第9章 消息队列(Message Queue)

现代的实时应用通常被构建成一组相互通信、但又相互独立的任务。这些任务可以通过若干机制来与对方进行通信，其中常用的一种就是消息队列。在这一情况下，基本的通信模式是：发送者（或生产者）任务将消息放入消息队列，而接收者（或消费者）任务从此队列中取出消息。这当然只是消息队列的使用方式之一。在接下来的讨论中，我们将看到若干不同的使用消息队列的例子。

ACE中的消息队列是仿照UNIX系统V的消息队列设计的，如果你已经熟悉系统V的话，就很容易掌握ACE的消息队列的使用。在ACE中有多种不同类型的消息队列可用，每一种都使用不同的调度算法来进行队列的入队和出队操作。

9.1 消息块

在ACE中，消息作为消息块（Message Block）被放入消息队列中。消息块包装正被存储的实际消息数据，并提供若干数据插入和处理操作。每个消息块“包含”一个头和一个数据块。注意在这里“包含”是在宽松的意义上使用的。消息块可以不对与数据块（Data Block）或是消息头（Message Header）相关联的内存进行管理（尽管你可以让消息块进行这样的管理）。它仅仅持有指向两者的指针。所以包含只是逻辑上的。数据块持有指向实际的数据缓冲区的指针。如图9-1所示，这样的设计带来了多个消息块之间的数据的灵活共享。注意在图中两个消息块共享一个数据块。这样，无需带来数据拷贝开销，就可以将同一数据放入不同的队列中。

消息块类名为ACE_Message_Block，而数据块类名为ACE_Data_Block。ACE_Message_Block的构造器是实际创建消息块和数据块的方便办法。

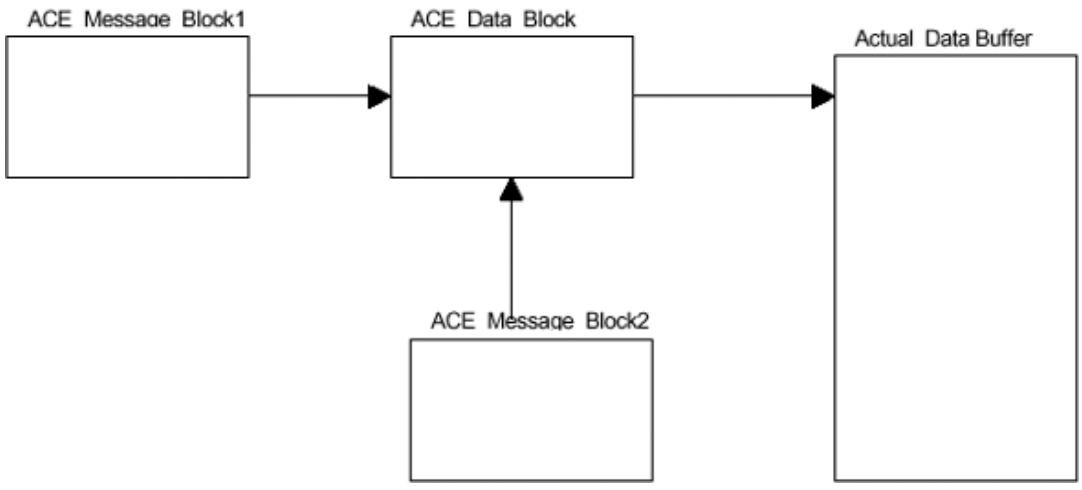

图9-1 ACE消息块的构造

9.1.1 构造消息块

ACE_Message_Block类包含有若干不同的构造器。你可以使用这些构造器来帮助你管理隐藏在消息和数据块后面的消息数据。ACE_Message_Block类可用于完全地隐藏ACE_Data_Block，并为你管理消息数据；或者，如果你需要，你可以自己创建和管理数据块及消息数据。下一部分将考查怎样使用ACE_Message_Block来管理消息内存和数据块。然后我们将考查怎样独立地进行这样的管理，而不用依赖ACE_Message_Block的管理特性。

9.1.1.1 ACE_Message_Block分配和管理数据内存

要创建消息块，最容易的方法是将底层数据块的大小传给ACE_Message_Block的构造器，从而创建ACE_Data_Block，并为消息数据分配空的内存区。在创建消息块后，你可以使用rd_ptr()和wr_ptr()方法来在消息块中插入和移除数据。让ACE_Message_Block来为数据和数据块创建内存区的主要优点是，它会为你正确地管理所有内存，从而使你免于在将来为许多内存泄漏而头疼。

ACE_Message_Block的构造器还允许程序员指定ACE_Message_Block在内部分配内存时所应使用的分配器。如果你传入一个分配器，消息块将用它来为数据块和消息数据区的创建分配内存。ACE_Message_Block的构造器为：

```
ACE_Message_Block (size_t size,
ACE_Message_Type type = MB_DATA,
ACE_Message_Block *cont = 0,
const char *data = 0,
ACE_Allocator *allocator_strategy = 0,
ACE_Lock *locking_strategy = 0,
u_long priority = 0,
const ACE_Time_Value & execution_time = ACE_Time_Value::zero,
const ACE_Time_Value & deadline_time = ACE_Time_Value::max_time);
```

上面的构造器的参数为：

1. 要与消息块相关联的数据缓冲区的大小。注意消息块的大小是size，但长度将为 0 ，直到wr_ptr被设置为止。这将在后面进一步解释。
2. 消息的类型。（在ACE_Message_Type枚举中有若干类型可用，其中包括缺省的数据消息）。
3. 指向“片段链”（fragment chain）中的下一个消息块的指针。消息块可以实际地链接在一起来形成链。随后链可被放入消息队列中，就好像它是单个数据块一样。该参数缺省为 0 ，意味着此块不使用链。
4. 指向要存储在此消息块中的数据缓冲区的指针。如果该参数的值为零，就会创建缓冲区（大小由第一个参数指定），并由该消息块进行管理。当消息块被删除时，相应的数据缓冲区也被删除。但是，如果在此参数中指定了数据缓冲区，也就是，参数不为空，当消息块被销毁时它就**不会**删除数据缓冲区。这是一个重要特性，必须牢牢记住。

5. 用于分配数据缓存（如果需要）的allocator_strategy，在第四个参数为空时使用（如上面所解释的）。任何ACE_Allocator的子类都可被用作这一参数。（关于ACE_Allocator的更多信息，参见“内存管理”一章）。
6. 如果locking_strategy不为零，它就将用于保护访问共享状态（例如，引用计数）的代码区，以避免竞争状态。
7. 这个参数以及后面两个参数用于ACE中的实时消息队列的调度，目前应保留它们的缺省值。

9.1.1.2 用户分配和管理消息内存

如果你正在使用ACE_Message_Block，你并不一定要让它来为你分配内存。消息块的构造器允许你：

- 创建并传入你自己的指向消息数据的数据块。
- 传入指向消息数据的指针，消息块将创建并设置底层的数据块。消息块将为数据块、而不是消息数据管理内存。

下面的例子演示怎样将指向消息数据的指针传给消息块，以及ACE_Message_Block怎样创建和管理底层的ACE_Data_Block。

```
//The data

char data[size];

data = ”This is my data”;


//Create a message block to hold the data

ACE_Message_Block *mb = new ACE_Message_Block (data, // data that is stored
                                                    //
                                                    in
                                                    the
                                                    newly
                                                    created
                                                    data
                                                    //
                                               blocksize);
                                               //size
                                               of
                                               the
                                               block
                                               that
                                                    //is
                                                    to
                                                    be
                                                    stored.
```

该构造器创建底层数据块，并将它设置为指向传递给它的数据的开头。被创建的消息块并不拷贝该数据，也不假定自己拥有它的所有权。这就意味着在消息块mb被销毁时，相关联的数据缓冲区data将**不**

会被销毁。这是有意义的：消息块没有拷贝数据，因此内存也不是它分配的，这样它也不应该负责销毁它。

9.1.2 在消息块中插入和操作数据

除了构造器，ACE_Message_Block还提供若干方法来直接在消息块中插入数据。另外还有一些方法可用来操作已经在消息块中的数据。

每个ACE_Message_Block都有两个底层指针：rd_ptr和wr_ptr，用于在消息块中读写数据。它们可以通过调用rd_ptr()和wr_ptr()方法来直接访问。rd_ptr指向下一次读取数据的位置，而wr_ptr指向下一次写入数据的位置。程序员必须小心地管理这些指针，以保证它们总是指向正确的位置。在使用这些指针读写数据时，程序员必须自己来增加它们的值，它们不会魔法般地自动更新。大多数内部的消息块方法也使用这两个指针，从而使它们能够在你调用消息块方法时改变指针状态。程序员必须保证自己了解这些指针的变化。

9.1.2.1 拷贝与复制（Copying and Duplicating）

可以使用ACE_Message_Block的copy()方法来将数据拷贝进消息块。

```
int copy(const char *buf, size_t n);
```

copy方法需要两个参数，其一是指向要拷贝进消息块的缓冲区的指针，其二是要拷贝的数据的大小。该方法从wr_pt指向的位置开始往前写，直到它到达参数n所指示的数据缓冲区的末尾。copy()还会保证wr_ptr的更新，使其指向缓冲区的新末尾处。注意该方法将实际地执行物理拷贝，因而应该小心使用。

base()和length()方法可以联合使用，以将消息块中的整个数据缓冲区拷贝*出来*。base()返回指向数据块的第一个数据条目的指针，而length()返回队中数据的总大小。将base和length相加，可以得到指向数据块末尾的指针。合起来使用这些方法，你就可以写一个例程来从消息块中取得数据，并做一次外部拷贝。

duplicate()和clone()方法用于制作消息块的“副本”。如它的名字所暗示的，clone()方法实际地创建整个消息块的新副本，包括它的数据块和附加部分；也就是说，这是一次“深度复制”。而另一方面，duplicate()方法使用的是ACE_Message_Block的引用计数机制。它返回指向要被复制的消息块的指针，并在内部增加内部引用计数。

9.1.2.2 释放消息块

一旦使用完消息块，程序员可以调用它的release()方法来释放它。如果消息数据内存是由该消息块分配的，调用release()方法就也会释放此内存。如果消息块是引用计数的，release()就会减少计数，直到到达 0 为止；之后消息块和与它相关联的数据块才从内存中被移除。

9.2 ACE的消息队列

如先前所提到的，ACE有若干不同类型的消息队列，它们大体上可划分为两种范畴：静态的和动态的。静态队列是一种通用的消息队列（ACE_Message_Queue），而动态消息队列（ACE_Dynamic_Message_Queue）是实时消息队列。这两种消息队列的主要区别是：静态队列中的消息具有静态的优先级，也就是，一旦优先级被设定就不会再改变；而另一方面，在动态消息队列中，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。

下面的例子演示怎样创建简单的静态消息队列，以及怎样在其上进行消息块的入队和出队操作。

例9-1a

```
#ifndef MQ_EG1_H_
#define MQ_EG1_H_

#include "ace/Message_Queue.h"

class QTest
{
public:
//Constructor creates a message queue with no synchronization
QTest(int num_msgs);

//Enqueue the num of messages required onto the message mq.
int enq_msgs();

//Dequeue all the messages previously enqueued.
int deq_msgs ();

private:
//Underlying message queue
ACE_Message_Queue<ACE_NULL_SYNCH> *mq_;

//Number of messages to enqueue.
int no_msgs_;
};
#endif /*MQ_EG1.H_*/
```

例9-1b

```
#include "mq_eg1.h"

QTest::QTest(int num_msgs)
:no_msgs_(num_msgs)
```

```
{
ACE_TRACE("QTest::QTest");

//First create a message queue of default size.
if(!(this->mq_=new ACE_Message_Queue<ACE_NULL_SYNCH> ()))
      ACE_DEBUG((LM_ERROR,"Error in message queue initialization \n"));
}

int  QTest::enq_msgs()
{
ACE_TRACE("QTest::enq_msg");
for(int i=0; i<no_msgs_;i++)
{
      //create a new message block specifying exactly how large
      //an underlying data block should be created.
      ACE_Message_Block *mb;
      ACE_NEW_RETURN(mb,
                              ACE_Message_Block(ACE_OS::strlen("This
                              is message 1\n")),
                        -1);

      //Insert data into the message block using the wr_ptr
      ACE_OS::sprintf(mb->wr_ptr(), "This is message %d\n", i);

      //Be careful to advance the wr_ptr by the necessary amount.
      //Note that the argument is of type "size_t" that is mapped to
      //bytes.
      mb->wr_ptr(ACE_OS::strlen("This is message 1\n"));

      //Enqueue the message block onto the message queue
      if(this->mq_->enqueue_prio(mb)==-1)
      {
            ACE_DEBUG((LM_ERROR,"\nCould not enqueue on to mq!!\n"));
            return -1;
      }

      ACE_DEBUG((LM_INFO,"EQ'd data: %s\n", mb->rd_ptr() ));
} //end for

//Now dequeue all the messages
this->deq_msgs();
```

```
return 0;
}

int  QTest::deq_msgs()
{
ACE_TRACE("QTest::dequeue_all");
ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n"
            ,mq_->message_count(),mq_->message_bytes()));
      ACE_Message_Block *mb;

//dequeue the head of the message queue until no more messages are
//left. Note that I am overwriting the message block mb and I since
//I am using the dequeue_head() method I dont have to worry about
//resetting the rd_ptr() as I did for the wrt_ptr()
for(int i=0;i <no_msgs_; i++)
{
      mq_->dequeue_head(mb);
      ACE_DEBUG((LM_INFO,"DQ'd data %s\n", mb->rd_ptr() ));

      //Release the memory associated with the mb
      mb->release();
}

return 0;
}

int main(int argc, char* argv[])
{
if(argc <2)
      ACE_ERROR_RETURN((LM_ERROR, "Usage %s num_msgs", argv[0]), -1);

QTest test(ACE_OS::atoi(argv[1]));
if(test.enq_msgs() == -1)
      ACE_ERROR_RETURN( (LM_ERROR,"Program failure \n"), -1);
}
```

上例演示了消息队列类的若干方法。例子由一个Qtest类组成，它通过ACE_NULL_SYNCH锁定来实例化缺省大小的消息队列。锁（互斥体和条件变量）被消息队列用来：

- 保证由消息块维护的引用计数的安全，防止在有多个线程访问时的竞争状态。

- “唤醒”所有因为消息队列空或满而休眠的线程。

在此例中，因为只有一个线程，消息队列的模板同步参数被设置为空（ACE_NULL_SYNCH，意味着使用ACE_Null_Mutex和ACE_Null_Condition）。随后Qtest的enq_msgs()方法被调用，它进入循环，创建消息、并将其放入消息队列中。消息数据的大小作为参数传给ACE_Message_Block的构造器。使用该构造器使得内存被自动地管理（也就是，内存将在消息块被删除时，即release()时被释放）。wr_ptr随后被获取（使用wr_ptr()访问方法），且数据被拷贝进消息块。在此之后，wr_ptr向前增长。然后使用消息队列的enqueue_prio()方法来实际地将消息块放入底层消息队列中。

在no_msgs_个消息块被创建、初始化和插入消息队列后，enq_msgs()调用deq_msgs()方法。该方法使用ACE_Message_Queue的dequeue_head()方法来使消息队列中的每个消息出队。在消息出队后，就显示它的数据，然后再释放消息。

9.3 水位标

水位标用于在消息队列中指示何时在其中的数据已过多（消息队列到达了高水位标），或何时在其中的数据的数量不足（消息队列到达了低水位标）。两种水位标都用于流量控制栗例如，low_water_mark可用于避免像TCP中的“傻窗口综合症”（silly window syndrome）那样的情况，而high_water_mark可用于“阻止“或减缓数据的发送或生产。

ACE中的消息队列通过维护已经入队的总数据量的字节计数来获得这些功能。因而，无论何时有新消息块被放入消息队列中，消息队列都将先确定它的长度，然后检查是否能将此消息块放入队列中（也就是，确认如果将此消息块入队，消息队列没有超过它的高水位标）。如果消息队列不能将数据入队，而它又持有一个锁（也就是，使用了ACE_SYNC，而不是ACE_NULL_SYNCH作为消息队列的模板参数），它就会阻塞调用者，直到有足够的空间可用，或是入队方法的超时（timeout）到期。如果超时已到期，或是队列持有一个空锁，入队方法就会返回-1，指示无法将消息入队。

类似地，当ACE_Message_Queue的dequeue_head方法被调用时，它检查并确认在出队之后，剩下的数据的数量高于低水位标。如果不是这样，而它又持有一个锁，它就会阻塞；否则就返回-1，指示失败（和入队方法的工作方式一样）。

分别有两个方法可用于设置和获取高低水位标：

```
//get the high water mark
size_t high_water_mark(void)

//set the high water mark
void high_water_mark(size_t hwm);

//get the low water_mark
```

```
size_t low_water_mark(void)


//set the low water_mark

void low_water_mark(size_t lwm)
```

9.4 使用消息队列迭代器(Message Queue Iterator)

和其它容器类的常见情况一样，可将前进（forward）和后退（reverse）迭代器用于ACE中的消息队列。这两个迭代器名为ACE_Message_Queue_Iterator和ACE_Message_Queue_Reverse_Iterator。它们都需要一个模板参数，用于在遍历消息队列时进行同步。如果有多个线程使用消息队列，该参数就应设为ACE_SYNCH；否则，就可设为ACE_NULL_SYNCH。在迭代器对象被创建时，必须将我们想要进行迭代的消息队列的引用传给它的构造器。

下面的例子演示水位标和迭代器的使用：

例9-2

```
#include ”ace/Message_Queue.h”

#include ”ace/Get_Opt.h”

#include ”ace/Malloc_T.h”

#define SIZE_BLOCK 1


class Args

{

public:

Args(int argc, char*argv[],int& no_msgs, ACE_Message_Queue<ACE_NULL_SYNCH>* &mq)

{

      ACE_Get_Opt get_opts(argc,argv,”h:l:t:n:xsd”);

      while((opt=get_opts())!=-1)

            switch(opt)

            {

            case ’n’:

                        //set the number of messages we wish to enqueue and
                        dequeue

                        no_msgs=ACE_OS::atoi(get_opts.optarg);

                        ACE_DEBUG((LM_INFO,”Number of Messages %d
                        \n”,no_msgs));

                        break;


            case ’h’:

                        //set the high water mark

                        hwm=ACE_OS::atoi(get_opts.optarg);
```

```
                        mq->high_water_mark(hwm);
                        ACE_DEBUG((LM_INFO,"High Water Mark %d msgs \n",hwm));
                        break;

            case 'l':
                        //set the low water mark
                        lwm=ACE_OS::atoi(get_opts.optarg);
                        mq->low_water_mark(lwm);
                        ACE_DEBUG((LM_INFO,"Low Water Mark %d msgs \n",lwm));
                        break;

            default:
                        ACE_DEBUG((LM_ERROR,
                              "Usage -n<no. messages> -h<hwm> -
                              l<lwm>\n"));
                        break;
            }
}

private:
int opt;
int hwm;
int lwm;
};

class Qtest
{
public:
QTest(int argc, char*argv[])
{
      //First create a message queue of default size.
      if(!(this->mq_=new ACE_Message_Queue<ACE_NULL_SYNCH> ()))
            ACE_DEBUG((LM_ERROR,"Error in message queue initialization \n"));

      //Use the arguments to set the water marks and the no of messages
      args_ = new Args(argc,argv,no_msgs_,mq_);
}

int start_test()
{
      for(int i=0; i<no_msgs_;i++)
      {
```

```
            //Create a new message block of data buffer size 1
            ACE_Message_Block * mb= new ACE_Message_Block(SIZE_BLOCK);

            //Insert data into the message block using the rd_ptr
            *mb->wr_ptr()=i;

            //Be careful to advance the wr_ptr
            mb->wr_ptr(1);

            //Enqueue the message block onto the message queue
            if(this->mq_->enqueue_prio(mb)==-1)
            {
                        ACE_DEBUG((LM_ERROR,"\nCould not enqueue on to
                        mq!!\n"));
                        return -1;
            }

            ACE_DEBUG((LM_INFO,"EQ'd data: %d\n",*mb->rd_ptr()));
      }

      //Use the iterators to read
      this->read_all();

      //Dequeue all the messages
      this->dequeue_all();

      return 0;
}

void read_all()
{
      ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n"
                        ,mq_->message_count(),mq_-
                        >message_bytes()));
      ACE_Message_Block *mb;

      //Use the forward iterator
      ACE_DEBUG((LM_INFO,"\n\nBeginning Forward Read \n"));
      ACE_Message_Queue_Iterator<ACE_NULL_SYNCH> mq_iter_(*mq_);
      while(mq_iter_.next(mb))
      {
            mq_iter_.advance();
```

```
            ACE_DEBUG((LM_INFO,"Read data %d\n",*mb->rd_ptr()));
      }

      //Use the reverse iterator
      ACE_DEBUG((LM_INFO,"\n\nBeginning Reverse Read \n"));
      ACE_Message_Queue_Reverse_Iterator<ACE_NULL_SYNCH>
      mq_rev_iter_(*mq_);
      while(mq_rev_iter_.next(mb))
      {
            mq_rev_iter_.advance();
            ACE_DEBUG((LM_INFO,"Read data %d\n",*mb->rd_ptr()));
      }
}

void dequeue_all()
{
      ACE_DEBUG((LM_INFO,"\n\nBeginning DQ \n"));
      ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n",
            mq_->message_count(),mq_->message_bytes()));
      ACE_Message_Block *mb;

      //dequeue the head of the message queue until no more messages
      //are left
      for(int i=0;i<no_msgs_;i++)
      {
            mq_->dequeue_head(mb);
            ACE_DEBUG((LM_INFO,"DQ'd data %d\n",*mb->rd_ptr()));
      }
}

private:
Args *args_;
ACE_Message_Queue<ACE_NULL_SYNCH> *mq_;
int no_msgs_;
};

int main(int argc, char* argv[])
{
QTest test(argc,argv);
if(test.start_test()<0)
      ACE_DEBUG((LM_ERROR,"Program failure \n"));
```

```
}
```

这个例子使用ACE_Get_Opt类（更多关于这个工具类的信息见附录）来获取低水位标和高水位标（在Args类中）。使用low_water_mark()和high_water_mark()访问函数可对它们进行设置。除此而外，还有一个read_all()方法使用ACE_Message_Queue_Iterator和ACE_Message_Queue_Reverse_Iterator来向前读和反向读。

9.5 动态或实时消息队列

如上面所提到的，动态消息队列是其中的消息的优先级随时间变化的队列。实时应用需要这样的行为特性，因而这样的队列在实时应用中天生更为有用。

ACE目前提供两种动态消息队列：基于最终期限（deadline）的和基于松弛度（laxity）的（参见[IX]）动态消息队列。基于最终期限的消息队列通过每个消息的最终期限来设置它们的优先级。在使用最早deadline优先算法来调用dequeue_head()方法时，队列中有着最早的最终期限的消息块将最先出队。而基于松弛度的消息队列，同时使用执行时间和最终期限来计算松弛度，并将其用于划分各个消息块的优先级。松弛度是十分有用的，因为在根据最终期限来调度时，被调度的任务有可能有最早的最终期限，但同时又有相当长的执行时间，以致于即使它被立即调度，也不能够完成。这会消极地影响其它任务，因为它可能阻塞那些可以调度的任务。松弛度把这样的长执行时间考虑在内，并保证任务如果不能完成，就不会被调度。松弛度队列中的调度基于最小松弛度优先算法。

基于松弛度的消息队列和基于最终期限的消息队列都实现为ACE_Dynamic_Message_Queue。ACE使用**策略**（STRATEGY）模式来为动态队列提供不同的调度特性。每种消息队列使用不同的“策略”对象来动态地设置消息队列中消息的优先级。每个这样的“策略”对象都封装了一种不同的算法来基于执行时间、最终期限，等等，计算优先级；并且无论何时消息入队或是出队，都会调用这些策略对象来完成前述计算工作。（有关策略模式的更多信息，请参见“设计模式”）。消息策略模式派生自ACE_Dynamic_Message_Strategy，目前有两种策略可用：ACE_Laxity_Message_Strategy和ACE_Deadline_Message_Strategy。因此，要创建基于松弛度的动态消息队列，首先必须创建ACE_Laxity_Message_Strategy对象。随后，应该对ACE_Dynamic_Message_Queue对象进行实例化，并将新创建的策略对象作为参数之一传给它的构造器。

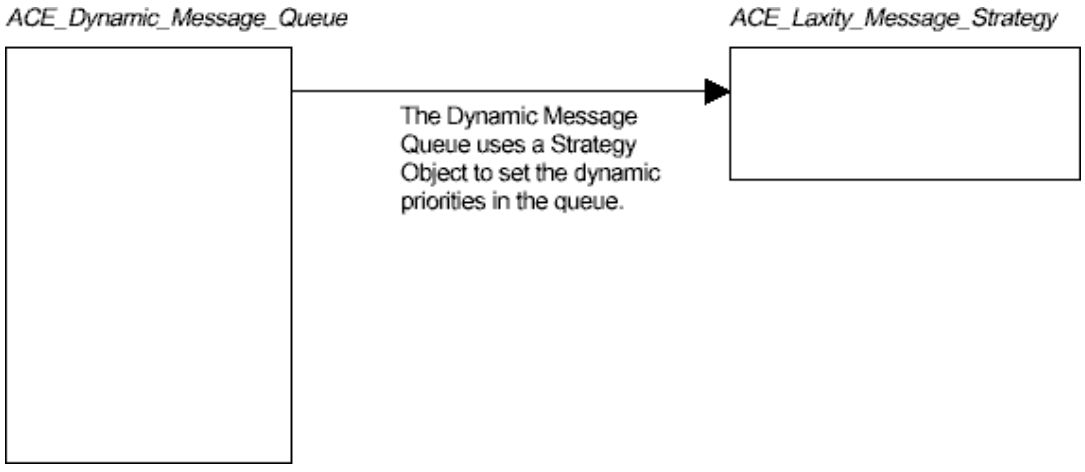

图9-2 ACE_Dynamic_Message_Queue与ACE_Laxity_Message_Strategy的关系

创建消息队列

为简化这些不同类型的消息队列的创建，ACE提供了名为ACE_Message_Queue_Factory的具体消息队列工厂，它使用工厂方法（FACTORY METHOD，更多信息参见“设计模式”）模式的一种变种来创建适当类型的消息队列。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：

```
static ACE_Message_Queue<ACE_SYNCH_USE> *
create_static_message_queue();

static ACE_Dynamic_Message_Queue<ACE_SYNCH_USE> *
create_deadline_message_queue();

static ACE_Dynamic_Message_Queue<ACE_SYNCH_USE> *
create_laxity_message_queue();
```

每个方法都返回指向刚创建的消息队列的指针。注意这些方法都是静态的，而create_static_message_queue()方法返回的是ACE_Message_Queue，其它两个方法则返回ACE_Dynamic_Message_Queue。

下面这个简单的例子演示动态和静态消息队列的创建和使用：

例9-3

```
#include ”ace/Message_Queue.h”
#include ”ace/Get_Opt.h”
#include ”ace/OS.h”

class Args
{
public:
Args(int argc, char*argv[],int& no_msgs, int& time,
      ACE_Message_Queue<ACE_NULL_SYNCH>*&mq)
{
      ACE_Get_Opt get_opts(argc,argv,”h:l:t:n:xsd”);
      while((opt=get_opts())!=-1)
            switch(opt)
            {
            case ’t’:
```

```
            time=ACE_OS::atoi(get_opts.optarg);
            ACE_DEBUG((LM_INFO,"Time: %d \n",time));
            break;

            case 'n':
            no_msgs=ACE_OS::atoi(get_opts.optarg);
            ACE_DEBUG((LM_INFO,"Number of Messages %d
            \n",no_msgs));
            break;

case 'x':
            mq=ACE_Message_Queue_Factory<ACE_NULL_SYNCH>::
            create_laxity_message_queue();
            ACE_DEBUG((LM_DEBUG,"Creating laxity q\n"));
            break;

case 'd':
            mq=ACE_Message_Queue_Factory<ACE_NULL_SYNCH>::
            create_deadline_message_queue();
            ACE_DEBUG((LM_DEBUG,"Creating deadline q\n"));
            break;

case 's':
            mq=ACE_Message_Queue_Factory<ACE_NULL_SYNCH>::
            create_static_message_queue();
            ACE_DEBUG((LM_DEBUG,"Creating static q\n"));
            break;

case 'h':
            hwm=ACE_OS::atoi(get_opts.optarg);
            mq->high_water_mark(hwm);
            ACE_DEBUG((LM_INFO,"High Water Mark %d msgs \n",hwm));
            break;

case 'l':
            lwm=ACE_OS::atoi(get_opts.optarg);
            mq->low_water_mark(lwm);
            ACE_DEBUG((LM_INFO,"Low Water Mark %d msgs \n",lwm));
            break;

default:
            ACE_DEBUG((LM_ERROR,"Usage specify queue type\n"));
```

```
                        break;
            }
}

private:
int opt;
int hwm;
int lwm;
};

class Qtest
{
public:
QTest(int argc, char*argv[])
{
      args_ = new Args(argc,argv,no_msgs_,time_,mq_);
      array_ =new ACE_Message_Block*[no_msgs_];
}

int start_test()
{
      for(int i=0; i<no_msgs_;i++)
      {
            ACE_NEW_RETURN (array_[i], ACE_Message_Block (1), -1);
                        set_deadline(i);
            set_execution_time(i);
            enqueue(i);
      }

      this->dequeue_all();

      return 0;
}

//Call the underlying ACE_Message_Block method msg_deadline_time() to
//set the deadline of the message.
void set_deadline(int msg_no)
{
      float temp=(float) time_/(msg_no+1);
      ACE_Time_Value tv;
      tv.set(temp);
```

```
      ACE_Time_Value deadline(ACE_OS::gettimeofday()+tv);
      array_[msg_no]->msg_deadline_time(deadline);
      ACE_DEBUG((LM_INFO,"EQ'd with DLine %d:%d,", deadline.sec(),deadline.usec()));
}

//Call the underlying ACE_Message_Block method to set the execution time
void set_execution_time(int msg_no)
{
      float temp=(float) time_/10*msg_no;
      ACE_Time_Value tv;
      tv.set(temp);
      ACE_Time_Value xtime(ACE_OS::gettimeofday()+tv);
      array_[msg_no]->msg_execution_time (xtime);
      ACE_DEBUG((LM_INFO,"Xtime %d:%d,",xtime.sec(),xtime.usec()));
}

void enqueue(int msg_no)
{
      //Set the value of data at the read position
      *array_[msg_no]->rd_ptr()=msg_no;

      //Advance write pointer
      array_[msg_no]->wr_ptr(1);

      //Enqueue on the message queue
      if(mq_->enqueue_prio(array_[msg_no])==-1)
      {
            ACE_DEBUG((LM_ERROR,"\nCould not enqueue on to mq!!\n"));
            return;
      }

      ACE_DEBUG((LM_INFO,"Data %d\n",*array_[msg_no]->rd_ptr()));
      }

void dequeue_all()
{
      ACE_DEBUG((LM_INFO,"Beginning DQ \n"));
      ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n",
      mq_->message_count(),mq_->message_bytes()));

      for(int i=0;i<no_msgs_ ;i++)
```

```
      {
            ACE_Message_Block *mb;
            if(mq_->dequeue_head(mb)==-1)
            {
                  ACE_DEBUG((LM_ERROR,"\nCould not dequeue from
                  mq!!\n"));
                  return;
            }

            ACE_DEBUG((LM_INFO,"DQ'd data %d\n",*mb->rd_ptr()));
      }
}

private:
Args *args_;
ACE_Message_Block **array_;
ACE_Message_Queue<ACE_NULL_SYNCH> *mq_;
int no_msgs_;
int time_;
};

int main(int argc, char* argv[])
{
QTest test(argc,argv);
if(test.start_test()<0)
      ACE_DEBUG((LM_ERROR,"Program failure \n"));
}
```

上面这个例子和前面的例子很相似，只是在其中增加了动态消息队列。在Args类中，我们增加了选项来使用ACE_Message_Queue_Factory创建所有不同类型的消息队列。此外，在Qtest类中增加了两个新方法，用以在每个消息块创建时设置它们的最终期限和执行时间（set_deadline()和set_execution_time()）。这两个方法使用了ACE_Message_Block的msg_execution_time()和msg_deadline_time()方法。注意它们采用的是绝对时间**而非**相对时间，这也是它们和ACE_OS::gettimeofday()方法一起使用的原因。

最终期限和执行时间通过time参数的帮助来设置。最终期限是这样来设置的：第一个消息将拥有最后的最终期限，在最终期限消息队列的情形中，它应该最后被调度。但是在使用松弛度队列时，执行时间和最终期限都将被考虑在内。