ACE 线程、同步、事件

Allen Long
ihuihoo@gmail.com
http://www.huihoo.com
huihoo - Enterprise Open Source

内容安排

- 进程和线程管理
- 反应堆(Reactor)框架：用于事件多路分离和分发的体系结构
- 为I/O，定时器，信号处理实现事件处理器

ACE进程管理

ACE通过以下一些类，完成进程创建、管理和控制

ACE类	说明
ACE_Process	可移植地创建和同步进程
ACE_Process_Options	指定“与平台无关”和“与平台相关”的选项
ACE_Process_Manager	可移植地创建和同步多组进程

ACE_Process类

提供一下能力：

- 创建和终止进程；
- 在进程退出时保持同步；
- 访问进程属性信息，如进程ID等

方法	说明
prepare()	子进程创建之前,由spawn()调用,可以对新进程中将要用到的选项进行检测、修改，可方便设置“和平台相关”的选项
spawn()	创建新的进程地址空间，并运行指定的程序映像
unmanage()	进程退出时，被ACE_Process_Manager调用
getpid()	返回新子进程的进程ID
exit_code()	返回子进程的退出代码
wait()	等待创建的进程退出
terminate()	突然终止进程，不给进程清理自己的机会(要慎用)

Process Example

```
#include "ace/Process.h"
#include "ace/Log_Msg.h"

int ACE_TMAIN (int, ACE_TCHAR *[])
{
    ACE_Process new_process;
    ACE_Process_Options options;
     options.command_line(ACE_TEXT("calc"));
     pid_t pid;
     pid = new_process.spawn(options);
     ACE_DEBUG ((LM_DEBUG, "Process ID %d\n", pid));
     return 0;
}
```

ACE_Process类

只在POSIX/UNIX中有效的方法

方法	说明
parent()	可重写,提供“与应用程序相关”的行为;在父进程环境中,若fork()成功创建子进程,此方法将被调用
child()	可重写,提供“与应用程序相关”的行为;此方法在子进程环境中调用,且在fork()之后,exec()之前
kill()	向进程发送信号,仅在能向指定进程发送信号的UNIX/POSIX系统中具有可移植性

ACE_Process_Options类

提供以下能力：

- 允许应用程序指定需要的“进程控制信息”
- 允许“进程控制项(process control item)”随着平台的改变而扩展

方法	说明
command_line()	指定程序的命令和参数，在“被创建进程”中启动新程序时会使用它们
setenv()	指定环境变量，并将其添加到被创建进程的环境中
working_directory()	指定一个新路径，使被创建进程在启动新程序映象之前切换到这个路径
set_handles()	设置某些特殊的文件句柄；被创建进程可被这些进程用于它的STDIN,STDOUT,STDERR
pass_handle()	表示一个“应被传递给新创建的进程”的句柄

$ACE_wrappers/examples/OS/Process

ACE_Process_Manager类

可将多个进程作为一个单元来管理

- 保存内部记录，管理和监视ACE_Process类创建的多组进程
- 允许一个进程创建一组进程

方法	说明
open()	初始化ACE_Process_Manager
close()	释放所有资源(不等待进程退出)
spawn()	创建一个进程，将其添加到被管理的一个进程组中
spawn_n()	创建n个进程，这些进程都属于同一进程组
wait()	等待进程组中的一些或全部进程退出
instance()	返回一个指向ACE_Process_Manager singleton的指针的静态方法

使用ACE_Process_Manager

实验：

1、使用ACE_Process_Options 设置相关属性，调用计算器
2、使用 ACE_Process_Manager 完成多个线程的创建
3、...

ACE线程管理

通过以下一些类，网络应用程序可以在一个进程中创建、管理一个或多个控制线程(threads of control)

- ACE_Thread提供了对POSIX pthreads接口、Solaris线程、Win32线程调用的简单包装，这些调用处理线程创建、挂起、取消和删除等问题。
- ACE_Thread_Manager提供了ACE_Thread中的功能的超集。

ACE类	说明
ACE_Thread_Manager	使应用程序能够可移植地创建和管理线程的生命周期、同步和属性
ACE_Sched_Params	可移植地封装了OS”调度”类特性；可结合ACE_OS::sched_params()包装器方法一起使用，控制实时线程的各种属性
ACE_TSS	封装了“线程专有存储”机制

ACE_Thread_Manager

方法	说明
spawn()	创建一个新的控制线程,并传递给它一个“函数”和“函数的参数”,作为线程执行的入口点
spawn_n()	创建n个“同属一个线程组”的新线程,其他线程会等待这整组线程的退出
wait()	阻塞，直到线程管理器中的所有线程都已退出，并获得所有“可连接”线程的退出状态为止
join()	等待某一线程退出，并获得其退出状态
cancel_all()	请求ACE_Thread_Manager对象管理的线程全部退出
testcancel()	询问某一线程是否已被请求退出
exit()	退出一个线程，并释放这个线程的资源
close()	关闭并释放所有被管理线程的资源
instance()	这是一个静态方法,返回一个指向ACE_Thread_Manager singleton的指针

使用ACE_Thread_Manager创建一组线程

```
#include "ace/Thread_Manager.h"
#include <iostream>

void print (void* args)
{
  int id = ACE_Thread_Manager::instance()->thr_self();
  std::cout << "Thread Id: " << id << std::endl;
}

int ACE_TMAIN (int argc, ACE_TCHAR* argv[])
{
  ACE_Thread_Manager::instance()->spawn_n(
      4, (ACE_THR_FUNC) print, 0, THR_JOINABLE | THR_NEW_LWP);

  ACE_Thread_Manager::instance()->wait();
  return 0;
```

ACE_Sched_Params

方法	说明
ACE_Sched_Params()	使应用程序能够可移植地创建和管理线程的生命周期、同步和属性
priority_min()	返回某一调度策略和范围的最低优先级
priority_max()	返回某一调度策略和范围的最高优先级
next_priority()	给定一个策略、优先级和范围，返回下一个较高的优先级，其中“较高”指的是调度优先级，而不是优先级的值。如果给定的优先级已经是指定策略的最高优先级，那么返回的就是这个优先级本身
previous_priority()	给定一个策略、优先级和范围，返回下一个较低的优先级，其中“较高”指的是调度优先级，而不是优先级的值。如果给定的优先级已经是指定策略的最低优先级，那么返回的就是这个优先级本身

ACE_Sched_Params

```
virtual int handle_data (ACE_SOCK_Stream * logging_clienet) {
  int prio =
    ACE_Sched_Params::next_priority
      (ACE_SCHED_FIFO,  // 实施FIFO调度
       ACE_Sched_Params::priority_min (ACE_SCHED_FIFO),
       ACE_SCOPE_THREAD);
  ACE_OS::thr_setprio (prio);
  return Thread_Per_Connection_Logging_Server::handle_data
            (logging_client)
}
```

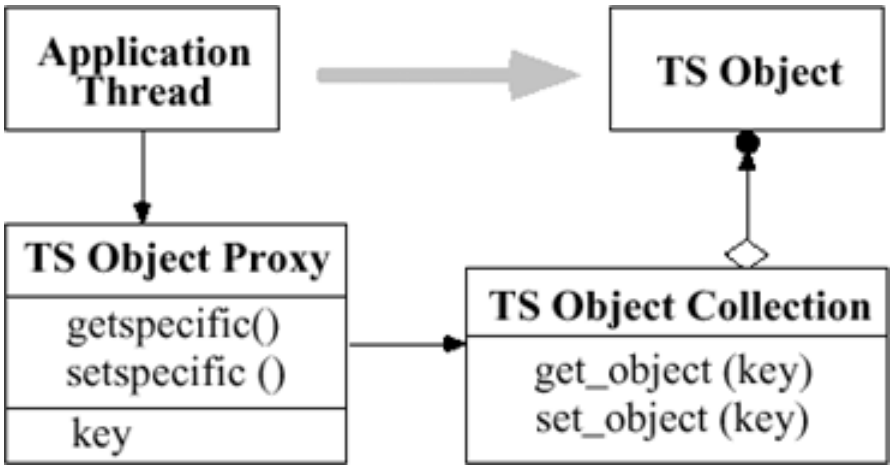
对每一个处理(来自“已连接客户”的)日志记录的线程，我们提高他们的优先级。

ACE_TSS

- 线程专有存储（Thread-Specific Storage）模式可减轻多线程性能和编程复杂性的若干问题。通过允许多个线程使用一个逻辑上的全局访问点来获取线程专有数据，而又不给每次访问带来锁定开销，线程专有存储模式可改善性能，并简化多线程应用。
- 在ACE中，TSS通过使用ACE_TSS模板类来实现。需要成为线程专有的类被传入ACE_TSS模板，然后可以使用C++的->操作符来调用它的全部公共方法。ACE_TSS模板是一个代理，它将普通的C++类转换为“类型安全”的类，这些类的实例位于“线程专有存储空间”中

方法	说明
operator->()	得到“和TSS键相关联”的“线程专有”对象
cleanup()	静态方法,线程退出时删除TSS对象

线程专有存储模式中的参与者的结构



- 应用线程使用TS Object Proxy(TS对象代理)来访问驻留在线程专有存储中的TS Object.线程专有存储模式的实现可以使用“智能指针”(smart pointer)来隐藏TS Object Proxy,以使应用看起来像是在直接访问TS Object.

ACE_TSS的已知应用

- 在支持POSIX和Solaris线程API的OS平台上实现的errno机制是被广泛使用的线程专有存储模式的例子.此外,与Win32一起提供的C运行时库支持线程专有的errno.Win32 GetLastError/SetLastError也实现了线程专有存储模式.
- 在Win32操作系统中,窗口属于线程.每个拥有窗口的线程都有一个私有的消息队列,OS会在其中放入用户接口事件.获取等待处理的下一消息的API调用使下一个消息从调用线程的消息队列中出队,而该消息队列就驻留在线程专有存储中.
- OpenGL是一个用于渲染三维图形的C API.程序根据多边形来渲染图形;多边形通过反复调用glVertex函数、以传递多边形的每一顶点给库来描述.在顶点被传递给库之前设置的状态变量精确地决定OpenGL在接收到顶点时如何进行绘制.该状态在OpenGL库中、或是在图形卡上作为封装的全局变量存储.在Win32平台上,OpenGL库为每个使用该库的线程在线程专有存储中维护一组唯一的状态变量.
- 线程专有存储被用于在ACE网络编程工具包中实现它的错误处理方案,该方案与ACE中日志的Logger方法相类似.此外,ACE还实现了线程安全的线程专有存储模板包装.

使用ACE_TSS

实验：

1、设置 ACE_TSS 变量
2、设置当前线程的ID为线程局部变量
3、创建10个这样的线程
4、访问并打印线程局部变量

ACE的同步和线程管理机制

处理同步的类

- ACE_Lock类属(互斥体、信号量、读／写互斥体和令牌)
- ACE_Guard类属
- ACE_Condition类属

杂项ACE Synchronization类(如ACE_Barrier和ACE_Atomic_Op)

ACE线程/进程同步类

ACE类	说明
ACE_Guard ACE_Read_Guard ACE_Write_Guard	运用Scoped Locking技术，确保进入或退出一段C++代码时，能够自动得到或释放锁(Lock)
ACE_Thread_Mutex ACE_Process_Mutex ACE_Null_Mutex	为并发应用程序提供一种简单高效的机制，用以串行访问共享资源
ACE_RW_Thread_Mutex ACE_RW_Process_Mutex	可以高效、并发地访问“内容经常被读取但较少被修改”的资源
ACE_Thread_Semaphore ACE_Process_Semaphore ACE_Null_Semaphore	实现“计数信号量”(counting semaphore),这是一种“对多个控制线程实施同步”的通用机制
ACE_Condition_Thread_Mutex ACE_Null_Condition	使线程可以高效地协调和调度它们的处理。

使用ACE_Mutex

实验：

1、使用ACE_Thread_Mutex创建互斥体
2、控制两个线程并控制它们对资源的访问

Scoped Locking：ACE 区域锁

目的：区域锁习惯用法确保当控制进入一个区域时,一个锁将自动被获取,当控制离开这个区域时,这个锁将被自动释放.

定义一个卫兵(guard)类,当控制进入一个区域时,它的构造函数将自动获取锁,当控制离开这个区域时,它的析构函数将自动释放锁.在一个方法中实例化一个guard类实例用于获取/释放锁,同时定义了临界区的阻塞范围.

使用范围锁带来的两个益处：

1、增加程序的健壮性.通过使用这个习惯用法,当控制进入/离开被c++方法和阻塞区域定义的临界区时,将自动实现加锁/解锁的操作.因此,这个方法通过减少并发编程中的一类常见的错误来增加并发应用程序的健壮性.

2、降低维护的工作量.如果参数化方法或多态方法被用于实现guard/lock类,它将直接增加聚合,减少错误.因为这里只有一个实现,而不是针对每个guard类型都有一个分别的实现.这种集中化的概念避免了版本的爆炸.

在并发应用和组件中使用区域锁习惯用法将带来如下的缺陷：

1、当进行递归使用时,存在潜在的死锁.如果一个使用范围锁的方法递归的调用自己,那么将导致自死锁,除非使用的是可递归的mutex.线程安全的接口模式通过确保只有接口方法可以使用范围锁来避免了这个问题,但是目前的方法实现没有使用线程安全接口模式.

2、受到编程语言规定的语义的限制.因为范围锁习惯用法是基于c++的语言特性,没有必要绑定操作系统规定的系统调用.因此,当线程或进程在一个保护的临界区中被中止,范围锁将不能被释放.

使用ACE_Guard

实验：

1、使用ACE_Guard修改前一个例子
2、对比ACE_Guard和ACE_Thread_Mutux之间的联系与区别

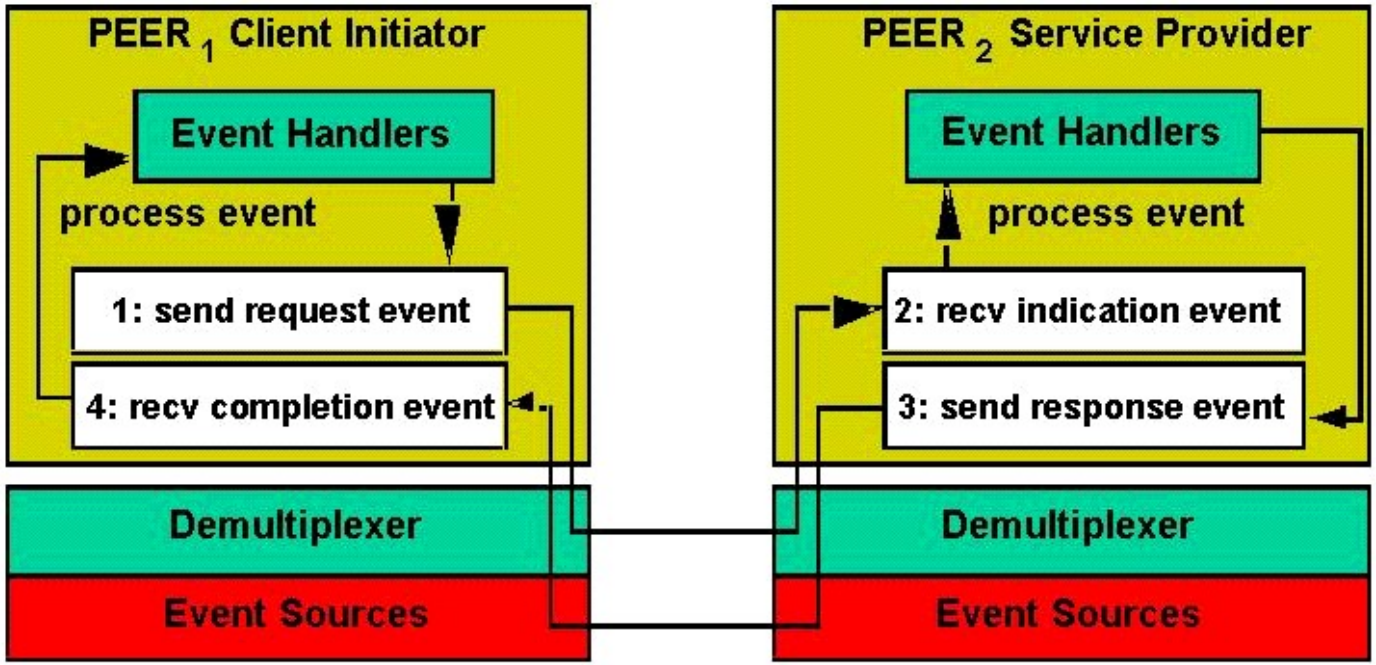
Thread-safe Interface: ACE中的线程安全接口模式

目的：线程安全接口模式保证组件内方法互调用时不会发生自死锁现象，同时能够最小化加锁带来的负载.

- 这种模式将加锁的范围扩大,由原来的组件内部关键数据扩大到整个组件,感觉上降低了组件的并发能力.如果确保组件方法不会存在互相调用的可能,建议不要使用此模式.
- 该模式借鉴了Java语言中的同步机制,实现对象加锁和方法同步.
- 为实现C++的Monitor对象提供的基础.

ACE Reactor(反应堆)

- 很多网络应用是基于事件驱动开发的.
- 这些应用的共同资源包括：IPC Stream for I/O operations, POSIX Signals, Windows handle signaling, & timer expirations
- 为提高扩展性, 灵活性, 需提供一个事件获取和分发的基础设施.



ACE Reactor(反应堆)

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,它将UNIX和Windows NT事件多路分离机制(比如select和poll)的功能封装在一个可移植和可扩展的OO包装中.

ACE_Reactor使用在Event Handler中声明的虚方法,以集成基于I/O句柄、基于定时器和基于信号的事件的多路分离.

* 基于I/O句柄的事件经由handle_input、handle_output和handle_exceptions方法分派;
* 基于定时器的事件经由handle_timeout方法分派;
* 基于信号的事件经由handle_signal分派。

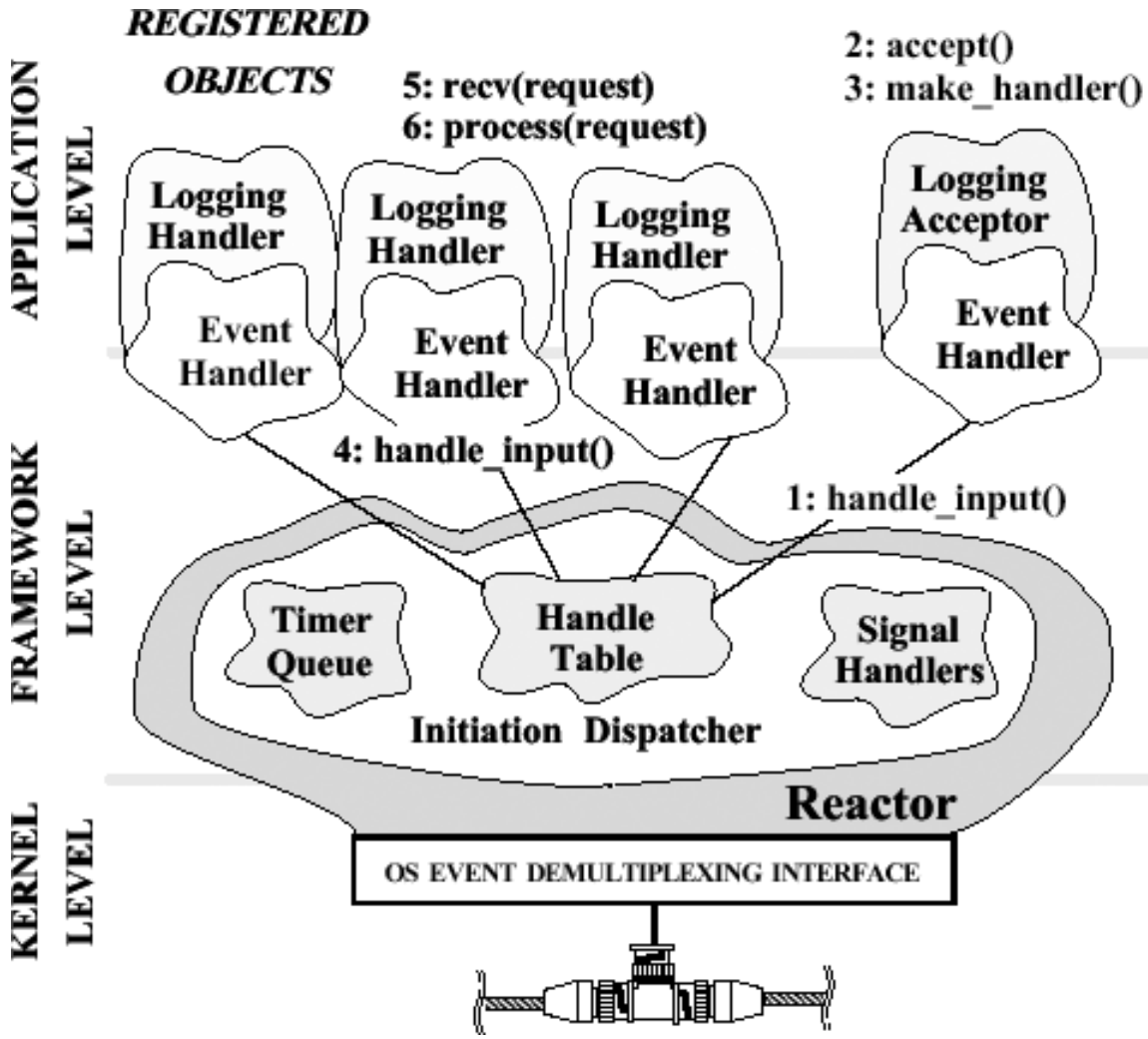
特性:

* 使用统一的OO多路分离和分派接口
* 使事件处理器分派自动化
* 支持透明的可扩展性
* 增加复用
* 增强类型安全性
* 改善可移植性
* 线程安全性
* 高效的多路分离

ACE_Reactor 实现

名称	说明
ACE_Select_Reactor	是除Windows之外的所有平台使用的默认的反应器实现.
ACE_WFMO_Reactor	是Windows上的默认的反应器实现.
ACE_Msg_WFMO_Reactor	与ACE_Msg_WFMO_Reactor很像, 但能分派Windows消息.
ACE_TP_Reactor	是对ACE_Select_Reactor的扩展, 运行在线程池中.
ACE_Priority_Reactor	是对ACE_Select_Reactor的扩展, 利用了ACE_Event_Handler类的priority()方法.
ACE_QtReactor (GUI)	即扩展了ACE_Select_Reactor, 又扩展了Qt的QObject类.
ACE_FlReactor (GUI)	集成了FastFlight工具包的FL::wait()方法.
ACE_TkReactor (GUI)	使用了Tcl/TK的Tcl_DoOneEvent()方法.
ACE_XtReactor (GUI)	使用XtWaitForMultipleEvents()集成了X Toolkit库.

ACE Reactor组件

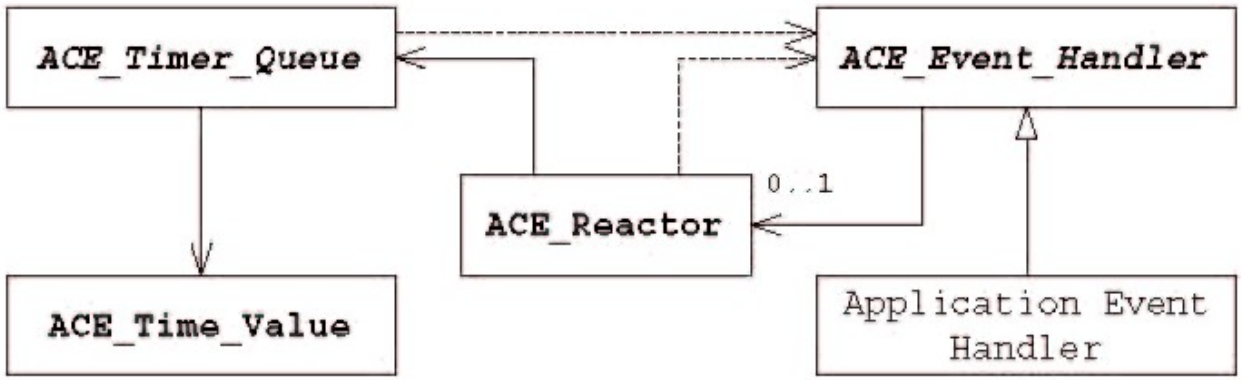


处理io事件

实验：

1、handle_input ()
2、reactor 和 acceptor 的使用
3、需要一个客户端发出请求

ACE Reactor框架



ACE类	描述
ACE_Time_Value	提供时间(Time)和持续时间(Duration)的可移植性、规范化的表示,使用C++运算符重载来简化与时间有关的算术和关系运算
ACE_Event_Handler	抽象类,其接口定义的Hook方法是ACE_Reactor回调的目标,大多数通过ACE并发的事件处理器都是ACE_Event_Handler的后代
ACE_Timer_Queue	抽象类,定义定时器队列的能力和接口.ACE含有多种派生自ACE_Timer_Queue的类,为不同的定时机制提供灵活的支持
ACE_Reactor	提供一个接口,用来在Reactor框架中管理事件处理器登记,并执行事件循环来驱动事件检测、多路分离和分派(它有多种实现)

ACE_Time_Value

ACE_Time_Value简化了可移植的、与时间和持续时间有关的运算的使用

通过反应器调度定时器

如：使用ACE_Event_Handler的子类处理来自ACE_Reactor的超时事件.

初始延迟3秒,时间间隔5秒

```
MyTimerHandler *time = new MyTimerHandler ();
ACE_Time_Value initialDelay (3);
ACE_Time_Value interval (5);
ACE_Reactor::instance()->schedule_time (time, 0, initialDelay,
   interval);
```

使用ACE_Time_Value

定义：

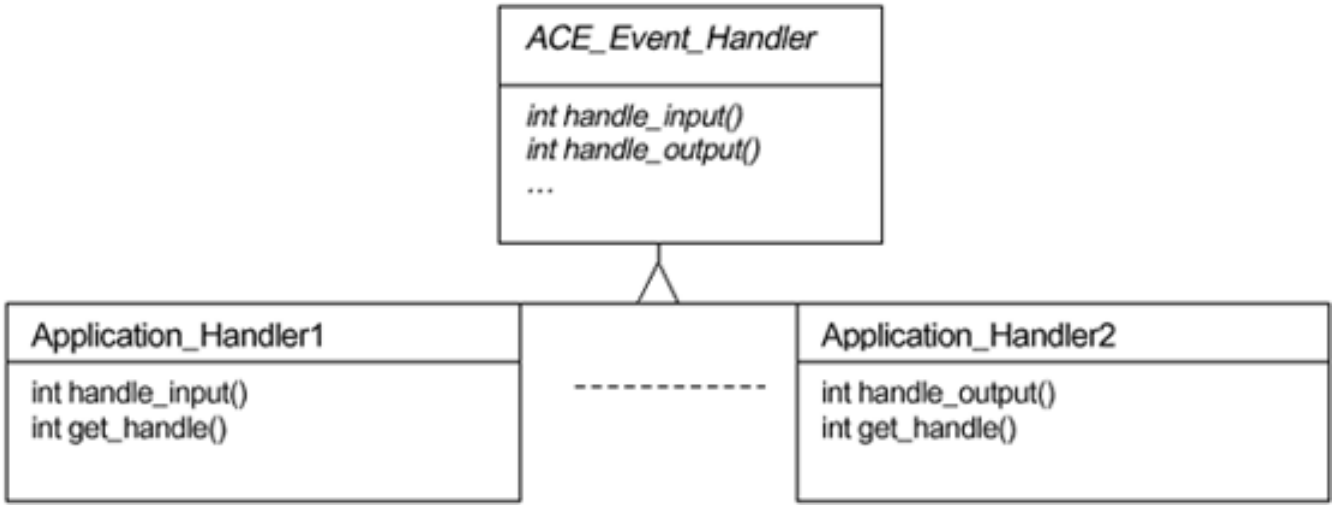
```
#include <ace/Event_Handler.h>
class TimeoutHandler : public ACE_Event_Handler
{
public:
  virtual int handle_timeout(const ACE_Time_Value& tv, const void* arg)
};
```

实现:

```
#include "TimeoutHandler.h"
int TimeoutHandler::handle_timeout(const ACE_Time_Value& tv, const void* arg)
{
  ACE_UNUSED_ARG(tv); // config-macros.h
  ACE_UNUSED_ARG(arg);
  cout << "Current largest prime is " << Calculator::getLargestPrime()
       << " and nonprime is " << Calculator::getLargestNonPrime() << endl;

  return 0;
}
```

ACE_Event_Handler



使用ACE_Reactor基本上有三个步骤

1. 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.
2. 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.
3. 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.

ACE_Event_Handler 对不同事件的处理

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()	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设备的输出. 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.

ACE_Event_Handler事件类型

掩码	回调方法	何时	和谁一起使用
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::EXCEPT_MASK	handle_exception()	指定异常事件,如紧急数据出现在Socket上	register_handler()
ACE_Event_Handler::SIGNAL_MASK	handle_close()	由反应器在handle_signal()返回-1时传递	handle_signal()
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()

使用ACE_Event_Handler

Ctrl-C的信号处理

定义：

```
#include <ace/Event_Handler.h>
class ExitHandler : public ACE_Event_Handler
{
public:
  virtual int handle_signal(int signum, siginfo_t*, ucontext_t*);
};
```

实现:

```
#include "ExitHandler.h"
int ExitHandler::handle_signal(int signum, siginfo_t*, ucontext_t*)
{
  ACE_UNUSED_ARG(signum);
  ACE_DEBUG((LM_DEBUG, "Server exits on Ctrl-C\n"));
  ACE_Reactor::instance()->end_reactor_event_loop();
  return 0;
}
```

处理信号

实验:

1、用Reactor捕捉信号
2、处理 Ctrl-C 信号
3、实现 handle_signal() 回调函数

定义：

```
#include <ace/Event_Handler.h>
class ExitHandler : public ACE_Event_Handler
{
public:
  virtual int handle_signal(int signum, siginfo_t*, ucontext_t*);
};
```

实现:

```
#include "ExitHandler.h"
int ExitHandler::handle_signal(int signum, siginfo_t*, ucontext_t*)
{
  ACE_UNUSED_ARG(signum);
  ACE_DEBUG((LM_DEBUG, "Server exits on Ctrl-C\n"));
  ACE_Reactor::instance()->end_reactor_event_loop();
  return 0;
}
```

处理信号事件

实验：

1、handle_signal ()
2、处理 SIGINT 和 SIGBREAK 信号

处理通知事件

实验：

1、用户自定义的通知
2、notify

ACE_Timer_Queue

ACE_Timer_Queue是抽象类,它结合设计模式、Hook方法和模板参数来提供四种定时器队列实现.

在事件超时的时候适当的调用事件处理器的handle_timeout()方法.为调度这样的定时器,反应器拥有一个schedule_timer()的方法.该方法接受事件处理器,以及以ACE_Time_Value对象形式出现的延迟参数.

此外,可以指定时间间隔,使定时器在它超时后自动恢复 .

反应器在内部维护ACE_Timer_Queue, 它以定时器要被调度的顺序对它们进行维护.实际使用的用于保存定时器的数据结构可以通过反应器的set_timer_queue()方法进行改变.反应器有若干不同的定时器结构可用:包括定时器轮(timer wheel)、定时器列表(timer listl)、定时器堆(timer heap)和定时器哈希(timer hash).

ACE_Timer_Queue实现

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)

使用ACE_Timer_Queue

```
#include "ace/Time_Value.h"
#include "ace/Log_Msg.h"
#include "ace/Synch.h"
#include "ace/Reactor.h"
#include "ace/Event_Handler.h"

class ACETimer : public ACE_Event_Handler
{
public:
   virtual int handle_timeout(const ACE_Time_Value &current_time,
      const void *act /* = 0 */)
   {
      const int *num = ACE_static_cast(const int*,act);
      ACE_DEBUG((LM_DEBUG, ACE_TEXT("%d "),num));
      return 0;
   }
};

int ACE_TMAIN(int, char *[])
{
   ACETimer *timer = new ACETimer;
   ACE_Time_Value tv(5),tv2(3);
   ACE_Reactor::instance()->schedule_timer(timer,(const int*)44,tv,tv2);
   while(1)
      ACE_Reactor::instance()->handle_events();
   return 0;
}
```

ACE_Reactor Class

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或WaitForMultipleObjects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.

六类方法:

1. 反应器初始化和析构方法
2. 事件处理器管理方法
3. 事件循环管理方法
4. 定时器管理方法
5. 通知方法
6. 实用方法

ACE_Reactor Class

1. 反应器初始化和析构方法

方法	描述
ACE_Reactor, open()	这两个方法创建并初始化反应器的实例
~ACE_Reactor, close()	这两个方法清理反应器在初始化时所分配的资源

2. 事件处理器管理方法

方法	描述
register_handler()	为基于I/O和信号的事件登记事件处理器
remove_handler()	移除某事件处理器，使其不再参与基于I/O和信号的事件分派
suspend_handler()	暂时停止分派事件给事件处理器
resume_handler()	恢复为先前挂起的事件处理器分派事件
mask_ops()	获取、设置、增加或是清除与某事件处理器相关联的事件类型及其分派掩码
schedule_wakeup()	将指定的掩码增加到某事件处理器的条目中，该处理器在此之前必须已经通过register_handler()作了登记
cancel_wakeup()	从某事件处理器的条目中清除指定的掩码，但并不将此处理器从反应堆中移除

ACE_Reactor Class

3. 事件循环管理方法

方法	描述
handle_events()	等待事件发生，并随即分派与之相关联的事件处理器。可通过超时参数限制花费在等待事件上的时间
run_reactor_event_loop()	反复调用handle_events()方法，直至失败，或是reactor_event_loop_done()返回1，或是发生超时(可选)
end_reactor_event_loop()	指示反应器关闭其事件循环
reactor_event_loop_done()	在反应器的事件循环被end_reactor_event_loop()调用结束时返回1

4. 定时器管理方法

方法	描述
schedule_timer()	登记一个事件处理器，它将在用户规定的时间之后被执行
cancel_timer()	取消一个或多个先前登记的定时器

ACE_Reactor Class

5. 通知方法

方法	描述
notify()	将事件(及可选的事件处理器)插入反应器的事件检测中，从而使其在反应器下次等待事件时被处理
max_notify_iterations()	设置反应器将在其通知机制中分派的处理器的最大数目
purge_pending_notification()	从反应器的通知机制中清除指定的事件处理器或所有的事件处理器

6. 实用方法

方法	描述
instance()	静态方法,返回指向一个单体(Singleton) ACE_Reactor的指针。这个ACE_Reactor是通过Singleton模式，结合Double-Checked Locking Optimization 模式来创建和管理的
owner()	使某个线程“拥有”反应器的事件循环

实现一个定时器

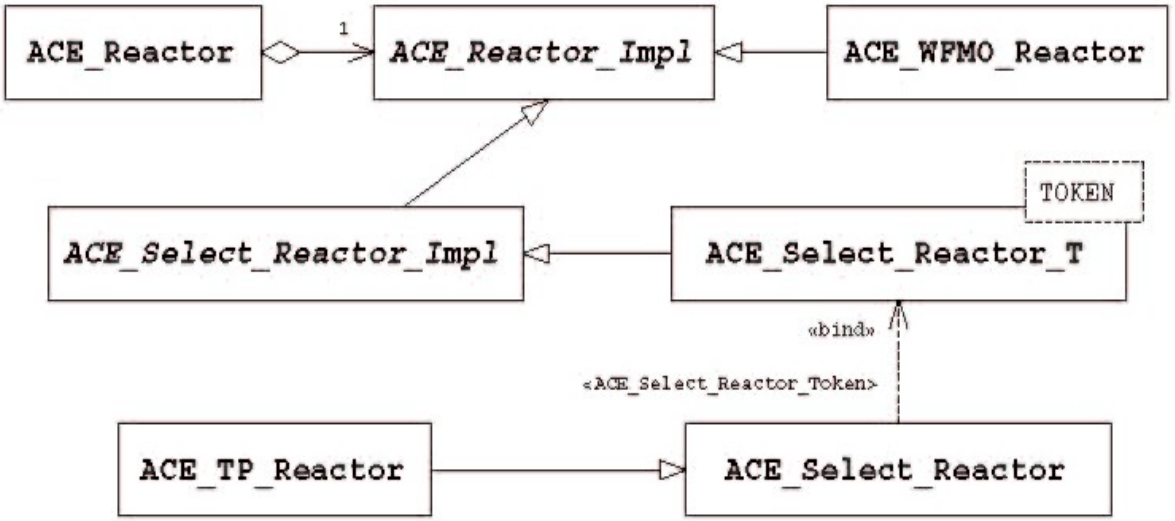
通过Reactor机制, 可以很容易的实现定时器的功能:

1. 编写一个事件反应器重载handle_timeout()方法, 该方法是定时器的触发时间到时, 会自动触发该方法;
2. 通过Reactor的schedule_timer()方法注册定时器;
3. 启动Reactor的handle_events()事件分发循环;
4. 当不想使用定时器时,可以通过Reactor的cancel_timer()方法注销定时器.

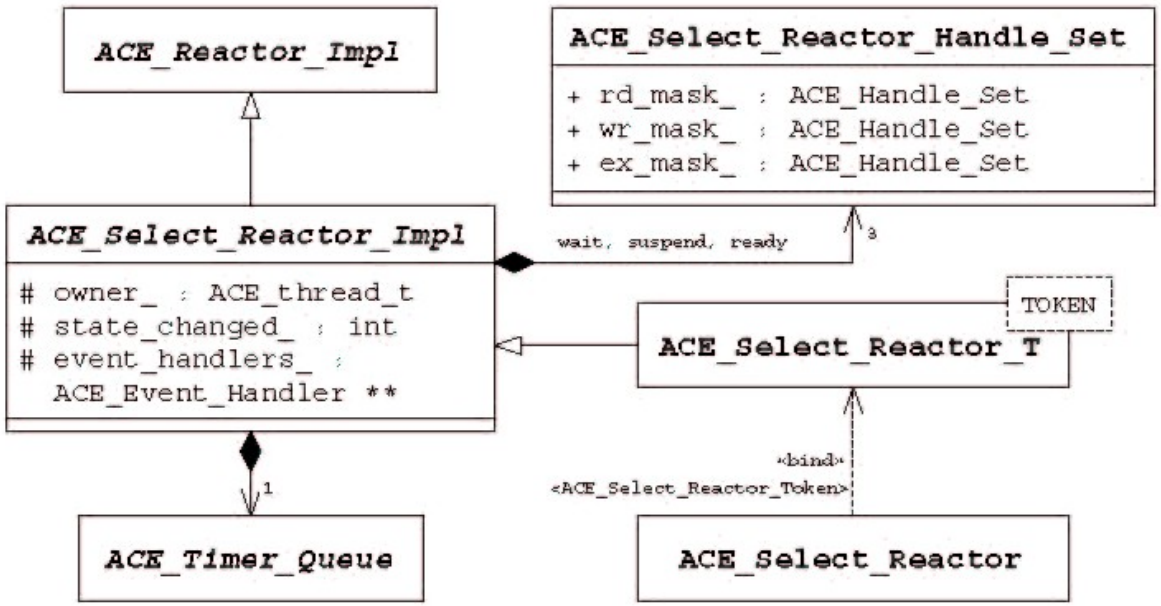
Reactor实现

为了实现一系列不同操作系统同步事件多路分离机制，包括select(),WaitForMultipleObjects(),XtAppMainLoop(),/dev/poll,ACE提供了Reactor的三种主要的实现方式.

* ACE_Select_Reactor ： 使用Select()同步事件多路分离器来检测I/O和定时器事件,并有序地结合了对POSIX信号的处理
* ACE_TP_Reactor ： 使用Leader/Follower模式来将ACE_Select_Reactor事件处理扩展到线程池(thread-pool based event dispatching)
* ACE_WFMO_Reactor ： 使用Windows WaitForMultipleObjects()事件多路分离器函数来检测socket I/O、超时以及Windows同步事件



ACE_Select_Reactor Class



- Select()函数是最常见的同步事件多路分离器
- ACE_Select_Reactor是ACE_Reactor在除Windows以外的所有平台的缺省实现

控制ACE_Select_Reactor大小

改变由ACE_Select_Reactor的某个实例所管理的事件处理器的数目

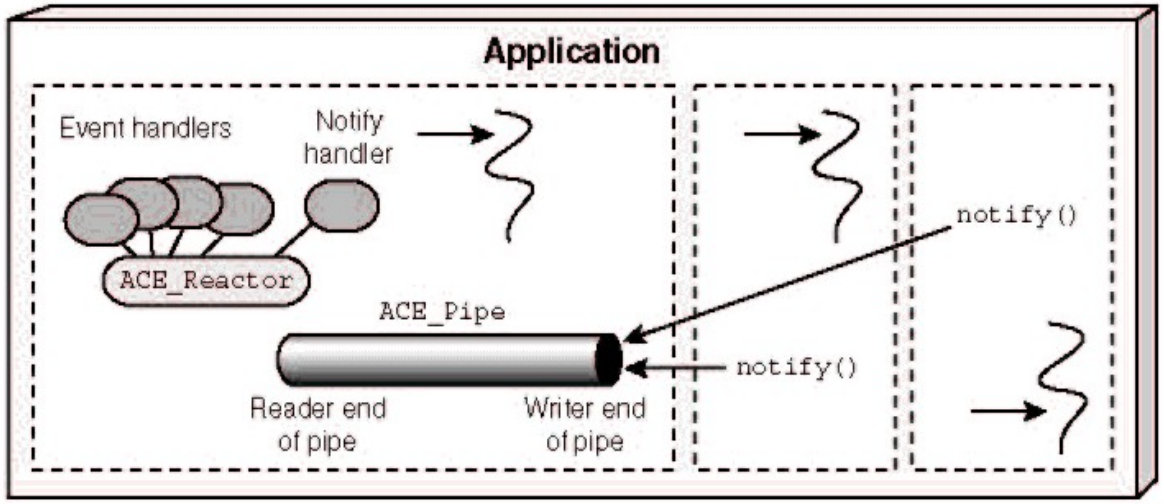
- ACE_Select_Reactor管理的事件处理器的数目缺省值是FD_SETSIZE宏的值
- 创建< FD_SETSIZE缺省尺寸的ACE_Select_Reactor,只需将该值传入ACE_Select_Reactor::open()方法,不需重新编译ACE库
- 创建> FD_SETSIZE缺省尺寸的ACE_Select_Reactor，需要在$ACE_ROOT/ace/config.h文件中改变FD_SETSIZE的值，并重新编译ACE库

处理大量I/O句柄的其他一些选择

- 可考虑使用某些UNIX平台上的ACE_Dev_Poll_Reactor
- 也可选择使用基于ACE Proactor框架的异步I/O

ACE_Select_Reactor的通知机制

ACE_Select_Reactor通过ACE_Pipe来实现其通知机制,ACE_IPC是一个双向IPC机制，用以在OS内核中传输数据



Pipe的两端扮演着下面角色

- writer
- reader

ACE_TP_Reactor

使用Leader/Follower模式来将ACE_Selectr_Reactor事件处理扩展到线程池

```
ACE_TP_Reactor *tp_reactor = new ACE_TP_Reactor;
ACE_Reactor *my_reactor = new ACE_Reactor (tp_reactor, 1)
```

ACE线程池

半同步/半异步（half-sync/half-asynv model）
一个侦听线程会异步地接收请求，并在某个队列中缓冲它们。另一组工作者线程负责同步地处理这些请求。
层次：
1.异步层：负责接收异步请求；
2.排队层：负责对请求进行缓冲；
3.同步层：含有若干阻塞在排队层上的控制线程；

优点：
＋排队层有助于处理爆发的客户，如果没有线程可以用于处理请求，这些请求会放在排队层中。
＋同步层很简单，并且与任何异步处理细节都无关。每个同步线程都阻塞在排队层上，等待请求到达。

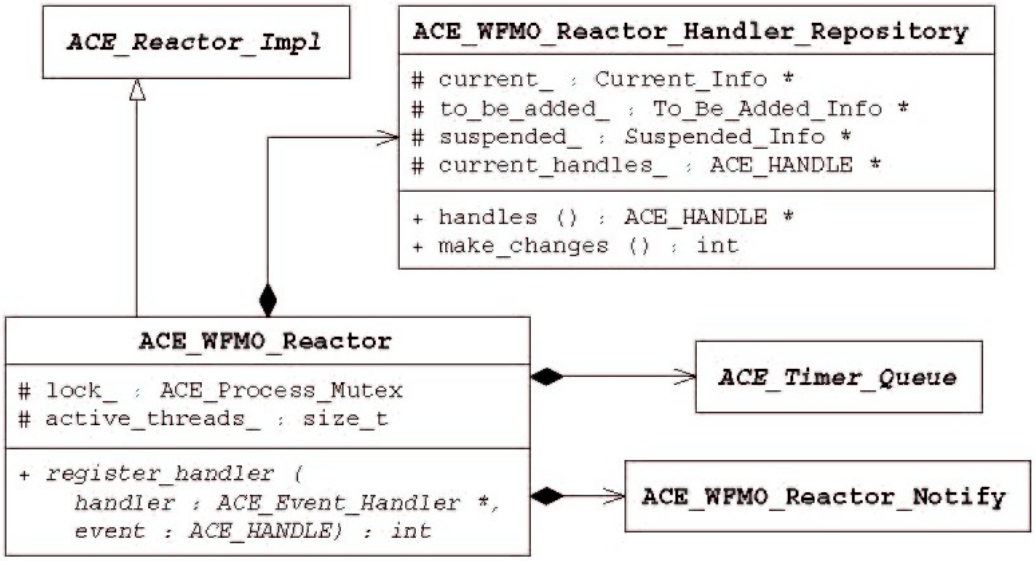
缺点：
－排队层会发生线程切换，造成同步和上下文切换开销。可能会产生数据拷贝和缓存相干性开销。
－不能把任何请求信息保存在栈上或线程专有存储中，因为请求是在另外的工作者线程中处理的。

领导者/跟随者（leader/followers model）
有一个线程是领导者，其余线程是在线程池中的跟随者。当请求到达时，领导者会拾取它，并从跟随者中选取一个新的领导者，然后继续处理请求。

优点：
＋性能提高，因为不用进行线程间上下文切换。

缺点：
－不容易处理爆发的客户，因为不一定有显示的排队层。
－实现更复杂。

ACE_WFMO_Reactor



ACE_Dev_Poll_Reactor

Linux下的EPOLL(一个IO多路服用模型),是Linux大规模接入的制胜法宝.
ACE对EPOLL进行了封装. 要让其支持EPOLL需要
在config_linux.h里添加这个语句:
#define ACE_HAS_EVENT_POLL

并重新编译.

用使用EPOLL, 需要预先设置栈内存和文件描述符上限.

如:
ulimit -n 16384
ulimit -s 4096

ACE_Dev_Poll_Reactor默认使用Poll,ACE_TP_Reactor默认使用Select.

Reactor设计准则 (1/3)

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而相反, 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.

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.

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).

Reactor设计准则 (2/3)

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.

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.

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.

Reactor设计准则 (3/3)

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.

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.

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.

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.

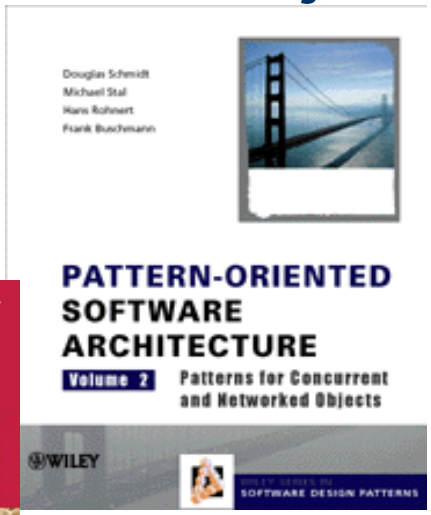
设计准则14：在handle_close方法中调用remove_handler时,总是传递给它DONT_CALL标志.该准则确保ACE_Reactor不会递归地调用handle_close方法.

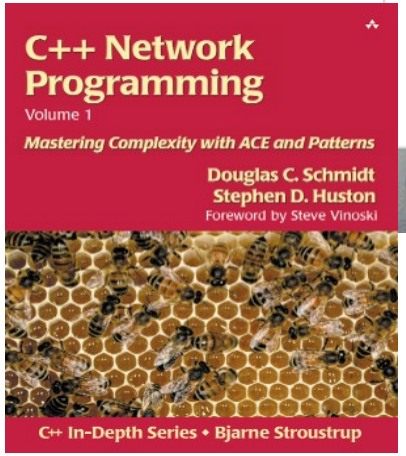
内容回顾

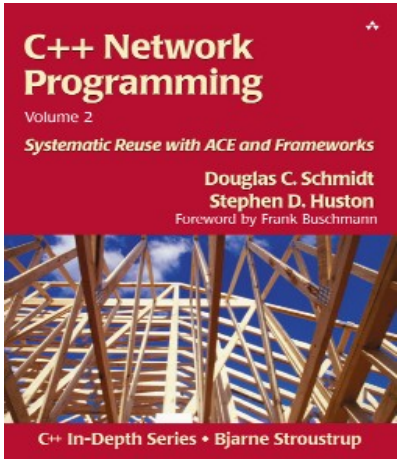
- 如何访问OS服务
- TCP/IP Socket编程接口
- 使用ACE的UDP类进行网络编程
- 进程和线程管理
- 反应堆（Reactor）和Proactor框架：用于事件多路分离和分发的体系结构
- 为I/O，定时器，信号处理实现事件处理器
- ACE Task框架：并发模式
- 消息队列(Message Queue)
- 接受器（Acceptor）和连接器（Connector）框架：接受连接模式

参考资料

- **Patterns & frameworks for concurrent & networked objects**
  - `www.posa.uci.edu`
- **ACE & TAO open-source middleware**
  - `www.cs.wustl.edu/~schmidt/ACE.html`
  - `www.cs.wustl.edu/~schmidt/TAO.html`







- **ACE research papers**
  - `www.cs.wustl.edu/~schmidt/ACE-papers.html`
- **Extended ACE & TAO tutorials**
  - **UCLA extension, January 21-23, 2004**
  - `www.cs.wustl.edu/~schmidt/UCLA.html`
- **ACE books**
  - `www.cs.wustl.edu/~schmidt/ACE/`

结束

谢谢大家！

ihuihoo@gmail.com
http://www.huihoo.com