第7章 ACE反应堆（Reactor）的设计和使用：用于事件多路分离的面向对象构架

Douglas C. Schmidt　　Irfan Pyarali

7.1 介绍

本文关注ACE构架[2]中的反应堆（Reactor）模式[1]的设计和实现。反应堆模式处理由一或多个客户并发地递送给应用的服务请求。应用的每个服务可由一或多个方法组成，并由一个单独的事件处理器代表；事件处理器负责分派服务特有的请求。

在本论文描述的反应堆模式的实现中，事件处理器分派由ACE_Reactor对象来完成。ACE_Reactor结合了I/O事件，以及其他类型的事件，比如定时器和信号的多路分离。在此实现的核心是一个同步的事件多路分离器，例如，select[3]或WaitForMultipleObjects[4]。当事件发生时，ACE_Reactor自动分派预登记的事件处理器的方法，后者随即执行应用指定的服务。

本论文被组织如下：7.2描述ACE_Reactor构架中的主要特性；7.3概述ACE_Reactor实现的OO设计[2]；7.4检查若干例子、演示ACE_Reactor怎样简化并发的事件驱动网络应用的开发；7.5描述使用ACE_Reactor开发事件驱动应用时所应遵循的设计规则；7.6给出结束语。

7.2 反应堆的特性

ACE_Reactor提供OO多路分离和分派构架，它简化了事件驱动应用的开发。下面的段落描述反应堆构架提供给应用开发者的特性：

统一的OO多路分离和分派接口：使用ACE_Reacotr的应用并不直接访问诸如select或WaitForMultipleObjects这样的低级事件多路分离系统调用，而是通过从ACE_Event_Handler基类继承来创建具体的事件处理器。该类指定处理多种类型事件的虚方法，比如I/O事件、定时器事件、信号和同步事件。图7-1演示ACE_Reactor中的关键组件。它演示了实现7.4描述的日志服务器的具体事件处理器。使用反应堆构架的应用创建具体事件处理器，并将它们登记到ACE_Reactor。

使事件处理器分派自动化：当ACE_Reactor管理的处理器上有活动发生时，反应堆自动调用适当的预登记的具体事件处理器的虚方法。C++*对象*被登记到ACE_Reactor。对象、而不是单独的函数的使用，允许在对具体事件处理器的挂钩方法的调用之间，状态可以方便地保持。这种风格的OO编程对于开发在多次客户调用间保持状态的事件处理器来说很有用。

支持透明的可扩展性：ACE_Reactor的功能和它的已登记的事件处理器可被透明地扩展，而*无须*修改或重编译现有代码。为实现这样的扩展性，反应堆构架采用继承和动态绑定来去耦（1）它的较低级的事件多路分离和分派*机制*与（2）应用定义的处理事件的较高级的*策略*。

ACE_Reactor管理的低级机制包括检测多个I/O句柄上的事件、使定时器到期，以及分派适当的事件处理器方法来处理这些事件。应用特有的具体事件处理器执行的较高级策略包括连接建立策略、数据编码和解码，以及处理来自客户的服务请求。例如，TAO[5]实时CORBA ORB使用反应堆构架来分离它的低级事件多路分离机制和它的GIOP连接管理和协议处理[5]的较高级策略。

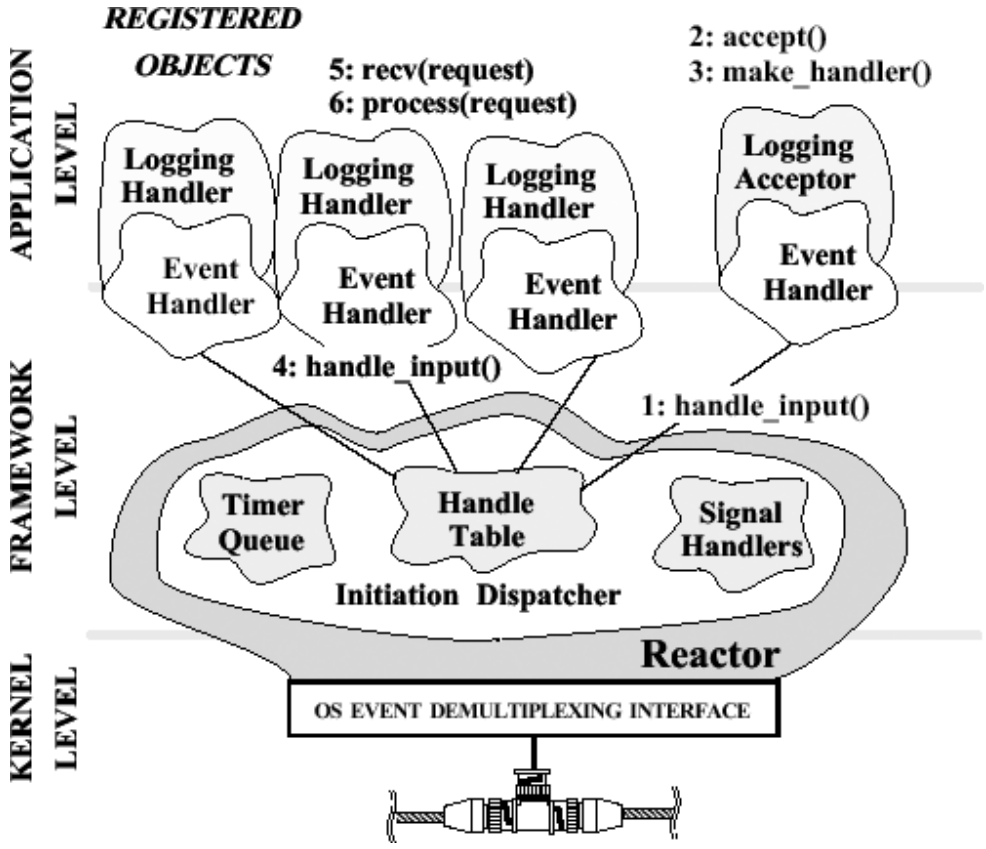

图7-1 反应堆组件

增加复用：ACE_Reactor的多路分离和分派机制可被许多网络应用复用。通过*复用*，而不是*重新发明*，开发者可以专注于较高级的应用特有的策略，而不是反复与较低级事件多路分离和分派机制相纠缠。

相反，直接使用像select和WaitForMultipleObject这样的低级事件多路分离操作的程序员必须为每一个应用重新实现、调试和调谐同样的多路分离和分派代码。而所有利用ACE_Reactor的应用自动地复用它的特性，以及将来的增强和优化。

增强类型安全性：ACE_Reactor将应用开发者和易错的细节（它们与使用像select这样的低级事件多路分离系统调用来编程是相关联的）屏蔽开来。这些细节涉及设置和清除位掩码、处理超时和中断，以及分派挂钩方法。特别地，ACE_Reacotr消除了导致select出错的若干微妙原因，这些错误涉及fd_set位掩码的误用。

改善可移植性：ACE_Reactor在若干事件多路分离机制上运行，其中包括Win32 WaitForMultipleObjects和select（它在UNIX和Win32上都可用）。ACE_Reactor将应用和底层事件多路分离机制的移植性差异屏蔽开来。如图7-6所示，ACE_Reactor向应用输出同样的接口，而不管底层系统调用是什么。而且，ACE_Reactor使用像桥接（Bridge）[6]这样的设计模式来增强它内部的可移植性。因而，将ACE_Reactor从select移植到WaitForMultipleObjects仅需要对构架进行一些局部变动[7]。

线程安全性：反应堆构架是完全多线程的。所以多个线程可安全地共享单个ACE_Reactor。同样地，多个ACE_Reactor也可在一个进程的不同线程中运行。反应堆构架提供必要的同步机制来防止条件竞争和类中方法的死锁[8]。

高效的多路分离：ACE_Reactor非常高效地执行它的事件多路分离逻辑。例如，基于select的ACE_Reactor使用7.3.2中描述的ACE_Handle_Set类来避免每次只查看fd_set位掩码的一位。这一优化基于一种成熟的算法，使用异或操作符来将运行时复杂度从O（*总位数*）降低到O（*已置位的位数*）。这充分地减少了运行时开销。

7.3 反应堆构架的OO设计

这一部分描述反应堆构架的OO设计。主要焦点在于它的组件的结构和关键的设计决策。在适当的地方还讨论了实现细节。7.3.1概述OS平台无关的组件，而7.3.2覆盖平台相关的组件。

7.3.1 平台无关的类组件

这一小部分总结反应堆构架中的平台无关的类，其中包括ACE_Reactor、ACE_Time_Value、ACE_Timer_Queue和ACE_Event_Handler。

7.3.1.1 ACE_Reactor类

ACE_Reactor定义反应堆构架的公共接口。图7-2演示ACE_Reactor类中关键的公共方法。此类中的方法大致被分组为以下几个种类：

管理器方法：构造器和open方法通过动态地分配多个实现对象来创建和初始化ACE_Reactor对象；这些实现对象在下面的7.3.2.1和7.3.2.2中描述。析构器和close方法释放这些对象。

基于I/O的方法：派生自ACE_Event_Handler的具体事件处理器通过反应堆的register_handler方法登记到ACE_Reactor；同样地，具体事件处理器可以通过它的remove_handler方法移除。

基于定时器的方法：ACE_Reactor的定时器策略使用最终期限来评估被调度的定时器的优先级。ACE_Reactor的定时器策略提供的操作包括（1）登记将在用户指定时间执行的具体事件处理器和（2）取消先前登记的事件处理器。

事件循环方法：在登记初始的具体事件处理器之后，应用最终进入一个事件循环，重复调用ACE_Reactor的handle_events方法中的一个。这些方法阻塞应用指定长度的时间，等待各种事件的发生，比如I/O句柄上的同步I/O事件或基于定时器的事件。在事件发生时，ACE_Reactor分派具体事件处理器的适当方法；这些方法已被应用登记以处理这些事件。

```
class ACE_Reactor

{

public:

// = Initialization and termination methods.

enum { DEFAULT_SIZE = FD_SETSIZE };


// Initialize a Reactor instance that may

// contain SIZE entries (<restart> indicates

// to restart system calls after interrupts).

ACE_Reactor (int size, int restart = 0);

virtual int open (int size = DEFAULT_SIZE,  int restart = 0);


// Perform cleanup activities to close down

// an instance of an <ACE_Reactor>.

void close (void);

?ACE_Reactor (void);


// = I/O-based event handler methods.

// Register an <ACE_Event_Handler> object according

// to the <ACE_Reactor_Mask>(s) (e.g., READ_MASK,

// WRITE_MASK, etc.).

virtual int register_handler (ACE_Event_Handler *, ACE_Reactor_Mask);


// Remove the handler associated with the

// appropriate <ACE_Reactor_Mask>(s).

virtual int remove_handler (ACE_Event_Handler *, ACE_Reactor_Mask);
```

```
// = Timer-based event handler methods.
// Register a handler to expire at time <delta>.
// When <delta> expires the <handle_timeout>
// method will be called with the current time
// and <act> as parameters. If <interval> is > 0
// then the handler is reinvoked periodically
// at that <interval>. The <delta> is interpreted
// "relative" to the current time of day.
virtual void schedule_timer
       (ACE_Event_Handler *,
       const void *act,
       const ACE_Time_Value &delta,
       const ACE_Time_Value &interval);

// Locate and cancel timer.
virtual void cancel_timer (ACE_Event_Handler *);

// = Event-loop methods
// Block process until I/O events occur or timer
// expires, then dispatch activated handler(s).
virtual int handle_events (void);

// Perform a timed event-loop that waits up to TV
// time units for events to occur; if no events
// occur then 0 is returned, otherwise return
// TV - (actual_time_waited).
virtual int handle_events (ACE_Time_Value &tv);

private:
// Pointer to the implementation class,
// e.g., <ACE_Select_Reactor> or
// <ACE_WFMO_Reactor>.
Reactor_Impl *reactor_impl_;
};
```

图7-2 ACE反应堆接口

7.3.1.2 ACE_Event_Handler类

该基类指定ACE_Reactor用以控制和协调具体事件处理器的多路分离和分派接口。ACE_Event_Handler接口中的虚方法如图7-3所示。

```
// Handle portability issues.
#if defined (UNIX)
typedef int ACE_HANDLE;
#elif defined (WIN32)
typedef HANDLE ACE_HANDLE;
#endif /* UNIX */
class ACE_Event_Handler
{
public:
// These values can be bitwise "or'd" together to
// instruct the <ACE_Reactor> to check for
// multiple I/O activities on a single handle.
enum {
       READ_MASK = 01,
       WRITE_MASK = 02,
       EXCEPT_MASK = 04,
       RWE_MASK = READ_MASK | WRITE_MASK | EXCEPT_MASK,
       DONT_CALL // Don't callback to handle_close().
};

// Returns the I/O handle associated with the
// derived object (must be supplied by a subclass).
virtual ACE_HANDLE get_handle (void) const = 0;

// Called when event handler is removed from
// an <ACE_Reactor>.
virtual int handle_close (ACE_HANDLE, ACE_Reactor_Mask);

// Called when input becomes available.
```

```
virtual int handle_input (ACE_HANDLE);


// Called when output is possible.

virtual int handle_output (ACE_HANDLE);


// Called when urgent data is available.

virtual int handle_except (ACE_HANDLE);


// Called when timer expires (<tv> stores the

// current time and <act> is the argument given

// when the handler was originally scheduled).

virtual int handle_timeout (const ACE_Time_Value &tv, const void *act = 0);


// Called when signal is triggered by OS.

virtual int handle_signal (int signum);

};
```

图7-3 ACE事件处理器接口

ACE_Reactor通过派生自ACE_Event_Handler的具体事件处理器来实现它的事件驱动回调机制。具体事件处理器可以有选择地重定义ACE_Event_Handler中的虚方法，以执行应用定义的处理来响应多种类型的事件。这些事件包括（1）同步I/O，例如，*读*、*写*和*异常*；（2）定时器；（3）信号；以及（4）同步，例如，将Win32互斥体从复位切换到置位[4]。

7.3.1.4中描述的ACE_Timer_Queue使用具体事件处理器来处理基于定时器的事件。当此队列管理的定时器到期时，先前排定的（scheduled）事件处理器的handle_timeout方法被调用。有两个参数被传递给该方法：（1）当前时间和（2）void * *异步完成令牌*（ACT）[9]，当事件处理器最初被排定时，此令牌也作为参数被传递给schedule_timer。

当具体事件处理器中的任何方法的返回值<0时，反应堆都自动调用处理器的handle_close清扫方法。应用可对该方法进行编程以执行终止活动，比如关闭日志文件或删除对象分配的动态内存。当handle_close方法返回时，ACE_Reactor从它的内部表中将与之相关联的具体事件处理器移除掉。

具体事件处理器通常要提供一个I/O句柄。当应用调用ACE_Reactor的register_handler方法时，该方法回调具体事件处理器的get_handle方法，以获取底层的I/O句柄。

当应用调用ACE_Reactor的handle_events方法时，所有已登记的具体事件处理器的句柄都被传递给底层的OS事件多路分离调用，例如，select或WaitForMultipleObjects。随后，当I/O事件变为“就绪”时，OS就*激活*这些句柄。在此时，ACE_Reactor通过调用被定义用来处理该事件的方法，通知适当的具体事件处理器。

7.3.1.3 ACE_Time_Value类

该C++包装封装底层OS平台的日期和时间结构，例如，在大多数UNIX平台上定义的struct timeval类型。timeval结构包括两个域，根据*秒*和*微秒*来表示时间。其他OS平台，比如POSIX和Win32，使用稍稍不同的时间表示法。为此，ACE_Time_Value类封装这些细节以提供可移植的C++接口。

ACE_Time_Value类的主要方法在图7-4中演示。ACE_Time_Value包装使用操作符重载来简化基于时间的比较。这样的重载允许标准的算术语法被用于涉及时间比较的关系表达式。

```
// Time value structure from /usr/include/sys/time.h
// struct timeval { long secs; long usecs; };
class ACE_Time_Value
{
public:
       ACE_Time_Value (long sec = 0, long usec = 0);
ACE_Time_Value (timeval t);

// Returns sum of two <ACE_Time_Value>s.
friend ACE_Time_Value operator +
       (const ACE_Time_Value &lhs,
       const ACE_Time_Value &rhs);

// Returns difference between two
// <ACE_Time_Value>s.
friend ACE_Time_Value operator -(
       const ACE_Time_Value &lhs,
       const ACE_Time_Value &rhs);

// Relational and comparison operators for
// normalized <ACE_Time_Value>s.
friend int operator <
       (const ACE_Time_Value &lhs,
       const ACE_Time_Value &rhs);

// Other relation operators...

private:
// ...
```

```
};
```

图7-4 ACE时间值接口

ACE_Time_Value的方法被实现用来“规格化”时间数量。规格化调整timeval结构中的两个域，使之使用能确保精确比较的规范的编码方式。例如，在规格化之后，数量ACE_Time_Value(1, 1000000)将与ACE_Time_Value(2)相等。相反，直接按位比较这些非规格化的类值将检测不到它们的相等。

下面的代码创建两个ACE_Time_Value对象，它们由用户提供的命令行参数加上当前时间来构造。随后显示两个对象之间的正确顺序关系：

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

{

if (argc != 3)

       ACE_ERROR_RETURN ((LM_ERROR, "usage: %d" "time1 time2\n"), 1);


ACE_Time_Value time = ACE_OS::gettimeofday ();

ACE_Time_Value timer1 = time + ACE_Time_Value (ACE_OS::atoi (argv[1]));

ACE_Time_Value timer2 = time + ACE_Time_Value (ACE_OS::atoi (argv[2]));


if (timer1 > timer2)

       ACE_DEBUG ((LM_DEBUG, "timer 1 is greater\n"));

else if (timer2 > timer1)

       ACE_DEBUG ((LM_DEBUG, "timer 2 is greater\n"));

else

       ACE_DEBUG ((LM_DEBUG, "timers are equal\n"));


return 0;

}
```

上面所示的代码对所有OS平台都是完全可移植的。注意像操作符重载和类这样的C++特性的使用是怎样简化与时间相关操作的使用的。

7.3.1.4 ACE_Timer_Queue类

ACE_Reactor的基于定时器的机制对于需要定时器支持的应用来说是很有用的。例如，WWW服务器需要看门狗定时器来释放资源，如果客户在它们连接后不在指定的时间间隔内发送HTTP请求的话。同样

地，像Windows NT Service Control Manager[4]这样的看守配置构架要求在它们控制之下的服务周期性地报告它们的当前状态。这些“心跳”消息用于确认服务没有异常地终止。

ACE_Timer_Queue类提供的机制允许应用登记派生自ACE_Event_Handler的、基于定时器的具体事件处理器。ACE_Timer_Queue确保这些事件处理器中的handle_timout方法将来在应用指定的时间被调用。图7-5中所示的ACE_Timer_Queue类的方法使得应用可以调度、取消，和调用定时器对象。

```
class ACE_Timer_Queue

{

public:

ACE_Timer_Queue (void);


// True if queue is empty, else false.

int is_empty (void) const;


// Returns earliest time in queue.

const ACE_Time_Value &earliest_time (void) const;


// Schedule a <handler> to be dispatched at

// the <future_time> and at subsequent

// <interval>s.

int virtual schedule

       (ACE_Event_Handler *handler,

       const void *act,

       const ACE_Time_Value &future_time,

       const ACE_Time_Value &interval);


// Cancel all registered <ACE_Event_Handlers>

// that match the address of <handler>, which

// can be registered multiple times.

int virtual cancel(ACE_Event_Handler *handler);


// Cancel the single <ACE_Event_Handler>

// matching the <timer_id> value returned

// from <schedule>.

int virtual cancel (int timer_id, const void **act = 0);
```

```
// Expire all timers <= <expire_time>

// (note, this method must be called manually

// since it is not invoked asynchronously).

void virtual expire(const ACE_Time_Value &expire_time);



private:

// ...

};
```

图7-5 ACE_Timer_Queue接口

应用调度具体事件处理器，在延迟一定数量的时间之后到期。如果它到期了，act作为值被传递给事件处理器的handle_timeout挂钩方法。如果interval不等于ACE_Time_Value::zero，该值就被用于自动重调度事件处理器。

Schedule方法返回一个定时器id，唯一地标识每个事件处理器在定时器队列的内部表中的登记。定时器id被cancel方法用于在事件处理器到期之前将其移除。如果一个非NULL act被传给cancel，这个act就被设置为应用在定时器最初被排定时所传入的*异步完成令牌*（ACT）[9]。 这使得程序可以释放动态分配的ACT，以避免内存泄漏。

缺省地，ACE_Reactor所用的ACE_Timer_Queue被作为堆来实现。堆是一种“部分有序的、几乎完全的二进制树”，它确保插入和删除一个具体事件处理器的平均和最坏情况的时间复杂度为O(lg n)。堆表示法对大多数应用实例、特别是实时应用来说都是很适用的。

ACE_Timer_Queue堆由含有ACE_Time_Value、ACE_Event_Handler *，和void *的三元组组成。ACE_Event_Handler * 域指向被排定的定时器对象，该对象的运行时间由ACE_Time_Value域指定。void * 域是在具体事件处理器被最初被排定时所提供的参数。当定时器到期时，这个参数被自动传给7.3.1.2中描述的handle_timeout方法。堆中的每个ACE_Time_Value都以*绝对*时间单元来存储，例如，按照UNIX gettimeofday系统调用所生成的时间。

在ACE_Timer_Queue接口中使用了虚方法。因而，应用可以扩展缺省的ACE实现来支持其他可选的数据结构，如*delta表*[11]和*定时轮*（timing wheel）[12]。delta表将时间存储为“相对的”单元，即相对于表的最前面的ACE_Time_Value的偏移或“delta”。定时轮使用循环缓冲区，使得程序有可能在O(1)时间内启动、停止和维护定时器。ACE构架提供了若干可选的定时器队列的实现。

7.3.2 平台相关的类组件

ACE_Reactor类是应用用以访问ACE反应堆构架的公共接口。ACE_Reactor接口由一些虚方法组成。因此，它可以通过继承来扩展。

但是，扩展ACE_Reactor最常用的方法并不是派生它的子类。相反，如图7-6所示，桥接模式[6]被用来使ACE_Reactor接口与它的ACE_Reactor_Impl子类实现去耦合。ACE构架提供的两个子类包括ACE_Select_Reactor和ACE_WMFO_Reactor，它们分别封装了select和WaitForMultipleObjects OS事件多路分离调用。

在不同OS平台上，ACE_Reactor_Impl子类的实现也不同。但是，ACE_Reactor接口提供的方法的名字和总的功能保持不变。这种统一性源于ACE_Reactor的设计的模块性，该设计还增强了反应堆的复用、可移植性和可维护性。ACE_Reactor的WaitForMultipleObjects和select版本概述如下。

7.3.2.1 ACE_Select_Reactor类

如图7-6(1)所示，基于select的ACE_Reactor包含有三个ACE_Event_Handler * 数组。这些数组存储的指针指向已登记用来处理各种类型事件的具体事件处理器。

ACE_Handle_Set类为底层的fd_set位掩码数据类型提供高效的C++包装。fd_set将I/O句柄名字空间映射到紧凑的位向量表示，并提供若干操作来置位、复位和测试与I/O句柄相对应的位。可传给select调用一或多个fd_set。

ACE_Handle_Set类通过以下方法来优化若干常用的fd_set操作：（1）使用“全字”（full-word）比较，以使不必要的位操作最少化，（2）缓存某些值，以避免每次调用都计算位偏移，以及（3）使用一种异或算法，它与fd_set中的活动句柄的数目、而非*潜在的*活动句柄的数目线性相关。

7.3.2.2 ACE_WMFO_Reactor类

WaitForMultipleObjects接口比select更为通用，它允许应用等待更广泛的事件，比如同步事件。因此，基于WaitForMultipleObjects的ACE_Reactor既不需要三个ACE_Event_Handler * 数组，也不需要ACE_Handle_Set类，而是在内部分配和使用单个的ACE_Event_Handler指针数组和句柄数组，以存储已登记的具体事件处理器。

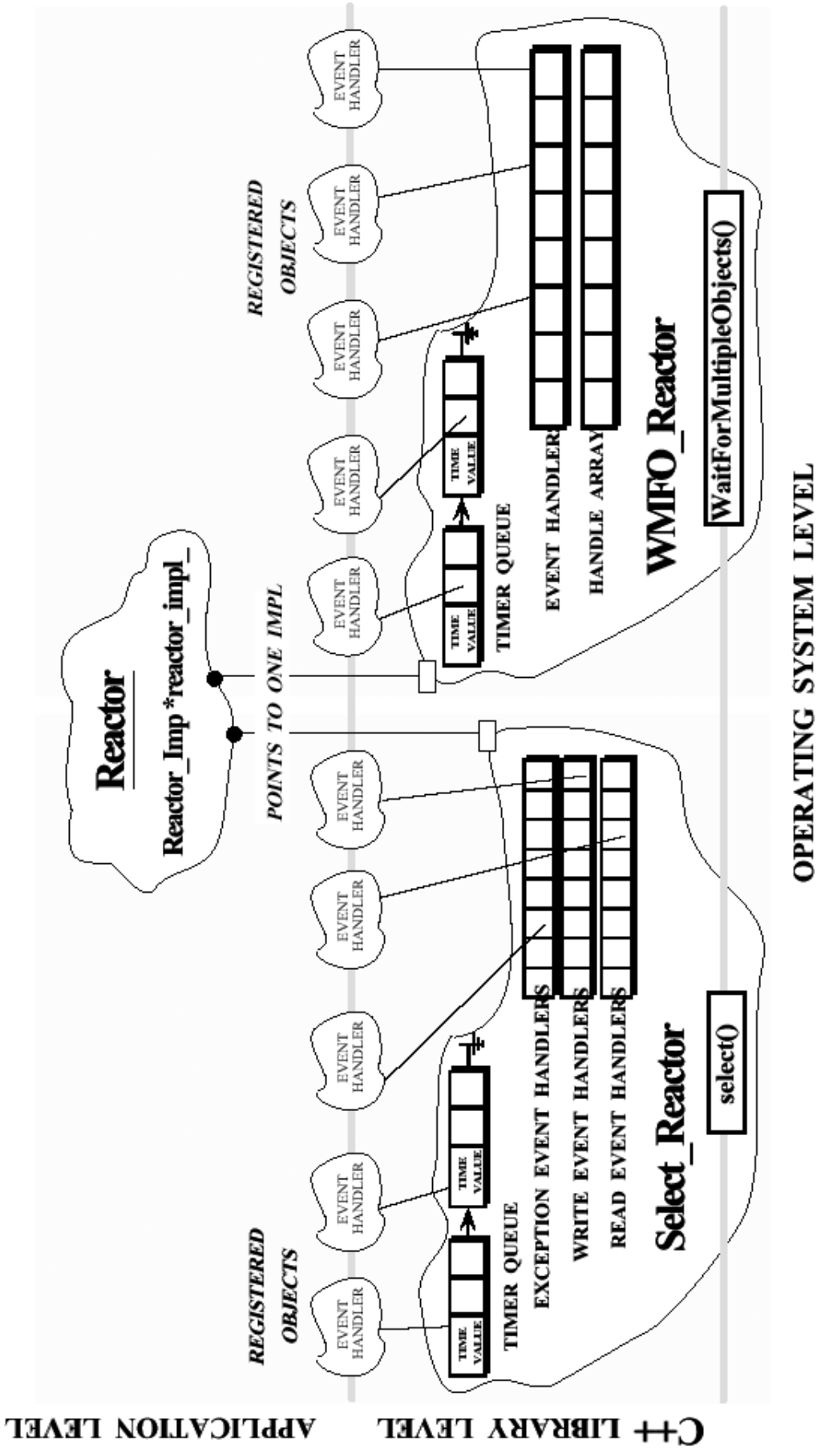

图7-6 将桥接模式用于反应堆实现

7.4 分布式日志服务例子

反应堆构架意在简化事件驱动应用的开发，比如Web服务器[13, 14]和CORBA对象请求代理[15]。这一部分描述一个分布式日志服务（distributed logging service）的设计和实现，以演示ACE_Reactor在实践中的应用。

7.4.1 综述

日志提供一种“只追加”的存储服务，记录由一或多个应用发来的诊断信息。日志的主要单元是*记录*。到来的记录被追加到日志的末尾，而所有其它类型的写访问都是被禁止的。

如图7-7所示，下面检查的日志服务使用客户/服务器体系结构，以使通过TCP/IP网络连接的工作站和服务器能够记录事件。日志服务将ACE_Reactor的多路分离与分派特性和[16]中描述的C++ IPC包装库提供的BSD sockets与系统V传输层接口（TLI）的OO接口结合在了一起。

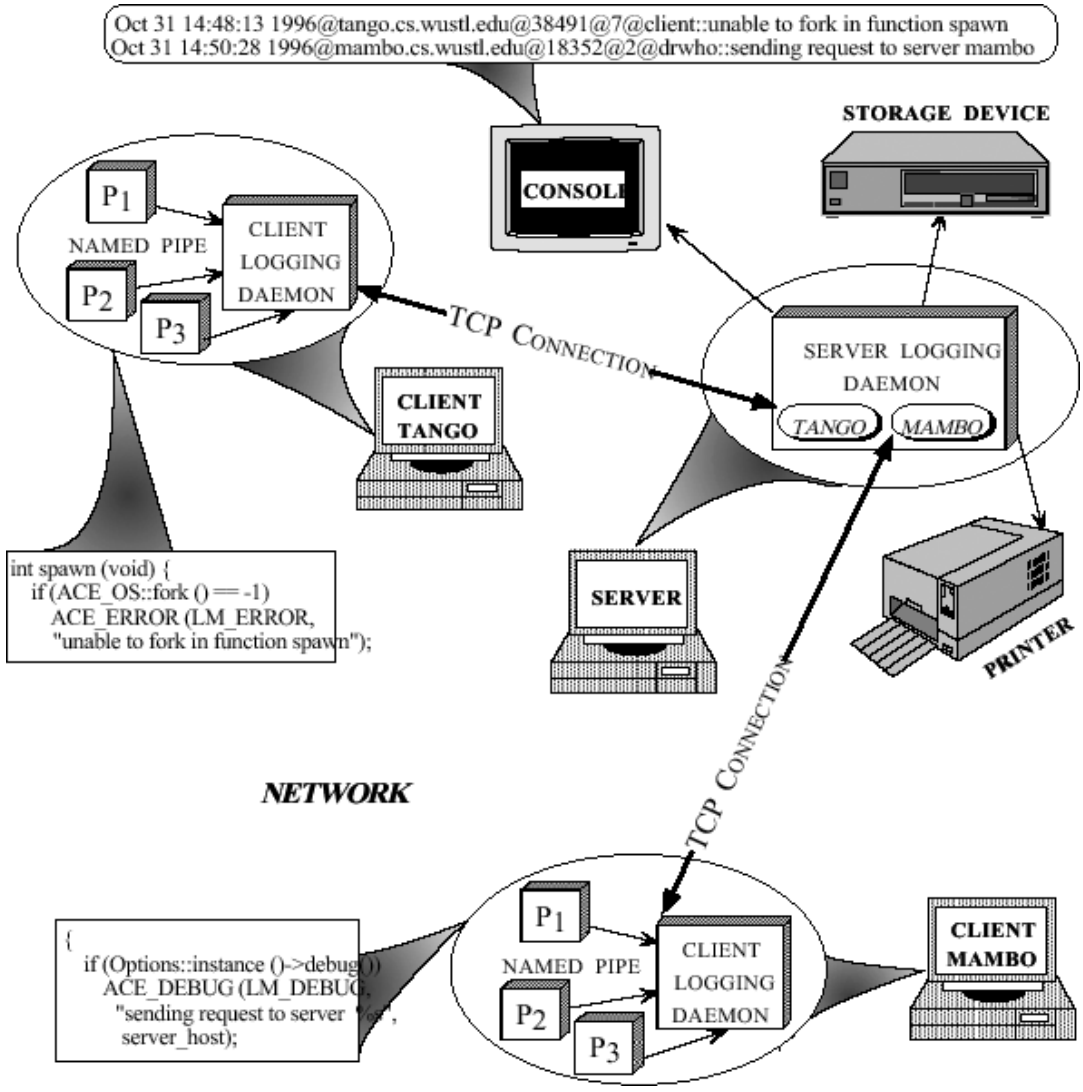

图7-7 分布式日志服务中的组件

日志服务中的关键组件描述如下：

应用日志接口（Application logging interface**）：**在客户主机上运行的应用进程，例如，P1、P2、P3，使用ACE_Log_Msg C++类来生成日志记录，比如LM_ERROR和LM_DEBUG。ACE_Log_Msg::log方法提供一种printf风格的接口。图7-8概述了在应用接口、客户和服务器日志看守之间交换的记录的若干优先级和数据格式。当被应用调用时，日志接口格式化日志记录，打上时间戳，将它们写到周知的STREAM管道[17]中去。如下所述，*客户日志看守*负责处理这些记录。

```
// The following enum indicates the relative

// priorities of the logging messages.

enum Log_Priority

{

// Messages that contain information normally

// used only when debugging a program.

LM_DEBUG,


// Critical conditions, e.g., hard device errors

LM_ERROR


// ...

};


struct Log_Record

{

enum {

       // Maximum number of bytes in logging record.

       MAXLOGMSGLEN = 1024

};


// Type of logging record.

Log_Priority type_;


// length of the logging record.

long length_;


// Time logging record generated.
```

```
long time_stamp_;


// Id of process that generated the record.
long pid_;


// Logging record data.
char rec_data_[MAXLOGMSGLEN];
};
```

图7-8 日志记录格式

客户日志看守（Client logging daemon**）：**客户日志看守运行在所有参与分布式日志服务的主机上。每个客户日志看守与STREAM管道的读端相连，后者用于从这台机器上的应用那里接收日志记录。使用STREAM管道是因为它们是只用于本地主机的IPC的高效形式。此外，STREAM管道的语义允许“结合优先级”的消息，可按照“重要性顺序”，以及“到达顺序”接收[18]。

客户日志看守从应用进程那里持续地接收日志记录。它随后将多字节的记录头字段转换为网络字节顺序。最后，它使用TCP将记录转发给*服务器日志看守*。服务器通常运行在远地主机上。

服务器日志看守（Server logging daemon**）：**服务器日志看守持续地收集、重格式化和输出到来的日志记录。这一部分的余下部分专注于服务器日志看守。在此例中演示和描述了多种ACE_Reactor和ACE C++ socket包装机制。

7.4.2 服务器日志看守

下面描述用来构造服务器日志看守的类的接口和实现。日志服务器在单独的进程中运行，并发地处理来自客户的日志记录。并发是由ACE_Reactor提供的，它将它的注意力以循环方式“分时”给每个活动的客户。

每次应用调用ACE_Reactor的handle_events方法，就从每个I/O句柄变为活动的客户那里读取一条日志记录。日志记录被写到服务器日志看守的标准输出。该输出可被重定向到多种设备，比如打印机、持久存储仓库或日志管理控制台。

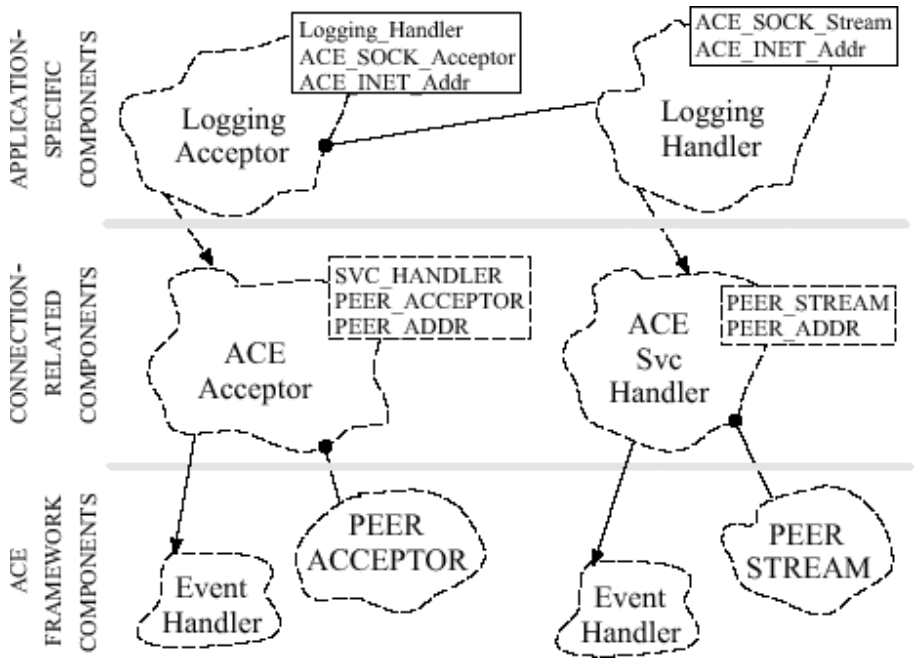

图7-9 服务器日志看守中的组件

有若干C++类组件出现在日志服务体系中。在图7-9中使用Booch表示法[19]演示了多种组件间的继承和模板参数化关系。为增强复用和可扩展性，图中所示的组件被设计用以使服务器日志看守体系结构的以下方面去耦合：

反应堆构架组件（Reactor framework component**）：**反应堆构架中的组件封装执行I/O多路分离和具体事件处理器分派的最底层机制。这些组件已在7.3中讨论。

连接相关组件（Connection-related component**）：**这些通用模板实现接受器（Acceptor）模式[20]，提供可复用的连接工厂组件。ACE_Acceptor是一个接受来自远地客户的网络连接、创建ACE_Svc_Handler的模板。ACE_Svc_Handler是与相连的客户交换数据的模板。这些组件在7.4.2.1中讨论。

应用特有组件（Application-specific component**）：**这些组件实现分布式日志服务的应用特有的部分。Logging_Acceptor类给ACE_Acceptor提供具体的参数化类型，后者创建专用于日志应用的连接处理实例。同样地，Logging_Handler类也通过具体类型来实例化，这种具体类型提供必需的应用特有的功能，以接收和处理来自远地客户的日志记录。这些组件在7.4.2.2中讨论。

使用这样的高度去耦合的OO分解极大地增强了服务器日志看守的开发和可扩展性。这些组件的每一个被描述如下。

7.4.2.1 连接相关组件

下面的类用于实现Acceptor模式[20]。该模式用于使（1）*被动*连接建立与（2）一旦服务的两端连接和初始化后、服务所进行的处理去耦合。

ACE_Acceptor**类：**该类为一族类提供一种通用模板，使从客户接受网络连接请求所必需的步骤标准化和自动化。图7-10演示了ACE_Acceptor类的接口。

```
// A template class that handles connection

// requests from a remote client.


template <class SVC_HANDLER,

class PEER_ACCEPTOR>

class ACE_Acceptor : public ACE_Event_Handler

{

public:

ACE_Acceptor (ACE_Reactor *r, const PEER_ACCEPTOR::PEER_ADDR &a);

~ACE_Acceptor (void);

protected:

virtual ACE_HANDLE get_handle (void) const;

virtual int handle_input (ACE_HANDLE);

virtual int handle_close

       (ACE_HANDLE = ACE_INVALID_HANDLE,

       ACE_Reactor_Mask = ACE_Event_Handler::READ_MASK);


private:

// Accept connections.

PEER_ACCEPTOR acceptor_;

};
```

图7-10 Acceptor类接口

ACE_Acceptor模板类继承自ACE_Event_Handler。这一派生使得ACE_Acceptor可以与反应堆构架无缝地交互。此外，该模板类通过具体的SVC_HANDLER（它知道怎样执行与客户的I/O）和PEER_ACCEPTOR类来参数化（它知道怎样接受客户连接）。

从ACE_Acceptor实例化的类有能力完成以下工作：

1. 接受远地客户发送的连接请求。
2. 动态分配SVC_HANDLER子类的对象。
3. 将此对象登记到ACE_Reactor的一个实例。随后，SVC_HANDLER类必须知道怎样处理与客户交换的数据。

ACE_Acceptor类的实现如图7-11所示。当一或多个连接请求到达时，handle_input方法被Reactor自动分派。该方法的行为如下：首先，它动态创建新的SVC_HANDLER对象，负责处理发送数据和接收来自新客户的数据。其次，它将一个到达的连接接受进SVC_HANDLER。最后，它调用新SVC_HANDLER的open挂钩。如下所示，该挂钩可以将新创建的SVC_HANDLER登记到ACE_Reactor，或是生成一个独立的线程控制，等等。

```
// Shorthand names

#define SH SVC_HANDLER

#define PA PEER_ACCEPTOR


template <class SH, class PA>

ACE_Acceptor<SH, PA>::ACE_Acceptor

(ACE_Reactor *reactor,

const PA::PEER_ADDR &addr)

: acceptor_ (addr)

{

// Register to accept connections.

reactor->register_handler

       (this,

       ACE_Event_Handler::ACCEPT_MASK);

}


template <class SH, class PA> ACE_HANDLE

ACE_Acceptor<SH, PA>::get_handle (void) const

{

// Return the underlying I/O handle

// when called by Reactor during

// registration.

return this->acceptor_.get_handle ();

}


template <class SH, class PA> int
```

```
ACE_Acceptor<SH, PA>::handle_close
(ACE_HANDLE, ACE_Reactor_Mask)
{
// Close down the Acceptor and
// release the handle resources.
return this->acceptor_.close ();
}

template <class SH, class PA>
ACE_Acceptor<SH, PA>::?ACE_Acceptor (void)
{
this->handle_close ();
}

// Template Method that accepts connections
// from client hosts, creates and activates
// a service handler.

template <class SH, class PA> int
ACE_Acceptor<SH, PA>::handle_input
(ACE_HANDLE)
{
// Create a new Svc_Handler.
SH *svc_handler = new SH;

// Accept connection into the handler.
this->acceptor_.accept (svc_handler->peer ());

// Activate the handler.
svc_handler->open (0);
}
```

图7-11 Aceptor类实现

ACE_Svc_Handler类：该参数化类型为处理与客户交换的数据提供一种通用模板。例如，在分布式日志服务中，I/O格式牵涉到日志记录。但是，也可以很容易地为其他应用换用不同的格式。ACE_Svc_Handler类的接口在图7-12中描述。和ACE_Acceptor类一样，该类继承ACE_Event_Handler基类

的功能。这使得从ACE_Svc_Handler实例化的具体事件处理器可被动态创建并登记到ACE_Reactor。ACE_Acceptor类中的handle_input方法自动完成这一行为。

```
// Receive client message from the remote clients.
template <class PEER_STREAM>
class ACE_Svc_Handler : public ACE_Event_Handler
{
public:
ACE_Svc_Handler (void);

// Must be filled in by subclass
virtual int open (void *) = 0;
PEER_STREAM &peer (void);

// Demultiplexing hooks.
virtual ACE_HANDLE get_handle (void) const;

protected:
// Connection open to the client.
PEER_STREAM peer_stream_;
};
```

图7-12 Svc_Handler类接口

图7-13演示ACE_Svc_Handler类实现。注意继承、动态绑定和参数化类型的结合是怎样使构架的通用部分（例如，连接建立）与应用特有的功能（例如，接收日志记录）去耦合的。

```
#define PS PEER_STREAM

// Extract the underlying PS (e.g., for
// use by accept()).

template <class PS> PS &
```

```
ACE_Svc_Handler<PS>::peer (void)

{

return this->peer_stream_;

}


template <class PS> ACE_HANDLE

ACE_Svc_Handler<PS>::get_handle (void) const

{

// Return the underlying I/O handle

// when called by Reactor during

// registration.

return this->peer_stream_.get_handle ();

}
```

图7-13 Svc_Handler类实现

当ACE_Reactor被指示从它的内部表中移除ACE_Svc_Handler时，它自动调用具体事件处理器的handle_close方法。缺省地，该方法释放处理器的内存；该内存原来是由ACE_Acceptor类中的handle_input方法分配的。具体事件处理器通常在客户进程关闭或发生严重传输错误时被移除。

7.4.2.2应用特有的服务

下面的类实现服务的应用特有部分，在此例中也就是日志服务器看守。

Logging_Acceptor类：为实现分布式日志应用的服务器看守部分，Logging_Acceptor类从通用的ACE_Acceptor模板实例化，如下所示：

```
typedef ACE_Acceptor <Logging_Handler, ACE_SOCK_Acceptor>Logging_Acceptor;
```

SVC_HANDLER模板参数通过下面描述的Logging_Handler类实例化。同样地，PEER_ACCEPTOR模板参数被ACE_SOCK_Acceptor类替换。ACE_SOCK_* 实例化类型是称为SOCK_SAP的C++包装的一部分[16]。SOCK_SAP封装socket接口，以在客户和服务器上的进程间可靠地传输数据。

通过使用参数化类型，用于IPC的类可以是任何遵从参数化类中所用API的网络编程接口。例如，取决于底层OS平台的特定属性（比如是UNIX的BSD还是系统V变种），日志应用可以使用SOCK_SAP或TLI_SAP（后者是系统V传输层接口（TLI）的ACE C++包装）来实例化ACE_Svc_Handler类。这种技术在下面演示：

```
// Logging application.


#if defined (USE_SOCKETS)

typedef ACE_SOCK_Stream PEER_STREAM;

#elif defined (USE_TLI)

typedef ACE_TLI_Stream PEER_STREAM;

#endif /* USE_SOCKETS */


class Logging_Handler

: public ACE_Svc_Handler<PEER_STREAM>

{

// ...

};
```

在开发必须跨越多种OS平台运行的应用时，基于模板的可扩展性所提供的灵活性十分有用。事实上，通过传输接口来参数化应用的能力对于跨越OS平台的变种也很有用。例如，Solaris的某些版本不提供线程安全的socket实现。

Logging_Handler**类：**该类通过实例化ACE_Svc_Handler类创建：

```
class Logging_Handler :

public ACE_Svc_Handler<ACE_SOCK_Stream>

{

public:

// Initialization hook called by

// the <ACE_Acceptor>.

virtual int open (void *)

{

       ACE_SOCK_Stream::PEER_ADDR addr;


       // Cache remove host name.

       peer ().get_remote_addr (addr);

       ACE_OS::strcpy (host_name_,

                                addr.get_host_name ());


       // Register ourselves with the Reactor so
```

```
        // we can be dispatched automatically when

        // I/O arrives from clients.

        ACE_Reactor::instance ()->register_handler

                (this, ACE_Event_Handler::READ_MASK);

}


// Demultiplexing hook called by

// the <ACE_Reactor>.

virtual int handle_input (ACE_HANDLE);


private:

char host_name_[MAXHOSTNAME];

};
```

当此类的对象被动态分配时，open挂钩缓存相关联的客户的主机地址。如图7-7中的“控制台”窗口所示，该主机的名字和从客户日志看守那里接收到的日志记录一起被打印出来。

PEER_STREAM参数被ACE_SOCK_Stream类替换。当输入到达底层的ACE_SOCK_Stream时，handle_input方法被ACE_Reactor自动调用。该方法可实现如下：

```
// Hook method for handling the reception of

// remote logging transmissions from clients.

int Logging_Handler::handle_input (ACE_HANDLE)

{

ssize_t n = peer_stream_.recv (&len, sizeof len);


if (n != sizeof len)

        // Trigger handle_close().

        return -1;

else

{

        ACE_Log_Record lr;


        size_t len = ntohl (len);

        n = this->peer_stream_.recv_n (&lr, len));


        if (n != len)
```

```
            ACE_ERROR_RETURN ((LM_ERROR,
                                                "%p at
                                                host
                                                %s\n",
                                                "client
                                                logger",
                                                this-
                                                >host_name_),
                                                -1);
      lr.decode ();

      if (lr.len == n)
            lr.print (this->host_name_, 0, stderr);
      else
            ACE_ERROR_RETURN ((LM_DEBUG,
                                                "lr.len
                                                = %d,
                                                n =
                                                %d\n",
                                                lr.len,
                                                n),
                                                -1);
return 0
}
}
```

该方法执行两个recv，以模拟经由底层TCP连接的面向消息的服务。这种行为是必需的，因为TCP提供面向字节流、而不是面向记录的服务。第一个recv读取跟随的日志记录的长度，此长度存储为定长整数。随后第二个recv读取该长度那么多的字节，以获取实际的记录。自然，发送此消息的客户必须遵从同样的消息帧协议。

7.4.2.3 main()驱动程序

下面的事件循环驱动基于ACE_Reactor的日志服务器：

```
int main (int argc, char *argv[])
{
// 1. Set the program name with the logger.
ACE_LOG_MSG->open (argv[0]);
```

```
// Ensure correct usage.

if (argc != 2)

       ACE_ERROR_RETURN ((LM_ERROR,

                                        "usage: %n port-
                                        number"),

                                        -1);



       // 2. Create an addr and an acceptor.

ACE_INET_Addr port (ACE_OS::atoi (argv[1]));

                                        Logging_Acceptor
                                        acceptor

                                  (ACE_Reactor::instance (), port);



// 3. Loop forever, handling client requests.

for (;;)

       ACE_Reactor::instance ()->handle_events ();



/* NOTREACHED */

return 0;

}
```

在步骤1中，一个ACE_Log_Msg被创建，以将服务器生成的任何日志记录定向到它自己的标准错误流。图7-11和图7-13中的代码演示服务器怎样使用应用日志接口来在本地记录它自己的诊断消息。因为此本地配置不使用服务器日志看守，也就不会有导致“无限日志循环”的危险。

在步骤2中，服务器创建Logging_Acceptor，它的构造器将它自己登记到ACE_Reactor单体。在步骤3中，服务器进入一个无穷循环，它阻塞在handle_events方法中，直到接收到来自客户日志看守的事件。

图7-14演示在两个客户被ACE_Reactor分派、并开始参与分布式日志服务之后，日志服务器看守的状态。如图所示，为每个客户，动态地实例化和登记了一个Logging_Handler。

当事件到达服务器，ACE_Reactor自动分派Logging_Acceptor和Logging_Handler的handle_input方法。例如，当连接请求从客户日志看守到达时，ACE_Reactor调用Logging_Acceptor的handle_input方法。该方法接受新连接，并创建Logging_Handler，由它读取客户发送的所有数据，并将其显示在标准输出流上。同样地，当日志记录或关闭消息从相连的客户日志看守到达时，ACE_Reactor调用相应的Logging_Handler的handle_input方法。

图7-7描绘整个运行中的系统。日志记录通过应用日志接口的ACE_Log_Msg::log方法生成。该方法将日志记录转发给本地客户日志看守。客户日志看守随即通过网络将记录传送给服务器日志看守，在这里它被显示在服务器的日志控制台上。

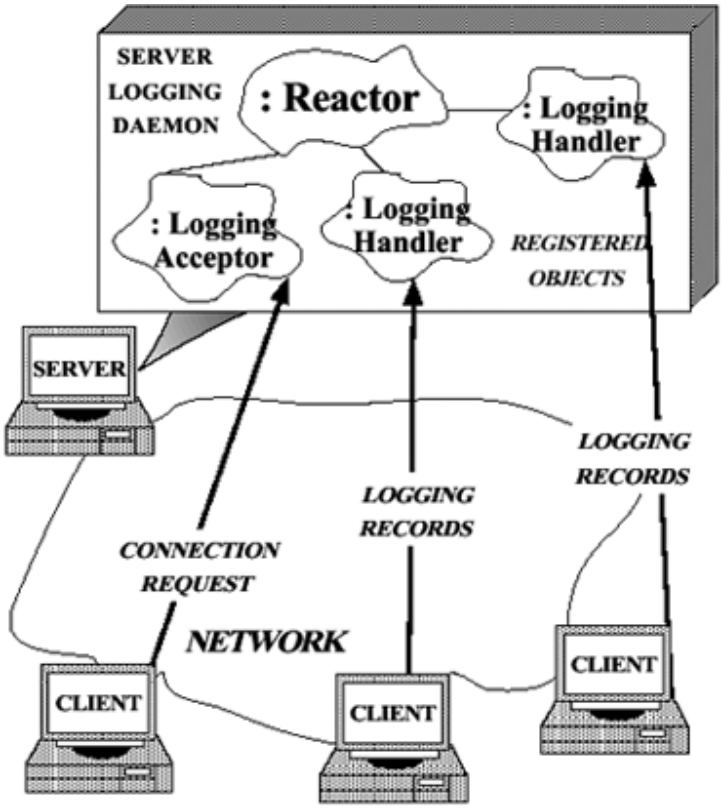

图7-14 服务器日志看守的运行时配置

服务器显示的日志信息指示（1）应用接口生成日志记录的时间，（2）应用所运行在其上的主机，（3）应用的进程标识符，（4）日志记录的优先级，（5）应用的命令行名字（也就是，“argv[0]”），以及（6）含有日志消息文本的任意文本串。

7.4.3 评估可选的日志工具实现

7.4.2中描述的分布式日志服务最初用C编写，并被用于一种商业的在线事务处理产品。这一部分根据若干软件质量因素，比如模块性、可扩展性、可复用性和可移植性，来比较该分布式日志服务的C++和C版本。

7.4.3.1 基于C的非OO日志服务

基于ACE_Reactor的分布式日志服务是对一种早期的、功能等价的非OO日志服务的面向对象的重新实现。原版本是为一种基于BSD UNIX的商业在线事务处理产品开发的。当时它用C编写，并直接使用BSD socket和select。随后，它被移植到其他操作平台上，比如系统V UNIX和Win32。

由于下面的一些问题，原来的C实现很难修改、扩展和移植：

紧耦合的功能：在原来的C日志服务中，事件多路分离、服务分派和事件处理操作都与接受客户连接请求及接收客户日志记录的代码紧耦合在一起。

全局变量的过多使用：若干全局数据结构被用于维持下面两者之间的关系：（1）每个客户的上下文信息（比如客户主机名和并发处理状态）和（2）标识相应的上下文记录的I/O句柄。因此，对程序的任何增强和修改都将直接影响现有源代码。

7.4.3.2 基于C++的OO日志服务

在本论文中描述的基于ACE_Reactor的日志服务的OO版本使用数据抽象、继承、动态绑定和模板来提供下面好处：

最小化对全局变量的依赖：基于ACE_Reactor的日志服务没有包含全局变量。相反，每个登记到ACE_Reactor的Logging_Handler对象都封装了客户地址和用于与客户通信的底层I/O句柄。

去耦策略和机制：使处理到来连接和数据的应用*策略*与执行多路分离和分派的底层机制去耦合。使策略与机制去耦合增强了下列软件质量因子：

- **可复用性：**ACE反应堆构架提供可复用的组件，执行所有较低级的事件多路分离和服务分派。因而，如7.4所示，要实现服务服务器日志看守，仅需要少量应用特有的代码。这些代码主要用于应用处理活动，比如接受新连接和接收客户日志记录。
- **可扩展性：**ACE_Reactor体系结构中策略和机制的分离同时增强了它的公共接口之上和之下的可扩展性。例如，要扩展服务器日志看守的功能，给它增加“认证日志”特性，相当地直接：只是简单地从ACE_Event_Handler基类继承，并有选择地实现必需的虚方法。同样地，通过实例化ACE_Acceptor和ACE_Svc_Handler模板，可以无需重新开发已有的基础构造，就制造出后续的应用。相反，要对原来的非OO的C版本完成同样的修改，需要直接改动已有的代码。
- **可移植性：**有可能修改ACE_Reactor底层的事件多路分离机制，而又不影响现有的应用代码。例如，从BSD平台将基于ACE_Reactor的分布式日志服务移植到系统V或Win32平台，无需对应用代码进行明显的修改。相反，将原有的C版本分布式日志服务从select移植到WaitForMultipleObjects是麻烦而易错的。例如，有若干被引入源代码的的微妙错误直到运行时才被发现。
- **效率：**在某些实时应用中，数据会立即在一或多个句柄上可用。因此，通过非阻塞的I/O轮询这些句柄可能比使用像select或WaitForMultipleObjects这样的OS事件多路分离器更为高效。扩展ACE_Reactor以支持这种可选的多路分离实现无需修改它的公共接口。

  > 在反应堆构架中使用OO的一个逻辑后果是它大量地使用动态绑定。[16]讨论了为什么在为socket设计“瘦”C++包装时，避免使用动态绑定常常是可取的。在有些编译器上，间接的虚表分派所带来的开销可能会相当地高。在这样的情况下，开发者可能需要避免大量使用动态绑定。
  >
  > 但是，一般而言，ACE_Reactor所带来的清晰性、可扩展性和模块性的显著增强已足够补偿效率的轻微下降。而且，ACE_Reactor通常用于开发分布式应用。分布式系统中开销的主要来源是像缓存、延迟、网络/主机接口硬件、表示层格式化、内存到内存的拷贝，以及进程管理这样的活动[21]。因

而，动态绑定所导致的额外的间接性通常是微不足道的[22]。此外，通过使用“adjustor thunks”，好的C++编译器可以将虚方法的额外开销完全优化掉[23]。

7.5 有效使用反应堆的设计准则

ACE_Reactor是事件多路分离和事件处理器分派的强大构架。但是，像其他构架一样，学习使用ACE_Reactor需要时间和努力。缩短学习曲线的的一种途径是去理解有效使用反应堆所必须遵从的*设计准则*。下面描述的设计准则基于帮助ACE用户正确进行反应堆构架编程所获得的大量经验。

7.5.1 理解具体事件处理器的返回值语义

具体事件处理器定义的各种handle_* 挂钩方法的返回值致使ACE_Reactor以不同的方式工作。使用返回值来触发不同行为意在降低ACE_Reactor的API的复杂度。但是，返回值常常使得程序员莫明其妙。因而，理解从handle_* 方法返回的值的效应非常重要；这些值分为三种情况：

零：handle_* 方法返回零（0）通知ACE_Reactor、事件处理器希望继续像前面一样被处理，也就是，它应该保持在ACE_Reactor的实现的一张表中。这样，当下一次ACE_Reactor的事件多路分离器系统调用经由handle_events被调用时，它还会继续包括该事件处理器的句柄。对于那些生存期超出一次handle_* 方法分派的事件处理器，这是一种“正常的”行为。

大于零：handle_* 方法返回大于0（> 0）的值通知ACE_Reactor、事件处理器希望在ACE_Reactor阻塞在它的事件多路分离器系统调用上面*之前*，再一次被分派。对协作的事件处理器来说，这种特性有助于增强全面的系统“公正性”。特别地，这种特性使得一个事件处理器在再次持有控制之前，允许其他事件处理器被分派。

小于零：handle_* 方法返回小于0（< 0 ）的值通知ACE_Reactor、事件处理器想要被关闭、并从ACE_Reactor的内部表中移除。为完成此工作、ACE_Reactor调用事件处理器的handle_close清扫方法。该方法可以执行用户定义的终止活动，比如删除对象分配的动态内存或关闭日志文件。handle_close方法返回后，ACE_Reactor将相关联的具体事件处理器从它的内部表中移除。

为减少handle_* 返回值所带来的问题，在实现具体事件处理器时，遵守下面的设计准则：

设计准则0：*不要手工删除事件处理器对象或显式调用handle_close*与*相反*，确保ACE_Reactor自动调用handle_close清扫方法。因而，应用必须遵从适当的协议来移除事件处理器，也就是，或者通过（1）从handle_* 挂钩方法中返回负值，或者通过（2）调用remove_handler。

该设计准则确保ACE_Reactor能够适当地清扫它的内部表。如果不服从这一准则，当ACE_Reactor试图移除已经在外部被删除的具体事件处理器时，就会带来不可预测的内存管理问题。后面的设计准则详细说明怎样确保ACE_Reactor调用handle_close清扫方法。

设计准则1：*从继承自ACE_Event_Handler的类的handle_* 方法中返回的表达式必须是常量（constant）*。这一设计准则有助于静态地检查是否handle_* 方法返回了恰当的值。如果必须违反此准则，开发者必须在return语句之前加一注释，解释为何要使用变量，而不是常量。

设计准则2：*如果从继承自ACE_Event_Handler的类的handle_* 方法中返回的值不为0，必须在return语句之前加一注释，说明该返回值的含义*。这一设计准则确保所有非0的返回值都是开发者有意使用的。

7.5.2 理解handle_close()清扫挂钩的语义

必须记住handle_close清扫挂钩方法只能由ACE_Reactor（1）*隐式地*调用，也就是，当handle_*方法返回-1这样的负值时，或是（2）*显式地*调用，也就是，如果应用调用remove_handler方法来解除具体事件处理器的登记。特别地，ACE_Reactor*不会*在本地应用或是远地应用关闭I/O句柄时自动调用handle_close。因此，应用必须确定何时I/O句柄已被关闭，并采取适当的步骤，以使ACE_Reactor触发handle_close清扫方法。

下面的来自7.4.2.2的Logging_Handler代码片段演示怎样正确地触发清扫挂钩：

```
// Hook method for handling the reception of

// remote logging transmissions from clients.

int Logging_Handler::handle_input (ACE_HANDLE)

{

ssize_t n = peer_stream_.recv (&len, sizeof len);


if (n == 0)

       // Trigger handle_close().

       return -1;


// ...


// Keep handler registered for ‘‘normal’’ case.

return 0;

}
```

当handle_input方法从recv那里收到0，它就返回-1。该值触发ACE_Reactor调用handle_close清扫挂钩。

为最少化handle_* 返回值所带来的问题，在实现具体事件处理器时，应遵守下面的设计准则：

设计准则3：*当你想要触发具体事件处理器的相应handle_close清扫方法时，从handle_* 方法中返回一个负值*。值-1通常用于触发清扫挂钩，因为它是ACE_OS系统调用包装中一个常用的错误代码。但是，任何来自handle_* 方法的负数都将触发handle_close。

设计准则4：*将所有Event_Handler清扫活动限制在handle_close清扫方法中*。一般而言，将所有的清扫活动合并到handle_close方法中，而不是分散在事件处理器的各个handle_* 方法中要更为容易。在处理动态分配的、必须用delete this来清除的事件处理器时，特别需要遵从此设计准则（见准则9）。

7.5.3 记住ACE_Time_Value参数是相对的

传递给ACE_Reactor的schedule_timer方法的两个ACE_Time_Value参数必须*相对于*当前时间指定。例如，下面的代码调度一个对象，延迟delay秒后开始，每interval秒打印一次可执行程序的名字（也就是，argv[0]）：

```
class Hello_World : public ACE_Event_Handler

{

public:

virtual int handle_timeout (const ACE_Time_Value &tv, const void *act)

{

       ACE_DEBUG ((LM_DEBUG,

                           "%[s] %d, %d\n",

                                  act,

                           tv.sec (),

                           tv.usec ()));

       return 0;

}

};



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

{

if (argc != 3)

       ACE_ERROR_RETURN ((LM_ERROR,
```

```
                                        "usage: %s delay
                                        interval\n",

                                        argv[0]), -1);


Hello_World handler; // timer object.


ACE_Time_Value delay = ACE_OS::atoi (argv[1]);

ACE_Time_Value interval = ACE_OS::atoi (argv[2]);


// Schedule the timer.

ACE_Reactor::instance ()->schedule_timer

       (&handler,

       (const void *) argv[0],

       delay,

       interval);


// Run the event loop.

for (;;)

       ACE_Reactor::instance ()->handle_events ();


/* NOTREACHED */

}
```

一种常见的错误是误将*绝对的*时间值传递给schedule_timer。例如，考虑一个不同的例子：

```
ACE_Time_Value delay = ACE_OS::atoi (argv[1]);

delay += ACE_OS::gettimeofday ();


// Callback every following 10 seconds.

ACE_Time_Value interval = delay + 10;


ACE_Reactor::instance ()->schedule_timer

(&handler,

0,

delay,

interval);
```

但是，该定时器在将来*很长*的时间内都不会到期，因为它将本日的当前时间加到了用户所要求的delay和interval上。

下面是实现具体事件处理器时，为最小化与绝对的ACE_Time_Value有关的问题，所应遵从的设计准则：

设计准则5：*不要将绝对时间用作ACE_Reactor::schedule_timer的第三或第四参数*。一般而言，这些参数应该小于一个极长的延迟，更远小于当前时间。

7.5.4 小心追踪ACE_Event_Handler的生存期

对登记到ACE_Reactor上的ACE_Event_Handler的跟踪失败会导致各种问题。不使用像Purify[24]这样的内存错误检测工具，很难去追踪这些问题；但这样的工具也只能捕捉下面的一些、而不是全部的与生存期相关的问题：

7.5.4.1 保守地使用非动态分配的事件处理器

考虑下面的具体事件处理器的定义：

```
class My_Event_Handler : public ACE_Event_Handler

{

public:

My_Event_Handler (const char *str = "hello")

       : str_ (ACE_OS::strnew (str)) {}


virtual int handle_close

       (ACE_HANDLE = ACE_INVALID_HANDLE,

       ACE_Reactor_Mask = ACE_Event_Handler::READ_MASK)

{

       // Commit suicide.

       delete this;

}


~My_Event_Handler (void)

{
```

```
delete [] this->str_;

}


// ...


private:

char *str_;

};
```

该类在从ACE_Reactor上移除时，通过它的handle_close清扫方法删除它自己。尽管这看起来有一点不太传统，它却是完全有效的C++习语。但是，它仅在正被删除的对象是动态分配的的情况下才能够工作。

相反，如果正被删除的对象*不是*动态分配的，全局动态内存堆将会被破坏。原因是delete操作符将把this解释为堆中有效的地址。当delete操作符试图将非堆的内存插入它的内部空闲表时，就会造成微妙的内存管理问题。

下面的例子演示一个常见的可导致堆崩溃的使用实例：

```
int main (void)

{

// Non-dynamically allocated.

My_Event_Handler my_event_handler;


ACE_Reactor::instance ()->register_handler

       (&my_event_handler,

       ACE_Event_Handler::READ_MASK);


       // ...

// Run event-loop.

while (/* ...event loop not finished... */)

       ACE_Reactor::instance ()->handle_events ();


// The <handle_close> method deletes an

// object that wasn't allocated dynamically...

ACE_Reactor::instance ()->remove_handler

       (&my_event_handler,

       ACE_Event_Handler::READ_MASK);
```

```
return 0;

}
```

上面代码的问题是remove_handler被调用时，ACE_Reactor将会调用My_Event_Handler的handle_close方法。遗憾的是，handle_close方法会对my_event_handler对象执行delete this操作，而此对象并非是动态分配的。

防止发生此问题的一种方法是将析构器放置在My_Event_Handler的私有区域，也就是：

```
class My_Event_Handler : public ACE_Event_Handler

{

public:

My_Event_Handler (const char *str);

// ...


private:

// Place destructor into the private section

// to ensure dynamic allocation.

?My_Event_Handler (void);

// ...

};
```

在此类中，My_Event_Handler的析构器被放置在类的私有访问控制区中。这种C++习语确保该类的所有实例都*必须*是动态分配的。如果实例被偶然地定义为static或auto，它在编译时就会被作为错误标记出来。

下面是实现具体事件处理器时，为最小化与具体事件处理器的生存期相关的问题，所应遵从的设计准则：

设计准则6：*不要delete不是动态分配的事件处理器*。任何含有delete this、而其类又没有私有析构器的handle_close方法，都有可能违反这一设计准则。在缺乏一种能够静态地识别这一情况的规约检查器时，应该在delete this的紧前面加上注释，解释为何要使用这一习语。

7.5.4.2 适当地解除具体事件处理器的登记

下面的程序演示与具体事件处理器的生存期相关的另一种常见错误：

```
ACE_Reactor reactor;


int main (void)

{

My_Event_Handler my_event_handler;


ACE_Reactor::instance ()->register_handler

       (&my_event_handler,

       ACE_Event_Handler::READ_MASK);


while (/* ...event loop not finished... */)

       ACE_Reactor::instance ()->handle_events ();


// The destructor of the ACE_Reactor singleton

// will be called when the process exits. It

// removes all registered event handlers.

return 0;

}
```

my_event_handler的生存期由main函数的生存期决定。相反，ACE_Reactor单体的生存期由进程的生存期决定。因而，当进程退出时，反应堆的析构器将会被调用。通过调用所有仍然登记在册的事件处理器的handle_close方法，ACE_Reactor的析构器将这些处理器全部移除掉。但是，如果my_event_handler仍然登记在Reactor上，它的handle_close方法将会在该对象出了作用域、并被销毁*之后*调用。

下面是实现具体事件处理器时，为最小化与具体事件处理器的生存期相关的问题，所应遵从的其他三条设计准则：

设计准则7：*总是从堆中动态分配具体事件处理器*。这是解决许多与具体处理器的生存期有关的问题的相对直接的方法。如果不可能遵从此准则，必须在具体事件处理器登记到ACE_Reactor时给出注释，解释为什么不使用动态分配。该注释应该在将静态分配的具体处理器登记到ACE_Reactor的register_handler语句的紧前面出现。

设计准则8：*在ACE_Event_Handler退出它们“生活”的作用域之前，从与它们相关联的ACE_Reactor中将它们移除掉*。该准则应在未遵从准则7的情况下使用。

设计准则9：*只允许在handle_close方法中使用delete this习语*，也就是，不允许在其他handle_* 方法中使用delete this。该准则有助于检查是否有与删除非动态分配的内存有关的潜在错误。自然，与ACE_Reactor无关的组件可以拥有不同的对自删除进行管辖的准则。

设计准则10：*仅在为具体事件处理器所登记的最后一个事件已从ACE_Reactor中移除时执行delete this操作*。过早删除在ACE_Reactor上登记了多个事件的具体处理器会导致“晃荡的指针”，遵从此准则可以避免发生这样的情况。

例如，my_event_handler可以登记READ和WRITE事件，如下所示：

```
ACE_Reactor::instance ()->register_handler
(&my_event_handler,
ACE_Event_Handler::READ_MASK
| ACE_Event_Handler::WRITE_MASK);
```

在此情形下，当handle_input返回-1时，ACE_Reactor将调用handle_close清扫挂钩方法。在具体事件处理器登记的WRITE_MASK也被移除之前（例如，让它返回一个负值，或是通过下面的语句显式地将它移除），该方法*不能*执行delete this操作。

```
ACE_Reactor::instance ()->remove_handler
(&my_event_handler,
ACE_Event_Handler::WRITE_MASK);
```

下面的方法演示追踪此信息的一种途径：

```
class My_Event_Handler : public ACE_Event_Handler
{
public:
My_Event_Handler (void)
{
       // Keep track of which bits are enabled.
       ACE_SET_BITS (this->mask_,
                           ACE_Event_Handler::READ_MASK
              | ACE_Event_Handler::WRITE_MASK);

       // Register ourselves with the Reactor for
       // both READ and WRITE events.
       ACE_Reactor::instance ()->register_handler
       (this, this->mask_);
}
```

```
virtual int handle_close (ACE_HANDLE h, ACE_Reactor_Mask mask)
{
       if (mask == ACE_Event_Handler::READ_MASK)
       {
              ACE_CLR_BITS (this->mask_,
                            ACE_Event_Handler::READ_MASK);

              // Perform READ_MASK cleanup logic.
       }
       else if (mask == ACE_Event_Handler::WRITE_MASK)
       {
              ACE_CLR_BITS (this->mask_,
              ACE_Event_Handler::WRITE_MASK);

              // Perform WRITE_MASK cleanup logic.
       }

       // Only delete ourselves if we've been closed
       // down for both READ and WRITE events.
       if (this->mask_ == 0)
              delete this;
}

// ... handle_input() and handle_output() methods.

private:
ACE_Reactor_Mask mask_;
// Keep track of when to delete this.
};
```

上面的解决方案维护ACE_Reactor_Mask，追踪何时一个具体事件处理器登记的所有事件已被从ACE_Reactor移除。

7.5.5 注意WRITE_MASK语义

下面的代码指示ACE_Reactor，只要可以无阻塞地向一个句柄写，就回调一个event_handler。

```
ACE_Reactor::instance ()->mask_ops
(event_handler,
ACE_Event_Handler::WRITE_MASK,
ACE_Reactor::ADD_MASK);
```

但是，除非连接被流控制，否则*总是*可以向一个句柄写。因此，反应堆会持续地回调event_handler的handle_output方法，直到（1）发生连接流控制或（2）mask_ops方法被指示清除WRITE_MASK。一种常见的编程错误是忘记清除此掩码，导致ACE_Reactor不断地调用handle_output方法。应遵从下面的设计准则来避免这一问题：

设计准则11：*当你不再需要具体事件处理器的handle_output方法被回调时，清除WRITE_MASK*。

下面的代码演示怎样确保handle_output方法不再被回调：

```
ACE_Reactor::instance ()->mask_ops
(event_handler,
ACE_Event_Handler::WRITE_MASK,
ACE_Reactor::CLR_MASK);
```

ACE_Reactor还定义了完成同样操作的简捷方法：

```
ACE_Reactor::instance ()->cancel_wakeup
(event_handler,
ACE_Event_Handler::WRITE_MASK);
```

这些方法通常在已不再有在具体事件处理器上待决的输出消息时被调用。

为帮助自动查验此准则，程序员必须在他们的handle_output方法中插入注释，这些注释指示哪些返回路径*不会*清除WRITE_MASK，也就是，事件处理器想要在“可以写”时继续被回调。同样地，程序员还应该注释那些WRITE_MASK被清除的路径。如果在handle_output方法中没有路径清除WRITE_MASK，那就意味着可能违反了此准则。

例如，下面的handle_output方法演示此设计准则的可能的应用：

```
int My_Event_Handler::handle_output (ACE_HANDLE)
{
if (/* output queue is now empty */)
{
       ACE_Reactor::instance ()->cancel_wakeup
              (event_handler,
              ACE_Event_Handler::WRITE_MASK);
       /* Removing WRITE_MASK */
       return 0;
} else
{
       // ... continue to transmit messages
       // from the output queue.
       /* Not removing WRITE_MASK */
       return 0;
}
}
```

如果没有注释指示对WRITE_MASK的清除，就有可能违反了此设计准则。

7.5.6 适当地登记具体事件处理器

当为I/O操作在ACE_Reactor上登记具体事件处理器时，选择下面的方法中的一种：

显式地传递句柄：该方法使用下面的ACE_Reactor方法：

```
int register_handler
(ACE_HANDLE io_handle,
ACE_Event_Handler *event_handler,
ACE_Reactor_Mask mask);
```

并显式地传递I/O设备的ACE_HANDLE，也就是：

```
void register_socket (ACE_HANDLE socket,
```

```
                                    ACE_Event_Handler *handler)
{
ACE_Reactor::instance ()->register_handler
       (socket,
       handler,
       ACE_Event_Handler::READ_MASK);
       // ...
}
```

注意此register_handler方法允许同一个具体事件处理器与多个ACE_HANDLE一起进行登记。反应堆的这一特性使得我们有可能最小化处理许多客户所需的状态的数量；这些客户同时与同一事件处理器相连接。

隐式地传递句柄：该方法使用ACE_Reactor的另一个register_handler方法：

```
int register_handler
(ACE_Event_Handler *event_handler,
ACE_Reactor_Mask mask);
```

在这种情形下，ACE_Reactor执行一次“双重分派”（double-dispatch）[6]来通过具体事件处理器的get_handle方法从处理器中获取底层的ACE_HANDLE。该方法在ACE_Event_Handler基类中定义，具有特征const：

```
virtual ACE_HANDLE get_handle (void) const;
```

当使用隐式登记时，常见的一种错误是在从ACE_Event_Handler派生子类时忽略了get_handle上的const。这样的疏忽将导致编译器不能适当地在子类中重定义get_handle方法。相反，它将在子类中*隐藏*该方法，从而产生代码、调用*基类*的get_handle方法；该方法缺省返回-1。

因此，服从下面的设计准则十分重要：

设计准则12：*确定get_handle方法的特征与ACE_Event_Handler基类中的一致*。如果你不遵从此准则，并且你“隐式地”将ACE_HANDLE传递给ACE_Reactor，ACE_Event_Handler基类中的缺省get_handle将返回-1，而这是错误的。

7.5.7 从反应堆中移除已关闭的句柄/处理器

当连接被关闭时，句柄就不再能用于I/O。在这样的情况下，select将会持续地报告句柄“读就绪”，这样你就可以在句柄上调用close了。此步骤通常在handle_close清扫方法中完成。

一个常见的错误是对已死句柄及其事件处理器的移除的失败。这样ACE_Reactor将会持续地回调事件处理器的handle_input方法，直到它被从ACE_Reactor中移除。下面的设计准则有助于避免这一问题：

设计准则13：*当连接关闭时（或当连接上发生错误时），从handle_* 方法中返回一个负值。*

遵从此设计准则的代码通常被构造如下：

```
int handle_input (ACE_HANDLE handle)
{
// ...
ssize_t result = ACE_OS::read (handle, buf, bufsize);

if (result <= 0)
       // Connection has closed down or an
       // error has occurred.
       return -1;
else
       // ...
```

当返回-1时，ACE_Reactor将调用你的handle_close清扫方法。为避免资源泄漏，确定该方法给了事件处理器以机会来删除它自己，并关闭它的句柄（例如，ACE_OS::close (handle)）。一旦handle_close返回，ACE_Reactor就有机会从它的内部表中移除句柄/处理器对。

7.5.8 使用DONT_CALL标志来避免递归的handle_close()回调

前面的准则描述了在应用显式或隐式地（通过从handle_* 挂钩方法中返回负值）调用它的remove_handler时，ACE_Reactor怎样自动调用handle_close方法。但是，如果应用在handle_close清扫方法中调用remove_handler，就必须特别小心，因为这有可能触发无限递归。下面的准则处理这一问题。

设计准则14：*在handle_close方法中调用remove_handler时，总是传递给它DONT_CALL标志。*该准则确保ACE_Reactor不会递归地调用handle_close方法。下面的代码演示怎样应用此准则：

```
int My_Event_Handler::handle_close

(ACE_HANDLE,

ACE_Reactor_Mask)

{

// ...


ACE_Reactor::instance ()->remove_handler

       (this->get_handle (),

       // Remove all the events for which we're

       // registered. We must pass the DONT_CALL

       // flag here to avoid infinite recursion.

       ACE_Event_Handler::RWE_MASK |

       ACE_Event_Handler::DONT_CALL);


       // ...

}
```

顺便说一下，remove_handler通常在下列情况下在handle_close中被调用：（1）为多个事件登记了同一个具体事件处理器，以及（2）handle_close第一次被调用时需要触发事件处理器的完全关闭。因而，handle_close还应该移除在ACE_Reactor中与该事件处理器相关联的其他事件。

7.6 结束语

ACE_Reactor是设计用于简化并发的、事件驱动的分布式应用的OO构架。通过在OO C++接口中封装低级的OS事件多路分离机制，ACE_Reactor使得开发正确、简洁、可移植和高效的应用变得更为容易。同样地，通过分离*策略*与*机制*，ACE_Reactor增强了复用、改善了可移植性，并提供了透明的可扩展性。

下面的C++语言特性对ACE_Reactor的设计和使用它的功能的应用有所帮助：

类：C++类提供的封装改善了可移植性。例如，ACE_Reactor类将应用与像WaitForMultipleObjects和select这样的OS事件多路分离器之间的差异屏蔽开来。

对象：将C++*对象*、而不是单独的*函数*登记到ACE_Reactor有助于将应用特有的状态和使用此状态的方法集成在一起。

继承和动态绑定：通过允许开发者不修改现有代码就增强ACE_Reactor及与其相关联的应用的功能，这些特性促进了透明的可扩展性。

模板：通过将可变性引入统一的类（它们可被“插入“到通用模板中），C++参数化类型有助于增强可复用性。例如，除了Logging_Handler和ACE_SOCK_Acceptor，ACE_Acceptor还可用SVC_HANDLER和PEER_ACCEPTOR实例化。

使用ACE_Reactor的一个潜在的不利方面是一开始很难理解应用的主线程控制在哪里执行。这是与事件循环回调分派器（比如ACE_Reactor或X-windows）相关的一个常见问题。但是，在使用此方法编写若干应用后，围绕这种“间接事件回调”分派模型的迷惑通常就会消失了。

ACE_Reactor和ACE socket包装的C++源代码和文档可在 http://www.cs.wustl.edu/~schmidt/ACE.html找到。这一版本还包括一套测试程序和例子，以及许多其他封装命名管道、流管道、mmap和系统V IPC机制（也就是，消息队列、共享内存和信号量）的C++包装。

感谢

感谢西门子公司的Hans Rohnert和Bill Landi对本论文提出的有益意见。

参考文献

[1] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529－545, Reading, MA: Addison-Wesley, 1995.

[2] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[3] W.R.Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[4] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[5] D. C. Schmidt and C. Cleeland, “Applying Patterns to Develop Extensible ORB Middleware,” *Submitted to the IEEE Communications Magazine*, 1998.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[7] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.

[8] D. C. Schmidt, "Transparently Parameterizing Synchronization Mechanisms into a Concurrent Distributed Application," *C++ Report*, vol. 6, July/August 1994.

[9] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[10] R. E. Barkley and T. P. Lee, "A Heap-Based Callout Implementation to Meet Real-Time Needs," in *Proceedings of the USENIX Summer Conference*, pp. 213 - 222, USENIX Association, June 1988.

[11] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol II: Design, Implementation, and Internals*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[12] G. Varghese and T. Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility," in *The Proceedings of the 11th Symposium on Operating System Principles*, November 1987.

[13] J. Hu, S. Mungee, and D. C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM," i n *Proceeedings of INFOCOM '98*, March/April 1998.

[14] J. Hu, I. Pyarali, and D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks," in *Proceedings of the 2nd Global Internet Conference*, IEEE, November 1997.

[15] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294 - 324, Apr. 1998.

[16] D. C. Schmidt, "IPC SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol.4, November/December 1992.

[17] D. L. Presotto and D. M. Ritchie, "Interprocess Communication in the Ninth Edition UNIX System," *UNIX Research System Papers, Tenth Edition*, vol. 2, no. 8, pp. 523 - 530, 1990.

[18] UNIX Software Operations, *UNIX System V Release 4 Programmer's Guide: STREAMS*. Prentice Hall, 1990.

[19] G. Booch, *Object Oriented Analysis and Design with Applications 2nd Edition*). Redwood City, California: Benjamin/Cummings, 1993.

[20] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[21] D. C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 489 - 506, May 1993.

[22] A. Koenig, "When Not to Use Virtual Functions," *C++ Journal*, vol. 2, no. 2, 1992.

[23] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[24] P. S. Inc., *Purify User's Guide*. PureAtria Software Inc., 1996.