第5章 任务和主动对象（Active Object）：并发编程模式

这一章介绍前面提到过的ACE_Task类，另外还介绍了主动对象模式。基本上这一章将涵盖两个主题。首先，它将讲述怎样将ACE_Task构造作为高级面向对象机制使用，用以编写多线程程序。其次，它将讨论怎样在主动对象模式[II]中使用ACE_Task。

5.1 主动对象

那么到底什么是主动对象呢？传统上，所有的对象都是被动的代码段，对象中的代码是在对它发出方法调用的线程中执行的。也就是，调用线程（calling threads）被“借出”，以执行被动对象的方法。

而主动对象却不一样。这些对象持有它们自己的线程（甚或多个线程），并将这个线程用于执行对它们的任何方法的调用。因而，如果你想象一个传统对象，在里面封装了一个线程（或多个线程），你就得到了一个主动对象。

例如，设想对象“A”已在你的程序的main()函数中被实例化。当你的程序启动时，OS创建一个线程，以从main()函数开始执行。如果你调用对象A的任何方法，该线程将“流过”那个方法，并执行其中的代码。一旦执行完成，该线程返回调用该方法的点并继续它的执行。但是，如果”A”是主动对象，事情就不是这样了。在这种情况下，主线程不会被主动对象借用。相反，当”A”的方法被调用时，方法的执行发生在主动对象持有的线程中。另一种思考方法：如果调用的是被动对象的方法（常规对象），调用会阻塞（同步的）；而另一方面，如果调用的是主动对象的方法，调用不会阻塞（异步的）。

5.2 ACE_Task

ACE_Task是ACE中的任务或主动对象“处理结构”的基类。在ACE中使用了此类来实现主动对象模式。所有希望成为“主动对象”的对象都必须从此类派生。你也可以把ACE_TASK看作是更高级的、更为面向对象的线程类。

当我们在前一章中使用ACE_Thread包装时，你一定已经注意到了一些“不好”之处。那一章中的大多数程序都被分解为函数、而不是对象。这是因为ACE_Thread包装需要一个全局函数名、或是静态方法作为参数。随后该函数（静态方法）就被用作所派生的线程的“启动点”。这自然就使得程序员要为每个线程写一个函数。如我们已经看到的，这可能会导致非面向对象的程序分解。

相反，ACE_Task处理的是对象，因而在构造OO程序时更便于思考。因此，在大多数情况下，当你需要构建多线程程序时，较好的选择是使用ACE_Task的子类。这样做有若干好处。首要的是刚刚所提到的，这可以产生更好的OO软件。其次，你不必操心你的线程入口是否是静态的，因为ACE_Task的入口是一个常规的成员函数。而且，我们会看到ACE_Task还包括了一种用于与其他任务进行通信的易于使用的机制。

重申刚才所说的，ACE_Task可用作：

- 更高级的线程（我们称之为任务）。

- 主动对象模式中的主动对象。

5.2.1 任务的结构

ACE_Task的结构在本质上与基于Actor的系统[III]中的“Actor”的结构相类似。该结构如下所示：

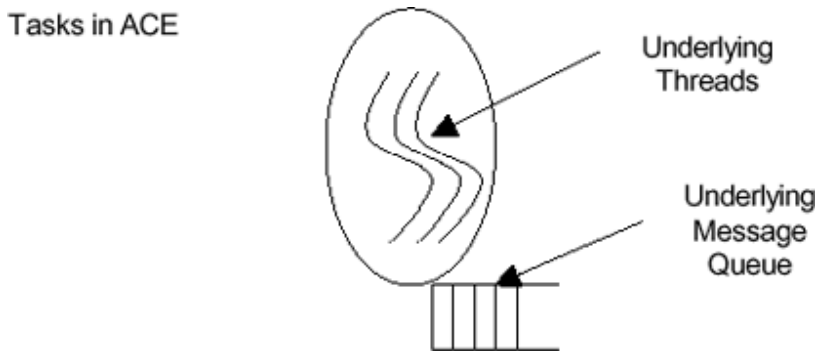

图5-1 任务结构示意图

图5-1说明每个任务都含有一或多个线程，以及一个底层消息队列。各个任务通过这些消息队列进行通信。但是，消息队列并非是程序员需要关注的对象。发送任务可以使用putq()调用来将消息插入到另一任务的消息队列中。随后接收任务就可以通过使用getq()调用来从它自己的消息队列里将消息提取出来。

因而，你可以设想一个系统，由多个自治的任务（或主动对象）构成，这些任务通过它们的消息队列相互通信。这样的体系结构有助于大大简化多线程程序的编程模型。

5.2.2 创建和使用任务

如上面所提到的，要创建任务或主动对象，你必须从ACE_Task类派生子类。在子类派生之后，必须采取以下步骤：

- **实现服务初始化和终止方法：**open()方法应该包含所有专属于任务的初始化代码。其中可能包括诸如连接控制块、锁和内存这样的资源。close()方法是相应的终止方法。
- **调用启用（Activation）方法：**在主动对象实例化后，你必须通过调用activate()启用它。要在主动对象中创建的线程的数目，以及其他一些参数，被传递给activate()方法。activate()方法会使svc()方法成为所有它生成的线程的启动点。
- **实现服务专有的处理方法：**如上面所提到的，在主动对象被启用后，各个新线程在svc()方法中启动。应用开发者必须在子类中定义此方法。

下面的例子演示怎样去创建任务：

例5-1

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

class TaskOne: public ACE_Task<ACE_MT_SYNCH>
{
public:
//Implement the Service Initialization and Termination methods
int open(void*)
{
      ACE_DEBUG((LM_DEBUG,"(%t) Active Object opened \n"));

      //Activate the object with a thread in it.
      activate();

      return 0;
}


int close(u_long)
{
      ACE_DEBUG((LM_DEBUG, "(%t) Active Object being closed down \n"));
      return 0;
}

int svc(void)
{
      ACE_DEBUG((LM_DEBUG,
            "(%t) This is being done in a separate thread \n"));

      // do thread specific work here
      //.......
      //.......
      return 0;
}
};

int main(int argc, char *argv[])
{
//Create the task
TaskOne *one=new TaskOne;
```

```
//Start up the task
one->open(0);

//wait for all the tasks to exit
ACE_Thread_Manager::instance()->wait();
ACE_DEBUG((LM_DEBUG,"(%t) Main Task ends \n"));
}
```

上面的例子演示怎样把ACE_Task当作更高级的线程来使用。在例子中，TaskOne类派生自ACE_Task，并实现了open()、close()和svc()方法。在此任务对象实例化后，程序就调用它的open()方法。该方法依次调用activate()方法，致使一个新线程被派生和启动。该线程的入口是svc()方法。主线程等待主动对象线程终止，然后就退出进程。

5.2.3 任务间通信

如前面所提到的，ACE中的每个任务都有一个底层消息队列（参见上面的图示）。这个消息队列被用作任务间通信的一种方法。当一个任务想要与另一任务“谈话”时，它创建一个消息，并将此消息放入它想要与之谈话的任务的消息队列。接收任务通常用getq()从消息队列里获取消息。如果队列中没有数据可用，它就进入休眠状态。如果有其他任务将消息插入它的队列，它就会苏醒过来，从队列中拾取数据并处理它。因而，在这种情况下，接收任务将从发送任务那里接收消息，并以应用特定的方式作出反馈。

下一个例子演示两个任务怎样使用它们的底层消息队列进行通信。这个例子包含了经典的生产者 - 消费者问题的实现。生产者任务生成数据，将它发送给消费者任务。消费者任务随后消费这个数据。使用ACE_Task构造，我们可将生产者和消费者看作是不同的ACE_Task类型的对象。这两种任务使用底层消息队列进行通信。

例5-2

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

//The Consumer Task.
class Consumer:
public ACE_Task<ACE_MT_SYNCH>
{
public:
int open(void*)
{
      ACE_DEBUG((LM_DEBUG, "(%t) Producer task opened \n"));
```

```
      //Activate the Task
      activate(THR_NEW_LWP,1);

      return 0;
}

//The Service Processing routine
int svc(void)
{
      //Get ready to receive message from Producer
      ACE_Message_Block * mb =0;
      do
      {
            mb=0;

            //Get message from underlying queue
            getq(mb);
            ACE_DEBUG((LM_DEBUG,
                        "(%t)Got message: %d from remote task\n",*mb-
                        >rd_ptr()));
      }while(*mb->rd_ptr()<10);

      return 0;
}

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

class Producer:
public ACE_Task<ACE_MT_SYNCH>
{
public:
Producer(Consumer * consumer):
      consumer_(consumer), data_(0)
{
      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;
}

//The Service Processing routine
int svc(void)
{
      while(data_<11)
      {
            //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;
}

int close(u_long)
{
      ACE_DEBUG((LM_DEBUG,"Producer closes down \n"));
      return 0;
}

private:
char data_;
Consumer * consumer_;
ACE_Message_Block * mb_;
};

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

```
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();

}
```

在此例中，生产者和消费者任务非常相似。它们都没有任何服务初始化或是终止代码。但两个类的svc()方法是不同的。生产者在open()方法中被启用后，svc()方法会被调用。在此方法中，生产者生成一个消息，将它插入消费者的队列。消息是使用ACE_Message_Block类来生成的（要更多地了解如何使用ACE_Message_Block，请阅读此教程及在线ACE指南中有关消息队列的章节）。生产者维护指向消费者任务（对象）的指针。它通过该指针来将消息放入消费者的消息队列中。该指针在main()函数中通过生产者的构造器设置。

消费者驻留在它的svc()方法的循环中，等待数据到达它的消息队列。如果队列中没有数据，消费者就会阻塞并休眠（这是由ACE_Task类魔术般地自动完成的）。一旦数据到达消费者的队列，它就会苏醒并消费此数据。

在此例中，生产者发送的数据由一个整数组成。生产者每次将这个整数加一，然后发送给消费者。

如你所看到的，生产者 - 消费者问题的解决方案十分简单，并且是面向对象的。在编写面向对象的多线程程序时，使用ACE_Task是比使用低级线程API更好的方法。

5.3 主动对象模式（Active Object Pattern）

主动对象模式用于降低方法执行和方法调用之间的耦合。该模式描述了另外一种更为透明的任务间通信方法。

该模式使用ACE_Task类作为主动对象。在这个对象上调用方法时，它就像是常规对象一样。就是说，方法调用是通过同样的->操作符来完成的，其不同在于这些方法的**执行**发生于封装在ACE_Task中的线程内。在使用被动或主动对象进行编程时，客户程序看不到什么区别，或仅仅是很小的区别。对于构架开发者来说，这是非常有用的，因为开发者需要使构架客户与构架的内部结构屏蔽开来。这样构架**用户**就不必去担心线程、同步、会合点（rendezvous），等等。

5.3.1 主动对象模式工作原理

主动对象模式是ACE实现的较为复杂的模式中的一个。该模式有如下参与者：

1. 主动对象（基于ACE_Task）。
2. ACE_Activation_Queue。
3. 若干ACE_Method_Object（主动对象的每个方法都需要有一个方法对象）。
4. 若干ACE_Future对象（每个要返回结果的方法都需要这样一个对象）。

我们已经看到，ACE_Task是怎样创建和封装线程的。要使ACE_Task成为主动对象，需要完成一些额外的工作：

必须为所有要从客户异步调用的方法编写方法对象。每个方法对象都派生自ACE_Method_Object，并会实现它的call()方法。每个方法对象还维护上下文信息（比如执行方法所需的参数，以及用于获取返回值的ACE_Future对象。这些值作为私有属性维护）。你可以把方法对象看作是方法调用的“罩子”（closure）。客户发出方法调用，使得相应的方法对象被实例化，并被放入启用队列（activation queue）中。方法对象是**命令**（Command）模式的一种形式（参见有关设计模式的参考文献）。

ACE_Activation_Queue是一个队列，方法对象在等待执行时被放入其中。因而启用队列中含有所有等待调用的方法（以方法对象的形式）。封装在ACE_Task中的线程保持阻塞，等待任何方法对象被放入启用队列。一旦有方法对象被放入，任务就将该方法对象取出，并调用它的call()方法。call()方法应该随即调用该方法在ACE_Task中的相应实现。在方法实现返回后，call()方法在ACE_Future对象中设置（set()）所获得的结果。

客户使用ACE_Future对象获取它在主动对象上发出的任何异步操作的结果。一旦客户发出异步调用，立即就会返回一个ACE_Future对象。于是客户就可以在任何它喜欢的时候去尝试从“期货”（future）对象中获取结果。如果客户试图在结果被设置之前从期货对象中提取结果，客户将会阻塞。如果客户不希望阻塞，它可以通过使用ready()调用来轮询（poll）期货对象。如果结果已被设置，该方法返回1；否则就返回0。ACE_Future对象基于“多态期货”（polymorphic futures）的概念。

call()方法的实现应该将返回的ACE_Future对象的内部值设置为从调用实际的方法实现所获得的结果（这个实际的方法实现在ACE_Task中编写）。

下面的例子演示主动对象模式是怎样实现的。在此例中，主动对象是一个“Logger”（日志记录器）对象。Logger使用慢速的I/O系统来记录发送给它的消息。因为此I/O系统很慢，我们不希望主应用任务的执行因为相对来说并非紧急的日志记录而减慢。为了防止此情况的发生，并且允许程序员像发出普通的方法调用那样发出日志调用，我们使用了主动对象模式。

Logger类的声明如下所示：

例5-3a

```
//The worker thread with which the client will interact

class Logger: public ACE_Task<ACE_MT_SYNCH>

{

public:

//Initialization and termination methods

Logger();

virtual ~Logger(void);

virtual int open (void *);
```

```
virtual int close (u_long flags = 0);

//The entry point for all threads created in the Logger
virtual int svc (void);

//////////////////////////////////////////////////////
//Methods which can be invoked by client asynchronously.
//////////////////////////////////////////////////////

//Log message
ACE_Future<u_long> logMsg(const char* msg);

//Return the name of the Task
ACE_Future<const char*> name (void);

//////////////////////////////////////////////////////
//Actual implementation methods for the Logger
//////////////////////////////////////////////////////
u_long logMsg_i(const char *msg);
const char * name_i();

private:
char *name_;
ACE_Activation_Queue activation_queue_;
};
```

如我们所看到的，Logger主动对象派生自ACE_Task，并含有一个ACE_Activation_Queue。Logger支持两个异步方法：logMsg()和name()。这两个方法应该这样来实现：当客户调用它们时，它们实例化相应的方法对象类型，并将它放入任务的私有启用队列。这两个方法的实际实现（也就是“真正地”完成所需工作的方法）是logMsg_i()和name_i()。

下面的代码段显示我们所需的两个方法对象的接口，分别针对Logger主动对象中的两个异步方法。

例5-3b

```
//Method Object which implements the logMsg() method of the active
//Logger active object class
class logMsg_MO: public ACE_Method_Object
{
public:
//Constructor which is passed a reference to the active object, the
```

```
//parameters for the method, and a reference to the future which
//contains the result.
logMsg_MO(Logger * logger, const char * msg,
      ACE_Future<u_long> &future_result);
virtual ~logMsg_MO();

//The call() method will be called by the Logger Active Object
//class, once this method object is dequeued from the activation
//queue. This is implemented so that it does two things. First it
//must execute the actual implementation method (which is specified
//in the Logger class. Second, it must set the result it obtains from
//that call in the future object that it has returned to the client.
//Note that the method object always keeps a reference to the same
//future object that it returned to the client so that it can set the
//result value in it.
virtual int call (void);

private:
Logger * logger_;
const char* msg_;
ACE_Future<u_long> future_result_;
};

//Method Object which implements the name() method of the active Logger
//active object class
class name_MO: public ACE_Method_Object
{
public:
//Constructor which is passed a reference to the active object, the
//parameters for the method, and a reference to the future which
//contains the result.
name_MO(Logger * logger, ACE_Future<const char*> &future_result);
virtual ~name_MO();

//The call() method will be called by the Logger Active Object
//class, once this method object is dequeued from the activation
//queue. This is implemented so that it does two things. First it
//must execute the actual implementation method (which is specified
//in the Logger class. Second, it must set the result it obtains from
//that call in the future object that it has returned to the client.
//Note that the method object always keeps a reference to the same
```

```
//future object that it returned to the client so that it can set the

//result value in it.

virtual int call (void);


private:

Logger * logger_;

ACE_Future<const char*> future_result_;

};
```

每个方法对象都有一个构造器，用于为方法调用创建“罩子”（closure）。这意味着构造器通过将调用的参数和返回值作为方法对象中的私有成员数据记录下来，来确保它们被此对象“记住”。调用方法包含的代码将对在Logger主动对象中定义的实际方法实现（也就是，logMsg_i()和name_i()）进行委托。

下面的代码段含有两个方法对象的实现：

例5-3c

```
//Implementation for the logMsg_MO method object.

//Constructor

logMsg_MO::logMsg_MO(Logger * logger, const char * msg, ACE_Future<u_long>

&future_result)

      :logger_(logger), msg_(msg), future_result_(future_result)

{

ACE_DEBUG((LM_DEBUG, "(%t) logMsg invoked \n"));

}


//Destructor

logMsg_MO::~logMsg_MO()

{

ACE_DEBUG ((LM_DEBUG, "(%t) logMsg object deleted.\n"));

}


//Invoke the logMsg() method

int logMsg_MO::call (void)

{

return this->future_result_.set (

      this->logger_->logMsg_i (this->msg_));

}


//Implementation for the name_MO method object.

//Constructor
```

```
name_MO::name_MO(Logger * logger, ACE_Future<const char*> &future_result):
logger_(logger), future_result_(future_result)
{
ACE_DEBUG((LM_DEBUG, "(%t) name() invoked \n"));
}

//Destructor
name_MO::~name_MO()
{
ACE_DEBUG ((LM_DEBUG, "(%t) name object deleted.\n"));
}

//Invoke the name() method
int name_MO::call (void)
{
return this->future_result_.set (this->logger_->name_i ());
}
```

这两个方法对象的实现是相当直接的。如上面所解释的，方法对象的构造器负责创建“罩子”（捕捉输入参数和结果）。call()方法调用实际的方法实现，随后通过使用ACE_Future::set()方法来在期货对象中设置值。

下面的代码段显示Logger主动对象自己的实现。大多数代码都在svc()方法中。程序在这个方法中从启用队列里取出方法对象，并调用它们的call()方法。

例5-3d

```
//Constructor for the Logger
Logger::Logger()
{
this->name_= new char[sizeof("Worker")];
ACE_OS:strcpy(name_,"Worker");
}

//Destructor
Logger::~Logger(void)
{
delete this->name_;
}

//The open method where the active object is activated
int Logger::open (void *)
```

```
{
ACE_DEBUG ((LM_DEBUG, "(%t) Logger %s open\n", this->name_));
return this->activate (THR_NEW_LWP);
}

//Called then the Logger task is destroyed.
int Logger::close (u_long flags = 0)
{
ACE_DEBUG((LM_DEBUG, "Closing Logger \n"));
return 0;
}

//The svc() method is the starting point for the thread created in the
//Logger active object. The thread created will run in an infinite loop
//waiting for method objects to be enqueued on the private activation
//queue. Once a method object is inserted onto the activation queue the
//thread wakes up, dequeues the method object and then invokes the
//call() method on the method object it just dequeued. If there are no
//method objects on the activation queue, the task blocks and falls
//asleep.
int Logger::svc (void)
{
while(1)
{
      // Dequeue the next method object (we use an auto pointer in
      // case an exception is thrown in the <call>).
      auto_ptr<ACE_Method_Object> mo
      (this->activation_queue_.dequeue ());
      ACE_DEBUG ((LM_DEBUG, "(%t) calling method object\n"));

      // Call it.
      if (mo->call () == -1)
            break;

      // Destructor automatically deletes it.
}

return 0;
}

//////////////////////////////////////////////////////////////
```

```
//Methods which are invoked by client and execute asynchronously.
//////////////////////////////////////////////////////////////
//Log this message
ACE_Future<u_long> Logger::logMsg(const char* msg)
{
ACE_Future<u_long> resultant_future;

//Create and enqueue method object onto the activation queue
this->activation_queue_.enqueue
      (new logMsg_MO(this,msg,resultant_future));

return resultant_future;
}

//Return the name of the Task
ACE_Future<const char*> Logger::name (void)
{
ACE_Future<const char*> resultant_future;

//Create and enqueue onto the activation queue
this->activation_queue_.enqueue
      (new name_MO(this, resultant_future));

return resultant_future;
}

//////////////////////////////////////////////////////
//Actual implementation methods for the Logger
//////////////////////////////////////////////////////
u_long Logger::logMsg_i(const char *msg)
{
ACE_DEBUG((LM_DEBUG,"Logged: %s\n",msg));

//Go to sleep for a while to simulate slow I/O device
ACE_OS::sleep(2);

return 10;
}

const char * Logger::name_i()
{
```

```
//Go to sleep for a while to simulate slow I/O device
ACE_OS::sleep(2);
return name_;
}
```

最后的代码段演示应用代码，它实例化Logger主动对象，并用它来进行日志记录：

例5-3e

```
//Client or application code.
int main (int, char *[])
{
//Create a new instance of the Logger task
Logger *logger = new Logger;

//The Futures or IOUs for the calls that are made to the logger.
ACE_Future<u_long> logresult;
ACE_Future<const char *> name;

//Activate the logger
logger->open(0);

//Log a few messages on the logger
for (size_t i = 0; i < n_loops; i++)
{
      char *msg= new char[50];
      ACE_DEBUG ((LM_DEBUG,
                        Issuing a non-blocking logging call\n"));
      ACE_OS::sprintf(msg, "This is iteration %d", i);
      logresult= logger->logMsg(msg);

      //Don’t use the log result here as it isn't that important...
}

ACE_DEBUG((LM_DEBUG,
      "(%t)Invoked all the log calls \
      and can now continue with other work \n"));

//Do some work over here...
// ...
```

```
// ...

//Find out the name of the logging task
name = logger->name ();

//Check to "see" if the result of the name() call is available
if(name.ready())
      ACE_DEBUG((LM_DEBUG,"Name is ready! \n"));
else
      ACE_DEBUG((LM_DEBUG,
            "Blocking till I get the result of that call \n"));

//obtain the underlying result from the future object.
const char* task_name;
name.get(task_name);
ACE_DEBUG ((LM_DEBUG,
      "(%t)==> The name of the task is: %s\n\n\n", task_name));

//Wait for all threads to exit.
ACE_Thread_Manager::instance()->wait();
}
```

客户代码在Logger主动对象上发出若干非阻塞式异步调用。注意这些调用看起来就像是在针对常规被动对象一样。事实上，调用是在另一个单独的线程控制里执行。在发出调用记录多个消息后，客户发出调用来确定任务的名字。该调用返回一个期货给客户。于是客户就开始使用ready()方法去检查结果是否已在期货对象中设置。然后它使用get()方法去确定期货的底层值。注意客户的代码是何等的优雅，没有用到线程、同步，等等。所以，主动对象模式可以使你更加容易地编写你的客户代码。