ACE 异步、并发、消息

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

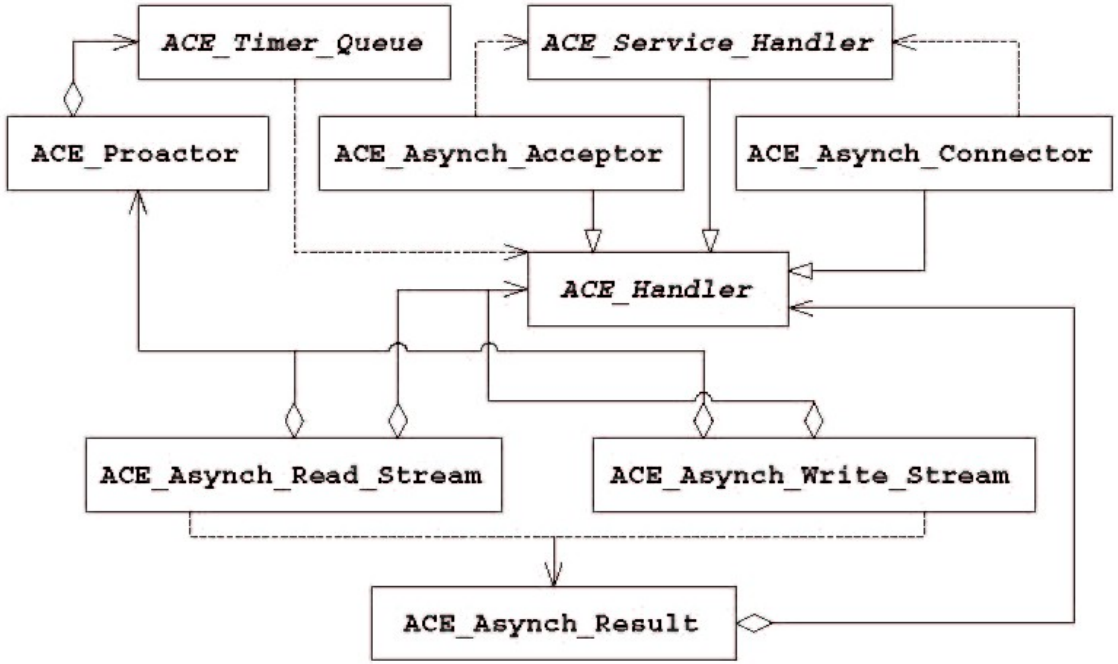
内容安排

- 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

ACE_Proactor(前镊器)

前摄器模式支持多个事件处理器的多路分离和分派,这些处理器由异步事件的完成来触发.通过集成完成事件(completion event)的多路分离和相应的事件处理器的分派,该模式简化了异步应用的开发.

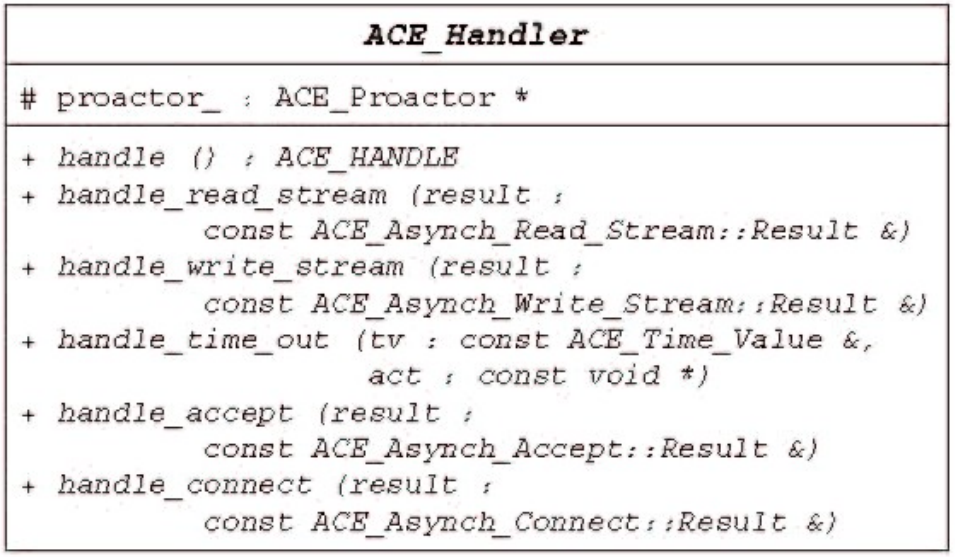
它封装Windows NT上的I/O完成端口,以及POSIX平台上的aio API.



ACE_Proactor框架类

ACE类	描述
ACE_Handler	定义用于接收异步I/O操作的结果和处理定时器到期的接口
ACE_Asynch_Read_Stream ACE_Asynch_Write_Stream ACE_Asynch_Result	在I/O流上发起异步读和异步写操作,并使每个操作与一个用于接收操作结果的ACE_Handler对象关联起来.
ACE_Asynch_Acceptor ACE_Asynch_Connector	Acceptor-Connector模式的一种实现,它异步地建立新的TCP/IP连接.
ACE_Service_Handler	定义ACE_Asynch_Acceptor和ACE_Asynch_Connector连接工厂的目标,并提供挂钩方法来初始化通过TCP/IP连接的服务.
ACE_Proactor	管理定时器和异步I/O完成事件多路分离。这个类是ACE Reactor框架中的ACE_Reactor类的类似物.

ACE_Handler 类



ACE_Handler是ACE Proactor框架中的所有异步完成处理器的基类

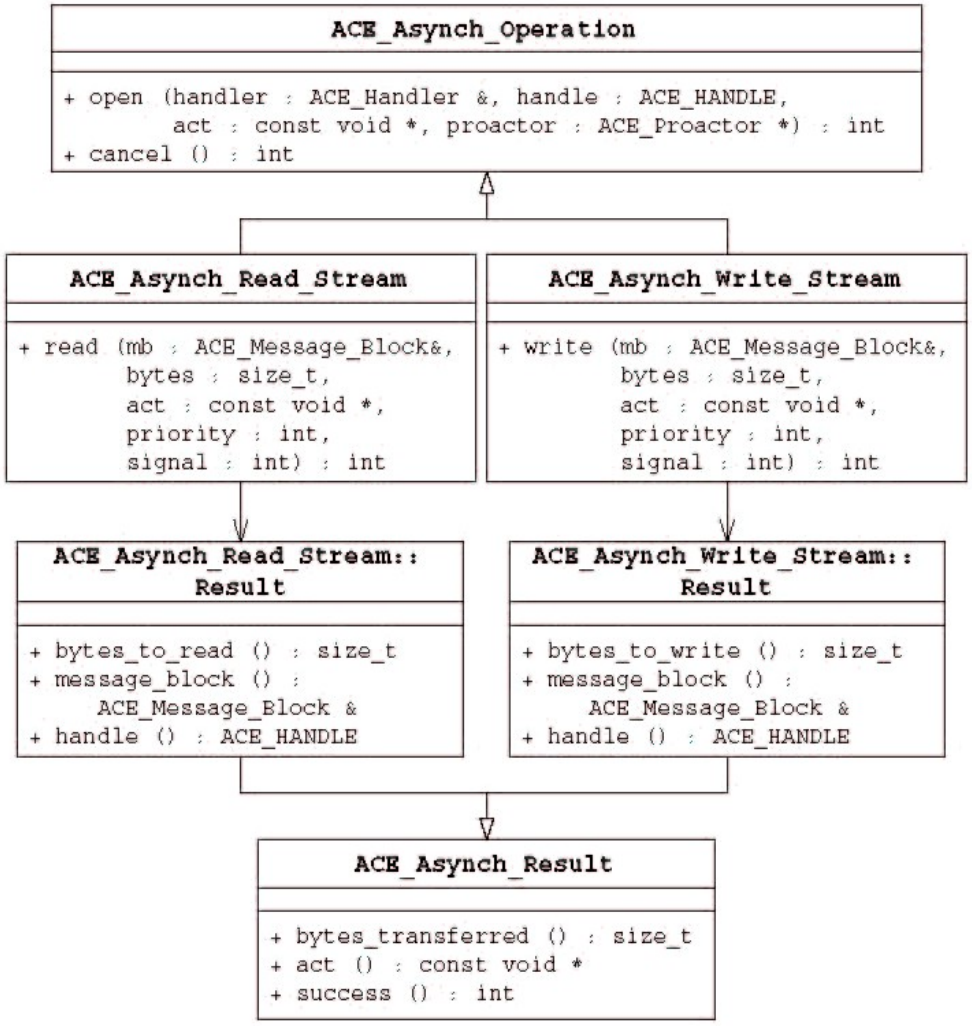
。它提供了多个Hook方法来处理ACE中定义的所有异步I/O操作的完成,包括连接建立和在IPC流上的I/O操作

。提供一个Hook方法来处理定时器到期

使用ACE_Handler

```
#include "ace/Proactor.h"
#include "ace/OS_main.h"

#if defined (ACE_HAS_WIN32_OVERLAPPED_IO) || defined (ACE_HAS_AIO_CALLS)

class Timeout_Handler : public ACE_Handler
{
public:
  Timeout_Handler (void)
    : count_ (0),
      start_time_ (ACE_OS::gettimeofday ())  { }

  virtual void handle_time_out (const ACE_Time_Value &tv,
                                const void *arg)
    {
      ACE_DEBUG ((LM_DEBUG, "(%t) %d timeout occurred for %s @ %d.\n",
                  ++count_,
                  (char *) arg,
                  (tv - this->start_time_).sec ()));
    }

private:
  int count_;
  ACE_Time_Value start_time_;
};
```

ACE Async Read/Write Stream Classes



ACE_Asynch_Read_Stream和ACE_Asynch_Write_Stream是两个工厂类,它们使得应用能够发起可移植的异步read()和write()操作

另外还提供:

ACE_Asynch_Read_Dgram

ACE_Asynch_Write_Dgram

ACE_Asynch_Read_File

ACE_Asynch_Write_File

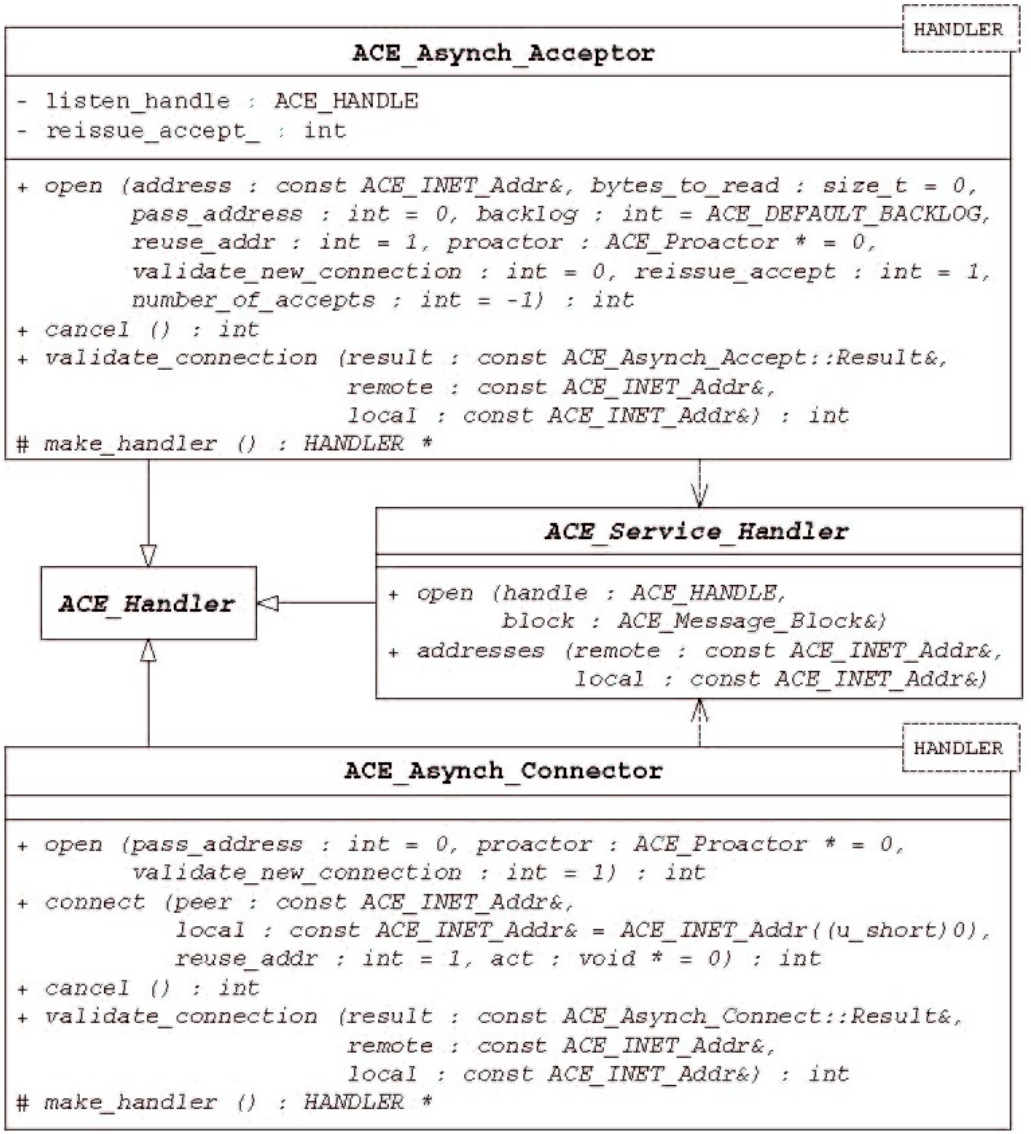
ACE_Asynch_Transmit_File

使用ACE Async Read/Write Stream

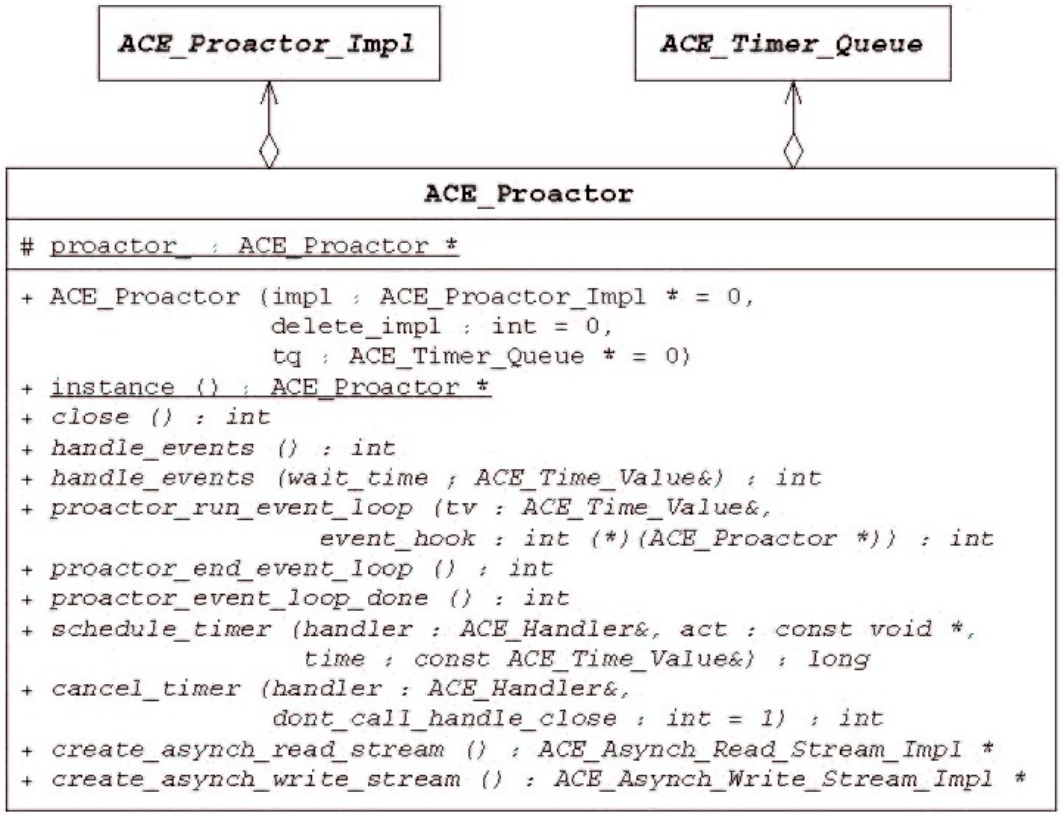
```
#include "ace/Message_Queue.h"
#include "ace/Asynch_IO.h"
#include "ace/OS.h"
#include "ace/Proactor.h"
#include "ace/Asynch_Connector.h"

client.cpp // 客户端异步写数据, 异步写完成后会调用此函数
virtual void handle_write_dgram
(const ACE_Asynch_Write_Stream::Result &result)
{
    ACE_Message_Block &mb = result.message_block ();
    mb.release();
    return;
}
server.cpp // 服务端异步读数据, 异步读完成后会调用此函数
virtual void handle_read_stream
(const ACE_Asynch_Read_Stream::Result &result)
{
    ACE_Message_Block &mb = result.message_block ();
    if (!result.success () || result.bytes_transferred () == 0)
    {
       mb.release ();
       delete this;
       return;
    }
```

前镊式Acceptor-Connector类

- **ACE_Asynch_Acceptor**是Acceptor-Connector模式中接收器角色的另一种实现
- **ACE_Asynch_Connector**是Acceptor-Connector模式中连接器角色的另一种实现



ACE_Proactor类



我们通过两个步骤来处理异步I/O操作:发起(initiation)和完成(completion)

ACE_Proactor Class

1. 生命周期管理方法

方法	描述
ACE_Proactor , open()	这两个方法创建并初始化前镊器的实例
~ACE_Proactor, close()	这两个方法清理前镊器初始化时所分配的资源
instance()	静态方法,返回指向一个单体(Singleton) ACE_Proactor的指针.这个ACE_Proactor是通过Singleton模式,结合Double-Checked Locking Optimization 模式来创建和管理的

2. 事件循环管理方法

方法	描述
handle_events()	等待完成事件发生,并随即分派与之相关联的完成处理器.可通过超时参数限制花费在等待事件上的时间
proactor_run_event_loop()	反复调用handle_events()方法,直至失败,或是proactor_event_loop_done()返回1,或是发生超时(可选)
proactor_end_event_loop()	指示前镊器关闭其事件循环
proactor_event_loop_done()	在前镊器的事件循环被proactor_end_event_loop()调用结束时返回1

ACE_Proactor Class

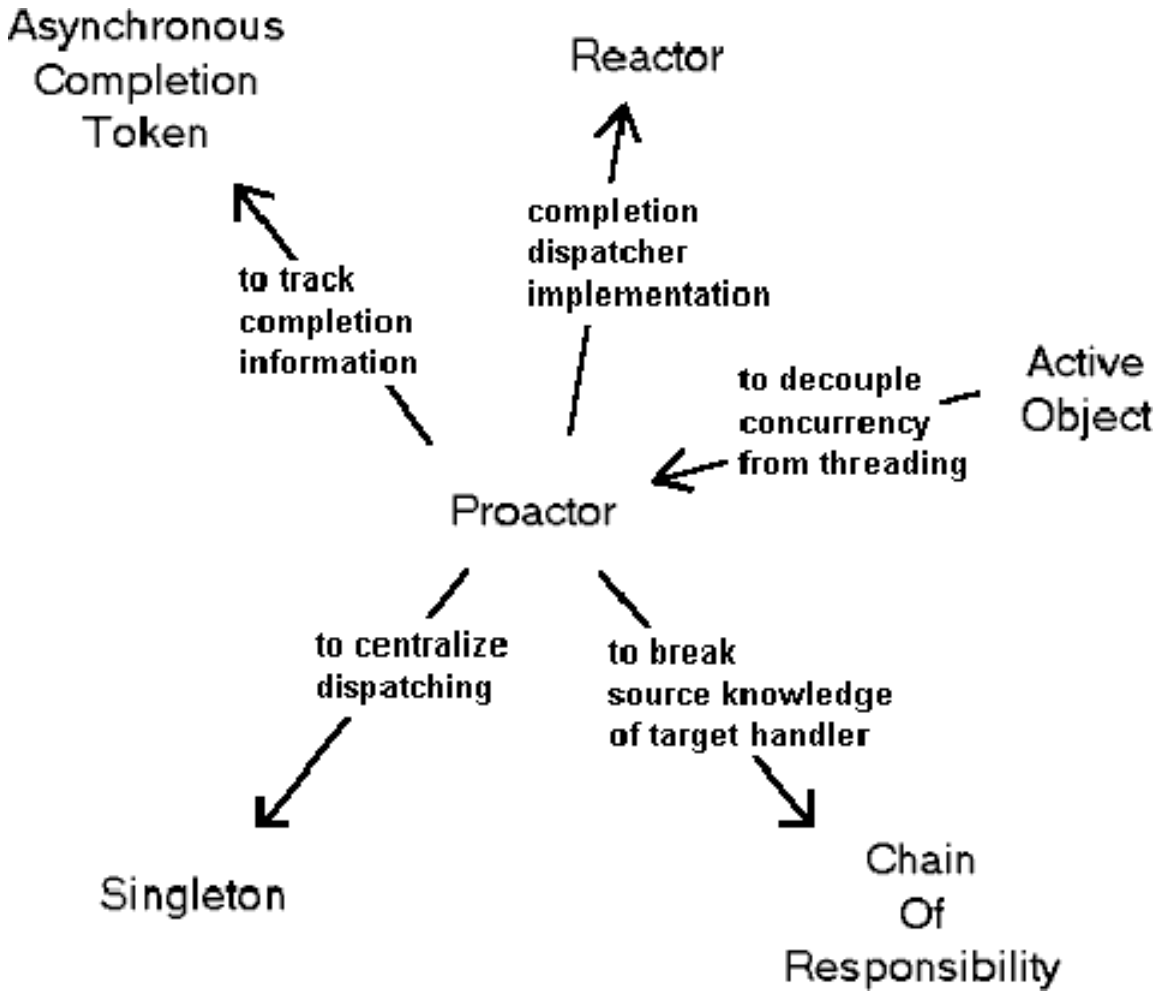
3. 定时器管理方法

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

4. I/O操作facilitator方法

方法	描述
create_asynch_read_stream()	创建ACE_Asynch_Read_Stream_Impl 的针对特定平台的子类的实例,适用于发起异步read()操作
create_asynch_write_stream()	创建ACE_Asynch_Write_Stream_Impl 的针对特定平台的子类的实例,适用于发起异步write()操作

与Proactor(前镊器)相关的模式



与Proactor(前摄器)相关的模式

异步完成令牌(ACT)模式通常与前摄器模式结合使用.当Asynchronous Operation完成时,应用可能需要比简单的通知更多的信息来适当地处理事件.异步完成令牌模式允许应用将状态高效地与Asynchronous Operation的完成相关联.

前摄器模式还与**观察者(Observer)模式**(在其中,当单个主题变动时,相关对象也会自动更新)有关.在前摄器模式中,当来自多个来源的事件发生时,处理器被自动地通知.一般而言,前摄器模式被用于异步地将多个输入源多路分离给与它们相关联的事件处理器,而观察者通常仅与单个事件源相关联.

前摄器模式可被认为是**同步反应堆模式**的一种异步的变体.反应堆模式负责多个事件处理器的多路分离和分派;它们在可以*同步地发起*操作而不会阻塞时被触发.相反,前摄器模式也支持多个事件处理器的多路分离和分派,但它们是被*异步事件的完成*触发的.

主动对象(Active Object)模式使方法执行与方法调用去耦合.前摄器模式也是类似的,因为Asynchronous Operation Processor代表应用的Proactive Initiator来执行操作.就是说,两种模式都可用于实现Asynchronous Operation.前摄器模式常常用于替代主动对象模式,以使系统并发策略与线程模型去耦合.

前摄器可被实现为**单体(Singleton)**.这对于在异步应用中,将事件多路分离和完成分派集中到单一的地方来说是有用的.

责任链(Chain of Responsibility, COR)模式使事件处理器与事件源去耦合.在Proactive Initiator与Completion Handler的隔离上,前摄器模式也是类似的.但是.在COR中,事件源预先不知道哪一个处理器将被执行(如果有的话).在前摄器中,Proactive Initiator完全知道目标处理器.但是,通过建立一个Completion Handler(它是由外部工厂动态配置的责任链的入口),这两种模式可被结合在一起.

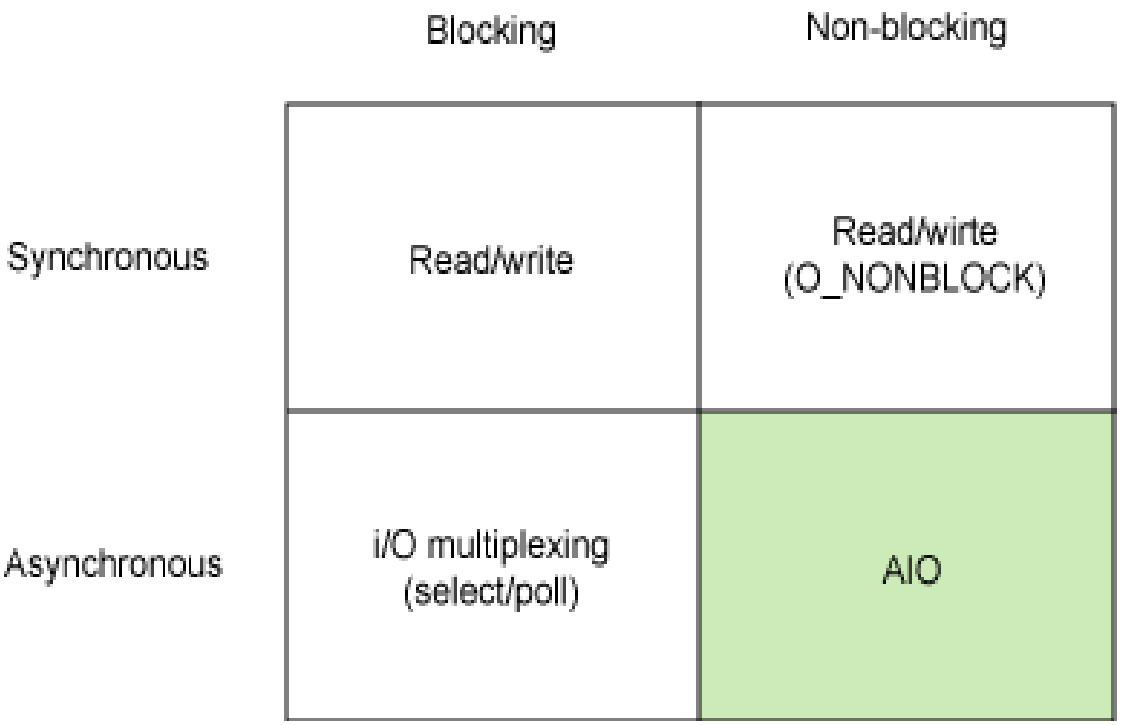
Proactor POSIX实现

- 在POSIX系统上的ACE Proactor实现给出了多种用于发起I/O操作及检测其完成的机制
- Many UNIX AIO implementations are buggy, however…

ACE Proactor变种	描述
ACE_POSIX_AIOCB_Proactor	这个实现平行地维护了aiocb结构和Result对象的列表.每个未完成的操作由各个列表中的一个条目代表.aio_suspend()函数挂起事件循环,直至有一个或多个异步I/O操作完成.
ACE_POSIX_SIG_Proactor	这个实现派生自ACE_POSIX_AIOCB_Proactor ,但是使用了POSIX实时信号来检测异步I/O完成.事件循环使用了sigtimedwait()和sigwaitinfo()函数来执行循环和获取与已完成操作有关的信息.使用这个前镊器启动的每个异步I/O操作都拥有一个唯一的与其aiocb关联在一起的值,用于与通知操作完成的信号进行通信.这一设计使其更易于定位aiocb和它的平行的Result对象,并分派正确的完成处理器
ACE_SUN_Proactor	这个实现也派生自ACE_POSIX_AIOCB_Proactor ，没使用POSIX 4 AIOS设施，而使用SUN自己的方式：使用aiowait()函数来检测I/O完成

- 在支持POSIX 4 AIO的平台上, ACE_POSIX_SIG_Proactor是缺省的前镊器实现
- 在linux上, ACE_POSIX_AIOCB_Proactor是缺省的前镊器实现

基本 Linux I/O 模型的简单矩阵



- 同步阻塞 I/O
- 同步非阻塞 I/O
- 异步阻塞 I/O
- 异步非阻塞 I/O (AIO)

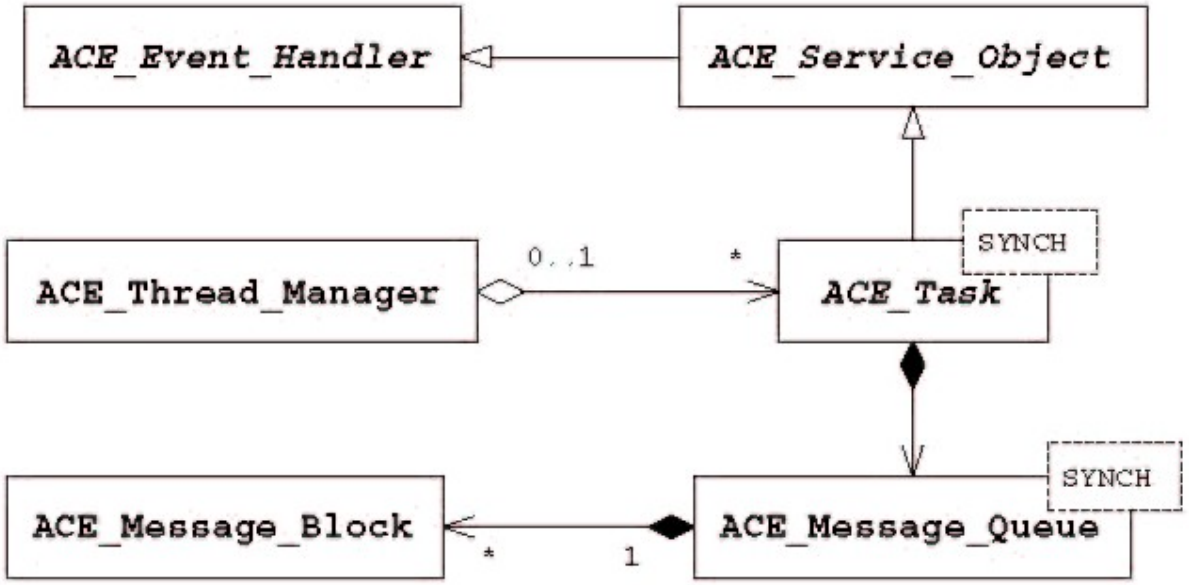
使用异步 I/O 能大大提高应用程序的性能. aiocb(Asynchronous I/O Control Block)结构

使用ACE_POSIX_AIOCB_Proactor

通过ACE_wrappers/examples/Reactor/Proactor$ vim test_aiocb.cpp 测试操作系统是否支持POSIX AIO

```
#if defined (ACE_HAS_WIN32_OVERLAPPED_IO) || defined (ACE_HAS_AIO_CALLS)

int
ACE_TMAIN (int, ACE_TCHAR *[])
{
  Timeout_Handler handler;

 // Register a 2 second timer.
 ACE_Time_Value foo_tv (2);
  if (ACE_Proactor::instance ()->schedule_timer (handler,
                                        (void *) "Foo",
```

ACE Task框架



ACE类	描述
ACE_Message_Block	实现Composite模式，使开发者能高效地操作定长和变长的消息
ACE_Message_Queue	提供一种进程内的消息队列，使应用能够在进程中的线程间传递和缓冲消息
ACE_Thread_Manager	允许应用可移植地创建和管理一个或多个线程的生命周期、同步以及各种属性
ACE_Task	允许应用创建被动或主动的对象，解除不同处理单元的耦合，使用消息来交流请求、响应、数据以及控制消息，并且可以顺序地或并发地排队和处理消息

用ACE_Thread创建线程

```
#include "ace/Thread.h"
#include "ace/OS.h"
#include <iostream>
using namespace std;

void* worker(void *arg)
{
    for(int i=0;i<10;i++)
    {
        ACE_OS::sleep(1);
        cout<<endl<<"hello world"<<endl;
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    ACE_thread_t threadId;
    ACE_hthread_t threadHandle;

    ACE_Thread::spawn(
        (ACE_THR_FUNC)worker,          //线程执行函数
        NULL,                          //执行函数参数
        THR_JOINABLE | THR_NEW_LWP,
        &threadId,
        &threadHandle
        );

    ACE_Thread::join(threadHandle);
    return 0;
}
```

用ACE_Task创建线程

```
#include "ace/Task.h"
#include "ace/OS.h"
#include <iostream>
using namespace std;

class TaskThread: public ACE_Task<ACE_MT_SYNCH>
{
public:
    virtual int svc(void)
    {
        for(int i=0;i<10;i++)
        {
            ACE_OS::sleep(1);
            cout<<endl<<"hello thread1"<<endl;
        }
        return 0;
    }
};
int main(int argc, char *argv[])
{
    TaskThread task;
    task.activate();

    while(true)
        ACE_OS::sleep(10);
    return 0;
}
```

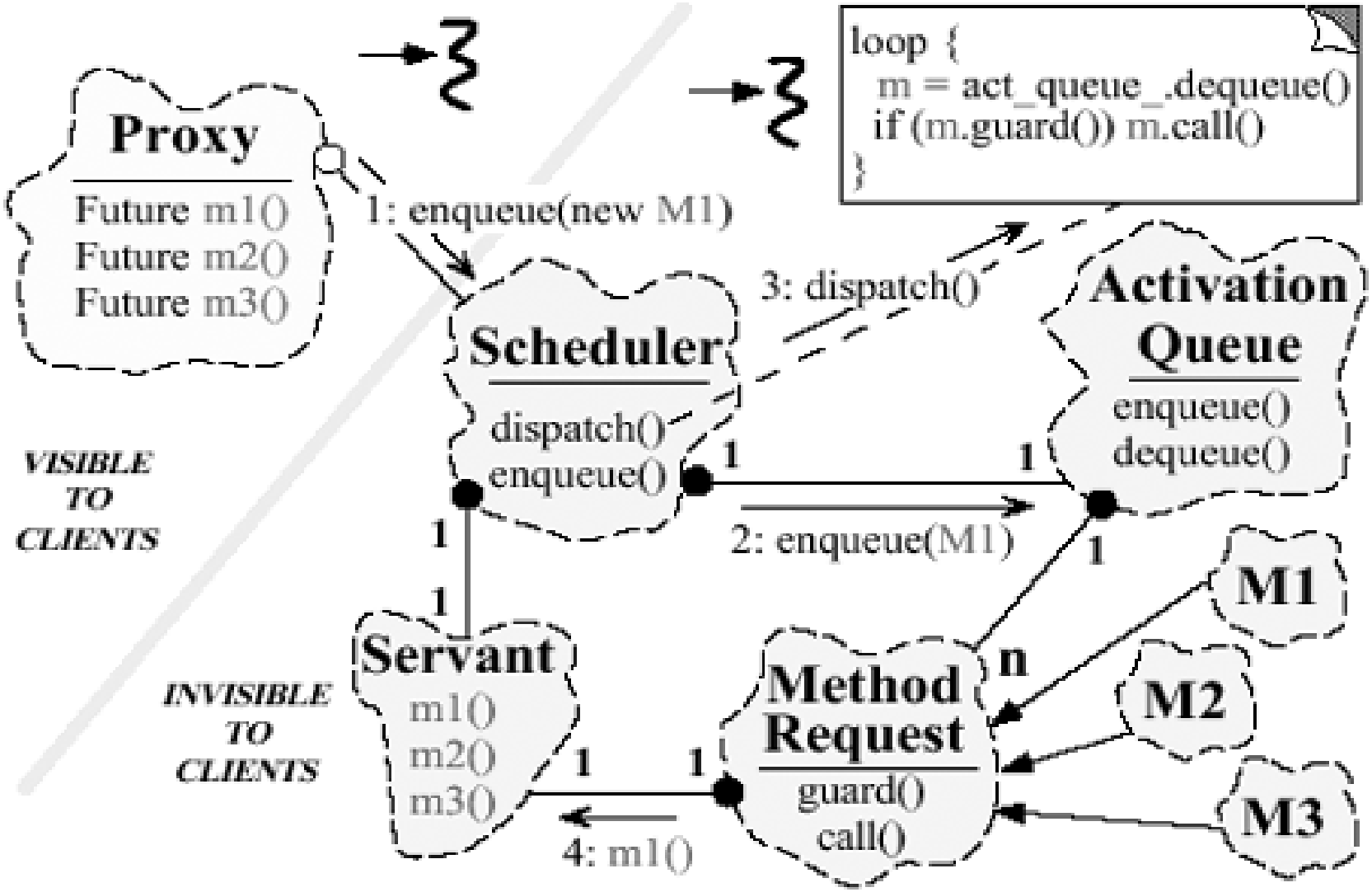
任务和主动对象(Active Object): 并发编程模式

ACE_Task：是ACE中的任务或主动对象“处理结构”的基类,ACE中使用它来实现主动对象模式.ACE提供了主动对象模式中的Method Request、Activation Queue和Future组件的可复用实现.

主动对象由以下组件组成：代理(Proxy)表示对象的接口，仆人(Servant)提供对象的实现. 代理和仆人运行在分离的线程中, 以使方法调用和方法执行能并发运行. 代理在客户线程中运行, 而仆人在不同的线程中运行. 在运行时, 代理将客户的方法调用(Method Invocation)转换为方法请求(Method Request), 并由调度者(Scheduler)将其存储在启用队列(Activation Queue)中. 调度者持续地运行在与仆人相同的线程中, 当启用队列中的方法请求变得可运行时, 就将它们出队, 并分派给实现主动对象的仆人. 客户可通过代理返回的"期货"(Future)获取方法执行的结果.

主动对象模式用于降低方法执行和方法调用之间的耦合.拥有更好的OO特性,更适合构建多线程程序,简化并发和异步的网络操作.

主动对象模式的结构


loop {
m = act_queue_.dequeue()
if (m.guard()) m.call()
}
Proxy
Future m1()
Future m2()
Future m3()
1: enqueue(new M1)
Scheduler
dispatch()
enqueue()
3: dispatch()
Activation Queue
enqueue()
dequeue()
2: enqueue(M1)
VISIBLE TO CLIENTS
INVISIBLE TO CLIENTS
Servant
m1()
m2()
m3()
4: m1()
Method Request
guard()
call()
M1
M2
M3
1
n

Active Object 的参与者

主动对象模式中有六个关键的参与者:

1. 代理(Proxy)
代理提供一个接口,允许客户使用标准的强类型程序语言特性,而不是在线程间传递松散类型的消息,来调用主动对象的可公共访问的方法.当客户调用代理定义的方法时,就会在调度者的启用队列上触发方法请求对象的构造和排队,所有这些都发生在客户的线程控制中.

2. 方法请求(Method Request)
方法请求用于将代理上的特定方法调用的上下文信息,比如方法参数和代码,从代理传递给运行在分离线程中的调度者.抽象方法请求类为执行主动对象方法定义接口.该接口还包含守卫(Guard)方法,可用于确定何时方法请求的同步约束已被满足.对于代理提供的每个主动对象方法(它们在其仆人中需要同步的访问),抽象方法请求类被子类化,以创建具体的方法请求类.这些类的实例在其方法被调用时由代理创建,并包含了执行这些方法调用和返回任何结果给客户所需的特定的上下文信息.

3. 启用队列(Activation Queue)
启用队列维护一个有界缓冲区,内有代理创建的待处理的方法请求.该队列跟踪哪些方法请求将要执行.它还使客户线程与仆人线程去耦合,以使两个线程能并发运行.

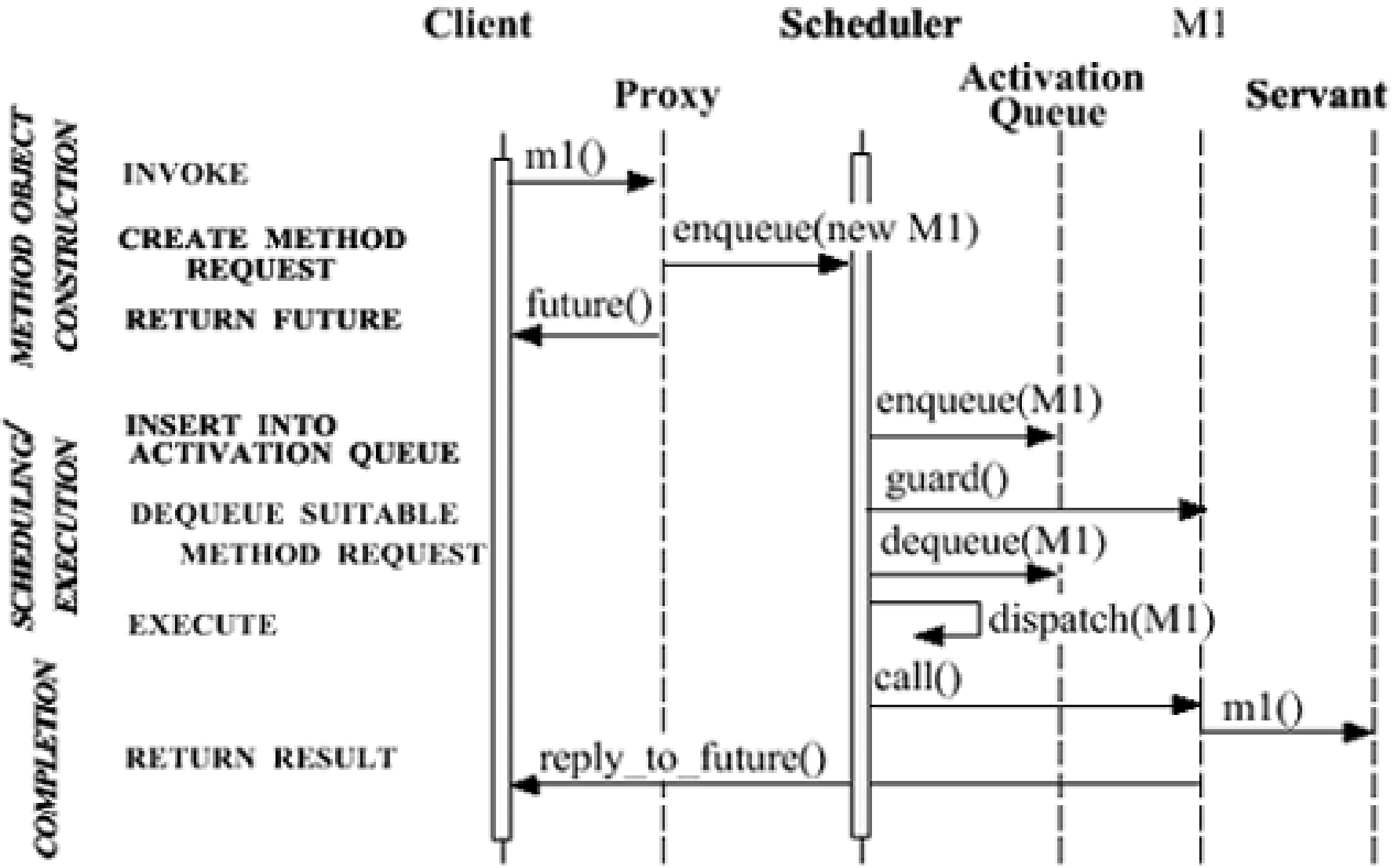
Active Object 的参与者

4. 调度者(Scheduler)
调度者运行在与其客户不同的线程中,它管理待处理的方法请求的启用队列.调度者决定下一个出队的方法请求,并在实现该方法的仆人上执行.这样的调度决策基于各种标准,比如像方法被插入到启用队列中的顺序以及同步约束,例如特定属性的满足或特定事件的发生,比如在有界数据结构中有新的条目空间变得可用.调度者通常使用方法请求守卫来对同步约束进行求值.

5. 仆人(Servant)
仆人定义被建模为主动对象的行为和状态.它实现在代理中定义的方法及相应的方法请求.仆人方法在调度者执行其相应的方法请求时被调用.因而,仆人在调度者的线程控制中执行.仆人还可提供其他方法,由方法请求用于实现它们的守卫.

6. 期货(Future)
期货允许客户在仆人结束方法的执行后获取方法调用的结果.当客户通过代理调用方法时,期货被立即返回给客户.期货为被调用的方法保留空间,以存储它的结果.当客户想要获取这些结果时,它可以阻塞或者轮询,直到结果被求值和存储到期货中,然后与期货“会合”.

Active Object 处理时序图



使用 Active Object

```
#include "ace/Method_Object.h"
#include "ace/Activation_Queue.h"
#include "ace/Future.h"
#include "ace/Auto_Ptr.h"

ACE_Activation_Queue cmdQueue; //命令队列

// 以主动的方式记录日志
void Logger::LogMsgActive(const string& msg,ACE_Future<string> *result)
{
    // 生成命令对象，插入到命令队列中
    cmdQueue.enqueue(new LogMsgCmd(this,msg,result));
}
...
```

Monitor Object: 并发编程的对象行为模式

Monitor对象模式通过同步方法执行来确保一个对象在一个时刻只有一个方法在被运行。它也允许一个对象的方法能够协同地安排它们执行的顺序。Monitor对象也叫线程安全的被动对象. (注意与Active Object的区别)

监视对象模式由以下组件组成:

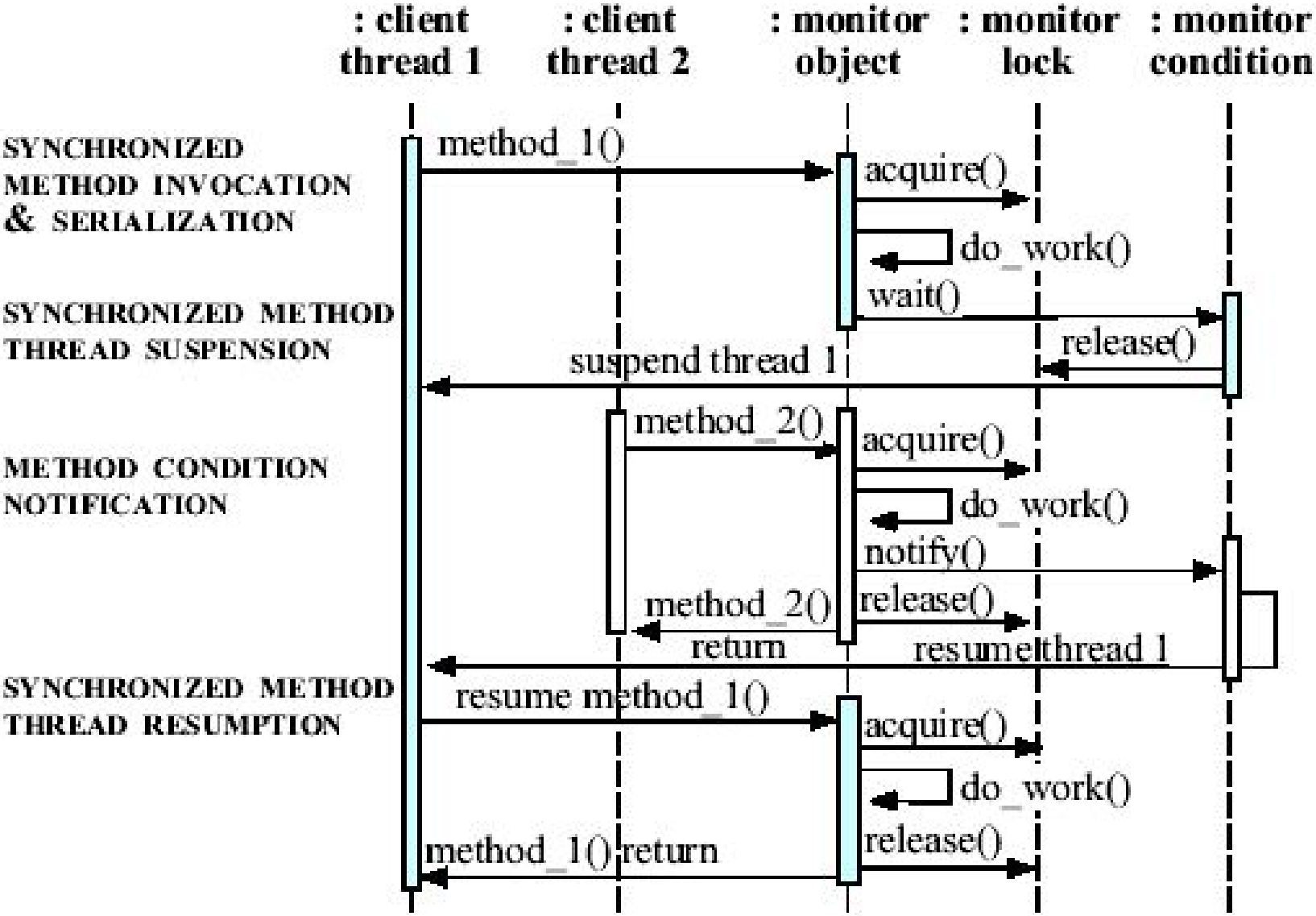
1、monitor对象: 一个monitor对象向客户暴露一个或多个接口方法.为了保护monitor对象的内部状态不受任意修改和竞争条件的破坏,所有的客户必须通过这些方法访问monitor对象.因为monitor本身不包含自己的控制线程,所以每个方法在调用它的客户线程上执行.

2、同步方法: 同步方法实现线程安全的被monitor对象暴露的服务.为了防止竞争条件,无论是否同时有多个线程并发调用同步方法,还是monitor对象类含有多个同步方法,在一个monitor对象内,在任意时间点只有一个同步方法能够被执行.

3、monitor锁: 每一个monitor对象包含自己的monitor锁.同步方法使用这个monitor锁来实现每个对象基础上的方法调用串行化.当方法进入/离开对象时,每个同步方法必须分别的获取/释放monitor锁.这个协议保证无论什时候一个方法访问或修改对象的状态时都应该先获取monitor锁.

4、monitor条件: 运行在分离线程上的多个同步方法可以经由monitor条件来相互等待和通知以实现协同地调度它们执行的顺序.同步方法可以使用monitor 条件来决定在何种环境下挂起或恢复它们的执行.

Monitor Object 处理时序图



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。

消息队列(Message Queue)

ACE中的每个任务都有一个底层消息队列,这个消息队列被用作任务间通信的一种方法.

ACE消息队列分为静态和动态两类

ACE_Message_Queue 静态队列

ACE_Dynamic_Message_Queue 动态消息队列（实时消息队列）

—基于最终期限(deadline)

—基于松弛度(laxity)

动态消息队列中,基于诸如执行时间和最终期限等参数,消息的优先级可以动态地改变

消息队列工厂有三个静态的工厂方法,可用来创建三种不同类型的消息队列:

create_static_message_queue();

create_deadline_message_queue();

create_laxity_message_queue();

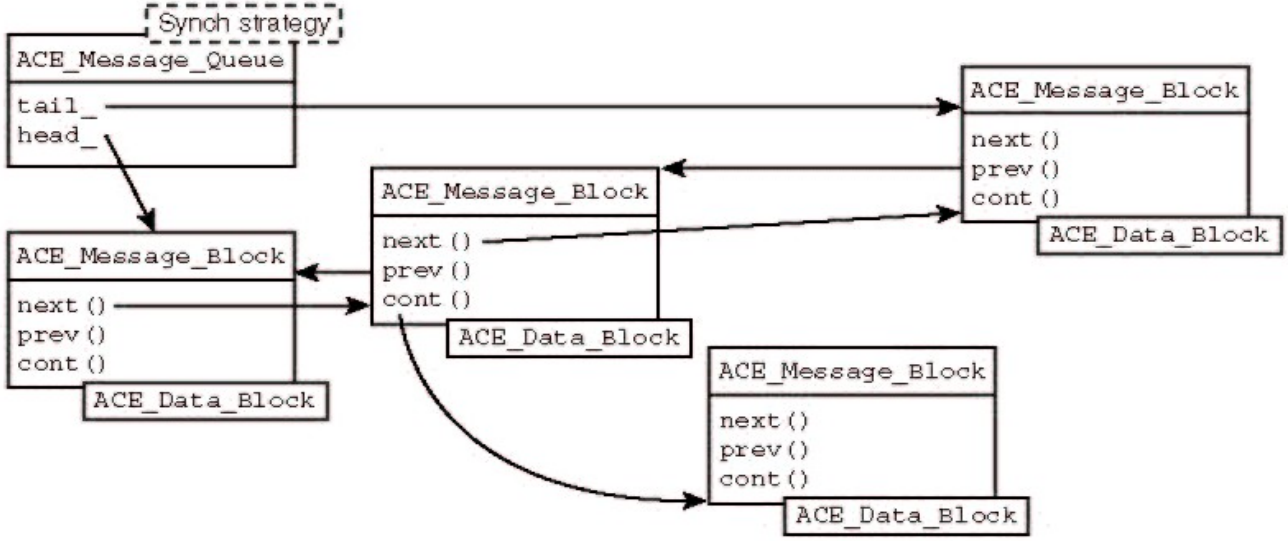
ACE_Message_Queue Class

。ACE_Message_Queue是一个可移植的轻量级进程内消息排队机制

。ACE_Message_Queue的设计基于System V STREAMS中的消息缓冲和排队设施

。 ACE_Message_Queue_Ex是ACE_Message_Queue的一个变种,所交换的是MESSAGE_TYPE模板参数的实例,而不是ACE_Message_Block

ACE_Message_Queue 结构



图演示了怎样将3个消息链接在一起形成一个ACE_Message_Queue的,队列中头消息和尾消息是简单消息,而中间的是复合消息,有一个消息块通过"继续”指针与它串联一起.ACE_Message_Queue的基本单元是ACE_Message_Block.

使用ACE_Message_Block

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

int main(int argc, char *argv[])
{
   ACE_Message_Block *mb = new ACE_Message_Block (30);
   ACE_OS::sprintf(mb->wr_ptr(),"%s","hello");
   ACE_OS::printf("%s\n",mb->rd_ptr ());
   mb->release();
   return 0;
}
```

Strategized Locking: ACE策略化的加锁模式

目的：策略化的加锁模式通过策略化一个组件的同步机制,在不降低其性能或可维护性的基础上增加组件的灵活性和可重用性.

- 定义组件的接口和实现.这个步骤的焦点在于定义简明的组件接口和一个不考虑同步问题的有效率的实现.ACE_TLI 系统V传输层接口
- 策略化易变的同步方面(aspect) 在这个步骤中,确定组件中的哪些是可能引起组件的接口和实现发生变化和修改的同步方面,将其策略化.许多可重用组件有相对简单的同步方面,它们可以通过象互斥量和信号量等普通的加锁策略来实现.这些同步方面可以使用参数化类型或多态方法来策略化成一种一致的行为.
- 定义锁策略族.这个家族中的每个成员都必须提供统一的接口,可以支持各种应用指定的并发用例.如果同步组件不存在或是存在,但有不兼容的接口,使用包装门面模式(Wrapper Facade Pattern)或适配方式使其适应组件同步方面希望的样式.除了使用包装门面模式定义的Thread_Mutex外,还包括读写锁、信号量、互斥量和文件锁等其他普通锁策略.一个令人惊奇的锁策略是NULL_MUTEX.

益处：

1、增加灵活性和性能的调节.因为组件的同步方面被策略化了,这将直接配置、调节一个组件以适应特殊的并发用例.

2、减少对组件的维护投入.它直接增强组件的功能,防止出错,因为这里只有一个实现而不是对每一个并发用例都有一个分离实现.这种集中化的考虑避免了版本的爆炸.

ACE_Message_Queue 使用策略锁

```
template <class SYNCH_STRATEGY>
class ACE_Message_Queue {
  // ...
protected:
  // C++ traits that coordinate concurrent access.
  ACE_TYPENAME SYNCH_STRATEGY::MUTEX lock_;
  ACE_TYPENAME SYNCH_STRATEGY::CONDITION notempty_;
  ACE_TYPENAME SYNCH_STRATEGY::CONDITION notfull_;
};
```

参数化的策略锁

- **The traits classes needn't derive from a common base class or use virtual methods!**

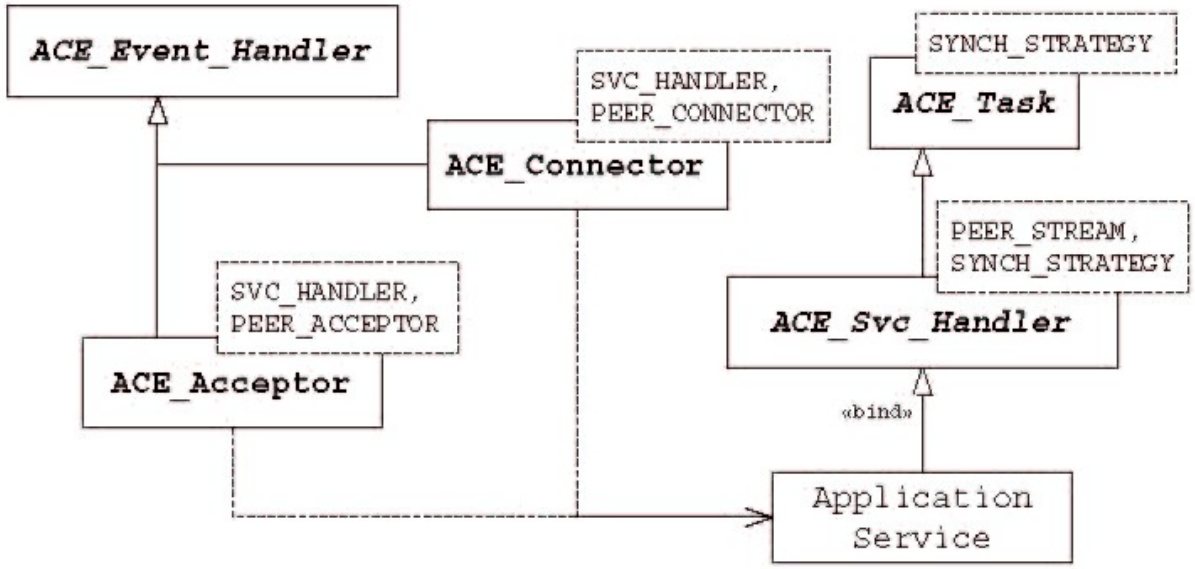
```
class ACE_NULL_SYNCH {
public:
  typedef ACE_Null_Mutex
          MUTEX;
  typedef ACE_Null_Condition
          CONDITION;
  typedef ACE_Null_Semaphore
          SEMAPHORE;
  // …
};
```

```
class ACE_MT_SYNCH {
public:
  typedef ACE_Thread_Mutex
          MUTEX;
  typedef ACE_Condition_Thread_Mutex
          CONDITION;
  typedef ACE_Thread_Semaphore
          SEMAPHORE;
  // …
};
```

使用ACE_Message_Queue

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

 //为每个消息创建一个消息块
 ACE_Message_Block *mb;
 //初始化消息块大小
ACE_NEW_RETURN (mb, ACE_Message_Block (20),-1);
 //把数据插入到消息块
 ACE_OS::sprintf (mb->wr_ptr(), "This is message %d\n", i);
 //移动消息块的指针
 mb->wr_ptr (ACE_OS::strlen("This is message 1\n"));
//把消息块入队
 if (this->mq_->enqueue_prio (mb) == -1)
       {
            ACE_DEBUG ((LM_ERROR, "\nCould not enqueue on to mq!!\n"));
            return -1;
       }

//取出消息
 for (int i=0; i<no_msgs_; ++i)
      {
            mq_->dequeue_head(mb);
            ACE_DEBUG ((LM_INFO, "DQ'd data %s\n", mb->rd_ptr()));
            mb->rd_ptr(ACE_OS::strlen(mb->rd_ptr()) + 1);
            mb->release();
 }
```

Connector(连接器)和Acceptor(接收器)框架



ACE类	描述
ACE_Svc_Handler	表示某个已连接服务的本地端,其中含有一个用于与相连对端通信的IPC端点
ACE_Acceptor	该工厂被动地等待接受连接,并随即初始化一个ACE_Svc_Handler来响应来自对端的主动连接请求
ACE_Connector	该工厂主动地连接到对端接收器,并随即初始化一个ACE_Svc_Handler来与其相连对端通信

Connector(连接器)和Acceptor(接收器)

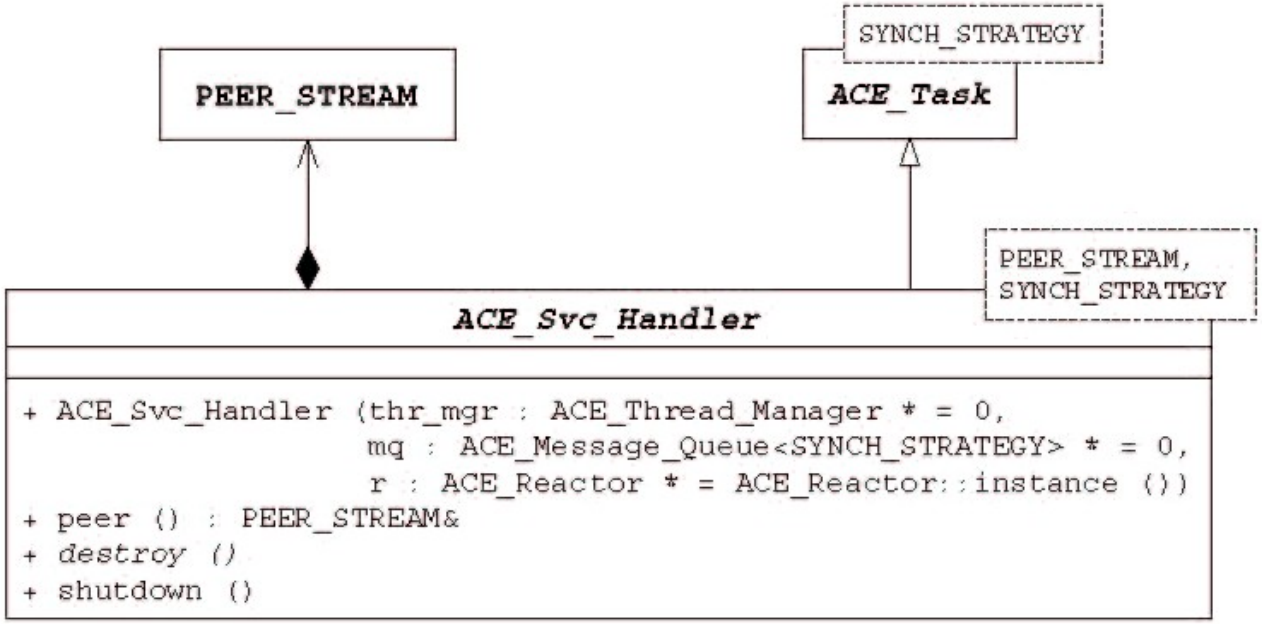
Connector(连接器)：主动建立连接
Acceptor(接受器)：被动建立连接

* 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。
* 接受器－连接器模式使用模板方法(Template Method)和工厂方法(Factory Method)模式
* 策略的调整是通过重载ACE_Acceptor或ACE_Connector类的handler方法来完成的
* 通过使用ACE_Cached_Connect_Strategy进行连接缓存,减少重建连接的开销

已有应用：

* UNIX网络超级服务器：比如inetd、listen
* CORBA ORB 核心对客户请求的处理方式
* WWW浏览器,HTML解析组件的处理方式

ACE_Svc_Handler 类

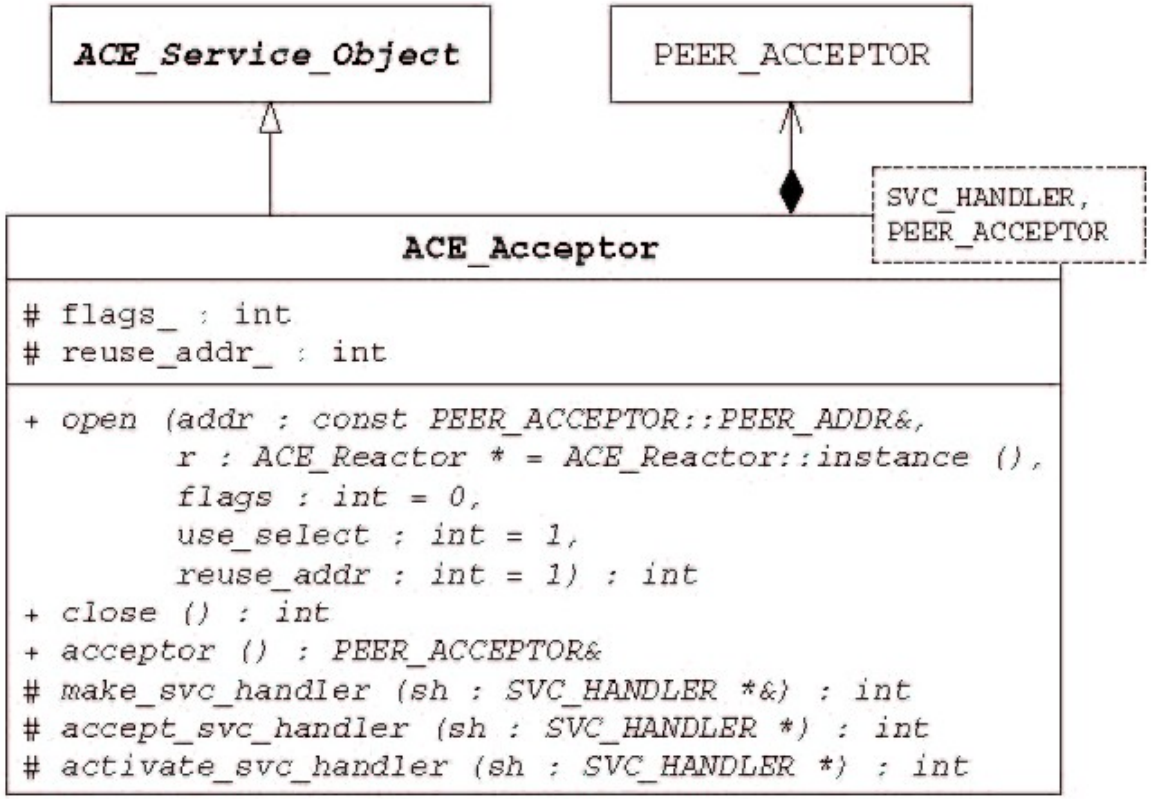


- ACE_Svc_Handler是ACE的同步和反应式数据传输及服务处理机制的基础
- ACE_Svc_Handler派生自ACE_Task,而ACE_Task派生自ACE_Event_Handler,所以继承了并发、同步、动态配置和事件处理能力

使用ACE_Svc_Handler

```
#include "ace/Svc_Handler.h"
#include "ace/Acceptor.h"
#include "ace/INET_Addr.h"

#define PORT_NUM 8000

class My_Svc_Handler:
    public ACE_Svc_Handler <ACE_SOCK_STREAM,ACE_NULL_SYNCH>{
        public:
            int open(void*){
                cout<< "Connection established" <<endl;
                return 0;
                }
            };

typedef ACE_Acceptor<My_Svc_Handler,ACE_SOCK_ACCEPTOR> MyAcceptor;

int main(int argc, char* argv[]){
    ACE_INET_Addr addr(PORT_NUM);
    MyAcceptor acceptor(addr, ACE_Reactor::instance());

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

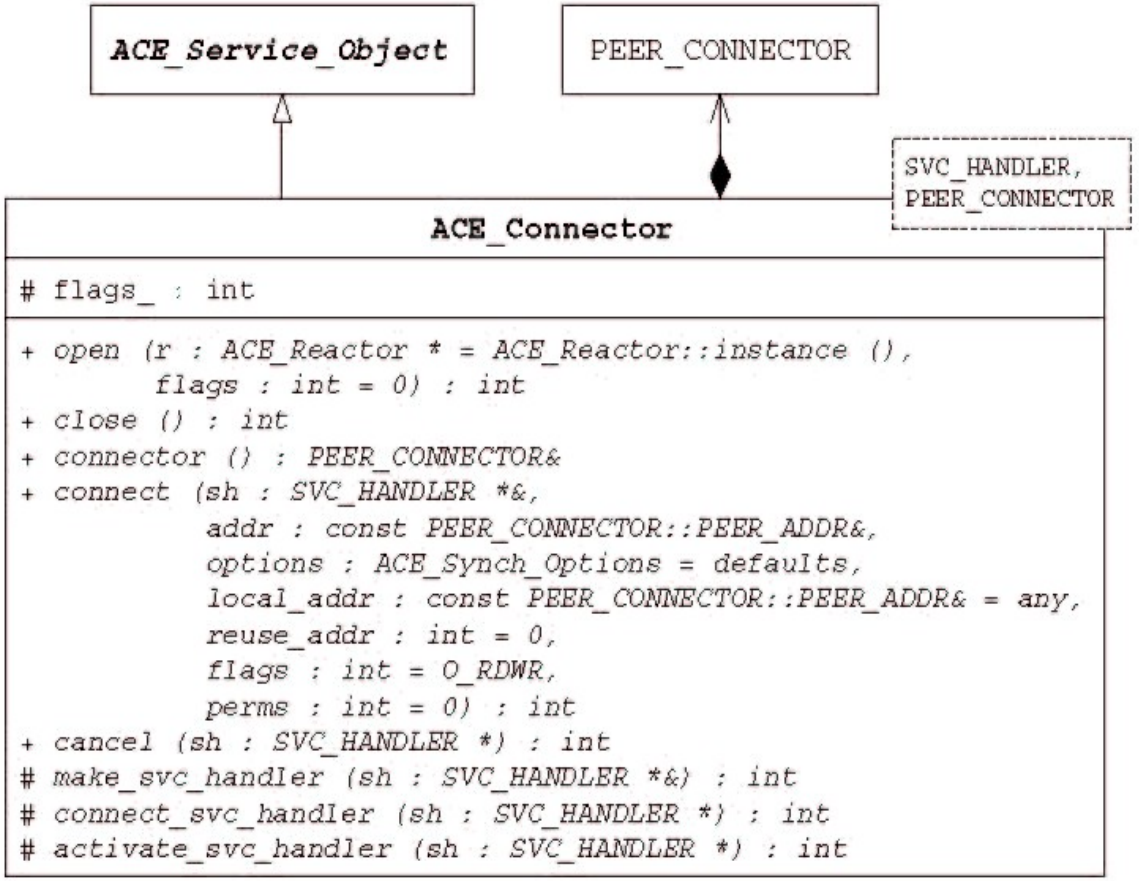
ACE_Acceptor Class



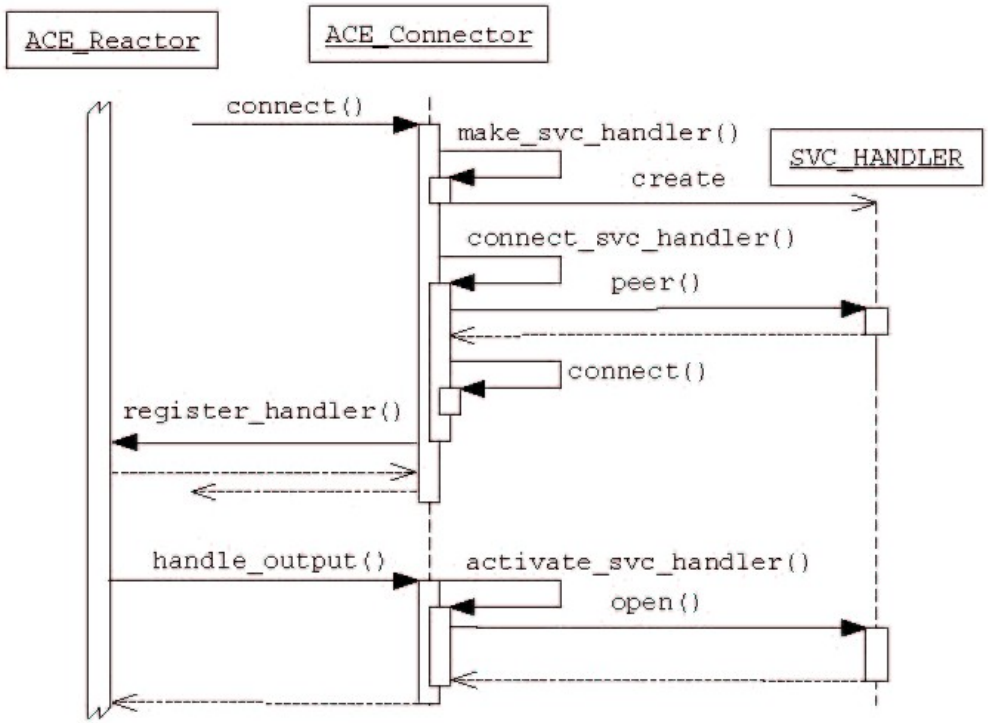
。许多面向连接的服务器应用将它们的连接建立和服务初始化代码紧密地耦合在一起，其耦合方式致使开发者难以复用现有的代码

。ACE Acceptor-Connector框架定义了ACE_Acceptor类，以使应用开发者无需再反复地重写这些代码

ACE_Connector Class



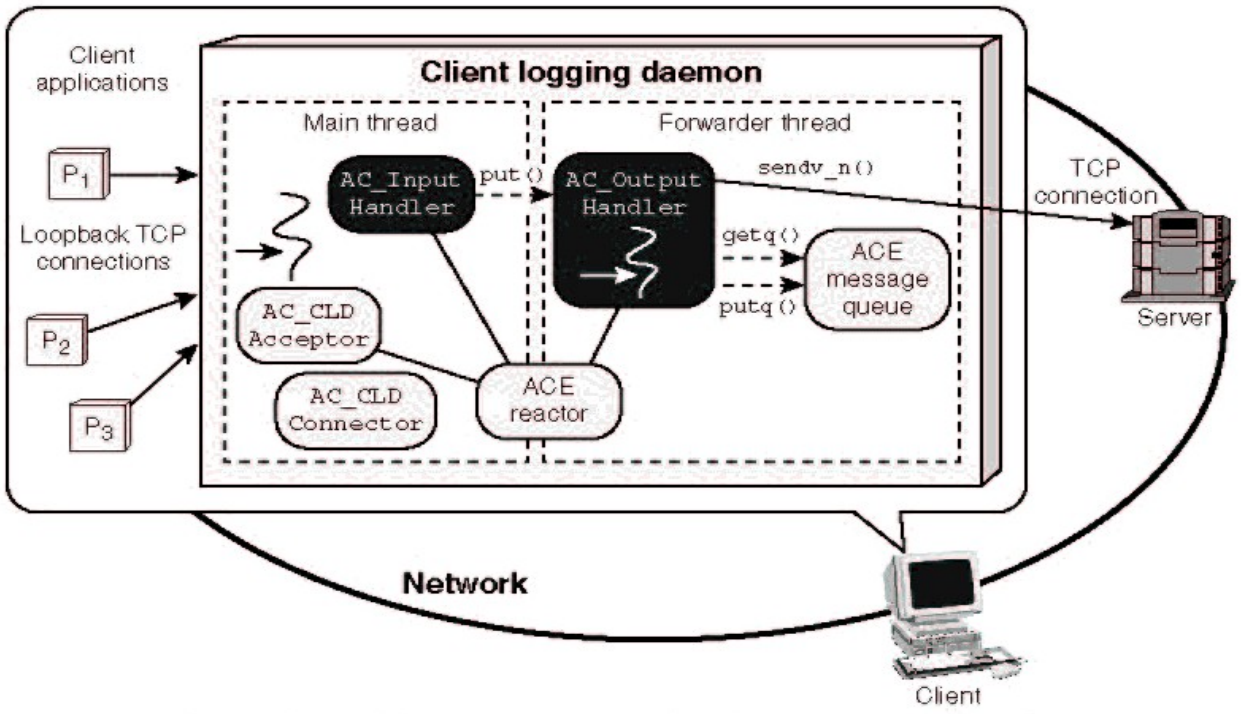
ACE_Connector Class



ACE_Connector异步连接建立中的各步骤

- ACE_Connector 是一个工厂
- 主动连接建立
- 连接建立后的处理

使用ACE_Connector Class



- client logging daemon 使用了两个线程处理输入、输出任务
- 输入处理：主线程使用单体

ACE_Reactor,ACE_Acceptor,ACE_Svc_Handler被动对象

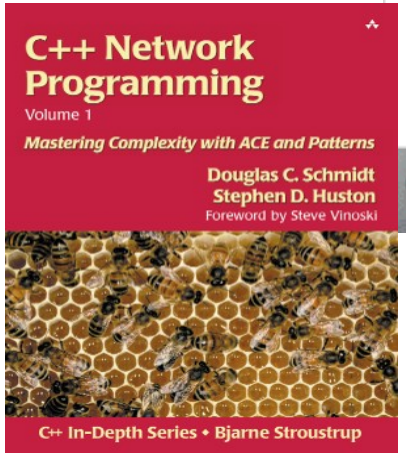
- 输出处理：主动对象ACE_Svc_Handler运行在自己的线程中

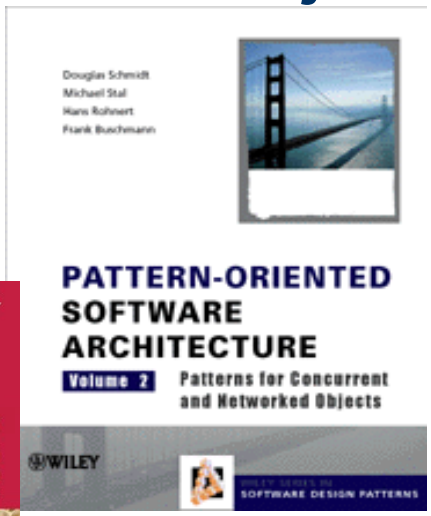
内容回顾

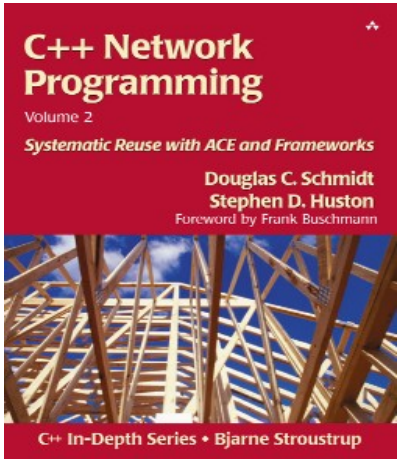
- 如何访问OS服务
- TCP/IP Socket编程接口
- 使用ACE的UDP类进行网络编程
- 进程和线程管理
- 反应堆（Reactor）和Proactor框架：用于事件多路分离和分发的体系结构
- 为I/O，定时器，信号处理实现事件处理器
- ACE Task框架：并发模式
- 消息队列(Message Queue)
- 接受器（Acceptor）和连接器（Connector）框架：接受连接模式

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