第6章 反应堆（Reactor）：用于事件多路分离和分派的体系结构模式

反应堆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构架。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。反应堆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，反应堆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，反应堆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堆用于对所有这些不同的事件类型进行多路分离。

6.1 反应堆组件

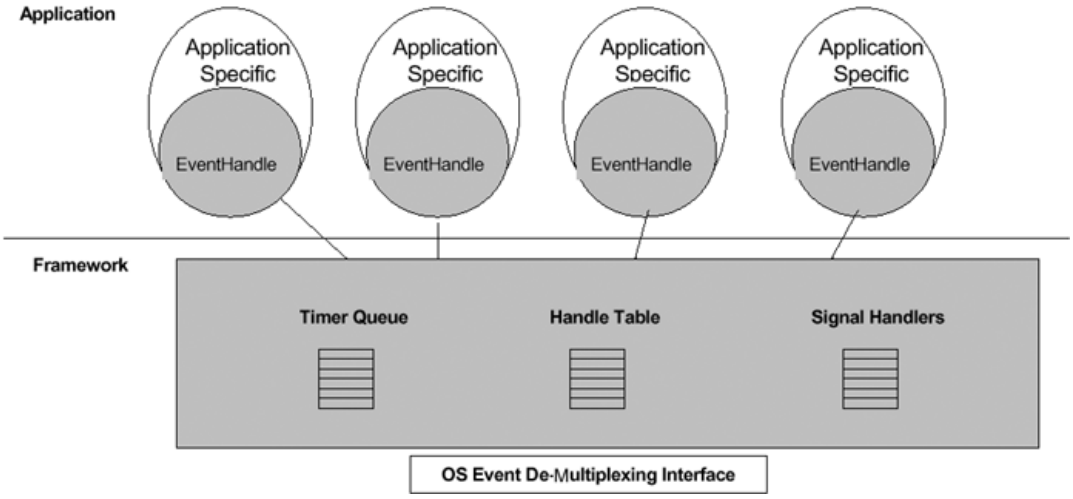

图6-1 反应堆中的内部组件和外部组件的协作

如图6-1所示，ACE中的反应堆与若干内部和外部组件协同工作。其基本概念是反应堆构架检测事件的发生（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. 在反应堆上登记，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堆。

随后反应堆构架将自动地：

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，反应堆发出对处理器中相应方法的回调。

6.2 事件处理器

反应堆模式在ACE中被实现为ACE_Reactor类，它提供反应堆构架的功能接口。

如上面所提到的，反应堆将事件处理器对象作为服务提供者使用。一旦反应堆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，反应堆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。当应用在反应堆上登记它的处理器对象，以处理特定类型的事件时，反应堆会创建这种事件和相应的事件处理器的关联。

因为反应堆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）。该构架提供名为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：

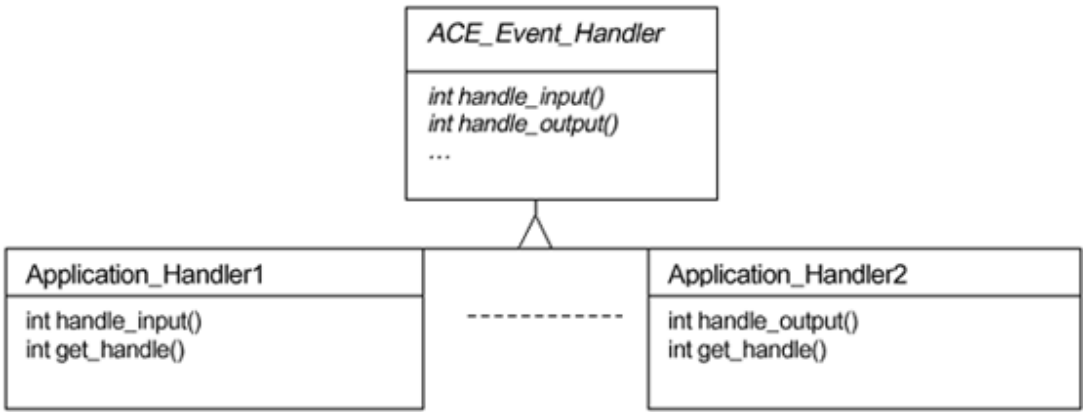

图6-2 ACE_Event_Handler类图

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，随后他就在反应堆上为特定事件“登记”他的事件处理器类。于是反应堆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。）
- 通过调用反应堆对象的register_handler()，将你的事件处理器登记到反应堆。
- 在事件发生时，反应堆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()来启动反应堆的事件循环。无论是发生哪一个事件，反应堆都将自动回调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）来获取全局反应堆对象的引用的。大多数应用都只需要一个反应堆，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。当任何在反应堆上登记的信号发生时，反应堆自动回调该方法。
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，反应堆自动回调该

	方法。
handle_exception()	异常事件。当已在反应堆上登记的异常事件发生时（例如，如果收到SIGURG（紧急信号）），反应堆自动回调该方法。
handle_timeout()	定时器。当任何已登记的定时器超时的时候，反应堆自动回调该方法。
handle_output()	I/O设备输出。当I/O设备的输出队列有可用空间时，反应堆自动回调该方法。

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，是在反应堆上调用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，并将它们加入反应堆的内部分派表。当有事件在handle上发生时，反应堆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堆上。因而，反应堆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，它就不再会被反应堆回调。

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堆上，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堆的分派队列中拆除，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堆里拆除是很重要的，因为，如果不这样做，反应堆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 从反应堆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堆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堆。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堆上的登记。但是，如果“handle_”方法返回的整数<0，反应堆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
}
```

在从反应堆注销、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从反应堆内部分派表中显式拆除事件处理器

另一种从反应堆的内部表中拆除事件处理器的方法是显式地调用反应堆的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堆的内部分派表中拆除。在remove_handler()被调用时，反应堆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 通过反应堆进行事件处理

在下面的几个部分，我们将演示怎样将反应堆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，反应堆可用于处理基于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，反应堆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堆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。

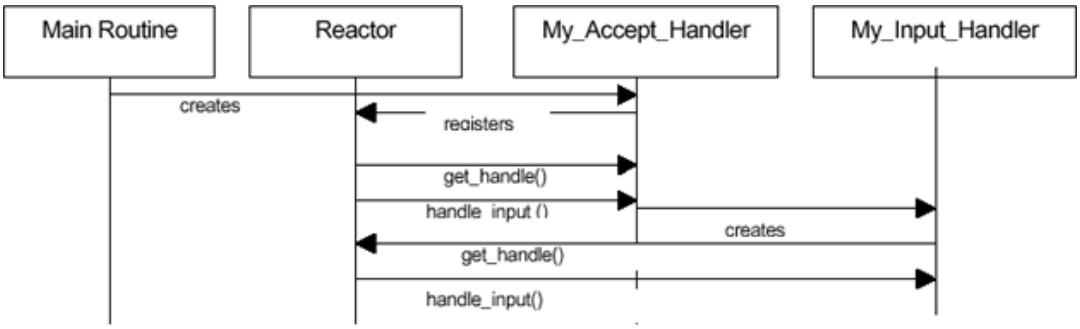

图6-3 反应堆的事件处理

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，它在反应堆上登记，

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()。

当反应堆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。因为反应堆使用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堆。

一旦在该句柄上接收到新的连接请求，反应堆会自动地回调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登记到反应堆，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，使自己从反应堆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，反应堆将自动回调My_Input_Handler::handle_input()。注意在My_Input_Handler的handle_input()方法中，返回给反应堆是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。每个掩码保证反应堆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。反应堆**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.	在反应堆的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 反应堆中的掩码

6.4 定时器（Timer）

反应堆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，反应堆拥有一个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。

反应堆在内部维护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堆的set_timer_queue()方法进行改变。反应堆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堆一起使用定时器。

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，并使用反应堆的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，程序通过调用反应堆的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，反应堆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。反应堆在内部使用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，无需反应堆也可以进行信号处理。

使用反应堆进行信号处理和使用ACE_Sig_Handlers类的重要区别是基于反应堆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）

反应堆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。这是通过反应堆的“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堆对特定的事件处理器对象发出回调。在反应堆与消息队列、或是协作任务协同使用时，这是十分有用的。可在ASX构架组件与反应堆一起使用时找到这种用法的一些好例子。

max_notify_iterations()方法通知反应堆，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。通过使用反应堆的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，反应堆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，它使用反应堆的notify()方法来请求反应堆通知处理器（它是在事件发生时被作为参数传入的）。随后反应堆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，从而使得反应堆回调该处理器的handle_input()方法。

因为我们已将max_notify_iterations设为5，所以在一次handle_events()调用过程中反应堆实际上只会发出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堆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堆线程。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堆，但这一次它们是从新线程、而不是反应堆所在的线程发送的。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。