Module06-04
C++ ACE: Task 框架

C++ ACE - Task 框架

- ACE 简介
- I/O 相关对象
- Reactor 框架
- Service Configuration 框架
- ➔ Task 框架
- Acceptor-Connector 框架
- Proactor 框架
- 杂项

Task 框架 - 概要

- Task 框架：
  - 概要
  - ACE_Message_Block
  - ACE_Message_Queue
  - ACE_Thread_Manager
  - ACE_Task

Task 框架 - 概要

- 关于 Task 框架
  - ACE Task 框架提供了强大而可扩展的面向对象并发能力，如在基于对象的上下文 (context) 中派生线程，以及在执行不同线程中的对象之间传递消息和对消息进行排队。 Task 框架可应用于实现一些关键的并发模式：
    - Active Object 模式，解除“调用某个方法的线程”与“执行该方法的线程”之间的耦合，该模式增强了并发性，简化多个线程对共享资源的同步访问
    - Half-Sync/Half-Async 模式，解除并发系统中的异步与同步处理的耦合，从而能够简化编程，同时又不会带来严重的性能下降，该模式引入了 3 个层次：一层用于异步 (或反应式) 处理，一层用于同步服务处理，还有一个排队层，负责协调异步 / 反应式和同步层之间通信

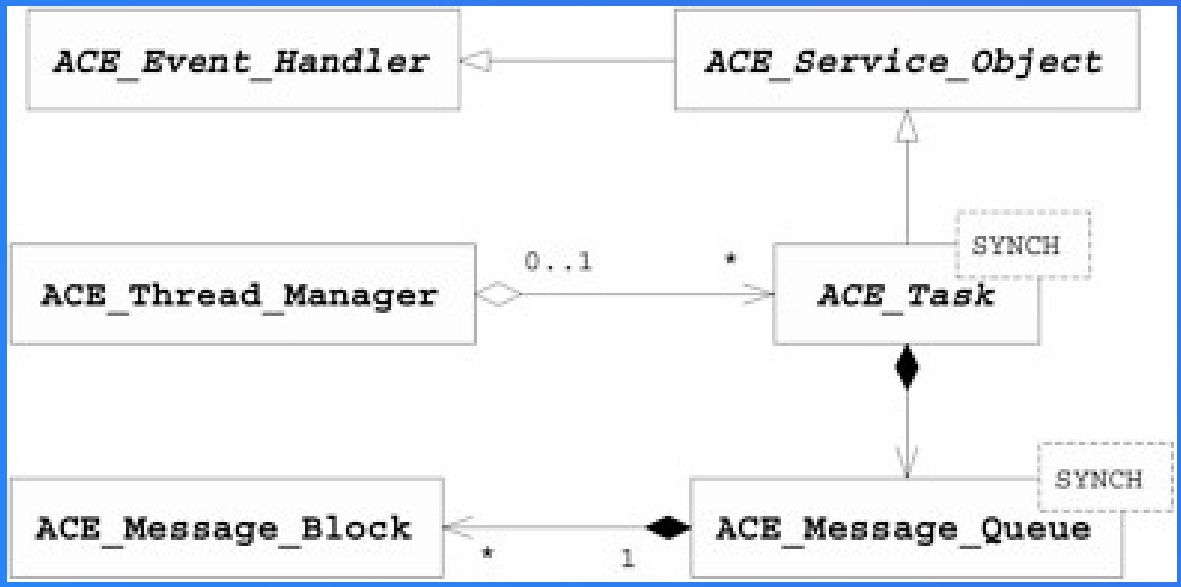
Task 框架 - 概要

- ACE Task 框架的主要参与者

ACE 类	说明
ACE_Message_Block	实现 Composite 模式，使开发者能够高效的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之间传递和缓冲消息
ACE_Thread_Manager	允许应用可移植的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对象，解除不同处理单元的耦合；使用消息来交换请求、响应、数据以及控制消息；且还可以序列化的或并行断点排队和消息处理

Task 框架 - 概要

- ACE Task 框架 类关系图



Task 框架 - 概要

- ACE Task 框架的优点
  - 改善编程风格的一致性

    ACE_Task 类提供了一种面向对象的编程抽象，将操作系统的线程与 C++ 对象关联起来
  - 将一组线程作为一个内聚的集合进行管理

    多线程化的网络化应用常需要成组的启动和结束多个线程。所以 ACE_Task 类提供了一种线程组能力，允许其它线程在继续其处理之前等待全组线程退出
  - 解除生产者与消费者线程的耦合

    让它们并发执行，并通过同步化的消息队列来传递消息，从而进行协作
  - 集成并发处理

    与 Reactor 框架或 Proactor 框架类似

Task 框架 - 概要

- ACE Task 框架的优点（续）
  - 方便对任务的动态配置

    通过与 ACE Service Configuration 框架进行集成，这样，开发者不必在设计时甚至运行时过早的做出并发决策。相反，应用可以被设计为根据配置来运行：这些配置可能会因部署站点资源和动态条件而发生变化

Task 框架 - ACE_Message_Block

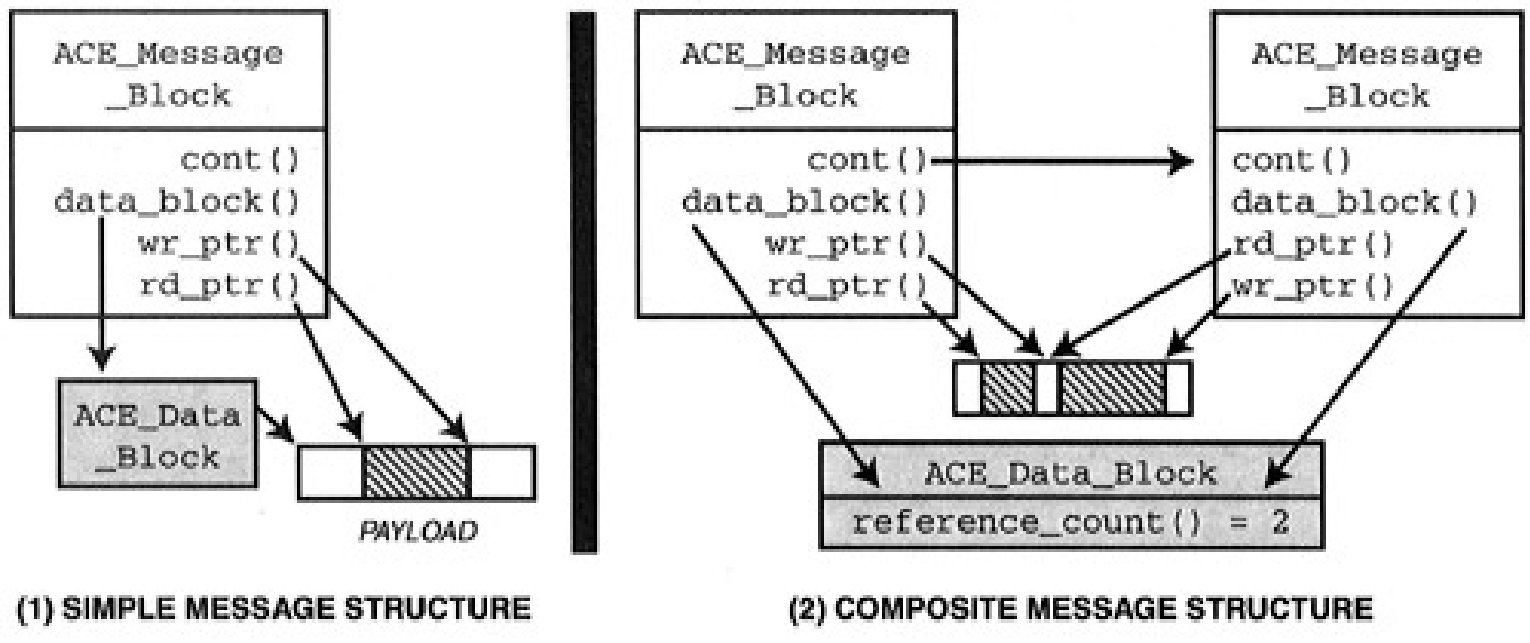
- Task 框架：
  - 概要
  - ACE_Message_Block
  - ACE_Message_Queue
  - ACE_Thread_Manager
  - ACE_Task

Task 框架 - ACE_Message_Block

- 关于 ACE_Message_Block
  - ACE_Message_Block 类可以让我们高效的管理定长或变长的消息，该类提供以下能力：
    - 每个 ACE_Message_Block 都包含一个指针，指向带引用技术的 ACE_Data_Block ，而 ACE_Data_Block 则指向和消息关联的实际数据。这样以来，我们可以灵活、高效的共享数据，并能降低内存复制带来的额外开销
    - 允许多条消息连接在一起，形成一个单链表，从而支持复合消息。这种复合消息可用于多态链表，也可以用于需要高效的添加 / 删除消息头 / 尾的层次化协议栈
    - 允许将多条消息连接起来，形成一个双链表，构成 ACE_Message_Queue 类的基础

Task 框架 - ACE_Message_Block

- 两种类型的 ACE_Message_Block



Task 框架 - ACE_Message_Block

- ACE_Message_Block 的关键方法

方法	描述
ACE_Message_Block() init()	对消息进行初始化
mst_type()	设置和获取消息类型
msg_priority()	设置和获取消息的优先级
clone()	返回整条消息的深拷贝
duplicate()	返回消息的浅拷贝，将引用计数加 1
release()	将引用计数减 1，如果引用计数降至 0，删除该消息
set_flags()	将指定的数据 bit 与一组已有的 flag 执行按位与操作，用以指定消息的语义（如，消息释放时是否删除缓冲区等）
clr_flags	清除指定的 flag
copy()	从缓冲区复制 n 个字节到消息

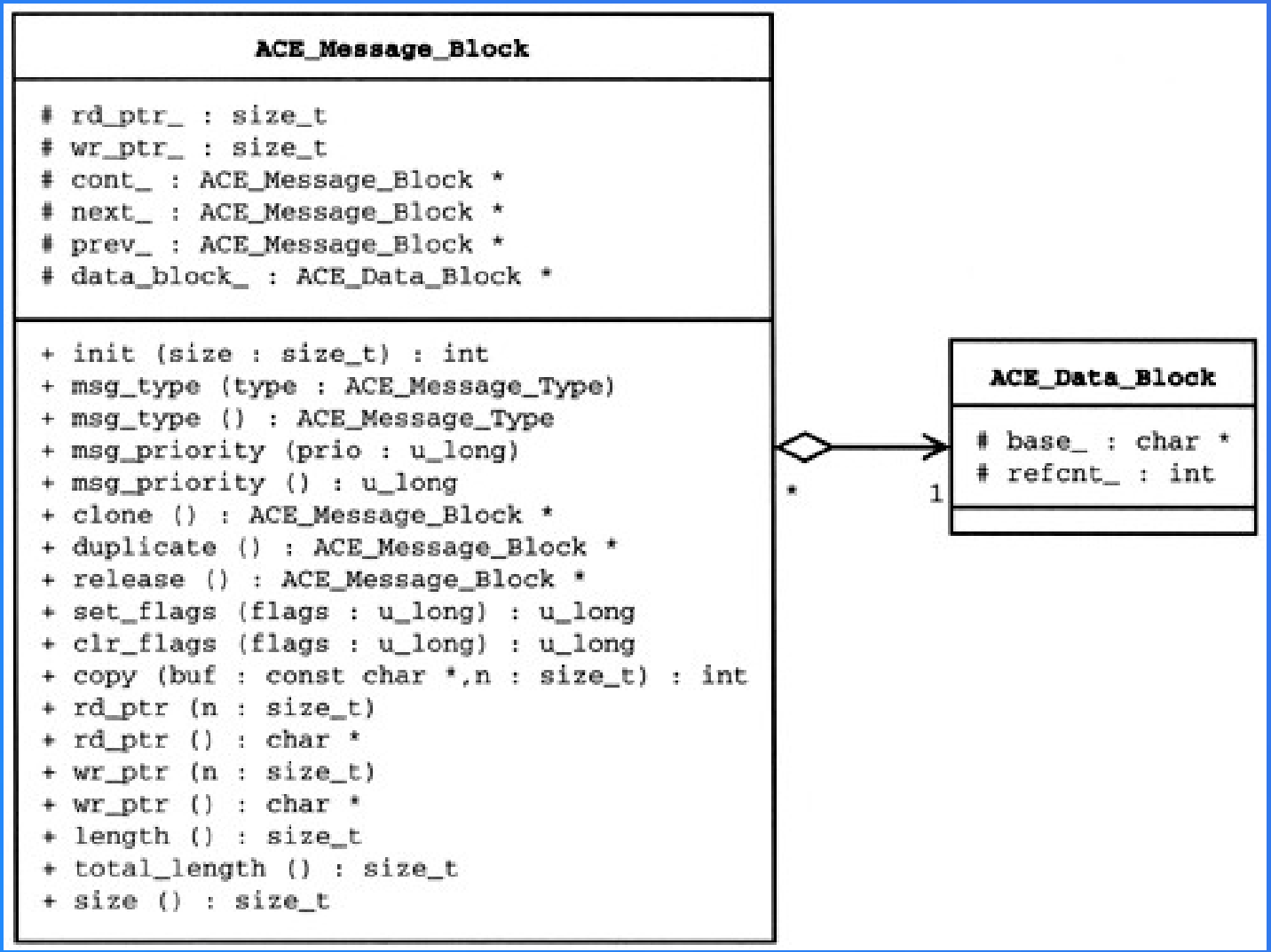
Task 框架 - ACE_Message_Block

- ACE_Message_Block 的关键方法（续）

方法	描述
rd_ptr()	设置和获取读指针
wr_ptr()	设置和获取写指针
cont()	设置和获取消息中的连接字段，该字段用于将复合消息连接在一起
next() prev()	设置和获取指向 ACE_Message_Queue 中的双向消息链表的指针
length()	设置和获取消息的当前长度，即 wr_ptr()-rd_ptr()
total_length()	获取消息的长度，包括所有被连接的消息块
size()	设置和或其消息的总容量，包括分配在 [rd_ptr(), wr_ptr()] 范围之前和之后的存储容量

Task 框架 - ACE_Message_Block

- ACE_Message_Block 类图



Task 框架 - ACE_Message_Block

- ACE_Message_Block 示例

```
#include <ace/ACE.h>
#include <ace/Message_Block.h>

int main() {
    ACE_Message_Block *head = new ACE_Message_Block(8);
    ACE_Message_Block *mblk = head;

    for (;;) {
        ssize_t nbytes = ACE::read_n(ACE_STDIN, mblk->wr_ptr(),
mblk->size());
        if (nbytes <= 0)
            break; // Break out at EOF or error.

        // Advance the write pointer to the end of the buffer.
        mblk->wr_ptr(nbytes);
```

Task 框架 - ACE_Message_Block

- ACE_Message_Block 示例

```
        // Allocate message block and chain it at the end of
list.
        mblk->cont(new ACE_Message_Block(8));
        mblk = mblk->cont();
    }
    // Print the contents of the list to the standard output.
    int i = 1;
    for (mblk = head; mblk != 0; mblk = mblk->cont()) {
        ACE_DEBUG((LM_DEBUG, "\n%dth message: ", i++));
        ACE::write_n(ACE_STDOUT, mblk->rd_ptr(), mblk-
>length());
    }

    head->release(); // This releases all the memory in the
chain.
    return 0;
}
```

Task 框架 - ACE_Message_Queue

- Task 框架：
  - 概要
  - ACE_Message_Block
  - ACE_Message_Queue
  - ACE_Thread_Manager
  - ACE_Task

Task 框架 - ACE_Message_Queue

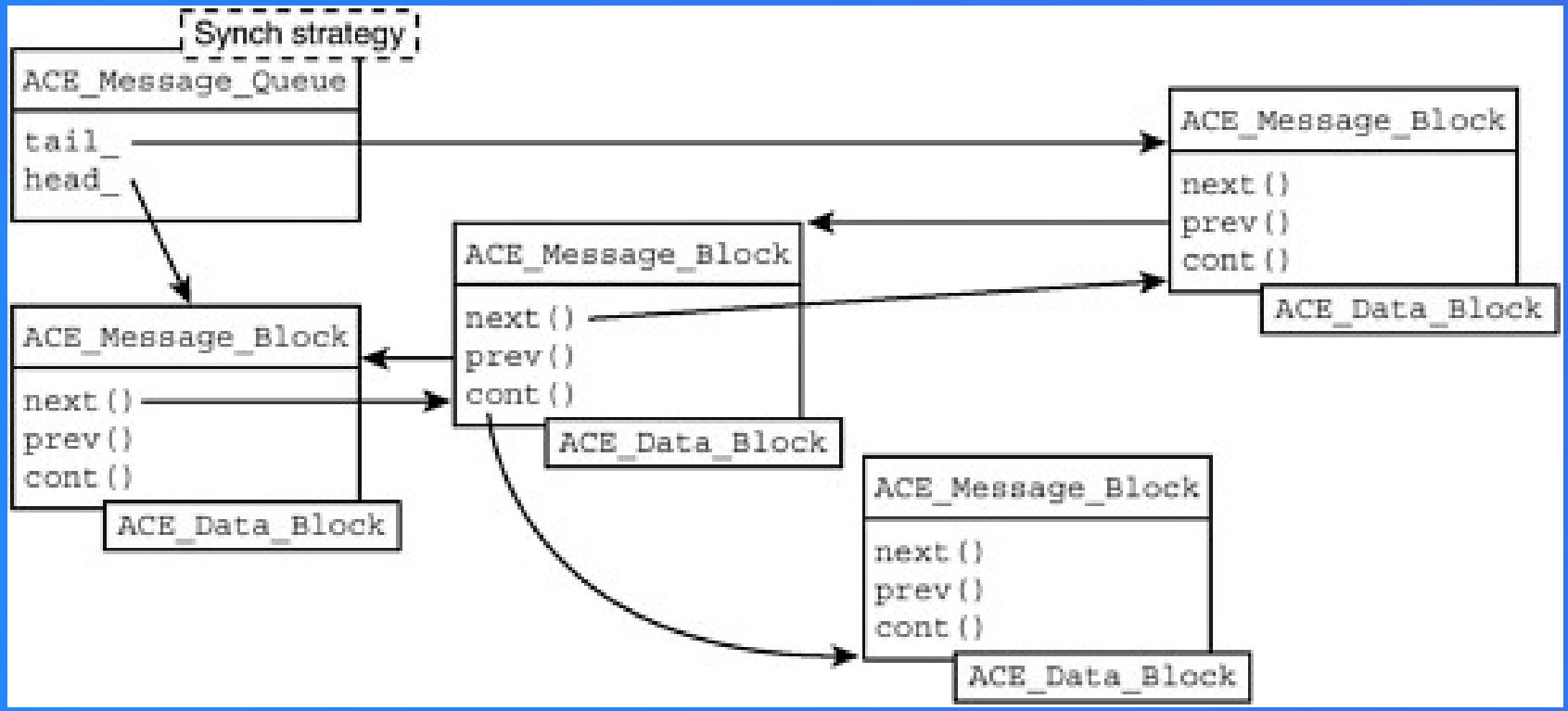
- 关于 ACE_Message_Queue
  - ACE_Message_Queue 是一个可移植的轻量级进程内的消息队列机制，该类提供以下能力：
    - 允许开发人员将消息（ ACE_Message_Block ）放入队列的前部、尾部或是基于消息的优先级、按优先级顺序放入队列。消息可以从队列的头部或尾部取出
    - 使用 ACE_Message_Block 来提供高效的消息缓冲机制，使动态内存分配和数据复制的需要降到最低
    - 既可以为多线程化配置，也可以为单线程化配置，并允许程序员在不需对队列进行并发访问时，取消严格的同步来换取更低开销
    - 在多线程化的配置中，它支持可配置的溢出控制 (Flow Control)，可以防止快速的生产者线程耗尽较慢的消息消费者线程的处理和内存消耗
    - 允许为入列和出列操作指定超时，以避免发生无限期的阻塞

Task 框架 - ACE_Message_Queue

- 关于 ACE_Message_Queue（续）
  - 可以同 ACE Reactor 框架的事件处理机制集成在一起
  - 提供了可定制策略的分配器，则消息所使用的内存可从多种来源获取，如共享内存、堆内存、栈内存、或是线程局部存储

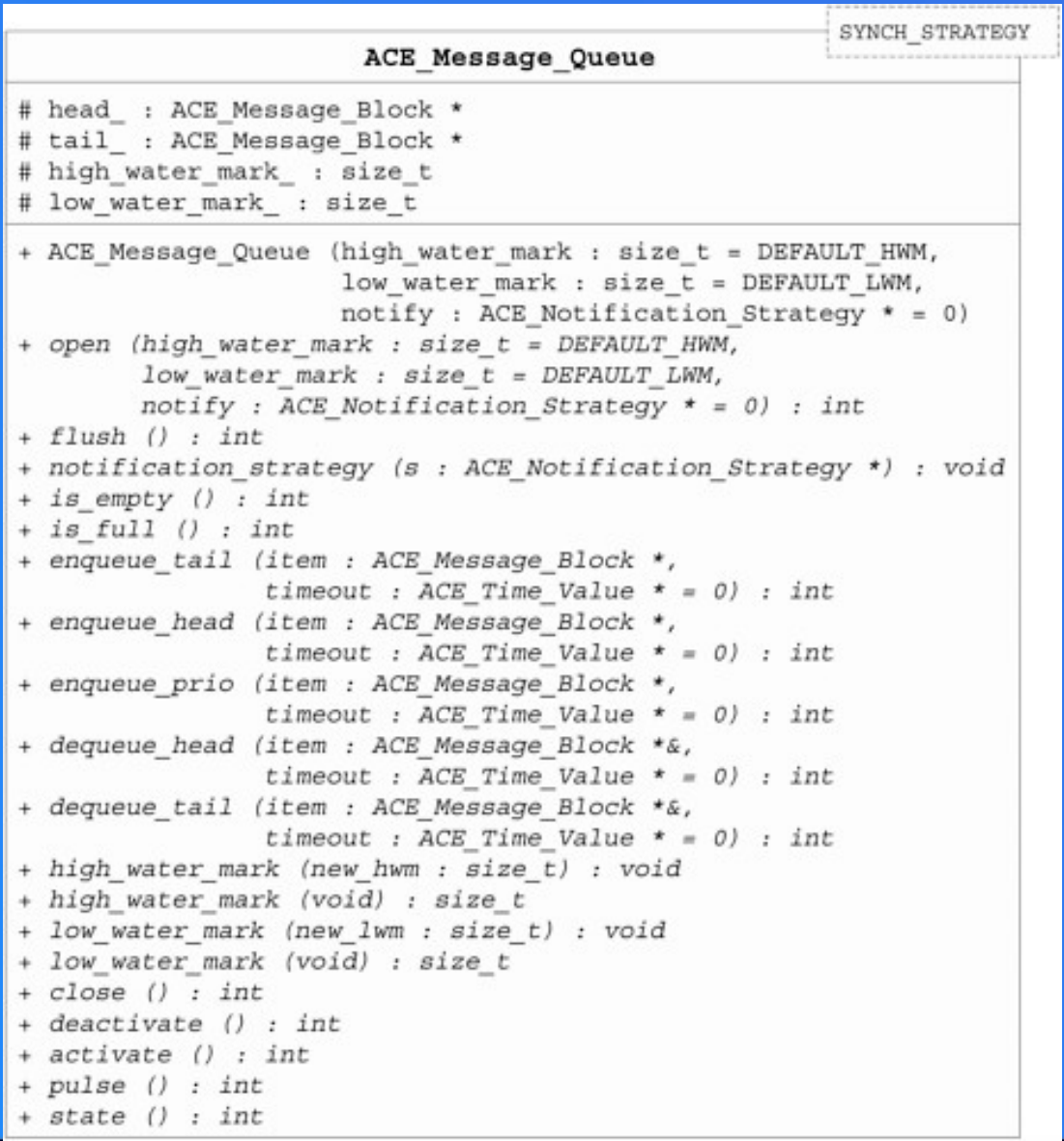
Task 框架 - ACE_Message_Queue

- 一个 ACE_Message_Queue 的结构



Task 框架 - ACE_Message_Queue

- ACE_Message_Queue 类图



Task 框架 - ACE_Message_Queue

- ACE_Message_Queue 的关键方法
  - 初始化与溢出控制方法

方法	描述
`ACE_Message_Queue()` `open()`	初始化队列，指定水位线和通知策略
`high_water_mark()`	设置和获取上溢的水位线
`low_water_mark()`	设置和获取下溢的水位线
`notification_strategy()`	设置和获取通知策略

Task 框架 - ACE_Message_Queue

- ACE_Message_Queue 的关键方法（续 1 ）
  - 入列 / 出列方法和消息缓冲

方法	描述
`is_empty()` `is_full()`	`is_full()` 方法在队列中的字节数高于高水位时返回 `true`
`enqueue_tail()`	将消息插入队列的尾部
`enqueue_head()`	将消息插入队列的头部
`enqueue_prio()`	根据优先级来插入消息
`dequeue_head()`	删除并返回队列头部的消息
`dequeue_tail()`	删除并返回队列尾部的消息

Task 框架 - ACE_Message_Queue

- ACE_Message_Queue 的关键方法（续 2 ）
  - 参数化的同步策略
    - 在 ACE_Message_Queue 类的内部，定义了以下类型：

```
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_;
};
```

Task 框架 - ACE_Message_Queue

- ACE_Message_Queue 的关键方法（续 2 ）
  - 参数化的同步策略（续 1 ）
    - ACE 提供了以下 2 种同步的 traits：
      - ACE_NULL_SYNCH：在单线程环境下使用该方式
      - ACE_MT_SYNCH：多线程需要同步的时候使用该方式
    - ACE_NULL_SYNCH：

```
class ACE_NULL_SYNCH {
public:
    typedef ACE_Null_Mutex MUTEX;
    typedef ACE_Null_Mutex NULL_MUTEX;
    typedef ACE_Null_Mutex PROCESS_MUTEX;
    typedef ACE_Null_Mutex RECURSIVE_MUTEX;
    typedef ACE_Null_Mutex RW_MUTEX;
    typedef ACE_Null_Condition CONDITION;
    typedef ACE_Null_Semaphore SEMAPHORE;
    typedef ACE_Null_Semaphore NULL_SEMAPHORE;
};
```

Task 框架 - ACE_Message_Queue

- ACE_Message_Queue 的关键方法（续 2 ）
  - 参数化的同步策略（续 2 ）
    - ACE_MT_SYNCH ：

```
class ACE_MT_SYNCH {
public:
    typedef ACE_Thread_Mutex MUTEX;
    typedef ACE_Null_Mutex NULL_MUTEX;
    typedef ACE_Process_Mutex PROCESS_MUTEX;
    typedef ACE_Recursive_Thread_Mutex RECURSIVE_MUTEX;
    typedef ACE_RW_Thread_Mutex RW_MUTEX;
    typedef ACE_Condition_Thread_Mutex CONDITION;
    typedef ACE_Thread_Semaphore SEMAPHORE;
    typedef ACE_Null_Semaphore NULL_SEMAPHORE;
};
```

- ACE_Message_Queue 的关键方法（续 3 ）
  - 关闭、释放方法

方法	描述
deactive()	将队列的状态变为 DEACTIVED，并唤醒所有在入列或出列操作上等待的线程。该方法不释放任何已经入列的消息
pulse()	将队列状态变为 PULSED(脉冲)，并唤醒所有在入列或出列操作上等待的线程。该方法不释放任何已经入列的消息
state()	返回队列的状态
active()	将队列的状态变为 ACTIVED
~ACE_Message_Queue() close()	使队列停止活动，并立即释放所有已经入列的消息
flush()	释放队列中的消息，但不改变队列的状态

Task 框架 - ACE_Thread_Manager

- Task 框架：
  - 概要
  - ACE_Message_Block
  - ACE_Message_Queue
  - ACE_Thread_Manager
  - ACE_Task

Task 框架 - ACE_Thread_Manager

- 关于 ACE_Thread_Manager
  - ACE_Thread_Manager 封装了不同 OS 下多线程 API 在语法、语义上的差异，该类提供以下能力：
    - 一次创建一个或多个线程，每个线程并发执行一个应用程序指定的函数
    - 针对每个被创建的线程，改变其最常见的线程属性，如调度优先级、栈大小等
    - 创建并管理一组线程，将其视为一个有机的集合：线程组
    - 管理 ACE_Task 中的线程
    - 实现线程的协同取消 (cooperative cancellation) 能力
    - 等待一个或多个线程退出

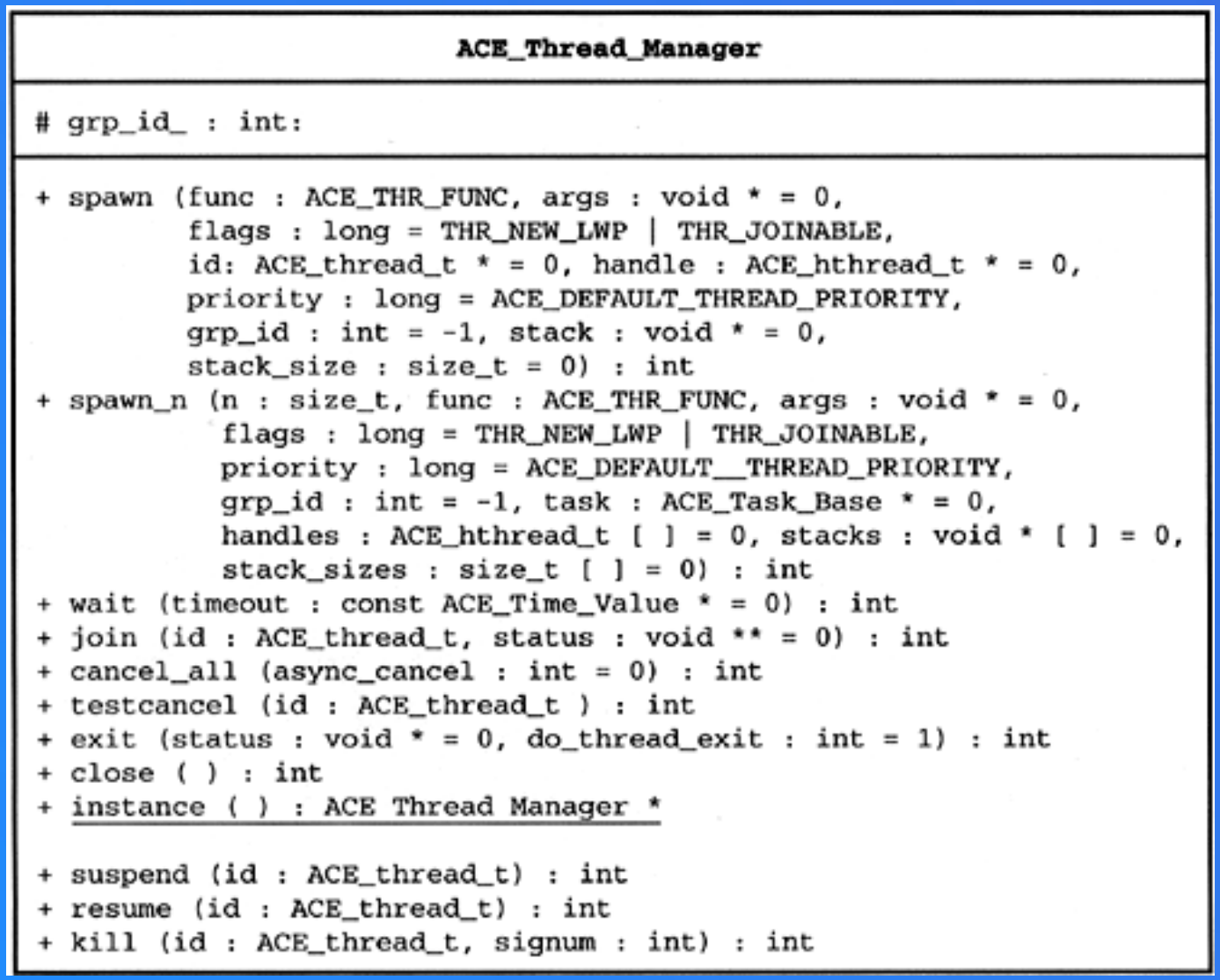
Task 框架 - ACE_Thread_Manager

- ACE_Thread_Manager 主要方法

方法	描述
spawn()	创建一个新的控制线程，并将一个函数和函数的参数传递给该线程，作为线程执行的入口
spawn_n()	与 spawn() 类似，不过是创建一个线程组
wait()	阻塞直至线程管理器中的所有线程都已退出，并获得所有 joinable 线程的退出状态
cancel_all()	请求 ACE_Thread_Manager 对象管理的线程全部退出
testcancel()	查看某个线程是否已被请求退出
exit()	退出一个线程，且释放该线程的资源
close()	关闭并释放所有被管理的线程的资源
instance()	静态方法，返回指向 ACE_Thread_Manager 单例的指针

Task 框架 - ACE_Thread_Manager

- ACE_Thread_Manager 类图



Task 框架 - ACE_Thread_Manager

- ACE_Thread_Manager::spawn() 的标志位

标志	描述
THR_SCOPE_SYSTEM	开启一个内核级线程，在系统范围内竞争
THR_SCOPE_PROCESS	开启一个用户级线程，在用户进程范围内竞争
THR_NEW_LWP	该标志影响进程的并发属性。对未绑定线程来说，其预想的并发级别是加 1，也就是添加一个新的内核线程到可用的线程池中，以运行用户线程，在不支持 N:M 用户 / 内核线程模型的 OS 平台上，此标志被忽略
THR_DETACHED	创建分离线程，该线程的退出状态不能被其它线程访问。当该线程退出时，其线程 ID 和其它资源会立即被 OS 回收。
THR_JOINABLE	创建可连接线程，也就是说其它线程可以通过 join() 方法得到它的退出状态

Task 框架 - ACE_Thread_Manager

- 示例 1： ACE_Thread_Manager 创建多线程

```
#include <ace/Thread_Manager.h>
#include <ace/Log_Msg.h>

ACE_Thread_Mutex nm, iom;
int n = 0;

static ACE_THR_FUNC_RETURN func(void* arg) {
    while (n < 100) {
        {
            ACE_Guard<ACE_Thread_Mutex> guard(nm);
            ++n;
        }
        ACE_OS::sleep(1);
        ACE_Guard<ACE_Thread_Mutex> guard2(iom);
        ACE_DEBUG((LM_DEBUG, "(%t) %d\n", n));
    }
    return 0;
}
```

Task 框架 - ACE_Thread_Manager

- 示例 1： ACE_Thread_Manager 创建多线程（续）

```
int main() {
    ACE_Thread_Manager::instance()->spawn_n(3, func, 0,
        THR_SCOPE_SYSTEM | THR_NEW_LWP);

    ACE_Thread_Manager::instance()->wait();
}
```

Task 框架 - ACE_Thread_Manager

- 示例 2： 一个并发的 Echo Server

```
#include <ace/INET_Addr.h>
#include <ace/SOCK_Acceptor.h>
#include <ace/SOCK_Stream.h>
#include <ace/Thread_Manager.h>
#include <ace/Log_Msg.h>

static ACE_THR_FUNC_RETURN handler(void* arg) {
    ACE_SOCK_Stream* peer = static_cast<ACE_SOCK_Stream*> (arg);
    char buf[512];
    ssize_t n = 0;
    for (;;) {
        memset(buf, 0, sizeof(buf));
        if ((n = peer->recv(buf, sizeof(buf))) == -1) {
            ACE_ERROR((LM_ERROR, "%p\n", "recv()"));
            break;
        } else if (peer->send(buf, n) == -1) {
            ACE_ERROR((LM_ERROR, "%p\n", "send()"));
            break;
        }
    }
```

Task 框架 - ACE_Thread_Manager

- 示例 2：一个并发的 Echo Server（续 1）

```
    peer->close();
    delete peer;
    return 0;
}

int main() {
    ACE_INET_Addr serverAddr;
    ACE_SOCK_Acceptor acceptor;

    if (serverAddr.set(8868) == -1)
        ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "set()"), 1);
    if (acceptor.open(serverAddr, 1) == -1)
        ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "open()"), 1);
```

Task 框架 - ACE_Thread_Manager

- 示例 2：一个并发的 Echo Server（续 2）

```
    for (;;) {
        ACE_SOCK_Stream* peer = new ACE_SOCK_Stream;
        if (acceptor.accept(*peer) == -1) {
            ACE_ERROR((LM_ERROR, "%p\n", "accept()"));
        } else {
            peer->disable(ACE_NONBLOCK); // Ensure blocking
<recv>s.
            ACE_Thread_Manager::instance()->spawn(handler, peer,
                THR_SCOPE_SYSTEM | THR_NEW_LWP | THR_DETACHED);
        }
    }

    return acceptor.close() == -1 ? 1 : 0;
}
```

Task 框架 - ACE_Task

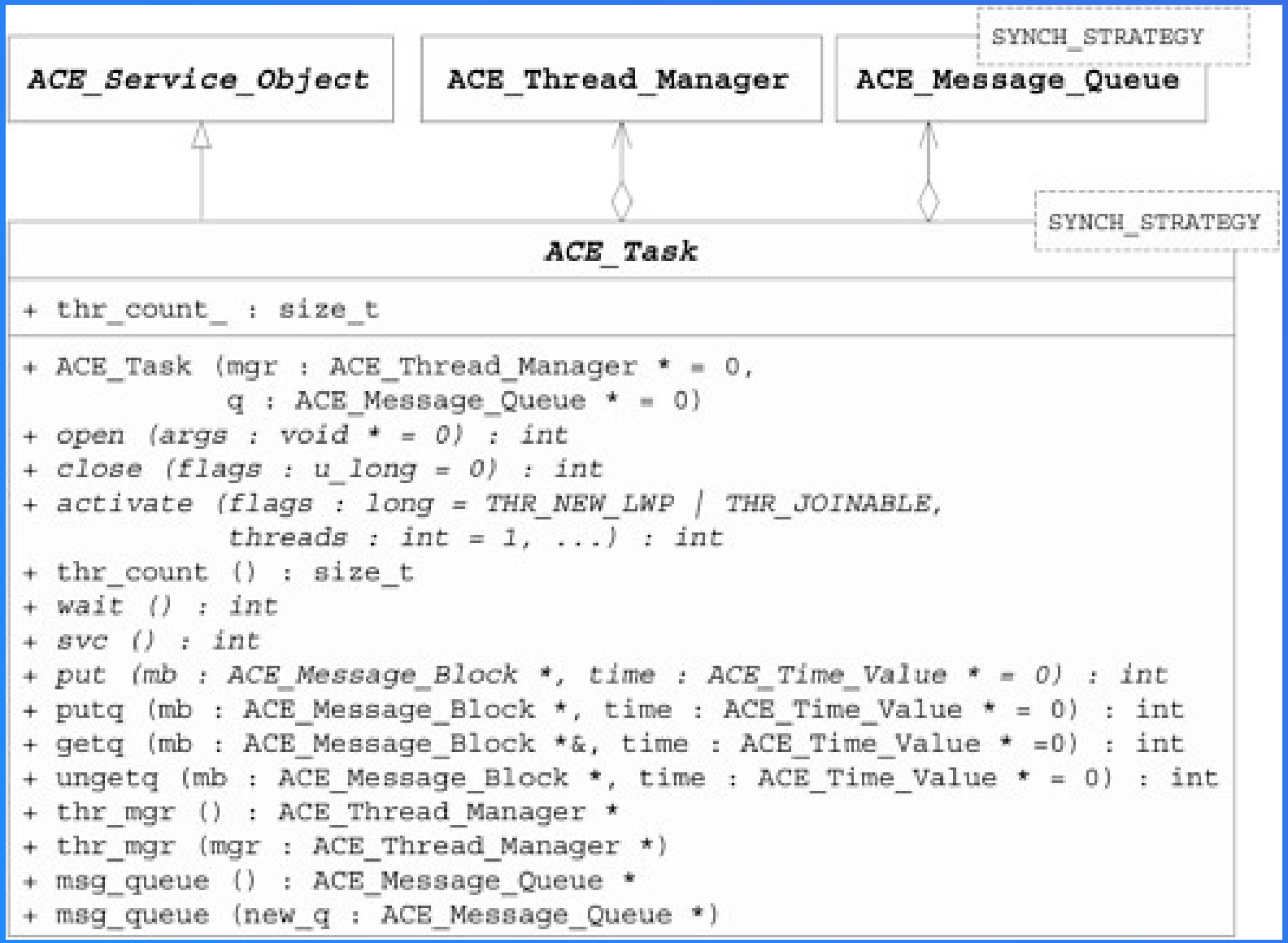
- Task 框架：
  - 概要
  - ACE_Message_Block
  - ACE_Message_Queue
  - ACE_Thread_Manager
  - ACE_Task

Task 框架 - ACE_Task

- 关于 ACE_Task
  - ACE_Task 是 ACE 面向对象并发框架的基础，该类提供以下能力：
    - 使用 ACE_Message_Queue 的一个实例来使数据和请求跟其处理相分离
    - 使用 ACE_Thread_Manager 类来激活任务，这样，任务就会作为主动对象而运行；这个主动对象在一个或多个控制线程中处理其队列中的消息。因为各个线程运行的是一个指定的类方法，所以它们可以直接访问任务的所有数据
    - 继承于 ACE_Service_Object，所以可以通过 ACE Service Configurator 框架对它进行动态配置
    - 继承于 ACE_Event_Handler，所以可以在 ACE_Reactor 框架中充当事件处理器

Task 框架 - ACE_Task

- ACE_Task 类图



Task 框架 - ACE_Task

- ACE_Task 主要方法
  - 任务初始化方法

方法	描述
ACE_Task()	构造函数，可以指定指向任务所用的 ACE_Manager_Queue 和 ACE_Thread_Manager 的指针
open()	挂钩方法，执行应用定义的初始化操作
thr_mgr()	获取和设置指向任务的 ACE_Thread_Manager 的指针
msg_queue()	获取和设置指向任务的 ACE_Message_Queue 的指针
activate()	将任务转换成运行在一个或多个线程中的主动对象

Task 框架 - ACE_Task

- ACE_Task 主要方法（续）
  - 任务通信、处理和同步方法

方法	描述
svc()	可以实现任务的服务处理的挂钩方法。所有通过 activate() 方法派生的线程都会执行它
put()	可被用于传递消息给任务的挂钩方法。消息可被立即处理，或是排队等待 svc() 挂钩方法随后处理
putq() getq() ungetq()	在任务的消息队列中插入、移除和替换消息。 putq()、 getq() 和 ungetq() 方法分别简化了对任务的消息队列的 enqueue_tail()、 dequeue—head() 和 enqueue()_head() 方法的访问

Task 框架 - ACE_Task

- 关于 put() 和 svc() 方法
  - put() 方法为被动处理
    - 该方法用于传递消息给某个 ACE_Task 。在任务之间传递的是 ACE_Message_Block 指针，以避免数据拷贝开销。
    - 该方法的执行占用调用者线程处理时间
  - svc() 方法为主动处理
    - Task 内的一个或多个线程会并发执行 svc() 方法
    - Task 以外的线程也可以通过调用 put() 方法将消息传递给 Task，而 put() 方法也可以不需处理该消息，仅需调用 putq() 将消息放入消息队列，由 svc() 方法调用 getq() ，而后并发的处理这些消息
    - svc() 方法不会被 Task 以外的线程调用

Task 框架 - ACE_Task

- ACE_Task 主要方法（续 2 ）
  - 任务析构

方法	描述
~ACE_Task()	删除在 ACE_Task 构造函数中分配的资源
close()	执行应用定义的关闭活动的挂钩方法。该方法通常不应由应用直接调用，特别是如果任务是主动对象的化
flush()	关闭与任务相关联的消息队列，从而释放所有在队列中的消息块，并释放所有阻塞在队列上的线程
thr_count()	返回活动在 ACE_Task 中的线程的总数
wait()	一个栅栏同步体，它在返回之前等待所有运行在此任务中 的 joinable 的线程退出

Task 框架 - ACE_Task

- ACE_Task 示例：生产者与消费者

```
#include "ace/Task.h"
#include "ace/Message_Block.h"

const int N = 10;

//The Consumer Task.
class Consumer: public ACE_Task<ACE_MT_SYNCH> {
public:
    int open(void*) {
        ACE_DEBUG((LM_DEBUG, "(%t)Consumer task opened \n"));
        //Activate the Task
        activate(THR_NEW_LWP, 3);
        return 0;
    }
```

- ACE_Task 示例：生产者与消费者（续 1 ）

```
    //The Service Processing routine
    int svc(void) {
        ACE_Message_Block* mb = 0;
        for (;;) {
            mb = 0; getq(mb);//Get message from underlying queue
            if (*mb->rd_ptr() < N) {
                ACE_DEBUG((LM_DEBUG, "(%t)Got message: %d from
remote task\n", *mb->rd_ptr()));
            } else if (*mb->rd_ptr() == N) {
                ACE_DEBUG((LM_DEBUG, "(%t)Got message: %d from
remote task\n", *mb->rd_ptr()));
                ++*mb->rd_ptr(); // *mb->rd_ptr() == N+1
                ungetq(mb); break;// 供后续线程查看
            } else { // *mb->rd_ptr() > N
                ungetq(mb); break;// 供后续线程查看
            }
        }
        return 0;
    }
```

Task 框架 - ACE_Task

- ACE_Task 示例：生产者与消费者（续 2 ）

```
    int close(u_long) {
        ACE_DEBUG((LM_DEBUG, "(%t)Consumer closes down \n"));
        return 0;
    }
};
```

Task 框架 - ACE_Task

- ACE_Task 示例：生产者与消费者（续 3 ）

```
class Producer: public ACE_Task_Base {
public:
    Producer(Consumer * consumer) :
        data_(0), consumer_(consumer) {
        mb_ = new ACE_Message_Block((char*) &data_,
                                    sizeof(data_));
    }
    int open(void*) {
        ACE_DEBUG((LM_DEBUG, "(%t)Producer task opened \n"));
        //Activate the Task
        activate(THR_NEW_LWP, 1);
        return 0;
    }
```

Task 框架 - ACE_Task

- ACE_Task 示例：生产者与消费者（续 4 ）

```
    //The Service Processing routine
    int svc(void) {
        while (data_ < N) {
            //Send message to consumer
            ACE_DEBUG((LM_DEBUG, "(%t)Sending message: %d to
remote task\n", data_));
            consumer_->putq(mb_);
            //Go to sleep for a sec.
            ACE_OS::sleep(1);
            ++data_;
        }
        return 0;
    }
```

Task 框架 - ACE_Task

- ACE_Task 示例：生产者与消费者（续 5 ）

```
    int close(u_long) {
        ACE_DEBUG((LM_DEBUG, "(%t)Producer closes down \n"));
        return 0;
    }
private:
    char data_;
    Consumer * consumer_;
    ACE_Message_Block * mb_;
};

int main() {
    Consumer *consumer = new Consumer;
    Producer * producer = new Producer(consumer);
    producer->open(0);
    consumer->open(0);
    //Wait for all the tasks to exit.
    ACE_Thread_Manager::instance()->wait();
}
```

Task 框架 - 小结

- ACE Task 框架使用 ACE_Task 类将多线程与面向对象的编程和队列集成在一起， ACE_Task 的队列机制使用了 ACE_Message_Queue 类在任务间高效的传递消息
- 该框架还可以与 ACE Reactor 框架结合起来，实现 half-Sync/Half-Async 模式