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堆和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）。

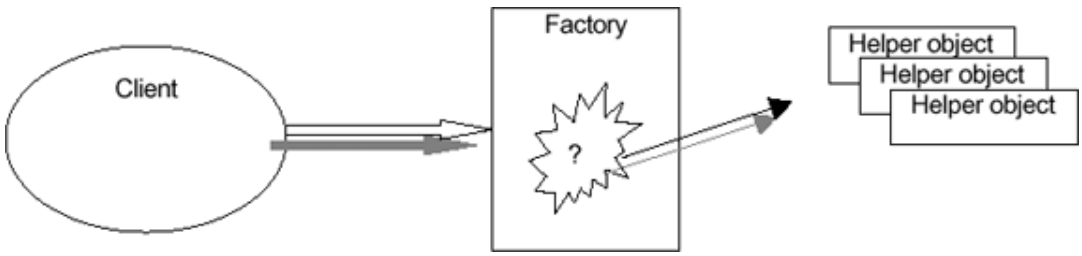

图7-1 工厂模式示意图

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。接受器构架假定服务处理类的类型是ACE_Event_Handler，这是ACE定义的接口类（该类已在反应堆一章中详细讨论过）。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（这是使用反应堆所必需的），同时还基于在ASX流构架中使用的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，并首先实现将被构架自动回调的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。
- **反应堆**：与ACE_Acceptor协同使用。如我们将看到的，在实例化接受器后，我们启动反应堆的事件处理循环。反应堆，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堆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堆。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堆和接受器的特性：

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堆上。因此，我们必须编写handle_input()方法；当数据在连接上到达时，这个方法会被反应堆回调。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，在内部使用反应堆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流构架的一部分）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流构架一起使用；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堆上，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，如果反应堆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堆。因而在接收到任何到来的连接请求时，反应堆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，并被反应堆自动回调。

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会在反应堆上登记自己，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，反应堆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，反应堆自动回调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堆，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，反应堆将回调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；并在反应堆上登记服务处理器，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	先在反应堆上登记服务处理器，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，反应堆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方法来作为反应堆在连接一旦建

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();
}
```