那将会方便得多。这样，服务开发者就不必再担心配置的服务。他所需关心的是服务如何完成工作。管理员就可以在应用中增加或替换新服务，而*不用*重新编译或关闭服务进程。

服务配置器模式可以完成所有这些任务。它使服务的*实现*与*配置*去耦合。无需关闭服务器，就可以在应用中增加新服务和移除旧服务。在大多数情况下，提供服务的服务器都被实现为看守（daemon）进程。

8.1 框架组件

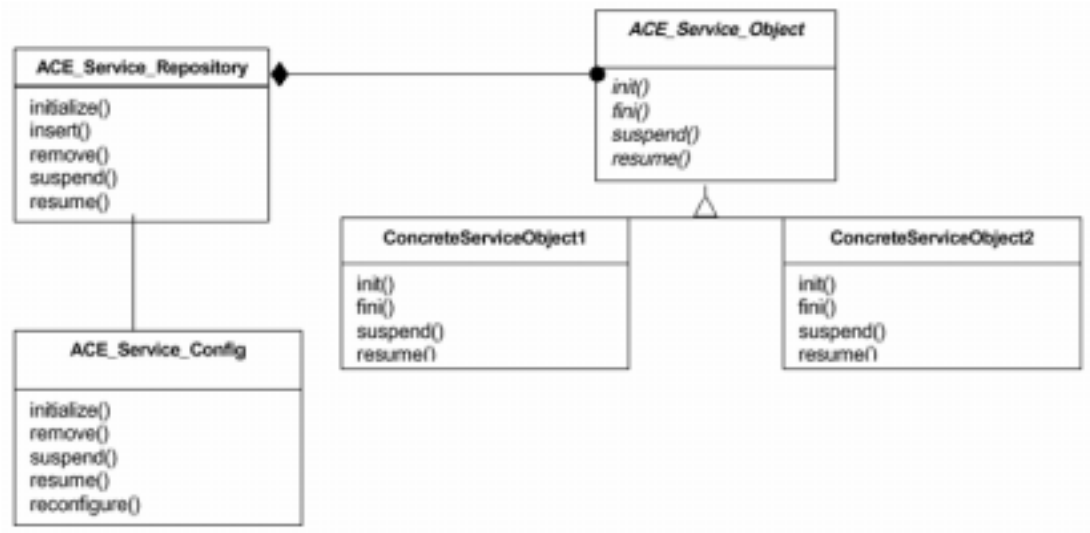

图 8-1 ACE 服务配置器中的组件及其关系

ACE 中的服务配置器由以下组件组成：

- 名为 ACE_Service_Object 的抽象类。应用开发者必须从它派生出子类，以创建他自己的应用特有的具体服务对象（Service Object）。
- 应用特有的具体服务对象。
- 服务仓库 ACE_Service_Repository。它记录服务器所运行的和所知道的服务。
- ACE_Service_Config。它是整个服务配置器框架的应用开发接口。
- 服务配置文件。该文件含有所有服务对象的配置信息。其缺省的名字是 svc.conf。当你的应用对 ACE_Service_Config 发出 open()调用时，服务配置器框架会读取并处理你写在此文件中的所有配置信息，随后相应地配置应用。

ACE_Service_Object 包括了一些由框架调用的方法，用于服务要启动（init()）、停止（fini()）、挂起（suspend()）或是恢复（resume()）时。ACE_Service_Object 派生自 ACE_Shared_Object 和 ACE_Event_Handler。ACE_Shared_Object 在应用想要使用操作系统的动态链接机制来进行加载时被用作抽象基类。ACE_Event_Handler 已在对反应器的讨论中进行了介绍。当开发者想要他的类响应来自反应器的事件时，他就从 ACE_Event_Handler 派生他的子类。

为什么服务对象要从 ACE_Event_Handler 继承？用户发起重配置的一种方法是生成一个信号；当这样的信号事件发生时，反应器被用于处理信号，并向 ACE_Service_Config 发出重配置请求。除此而外，软件的重配置也可能在某事件产生后发生。因而所有的服务对象都被构造为能对事件进行处理。

服务配置文件有它自己的简单脚本，用于描述你想要服务怎样启动和运行。你可以定义你是想要增加新服务，还是挂起、恢复或移除应用中现有的服务。另外还可以给服务发送参数。服务配置器还允许进行基于 ACE 的*流*（stream）的重配置。我们将在讨论了 ACE 流框架之后再来更多地讨论这一点。

8.2 定义配置文件

服务配置文件指定在应用中哪些服务要被加载和启动。此外，你可以指定哪些服务要被停止、挂起或恢复。还可以发送参数给你的服务对象的 init()方法。

8.2.1 启动服务

服务可以被静态或动态地启动。如果服务要动态启动，服务配置器实际上会从共享对象库（也就是，动态链接库）中加载服务对象。为此，服务配置器需要知道哪个库含有此对象，并且还需要知道对象在该库中的名字。因而，在你的代码文件中你必须通过你需要记住的名字来实例化服务对象。于是动态服务会这样被配置：

```
dynamic service_name type_of_service * location_of_shared_lib:name_of_object “parameters”
```

而静态服务这样被初始化：

```
static service_name “parameters_send_to_service_object”
```

8.2.2 挂起或恢复服务

如刚才所提到的，你在启动服务时分配给它一个名字。这个名字随后被用于挂起或恢复该服务。于是要挂起服务，你所要做的就是在 svc.conf 文件中指定：

```
suspend service_name
```

这使得服务对象中的 suspend()方法被调用。随后你的服务对象就应该挂起它自己（基于特定服务不同的“ 挂起 ”含义）。

如果你想要恢复这个服务，你所要做的就是在 svc.conf 文件中指定：

```
resume service_name
```

这使得服务对象中的 resume()方法被调用。随后你的服务对象就应该恢复它自己（基于特定服务不同的“ 恢复 ”含义。）

8.2.3 停止服务

停止并移除服务（如果服务是动态加载的）同样是很简单的操作，可以通过在你的配置文件中指定以下指令来完成：

```
remove service_name
```

这使得服务配置器调用你的应用的 fini()方法。该方法应该使此服务停止。服务配置器自己会负责将动态对象从服务器的地址空间里解除链接。

8.3 编写服务

为服务配置器编写你自己的服务相对比较简单。你可以让这个服务做任何你想做的事情。唯一的约束是它应该是 ACE_Service_Object 的子类。所以它必须实现 init()和 fini()方法。在 ACE_Service_Config 被打开（open()）时，它读取配置文件（也就是 svc.conf）并根据这个文件来对服务进行初始化。一旦服务被加载，它会调用该服务对象的 init()方法。类似地，如果配置文件要求移除服务，fini()方法就会被调用。这些方法负责分配和销毁服务所需的任何资源，比如内存、连接、线程等等。在 svc.conf 文件中指定的参数通过服务对象的 init()方法来传入。

下面的例子演示一个派生自 ACE_Task_Base 的服务。ACE_Task_Base 类含有 activate()方法，用于在对象里创建线程。（在“ 任务和主动对象 ”一章中讨论过的 ACE_Task 派生自 ACE_Task_Base，并包括了用于通信目的的消息队列。因为我们不需要我们的服务与其它任务通信，我们仅仅使用 ACE_Task_Base 来帮助我们完成工作。）更多详细信息，请阅读“ 任务和主动对象 ”一章。该服务是一个“ 无为 ”（do-nothing）的服务，一旦启动，它只是周期性地广播当天的时间。

例 8-1a

```
//The Services Header File.
#if !defined(MY_SERVICE_H)
#define MY_SERVICE_H

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

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

// A Time Service class. ACE_Task_Base already derives from
//ACE_Service_Object and thus  we don’t have to subclass from
//ACE_Service_Object in this case.
class TimeService: public ACE_Task_Base
{
public:
     virtual int init(int argc, ASYS_TCHAR *argv[]);
     virtual int fini(void);
     virtual int suspend(void);
     virtual int resume(void);
     virtual int svc(void);

private:
     int canceled_;
     ACE_Condition<ACE_Thread_Mutex> *cancel_cond_;
     ACE_Thread_Mutex *mutex_;
};

#endif
```

相应的实现如下所述：在时间服务接收到 init()调用时，它在任务中启用（activate()）一个线程。这将会创建一个新线程，其入口为 svc()方法。在 svc()方法中，该线程将会进行循环，直到它看到 canceled_标志被设置为止。此标志在服务配置框架调用 fini()时设置。但是，在 fini()方法返回底层的服务配置框架**之前**，它必须**确定**在底层的线程已经终止。因为服务配置器将要实际地卸载含有 TimeService 的共享库，从而将 TimeService 对象从应用进程中删除。如果在此**之前**线程并未终止，它将会对已经被服务配置器“ 蒸发 ”的代码发出调用！我们当然不需要这个。为了确保线程在服务配置器 “ 蒸发 ” TimeService 对象**之前终止**，程序使用了条件变量。（要更多地了解怎样使用条件变量，请阅读有关线程的章节）。

例 8-1b

```
#include "Services.h"

int TimeService::init(int argc, char *argv[])
{
     ACE_DEBUG((LM_DEBUG,"(%t)Starting up the time Service\n"));
     mutex_ = new ACE_Thread_Mutex;
     cancel_cond_ = new ACE_Condition<ACE_Thread_Mutex>(*mutex_);
     activate(THR_NEW_LWP|THR_DETACHED);
     return 0;
}

int TimeService::fini(void)
{
```

```
    ACE_DEBUG((LM_DEBUG,
        "(%t)FINISH!Closing down the Time Service\n"));

    //All of the following code is here to make sure that the
    //thread in the task is destroyed before the service configurator
    //deletes this object.
    canceled_=1;
    mutex_->acquire();
    while(canceled_)
        cancel_cond_->wait();
    mutex_->release();
    ACE_DEBUG((LM_DEBUG,"(%t)Time Service is exiting \n"));

    return 0;
}

//Suspend the Time Service.
int TimeService::suspend(void)
{
    ACE_DEBUG((LM_DEBUG,"(%t)Time Service has been suspended\n"));
    int result=ACE_Task_Base::suspend();
    return result;
}

//Resume the Time Service.
int TimeService::resume(void)
{
    ACE_DEBUG((LM_DEBUG,"(%t)Resuming Time Service\n"));
    int result=ACE_Task_Base::resume();
    return result;
}

//The entry function for the thread. The tasks underlying thread
//starts here and keeps sending out messages. It stops when:
// a) it is suspeneded
// b) it is removed by fini(). This happens when the fini() method
// sets the cancelled_ flag to true. Thus causes the TimeService
// thread to fall through the while loop and die. Before dying it
// informs the main thread of its imminent death. The main task
// that was previously blocked in fini() can then continue into the
// framework and destroy the TimeService object.
int TimeService::svc(void)
{
    char *time = new char[36];
```

```
    while(!canceled_)
    {
        ACE::timestamp(time,36);
        ACE_DEBUG((LM_DEBUG,"(%t)Current time is %s\n",time));
        ACE_OS::fflush(stdout);
        ACE_OS::sleep(1);
    }

    //Signal the Service Configurator informing it that the task is now
    //exiting so it can delete it.
    canceled_=0;
    cancel_cond_->signal();
    ACE_DEBUG((LM_DEBUG,
        "Signalled main task that Time Service is exiting \n"));
    return 0;
}

//Define the object here
TimeService time_service;
```

下面是一个简单的、只是用于启用时间服务的配置文件。可以去掉注释#号来挂起、恢复和移除服务。

例 8-1c

```
# To configure different services, simply uncomment the appropriate
#lines in this file!
#resume TimeService
#suspend TimeService
#remove TimeService
#set to dynamically configure the TimeService object and do so without
#sending any parameters to its init method
dynamic TimeService Service_Object * ./Server:time_service ""
```

最后，下面是启动服务配置器的代码段。这些代码还设置了一个信号处理器对象，用于发起重配置。该信号处理器已被设置成响应 SIGWINCH 信号（在窗口发生变化时产生的信号）。在启动服务配置器之后，应用进入一个反应式循环，等待 SIGWINCH 信号事件发生。一旦事件发生，就会回调事件处理器，由它调用 ACE_Service_Config 的 reconfigure()方法。如先前所讲述的，在此调用发生时，服务配置器重新读取配置文件，并处理用户放在其中的任何新指令。例如，在动态启动 TimeService 后，在这个例子中你可以改变 svc.conf 文件，只留下一个挂起命令在里面。当配置器读取它时，它将调用 TimeService 的挂起方法，从而使它挂起它的底层线程。类似地，如果稍后你又改变了 svc.conf，要求恢复服务，配置器就会调用 TimeService::resume()方法，从而恢复先前被挂起的线程。

例 8-1d

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

```
#include "ace/Event_Handler.h"
#include <signal.h>
//The Signal Handler which is used to issue the reconfigure()
//call on the service configurator.
class Signal_Handler: public ACE_Event_Handler
{
public:
    int open()
    {
        //register the Signal Handler with the Reactor to handle
        //re-configuration signals
        ACE_Reactor::instance()->register_handler(SIGWINCH,this);
        return 0;
    }

    int handle_signal(int signum, siginfo*,ucontext_t *)
    {
        if(signum==SIGWINCH)
            ACE_Service_Config::reconfigure();
        return 0;
    }
};

int main(int argc, char *argv[])
{
    //Instantiate and start up the Signal Handler. This is uses to
    //handle re-configuration events.
    Signal_Handler sh;
    sh.open();

    if (ACE_Service_Config::open (argc, argv) == -1)
        ACE_ERROR_RETURN ((LM_ERROR,
            "%p\n","ACE_Service_Config::open"),-1);

    while(1)
        ACE_Reactor::instance()->handle_events();
}
```

8.4 使用服务管理器

ACE_Service_Manager 是可用于对服务配置器进行远程管理的服务。它目前可以接受两种类型的请求。其一，你可以向它发送“ help ”消息，列出当前被加载进应用的所有服务。其二，你可以向服务管理器发送“ reconfigure ”消息，从而使得服务配置器重新配置它自己。

下面的例子演示了一个客户，它向服务管理器发送这两种类型的命令。

例 8-2

```
#include "ace/OS.h"
#include "ace/SOCK_Stream.h"
#include "ace/SOCK_Connector.h"
#include "ace/Event_Handler.h"
#include "ace/Get_Opt.h"
#include "ace/Reactor.h"
#include "ace/Thread_Manager.h"

#define BUFSIZE 128

class Client: public ACE_Event_Handler
{
public:
    ~Client()
    {
        ACE_DEBUG((LM_DEBUG,"Destructor \n"));
    }

    //Constructor
    Client(int argc, char *argv[]): connector_(), stream_()
    {
        //The user must specify address and port number
        ACE_Get_Opt get_opt(argc,argv,"a:p:");
        for(int c;(c=get_opt())!=-1;)
        {
            switch(c)
            {
            case 'a':
                addr_=get_opt.optarg;
                break;
            case 'p':
                port_= ((u_short)ACE_OS::atoi(get_opt.optarg));
                break;
            default:
                break;
            }
        }

        address_.set(port_,addr_);
    }
```

```
//Connect to the remote machine
int connect()
{
     connector_.connect(stream_,address_);
     ACE_Reactor::instance()->
          register_handler(this,ACE_Event_Handler::READ_MASK);
     return 0;
}

//Send a list_services command
int list_services()
{
     stream_.send_n("help",5);
     return 0;
}

//Send the reconfiguration command
int reconfigure()
{
     stream_.send_n("reconfigure",12);
     return 0;
}

//Handle both standard input and remote data from the
//ACE_Service_Manager
int handle_input(ACE_HANDLE h)
{
     char buf[BUFSIZE];

     //Got command from the user
     if(h== ACE_STDIN)
     {
          int result = ACE_OS::read (h, buf, BUFSIZ);
          if (result == -1)
               ACE_ERROR((LM_ERROR,"can't read from STDIN"));
          else if (result > 0)
          {
               //Connect to the Service Manager
               this->connect();
               if(ACE_OS::strncmp(buf,"list",4)==0)
                    this->list_services();
               else if(ACE_OS::strncmp(buf,"reconfigure",11)==0)
                    this->reconfigure();
          }
```

```
            return 0;
        }
        //We got input from remote
        else
        {
            switch(stream_.recv(buf,BUFSIZE))
            {
            case -1:
                //ACE_ERROR((LM_ERROR,  "Error in receiving from remote\n"));
                ACE_Reactor::instance()->remove_handler(this,
                        ACE_Event_Handler::READ_MASK);
                return 0;
            case 0:
                return 0;
            default:
                ACE_OS::printf("%s",buf);
                return 0;
            }
        }
    }

    //Used by the Reactor Framework
    ACE_HANDLE get_handle() const
    {
        return stream_.get_handle();
    }

    //Close down the underlying stream
    int handle_close(ACE_HANDLE,ACE_Reactor_Mask)
    {
        return stream_.close();
    }

private:
    ACE_SOCK_Connector connector_;
    ACE_SOCK_Stream stream_;
    ACE_INET_Addr address_;
    char *addr_;
    u_short port_;
};

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

```
    Client client(argc,argv);

    //Register the the client event handler as the standard
    //input handler
    ACE::register_stdin_handler(&client,
                ACE_Reactor::instance(),
                ACE_Thread_Manager::instance());

    ACE_Reactor::run_event_loop();
}
```

在此例中，Client 类是一个事件处理器，它处理两种类型的事件：来自用户的标准输入事件和来自 ACE_Service_Manager 的回复。如果用户输入“ list ”或“ reconfigure ”命令，相应的消息就被发送到远地的 ACE_Service_manager。服务管理器随之回复以当前配置的服务的列表或是“ done ”（指示服务的重配置已经完成）。因为 ACE_Service_Manager 是一个服务，你可以在 svc.conf 文件中指定你是想要静态还是动态地加载此服务，并使用服务配置框架来将它加载入应用中。

例如，下面的命令指定在 9876 端口静态地启动服务管理器：

```
static ACE_Service_Manager “-p 9876”
```

第 9 章 消息队列(Message Queue)

现代的实时应用通常被构建成一组相互通信、但又相互独立的任务。这些任务可以通过若干机制来与对方进行通信，其中常用的一种就是消息队列。在这一情况下，基本的通信模式是：发送者（或生产者）任务将消息放入消息队列，而接收者（或消费者）任务从此队列中取出消息。这当然只是消息队列的使用方式之一。在接下来的讨论中，我们将看到若干不同的使用消息队列的例子。

ACE 中的消息队列是仿照 UNIX 系统 V 的消息队列设计的，如果你已经熟悉系统 V 的话，就很容易掌握 ACE 的消息队列的使用。在 ACE 中有多种不同类型的消息队列可用，每一种都使用不同的调度算法来进行队列的入队和出队操作。

9.1 消息块

在 ACE 中，消息作为消息块（Message Block）被放入消息队列中。消息块包装正被存储的实际消息数据，并提供若干数据插入和处理操作。每个消息块“包含”一个头和一个数据块。注意在这里“包含”是在宽松的意义上使用的。消息块可以不对与数据块（Data Block）或是消息头（Message Header）相关联的内存进行管理（尽管你可以让消息块进行这样的管理）。它仅仅持有指向两者的指针。所以包含只是逻辑上的。数据块持有指向实际的数据缓冲区的指针。如图 9-1 所示，这样的设计带来了多个消息块之间的数据的灵活共享。注意在图中两个消息块共享一个数据块。这样，无需带来数据拷贝开销，就可以将同一数据放入不同的队列中。

消息块类名为 ACE_Message_Block，而数据块类名为 ACE_Data_Block。ACE_Message_Block 的构造器是实际创建消息块和数据块的方便办法。

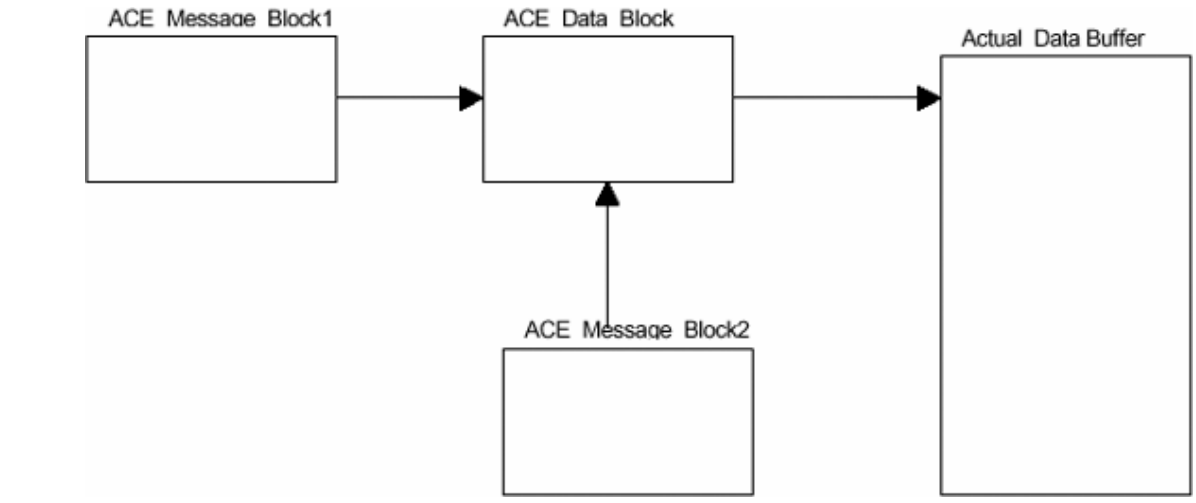

图 9-1 ACE 消息块的构造

9.1.1 构造消息块

ACE_Message_Block 类包含有若干不同的构造器。你可以使用这些构造器来帮助你管理隐藏在消息和数据块后面的消息数据。ACE_Message_Block 类可用于完全地隐藏 ACE_Data_Block，并为你管理消息数据；或者，如果你需要，你可以自己创建和管理数据块及消息数据。下一部分将考查怎样使用 ACE_Message_Block 来管理消息内存和数据块。然后我们将考查怎样独立地进行这样的管理，而不用依赖 ACE_Message_Block 的管理特性。

9.1.1.1 ACE_Message_Block 分配和管理数据内存

要创建消息块，最容易的方法是将底层数据块的大小传给 ACE_Message_Block 的构造器，从而创建 ACE_Data_Block，并为消息数据分配空的内存区。在创建消息块后，你可以使用 rd_ptr()和 wr_ptr()方法来在消息块中插入和移除数据。让 ACE_Message_Block 来为数据和数据块创建内存区的主要优点是，它会为你正确地管理所有内存，从而使你免于在将来为许多内存泄漏而头疼。

ACE_Message_Block 的构造器还允许程序员指定 ACE_Message_Block 在内部分配内存时所应使用的分配器。如果你传入一个分配器，消息块将用它来为数据块和消息数据区的创建分配内存。ACE_Message_Block 的构造器为：

```
ACE_Message_Block (size_t size,
    ACE_Message_Type type = MB_DATA,
    ACE_Message_Block *cont = 0,
    const char *data = 0,
    ACE_Allocator *allocator_strategy = 0,
    ACE_Lock *locking_strategy = 0,
    u_long priority = 0,
    const ACE_Time_Value & execution_time = ACE_Time_Value::zero,
    const ACE_Time_Value & deadline_time = ACE_Time_Value::max_time);
```

上面的构造器的参数为：

1. 要与消息块相关联的数据缓冲区的大小。注意消息块的大小是 size，但长度将为 0，直到 wr_ptr 被设置为止。这将在后面进一步解释。
2. 消息的类型。（在 ACE_Message_Type 枚举中有若干类型可用，其中包括缺省的数据消息）。
3. 指向“片段链”（fragment chain）中的下一个消息块的指针。消息块可以实际地链接在一起来形成链。随后链可被放入消息队列中，就好像它是单个数据块一样。该参数缺省为 0，意味着此块不使用链。
4. 指向要存储在此消息块中的数据缓冲区的指针。如果该参数的值为零，就会创建缓冲区（大小由第一个参数指定），并由该消息块进行管理。当消息块被删除时，相应的数据缓冲区也被删除。但是，如果在此参数中指定了数据缓冲区，也就是，参数不为空，当消息块被销毁时它就**不会**删除数据缓冲区。这是一个重要特性，必须牢牢记住。
5. 用于分配数据缓存（如果需要）的 allocator_strategy，在第四个参数为空时使用（如上面所解释的）。任何 ACE_Allocator 的子类都可被用作这一参数。（关于 ACE_Allocator 的更多信息，参见“内存管

理”一章）。

6. 如果 locking_strategy 不为零，它就将用于保护访问共享状态（例如，引用计数）的代码区，以避免竞争状态。
7. 这个参数以及后面两个参数用于 ACE 中的实时消息队列的调度，目前应保留它们的缺省值。

9.1.1.2 用户分配和管理消息内存

如果你正在使用 ACE_Message_Block，你并不一定要让它来为你分配内存。消息块的构造器允许你：

- 创建并传入你自己的指向消息数据的数据块。
- 传入指向消息数据的指针，消息块将创建并设置底层的数据块。消息块将为数据块、而不是消息数据管理内存。

下面的例子演示怎样将指向消息数据的指针传给消息块，以及 ACE_Message_Block 怎样创建和管理底层的 ACE_Data_Block。

```
//The data
char data[size];
data = ”This is my data”;

//Create a message block to hold the data
ACE_Message_Block *mb = new ACE_Message_Block (data, // data that is stored
                                                     // in the newly created data
                                                     //
                                          blocksize);  //size of the block that
                                                       //is to be stored.
```

该构造器创建底层数据块，并将它设置为指向传递给它的数据的开头。被创建的消息块并不拷贝该数据，也不假定自己拥有它的所有权。这就意味着在消息块 mb 被销毁时，相关联的数据缓冲区 data 将**不会**被销毁。这是有意义的：消息块没有拷贝数据，因此内存也不是它分配的，这样它也不应该负责销毁它。

9.1.2 在消息块中插入和操作数据

除了构造器，ACE_Message_Block 还提供若干方法来直接在消息块中插入数据。另外还有一些方法可用来操作已经在消息块中的数据。

每个 ACE_Message_Block 都有两个底层指针：rd_ptr 和 wr_ptr，用于在消息块中读写数据。它们可以通过调用 rd_ptr()和 wr_ptr()方法来直接访问。rd_ptr 指向下一次读取数据的位置，而 wr_ptr 指向下一次写入数据的位置。程序员必须小心地管理这些指针，以保证它们总是指向正确的位置。在使用这些指针读写数据时，程序员必须自己来增加它们的值，它们不会魔法般地自动更新。大多数内部的消息块方法也使用这两个指针，从而使它们能够在你调用消息块方法时改变指针状态。程序员必须保证自己了解这

些指针的变化。

9.1.2.1 拷贝与复制（Copying and Duplicating）

可以使用 ACE_Message_Block 的 copy()方法来将数据拷贝进消息块。

```
int copy(const char *buf, size_t n);
```

copy 方法需要两个参数，其一是指向要拷贝进消息块的缓冲区的指针，其二是要拷贝的数据的大小。该方法从 wr_pt 指向的位置开始往前写，直到它到达参数 n 所指示的数据缓冲区的末尾。copy()还会保证 wr_ptr 的更新，使其指向缓冲区的新末尾处。注意该方法将实际地执行物理拷贝，因而应该小心使用。

base()和 length()方法可以联合使用，以将消息块中的整个数据缓冲区拷贝*出来*。base()返回指向数据块的第一个数据条目的指针，而 length()返回队中数据的总大小。将 base 和 length 相加，可以得到指向数据块末尾的指针。合起来使用这些方法，你就可以写一个例程来从消息块中取得数据，并做一次外部拷贝。

duplicate()和 clone()方法用于制作消息块的“ 副本 ”。如它的名字所暗示的，clone()方法实际地创建整个消息块的新副本，包括它的数据块和附加部分 ;也就是说，这是一次“ 深度复制 ”。而另一方面，duplicate()方法使用的是 ACE_Message_Block 的引用计数机制。它返回指向要被复制的消息块的指针，并在内部增加内部引用计数。

9.1.2.2 释放消息块

一旦使用完消息块，程序员可以调用它的 release()方法来释放它。如果消息数据内存是由该消息块分配的，调用 release()方法就也会释放此内存。如果消息块是引用计数的，release()就会减少计数，直到到达 0 为止；之后消息块和与它相关联的数据块才从内存中被移除。

9.2 ACE 的消息队列

如先前所提到的，ACE 有若干不同类型的消息队列，它们大体上可划分为两种范畴：静态的和动态的。静态队列是一种通用的消息队列（ACE_Message_Queue），而动态消息队列（ACE_Dynamic_Message_Queue）是实时消息队列。这两种消息队列的主要区别是：静态队列中的消息具有静态的优先级，也就是，一旦优先级被设定就不会再改变；而另一方面，在动态消息队列中，基于诸如执行时间和最终期限等参数，消息的优先级可以动态地改变。

下面的例子演示怎样创建简单的静态消息队列，以及怎样在其上进行消息块的入队和出队操作。

例 9-1a

```
#ifndef MQ_EG1_H_
#define MQ_EG1_H_

#include "ace/Message_Queue.h"

class QTest
{
```

```
public:
     //Constructor creates a message queue with no synchronization
     QTest(int num_msgs);

     //Enqueue the num of messages required onto the message mq.
     int enq_msgs();

     //Dequeue all the messages previously enqueued.
     int deq_msgs ();

private:
     //Underlying message queue
     ACE_Message_Queue<ACE_NULL_SYNCH> *mq_;

     //Number of messages to enqueue.
     int no_msgs_;
};
#endif /*MQ_EG1.H_*/
```

例 9-1b

```
#include "mq_eg1.h"

QTest::QTest(int num_msgs)
     :no_msgs_(num_msgs)
{
     ACE_TRACE("QTest::QTest");

     //First create a message queue of default size.
     if(!(this->mq_=new ACE_Message_Queue<ACE_NULL_SYNCH> ()))
          ACE_DEBUG((LM_ERROR,"Error in message queue initialization \n"));
}

int  QTest::enq_msgs()
{
     ACE_TRACE("QTest::enq_msg");
     for(int i=0; i<no_msgs_;i++)
     {
          //create a new message block specifying exactly how large
          //an underlying data block should be created.
          ACE_Message_Block *mb;
          ACE_NEW_RETURN(mb,
                         ACE_Message_Block(ACE_OS::strlen("This is message 1\n")),
                         -1);
```

```
        //Insert data into the message block using the wr_ptr
        ACE_OS::sprintf(mb->wr_ptr(), "This is message %d\n", i);

        //Be careful to advance the wr_ptr by the necessary amount.
        //Note that the argument is of type "size_t" that is mapped to
        //bytes.
        mb->wr_ptr(ACE_OS::strlen("This is message 1\n"));

        //Enqueue the message block onto the message queue
        if(this->mq_->enqueue_prio(mb)==-1)
        {
            ACE_DEBUG((LM_ERROR,"\nCould not enqueue on to mq!!\n"));
            return -1;
        }

        ACE_DEBUG((LM_INFO,"EQ'd data: %s\n", mb->rd_ptr() ));
    } //end for

    //Now dequeue all the messages
    this->deq_msgs();

    return 0;
}

int  QTest::deq_msgs()
{
    ACE_TRACE("QTest::dequeue_all");
    ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n"
            ,mq_->message_count(),mq_->message_bytes()));
    ACE_Message_Block *mb;

    //dequeue the head of the message queue until no more messages are
    //left. Note that I am overwriting the message block mb and I since
    //I am using the dequeue_head() method I dont have to worry about
    //resetting the rd_ptr() as I did for the wrt_ptr()
    for(int i=0;i <no_msgs_; i++)
    {
        mq_->dequeue_head(mb);
        ACE_DEBUG((LM_INFO,"DQ'd data %s\n", mb->rd_ptr() ));

        //Release the memory associated with the mb
        mb->release();
    }
```

```
    return 0;
}

int main(int argc, char* argv[])
{
    if(argc <2)
        ACE_ERROR_RETURN((LM_ERROR, "Usage %s num_msgs", argv[0]), -1);

    QTest test(ACE_OS::atoi(argv[1]));
    if(test.enq_msgs() == -1)
        ACE_ERROR_RETURN( (LM_ERROR,"Program failure \n"), -1);
}
```

上例演示了消息队列类的若干方法。例子由一个 Qtest 类组成，它通过 ACE_NULL_SYNCH 锁定来实例化缺省大小的消息队列。锁（互斥体和条件变量）被消息队列用来：

- 保证由消息块维护的引用计数的安全，防止在有多个线程访问时的竞争状态。
- “ 唤醒 ” 所有因为消息队列空或满而休眠的线程。

在此例中，因为只有一个线程，消息队列的模板同步参数被设置为空（ACE_NULL_SYNCH，意味着使用 ACE_Null_Mutex 和 ACE_Null_Condition）。随后 Qtest 的 enq_msgs()方法被调用，它进入循环，创建消息、并将其放入消息队列中。消息数据的大小作为参数传给 ACE_Message_Block 的构造器。使用该构造器使得内存被自动地管理（也就是，内存将在消息块被删除时，即 release()时被释放）。wr_ptr 随后被获取（使用 wr_ptr()访问方法），且数据被拷贝进消息块。在此之后，wr_ptr 向前增长。然后使用消息队列的 enqueue_prio()方法来实际地将消息块放入底层消息队列中。

在 no_msgs_个消息块被创建、初始化和插入消息队列后，enq_msgs()调用 deq_msgs()方法。该方法使用 ACE_Message_Queue 的 dequeue_head()方法来使消息队列中的每个消息出队。在消息出队后，就显示它的数据，然后再释放消息。

9.3 水位标

水位标用于在消息队列中指示何时在其中的数据已过多（消息队列到达了高水位标），或何时在其中的数据的数量不足（消息队列到达了低水位标）。两种水位标都用于流量控制——例如，low_water_mark 可用于避免像 TCP 中的“ 傻窗口综合症 ”（silly window syndrome）那样的情况，而 high_water_mark 可用于“ 阻止 “ 或减缓数据的发送或生产。

ACE 中的消息队列通过维护已经入队的总数据量的字节计数来获得这些功能。因而，无论何时有新消息块被放入消息队列中，消息队列都将先确定它的长度，然后检查是否能将此消息块放入队列中（也就是，确认如果将此消息块入队，消息队列没有超过它的高水位标）。如果消息队列不能将数据入队，而它又持有一个锁（也就是，使用了 ACE_SYNC，而不是 ACE_NULL_SYNCH 作为消息队列的模板参数），它就会阻塞调用者，直到有足够的空间可用，或是入队方法的超时（timeout）到期。如果超时已到期，或是队列持有一个空锁，入队方法就会返回-1，指示无法将消息入队。

类似地，当 ACE_Message_Queue 的 dequeue_head 方法被调用时，它检查并确认在出队之后，剩下的数据的数量高于低水位标。如果不是这样，而它又持有一个锁，它就会阻塞；否则就返回-1，指示失败（和

入队方法的工作方式一样)。

分别有两个方法可用于设置和获取高低水位标：

```
//get the high water mark
size_t high_water_mark(void)

//set the high water mark
void high_water_mark(size_t hwm);

//get the low water_mark
size_t low_water_mark(void)

//set the low water_mark
void low_water_mark(size_t lwm)
```

9.4 使用消息队列迭代器(Message Queue Iterator)

和其它容器类的常见情况一样，可将前进（forward）和后退（reverse）迭代器用于 ACE 中的消息队列。这两个迭代器名为 ACE_Message_Queue_Iterator 和 ACE_Message_Queue_Reverse_Iterator。它们都需要一个模板参数，用于在遍历消息队列时进行同步。如果有多个线程使用消息队列，该参数就应设为 ACE_SYNCH；否则，就可设为 ACE_NULL_SYNCH。在迭代器对象被创建时，必须将我们想要进行迭代的消息队列的引用传给它的构造器。

下面的例子演示水位标和迭代器的使用：

例 9-2

```
#include ”ace/Message_Queue.h”
#include ”ace/Get_Opt.h”
#include ”ace/Malloc_T.h”
#define SIZE_BLOCK 1

class Args
{
public:
    Args(int argc, char*argv[],int& no_msgs, ACE_Message_Queue<ACE_NULL_SYNCH>* &mq)
    {
        ACE_Get_Opt get_opts(argc,argv,”h:l:t:n:xsd”);
        while((opt=get_opts())!=-1)
            switch(opt)
            {
            case ’n’:
                //set the number of messages we wish to enqueue and dequeue
                no_msgs=ACE_OS::atoi(get_opts.optarg);
                ACE_DEBUG((LM_INFO,”Number of Messages %d \n”,no_msgs));
```

```
                break;

            case 'h':
                //set the high water mark
                hwm=ACE_OS::atoi(get_opts.optarg);
                mq->high_water_mark(hwm);
                ACE_DEBUG((LM_INFO,"High Water Mark %d msgs \n",hwm));
                break;

            case 'l':
                //set the low water mark
                lwm=ACE_OS::atoi(get_opts.optarg);
                mq->low_water_mark(lwm);
                ACE_DEBUG((LM_INFO,"Low Water Mark %d msgs \n",lwm));
                break;

            default:
                ACE_DEBUG((LM_ERROR,
                    "Usage -n<no. messages> -h<hwm> -l<lwm>\n"));
                break;
            }
    }

private:
    int opt;
    int hwm;
    int lwm;
};

class Qtest
{
public:
    QTest(int argc, char*argv[])
    {
        //First create a message queue of default size.
        if(!(this->mq_=new ACE_Message_Queue<ACE_NULL_SYNCH> ()))
            ACE_DEBUG((LM_ERROR,"Error in message queue initialization \n"));

        //Use the arguments to set the water marks and the no of messages
        args_ = new Args(argc,argv,no_msgs_,mq_);
    }

    int start_test()
    {
```

```
    for(int i=0; i<no_msgs_;i++)
    {
        //Create a new message block of data buffer size 1
        ACE_Message_Block * mb= new ACE_Message_Block(SIZE_BLOCK);

        //Insert data into the message block using the rd_ptr
        *mb->wr_ptr()=i;

        //Be careful to advance the wr_ptr
        mb->wr_ptr(1);

        //Enqueue the message block onto the message queue
        if(this->mq_->enqueue_prio(mb)==-1)
        {
            ACE_DEBUG((LM_ERROR,"\nCould not enqueue on to mq!!\n"));
            return -1;
        }

        ACE_DEBUG((LM_INFO,"EQ'd data: %d\n",*mb->rd_ptr()));
    }

    //Use the iterators to read
    this->read_all();

    //Dequeue all the messages
    this->dequeue_all();

    return 0;
}

void read_all()
{
    ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n"
                ,mq_->message_count(),mq_->message_bytes()));
    ACE_Message_Block *mb;

    //Use the forward iterator
    ACE_DEBUG((LM_INFO,"\n\nBeginning Forward Read \n"));
    ACE_Message_Queue_Iterator<ACE_NULL_SYNCH> mq_iter_(*mq_);
    while(mq_iter_.next(mb))
    {
        mq_iter_.advance();
        ACE_DEBUG((LM_INFO,"Read data %d\n",*mb->rd_ptr()));
    }
```

```
        //Use the reverse iterator
        ACE_DEBUG((LM_INFO,"\n\nBeginning Reverse Read \n"));
        ACE_Message_Queue_Reverse_Iterator<ACE_NULL_SYNCH>
        mq_rev_iter_(*mq_);
        while(mq_rev_iter_.next(mb))
        {
            mq_rev_iter_.advance();
            ACE_DEBUG((LM_INFO,"Read data %d\n",*mb->rd_ptr()));
        }
    }

    void dequeue_all()
    {
        ACE_DEBUG((LM_INFO,"\n\nBeginning DQ \n"));
        ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n",
            mq_->message_count(),mq_->message_bytes()));
        ACE_Message_Block *mb;

        //dequeue the head of the message queue until no more messages
        //are left
        for(int i=0;i<no_msgs_;i++)
        {
            mq_->dequeue_head(mb);
            ACE_DEBUG((LM_INFO,"DQ'd data %d\n",*mb->rd_ptr()));
        }
    }

private:
    Args *args_;
    ACE_Message_Queue<ACE_NULL_SYNCH> *mq_;
    int no_msgs_;
};

int main(int argc, char* argv[])
{
    QTest test(argc,argv);
    if(test.start_test()<0)
        ACE_DEBUG((LM_ERROR,"Program failure \n"));
}
```

这个例子使用 ACE_Get_Opt 类（更多关于这个工具类的信息见附录）来获取低水位标和高水位标（在 Args 类中）。使用 low_water_mark()和 high_water_mark()访问函数可对它们进行设置。除此而外，还有一个 read_all()方法使用 ACE_Message_Queue_Iterator 和 ACE_Message_Queue_Reverse_Iterator 来向前读

和反向读。

9.5 动态或实时消息队列

如上面所提到的，动态消息队列是其中的消息的优先级随时间变化的队列。实时应用需要这样的行为特性，因而这样的队列在实时应用中天生更为有用。

ACE 目前提供两种动态消息队列：基于最终期限（deadline）的和基于松弛度（laxity）的（参见[IX]）动态消息队列。基于最终期限的消息队列通过每个消息的最终期限来设置它们的优先级。在使用最早 deadline 优先算法来调用 dequeue_head()方法时，队列中有着最早的最终期限的消息块将最先出队。而基于松弛度的消息队列，同时使用执行时间和最终期限来计算松弛度，并将其用于划分各个消息块的优先级。松弛度是十分有用的，因为在根据最终期限来调度时，被调度的任务有可能有最早的最终期限，但同时又有相当长的执行时间，以致于即使它被立即调度，也不能够完成。这会消极地影响其它任务，因为它可能阻塞那些可以调度的任务。松弛度把这样的长执行时间考虑在内，并保证任务如果不能完成，就不会被调度。松弛度队列中的调度基于最小松弛度优先算法。

基于松弛度的消息队列和基于最终期限的消息队列都实现为 ACE_Dynamic_Message_Queue。ACE 使用**策略**（STRATEGY）模式来为动态队列提供不同的调度特性。每种消息队列使用不同的“策略”对象来动态地设置消息队列中消息的优先级。每个这样的“策略”对象都封装了一种不同的算法来基于执行时间、最终期限，等等，计算优先级；并且无论何时消息入队或是出队，都会调用这些策略对象来完成前述计算工作。（有关策略模式的更多信息，请参见“设计模式”）。消息策略模式派生自 ACE_Dynamic_Message_Strategy，目前有两种策略可用：ACE_Laxity_Message_Strategy 和 ACE_Deadline_Message_Strategy。因此，要创建基于松弛度的动态消息队列，首先必须创建 ACE_Laxity_Message_Strategy 对象。随后，应该对 ACE_Dynamic_Message_Queue 对象进行实例化，并将新创建的策略对象作为参数之一传给它的构造器。

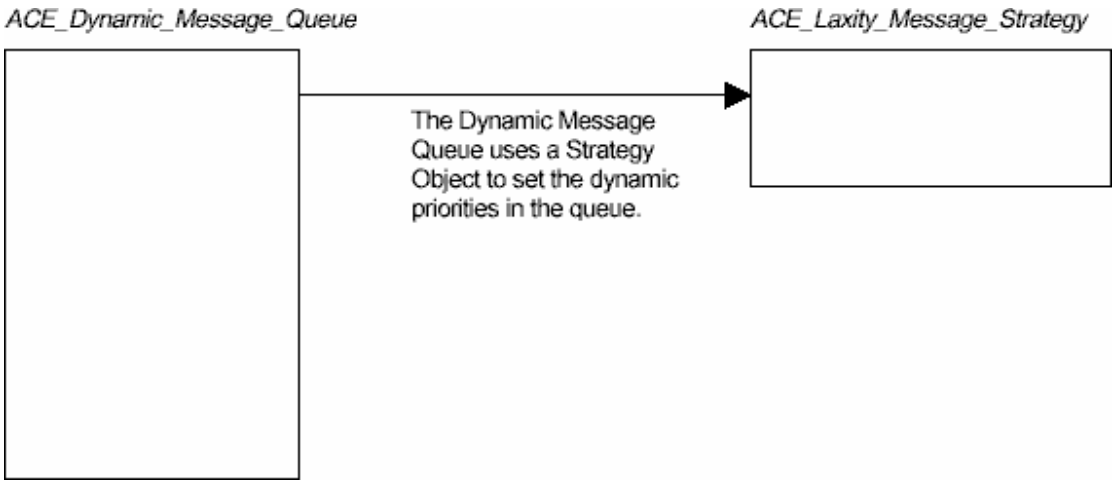

图 9-2 ACE_Dynamic_Message_Queue 与 ACE_Laxity_Message_Strategy 的关系

创建消息队列

为简化这些不同类型的消息队列的创建，ACE 提供了名为 ACE_Message_Queue_Factory 的具体消息队列工厂，它使用工厂方法（FACTORY METHOD，更多信息参见“设计模式”）模式的一种变种来创

建适当类型的消息队列。消息队列工厂有三个静态的工厂方法，可用来创建三种不同类型的消息队列：

```
static ACE_Message_Queue<ACE_SYNCH_USE> *
        create_static_message_queue();

static ACE_Dynamic_Message_Queue<ACE_SYNCH_USE> *
        create_deadline_message_queue();

static ACE_Dynamic_Message_Queue<ACE_SYNCH_USE> *
        create_laxity_message_queue();
```

每个方法都返回指向刚创建的消息队列的指针。注意这些方法都是静态的，而 create_static_message_queue() 方法返回的是 ACE_Message_Queue，其它两个方法则返回 ACE_Dynamic_Message_Queue。

下面这个简单的例子演示动态和静态消息队列的创建和使用：

例 9-3

```
#include ”ace/Message_Queue.h”
#include ”ace/Get_Opt.h”
#include ”ace/OS.h”

class Args
{
public:
    Args(int argc, char*argv[],int& no_msgs, int& time,
        ACE_Message_Queue<ACE_NULL_SYNCH>*&mq)
    {
        ACE_Get_Opt get_opts(argc,argv,”h:l:t:n:xsd”);
        while((opt=get_opts())!=-1)
            switch(opt)
            {
            case ’t’:
                time=ACE_OS::atoi(get_opts.optarg);
                ACE_DEBUG((LM_INFO,”Time: %d \n”,time));
                break;

            case ’n’:
                no_msgs=ACE_OS::atoi(get_opts.optarg);
                ACE_DEBUG((LM_INFO,”Number of Messages %d \n”,no_msgs));
                break;

            case ’x’:
                mq=ACE_Message_Queue_Factory<ACE_NULL_SYNCH>::
                create_laxity_message_queue();
```

```
                ACE_DEBUG((LM_DEBUG,”Creating laxity q\n”));
                break;

            case ’d’:
                mq=ACE_Message_Queue_Factory<ACE_NULL_SYNCH>::
                create_deadline_message_queue();
                ACE_DEBUG((LM_DEBUG,”Creating deadline q\n”));
                break;

            case ’s’:
                mq=ACE_Message_Queue_Factory<ACE_NULL_SYNCH>::
                create_static_message_queue();
                ACE_DEBUG((LM_DEBUG,”Creating static q\n”));
                break;

            case ’h’:
                hwm=ACE_OS::atoi(get_opts.optarg);
                mq->high_water_mark(hwm);
                ACE_DEBUG((LM_INFO,”High Water Mark %d msgs \n”,hwm));
                break;

            case ’l’:
                lwm=ACE_OS::atoi(get_opts.optarg);
                mq->low_water_mark(lwm);
                ACE_DEBUG((LM_INFO,”Low Water Mark %d msgs \n”,lwm));
                break;

            default:
                ACE_DEBUG((LM_ERROR,”Usage specify queue type\n”));
                break;
            }
    }

private:
    int opt;
    int hwm;
    int lwm;
};

class Qtest
{
public:
    QTest(int argc, char*argv[])
    {
```

```
    args_ = new Args(argc,argv,no_msgs_,time_,mq_);
    array_ =new ACE_Message_Block*[no_msgs_];
}

int start_test()
{
    for(int i=0; i<no_msgs_;i++)
    {
        ACE_NEW_RETURN (array_[i], ACE_Message_Block (1), -1);
        set_deadline(i);
        set_execution_time(i);
        enqueue(i);
    }

    this->dequeue_all();

    return 0;
}

//Call the underlying ACE_Message_Block method msg_deadline_time() to
//set the deadline of the message.
void set_deadline(int msg_no)
{
    float temp=(float) time_/(msg_no+1);
    ACE_Time_Value tv;
    tv.set(temp);
    ACE_Time_Value deadline(ACE_OS::gettimeofday()+tv);
    array_[msg_no]->msg_deadline_time(deadline);
    ACE_DEBUG((LM_INFO,"EQ'd with DLine %d:%d,", deadline.sec(),deadline.usec()));
}

//Call the underlying ACE_Message_Block method to set the execution time
void set_execution_time(int msg_no)
{
    float temp=(float) time_/10*msg_no;
    ACE_Time_Value tv;
    tv.set(temp);
    ACE_Time_Value xtime(ACE_OS::gettimeofday()+tv);
    array_[msg_no]->msg_execution_time (xtime);
    ACE_DEBUG((LM_INFO,"Xtime %d:%d,",xtime.sec(),xtime.usec()));
}

void enqueue(int msg_no)
{
```

```
        //Set the value of data at the read position
        *array_[msg_no]->rd_ptr()=msg_no;

        //Advance write pointer
        array_[msg_no]->wr_ptr(1);

        //Enqueue on the message queue
        if(mq_->enqueue_prio(array_[msg_no])==-1)
        {
            ACE_DEBUG((LM_ERROR,"\nCould not enqueue on to mq!!\n"));
            return;
        }

        ACE_DEBUG((LM_INFO,"Data %d\n",*array_[msg_no]->rd_ptr()));
    }

    void dequeue_all()
    {
        ACE_DEBUG((LM_INFO,"Beginning DQ \n"));
        ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n",
        mq_->message_count(),mq_->message_bytes()));

        for(int i=0;i<no_msgs_ ;i++)
        {
            ACE_Message_Block *mb;
            if(mq_->dequeue_head(mb)==-1)
            {
                ACE_DEBUG((LM_ERROR,"\nCould not dequeue from mq!!\n"));
                return;
            }

            ACE_DEBUG((LM_INFO,"DQ'd data %d\n",*mb->rd_ptr()));
        }
    }

private:
    Args *args_;
    ACE_Message_Block **array_;
    ACE_Message_Queue<ACE_NULL_SYNCH> *mq_;
    int no_msgs_;
    int time_;
};

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

```
{
    QTest test(argc,argv);
    if(test.start_test()<0)
        ACE_DEBUG((LM_ERROR,”Program failure \n”));
}
```

上面这个例子和前面的例子很相似，只是在其中增加了动态消息队列。在 Args 类中，我们增加了选项来使用 ACE_Message_Queue_Factory 创建所有不同类型的消息队列。此外，在 Qtest 类中增加了两个新方法，用以在每个消息块创建时设置它们的最终期限和执行时间（set_deadline()和 set_execution_time()）。这两个方法使用了 ACE_Message_Block 的 msg_execution_time()和 msg_deadline_time()方法。注意它们采用的是绝对时间**而非**相对时间，这也是它们和 ACE_OS::gettimeofday()方法一起使用的原因。

最终期限和执行时间通过 time 参数的帮助来设置。最终期限是这样来设置的：第一个消息将拥有最后的最终期限，在最终期限消息队列的情形中，它应该最后被调度。但是在使用松弛度队列时，执行时间和最终期限都将被考虑在内。

附录：工具类

地址包装类

ACE_INET_Addr

ACE_INET_Addr 类封装了 Internet 域地址族（AF_INET）。该类派生自 ACE 中的 ACE_Addr。它的多种构造器可用于初始化对象，使其具有特定的 IP 地址和端口。除此而外，该类还具有若干 set 和 get 方法，并且重载了比较操作，也就是==操作符和!=操作符。进一步的使用细节见参考手册。

ACE_UNIX_Addr

ACE_UNIX_Addr 类封装了 UNIX 域地址族（AF_UNIX），它也派生自 ACE_Addr。该类的功能与 ACE_INET_Addr 类相似。进一步的细节见参考手册。

时间包装类

ACE_Time_Value

通过 ACE_DEBUG 和 ACE_ERROR 记录日志

ACE_DEBUG 和 ACE_ERROR 宏用于打印调试及错误信息及记录相应的日志。它们的用法已在整个教程中作了演示。

这两个宏可以使用下面的格式修饰符。就如在 printf 格式串中一样，这些选项的每一个都冠有前缀’%’：

格式修饰符	描述
‘a’	在该点结束程序（var 参数是结束状态！）
‘c’	打印一个字符

‘i’, ‘d’	打印一个十进制数
‘I’	根据嵌套深度缩进
‘e’, ‘E’, ‘f’, ‘F’, ‘g’, ‘G’	打印一个双精度数
‘l’	打印错误发生处的行号
‘N’	打印错误发生处的文件名
‘n’	打印程序名（或“ <unknown> ”，如果没有设置的话）
‘o’	作为八进制数打印
‘P’	打印出当前的进程 id
‘p’	从 sys_errlist 中打印出适当的 errno 值
‘r’	调用相应参数所指向的函数
‘R’	打印返回状态
‘S’	根据 var 参数打印出适当的_sys_siglist 条目
‘s’	打印出一个字符串
‘T’	以 hour:minute:sec:usec 格式打印时间戳
‘D’	以 month/day/year hour:minute:sec:usec 格式打印时间戳
‘t’	打印线程 id（如果是单线程则为 1）
‘u’	作为无符号整数打印
‘X’, ‘x’	作为十六进制数打印
‘%’	打印出一个百分号，’%’

获取命令行参数

ACE_Get_opt

这个工具类基于 stdlib 中的 getopt()<3c>函数，用于从用户那里获取参数。传入该类的构造器的名为 optstring 的串指定应用想要响应的“ 转换子 ”（switch）。如果“ 转换子 ”字母紧跟着一个冒号，那就意味着该“ 转换子 ”期望一个参数。例如，如果 optstring 为“ ab:c ”，应用就期望没有参数的“ -a ”和“ -c ”，和有一个参数的“ -b ”。例如，可以这样来运行此应用：

```
MyApplication –a –b 10 –c
```

()操作符已经被重载，并被用于扫描 argv 的成员，以找到选项串（optstring）中指定的选项。
下面的例子将帮助阐明怎样使用这个类来从用户那里获取参数。

```
#include "ace/Get_Opt.h"
int main (int argc, char *argv[])
{
```

```
//Specify option string so that switches b, d, f and h all expect
//arguments. Switches a, c, e and g expect no arguments.
ACE_Get_Opt get_opt (argc, argv, "ab:cd:ef:gh:");
int c;

//Process the scanned options with the help of the overloaded ()
//operator.
while ((c = get_opt ()) != EOF)
    switch (c)
    {
    case 'a':
        ACE_DEBUG ((LM_DEBUG, "got a\n"));
        break;

    case 'b':
        ACE_DEBUG ((LM_DEBUG, "got b with arg %s\n",
            get_opt.optarg));
        break;

    case 'c':
        ACE_DEBUG ((LM_DEBUG, "got c\n"));
        break;

    case 'd':
        ACE_DEBUG ((LM_DEBUG, "got d with arg %s\n",
            get_opt.optarg));
        break;

    case 'e':
        ACE_DEBUG ((LM_DEBUG, "got e\n"));
        break;

    case 'f':
        ACE_DEBUG ((LM_DEBUG, "got f with arg %s\n",
            get_opt.optarg));
        break;

    case ’g’:
        ACE_DEBUG ((LM_DEBUG, "got g\n"));
        break;

    case ’h’:
        ACE_DEBUG ((LM_DEBUG, "got h with arg %s\n",
            get_opt.optarg));
```

```
            break;

        default:
            ACE_DEBUG ((LM_DEBUG, "got %c, which is unrecognized!\n",c));
            break;
        }

    //optind indicates how much of argv has been scanned so far, while
    //get_opt hasn't returned EOF. In this case it indicates the index in
    //argv from where the option switches have been fully recognized and the
    //remaining elements must be scanned by the called himself.
    for (int i = get_opt.optind; i < argc; i++)
        ACE_DEBUG ((LM_DEBUG, "optind = %d, argv[optind] = %s\n",
                i, argv[i]));

    return 0;
}
```

有关如何使用此工具包装类的更多信息，请参见参考手册。

ACE_Arg_Shifter

这个 ADT（抽象数据类型）将已知的参数或选项移动到 argv 向量的后面，这样在进行更深层次的参数解析时，就可以在 argv 向量的开始处定位还没有处理的参数。

ACE_Arg_Shifter 将 argv 向量的各个指针拷贝到临时数组中。当 ACE_Arg_Shifter 在临时数组中遍历时，它将已知参数放至 argv 的尾部，未知的放至首部。这样，在访问了临时向量中的所有参数后，ACE_Arg_Shifter 就以所有未知参数的初始顺序把它们放到了 argv 的前面。

这个类对解析来自命令行的选项也很有用。下面的例子将帮助演示这一点：

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

int main(int argc, char *argv[])
{
    ACE_Arg_Shifter arg(argc,argv);
    while(arg.is_anything_left ())
    {
        char *current_arg=arg.get_current();
        if(ACE_OS::strcmp(current_arg,"-region")==0)
        {
            arg.consume_arg();
            ACE_OS::printf("<region>= %s \n",arg.get_current());
        }
        else if(ACE_OS::strcmp(current_arg,"-tag")==0)
```

```
        {
            arg.consume_arg();
            ACE_OS::printf("<tag>= %s \n",arg.get_current());
        }
        else if(ACE_OS::strcmp(current_arg,"-view_uuid")==0)
        {
            arg.consume_arg();
            ACE_OS::printf("<view_uuid>=%s\n",arg.get_current());
        }

        arg.consume_arg();
    }

    for(int i=0;argv[i]!=NULL;i++)
        ACE_DEBUG((LM_DEBUG,”Resultant vector”: %s \n",argv[i]));
}
```

如果这样来运行上面的例子：

```
.../tests<330> arg -region missouri -tag 10 -view_uuid syyid -teacher schmidt -student
tim
```

所获得的结果为：

```
<region> missouri
<tag>= 10
<view_uuid>=syyid
Resultant Vector: tim
Resultant Vector: -student
Resultant Vector: schmidt
Resultant Vector: -teacher
Resultant Vector: syyid
Resultant Vector: -view_uuid
Resultant Vector: 10
Resultant Vector: -tag
Resultant Vector: missouri
Resultant Vector: -region
Resultant Vector: ./arg
```

参考文献

[I] “An introduction to programming with threads”, Andrew Birell, Digital Systems Research Center, 1989.
[II] “Active Object, An object behavioral Pattern for Concurrent Programming”, Douglas C. Schmidt, Greg Lavender. Pattern Languages of Programming Design 2, Addison Wesley 1996.
[III] “A model of Concurrent Computation in Distributed Systems”. MIT Press, 1986.
(Also see: http://www-osl.cs.uiuc.edu/)
[IV] “A Polymorphic Future and First-Class Function Type for Concurrent Object-Oriented Programming in C++”. R.G. Lavender and D.G. Kafura. Forthcoming 1995.
(Also see: http://www.cs.utexas.edu/users/lavender/papers/futures.ps)
[V] “Foundation Patterns”, Dwight Deugo. Proceedings PLoP’ 98.
[VI] “Design Patterns: Elements of reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Addison Wesley Longman Publishing 1995.
[VII] "Hashed and Hierarchical Timing Wheels: data structures to efficiently implement a timer facility", George Varghese. IEEE Trans Networking, Dec 97. Revised from a 1984 SOSP paper; used in X kernel, Siemens and DEC OS etc.
[VIII] “The x-Kernel: An architecture for implementing network protocols”. N. C. Hutchinson and L. L. Peterson. IEEE Transactions on Software Engineering, 17(1):64-76, Jan. 1991.
(Also see http://www.cs.arizona.edu/xkernel)
[IX] “Evaluating Strategies for Real-Time CORBA Dynamic Scheduling”, Douglas Schmidt, Chris Gill and David Levine (updated June 20th), Submitted to the International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware, guest editor Wei Zhao.