第3章 ACE的内存管理

ACE构架含有一组非常丰富的内存管理类。这些类使得你能够很容易和有效地管理动态内存（从堆中申请的内存）和共享内存（在进程间共享的内存）。你可以使用若干不同的方案来管理内存。你需要决定何种方案最适合你正在开发的应用，然后采用恰当的ACE类来实现此方案。

ACE含有两组不同的类用于内存管理。

第一组是那些基于ACE_Allocator的类。这组类使用动态绑定和策略模式来提供灵活性和可扩展性。它们只能用于局部的动态内存分配。

第二组类基于ACE_Malloc模板类。这组类使用C++模板和*外部多态性*（External Polymorphism）来为内存分配机制提供灵活性。在这组类中的类不仅包括了用于局部动态内存管理的类，也包括了管理进程间共享内存的类。这些共享内存类使用底层OS（OS）共享内存接口。

为什么使用一组类而不是另外一组呢？这是由在性能和灵活性之间所作的权衡决定的。因为实际的分配器对象可以在运行时改变，ACE_Allocator类更为灵活。这是通过动态绑定（这在C++里需要使用虚函数）来完成的，因此，这样的灵活性并非不需要代价。虚函数调用带来的间接性使得这一方案成了更为昂贵的选择。

另一方面，ACE_Malloc类有着更好的性能。在编译时，malloc类通过它将要使用的内存分配器进行配置。这样的编译时配置被称为“外部多态性”。基于ACE_Malloc的分配器不能在运行时进行配置。尽管ACE_Malloc效率更高，它不像ACE_Allocator那样灵活。

3.1 分配器(Allocator)

分配器用于在ACE中提供一种动态内存管理机制。在ACE中有若干使用不同策略的分配器可用。这些不同策略提供相同的功能，但是具有不同的特性。例如，在实时系统中，应用可能必须从OS那里预先分配所有它将要用到的动态内存，然后在内部对分配和释放进行控制。这样，分配和释放例程的性能就是高度可预测的。

所有的分配器都支持ACE_Allocator接口，因此无论是在运行时还是在编译时，它们都可以很容易地相互替换。这也正是灵活性之所在。所以，ACE_Allocator可以与策略模式协同使用，以提供非常灵活的内存管理。表3-1给出了对ACE中可用的各种分配器的简要描述。这些描述规定了每种分配器所用的内存分配策略。

分配器	**描述**
ACE_Allocator	ACE中的分配器类的接口类。这些类使用继承和动态绑定来提供灵活性。
ACE_Static_Allocator	该分配器管理固定大小的内存。每当收到分配内存的请求时，它就移动内部指针、以返回内存chunk（“大块”）。它还假定内存一旦被分配，就再也不会被释放。

ACE_Cached_Allocator	该分配器预先分配内存池，其中含有特定数目和大小的内存chunk。这些chunk在内部空闲表（free list）中进行维护，并在收到内存请求（malloc()）时被返回。当应用调用free()时，chunk被归还到内部空闲表、而不是OS中。
ACE_New_Allocator	为C++ new和delete操作符提供包装的分配器，也就是，它在内部使用new和delete操作符，以满足动态内存请求。

表3-1 ACE中的分配器

3.1.1 使用缓存式分配器（Cached Allocator）

ACE_Cached_Allocator预先分配内存，然后使用它自己内部的机制来管理此内存。这样的预分配发生在类的构造器中。所以，如果你使用此分配器，你的内存管理方案仅仅在开始时使用OS分配接口来完成预分配。在那以后，ACE_Cached_Allocator将照管所有的内存分配和释放。

为什么要这样做呢？答案是性能和可预测性。设想一个必须高度可预测的实时系统：通过OS来分配内存将涉及昂贵和不可预测的OS内核调用；相反，ACE_Cached_Allocator不会涉及这样的调用。每一次分配和释放都将花费固定数量的时间。

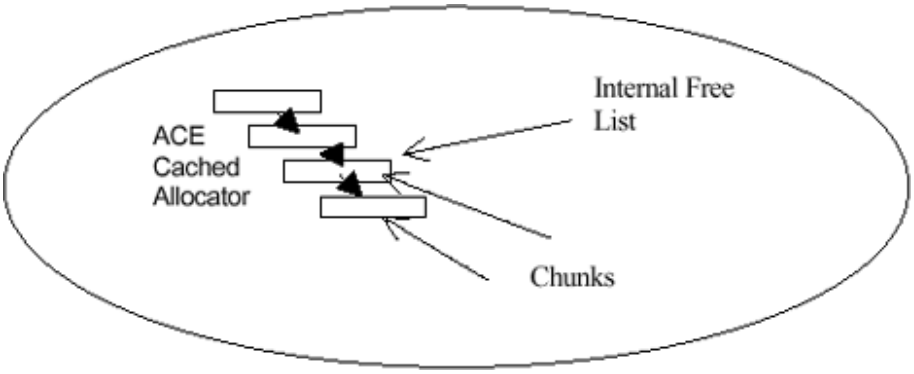

图3-1 缓存式分配器

图3-1演示缓存式分配器。在构造器中预分配的内存在空闲表中进行内部管理。该表将若干内存chunk作为它的节点。这些chunk可以是任何复杂的数据类型，你可以按你所希望的那样规定它们的实际类型。后面的例子会说明怎样去做。

在此系统中分配和释放涉及固定数量的空闲表指针操作。当用户请求内存chunk时，他将获得一个指针，而空闲表被调整。当用户释放内存时，它将回到空闲表中。如此循环往复，直到ACE_Cached_Allocator被销毁，所有的内存随之被归还给OS。在内存被用于实时系统时，需要考虑chunk的内部碎片。

下面的例子演示ACE_Cached_Allocator是怎样被用于预分配内存，然后处理内存请求的。

例3-1

```
#include "ace/Malloc.h"
//A chunk of size 1K is created. In our case we decided to use a simple array
//as the type for the chunk. Instead of this we could use any struct or class
//that we think is appropriate.
typedef char MEMORY_BLOCK[1024];



//Create an ACE_Cached_Allocator which is passed in the type of the
//"chunk" that it must pre-allocate and assign on the free list.
// Since the Cached_Allocator is a template class we can pretty much
//pass it ANY type we think is appropriate to be a memory block.
typedef ACE_Cached_Allocator<MEMORY_BLOCK,ACE_SYNCH_MUTEX> Allocator;
class MessageManager
{
public:
//The constructor is passed the number of chunks that the allocator
//should pre-allocate and maintain on its free list.
MessageManager(int n_blocks):
      allocator_(n_blocks),message_count_(0)
{
      mesg_array_=new char*[n_blocks];
}

//Allocate memory for a message using the Allocator. Remember the message
//in an array and then increase the message count of valid messages
//on the message array.
void allocate_msg(const char *msg)
{
      mesg_array_[message_count_]=allocator_.malloc(ACE_OS::strlen(msg)+1);
      ACE_OS::strcpy(mesg_array_[message_count_],msg);
      message_count_++;
}

//Free all the memory that was allocated. This will cause the chunks
//to be returned to the allocator's internal free list
//and NOT to the OS.
void free_all_msg()
```

```
{
      for(int i=0;i<message_count_;i++)
            allocator_.free(mesg_array_[i]);

      message_count_=0;
}

//Just show all the currently allocated messages in the message array.
void display_all_msg()
{
      for(int i=0;i<message_count_;i++)
            ACE_OS::printf("%s\n",mesg_array_[i]);
}

private:
char **mesg_array_;
Allocator allocator_;
int message_count_;
};

int main(int argc, char* argv[])
{
if(argc<2)
{
      ACE_DEBUG((LM_DEBUG, "Usage: %s <Number of blocks>\n", argv[0]));
      exit(1);
}

int n_blocks=ACE_OS::atoi(argv[1]);

//Instantiate the Memory Manager class and pass in the number of blocks
//you want on the internal free list.
MessageManager mm(n_blocks);

//Use the Memory Manager class to assign messages and free them.
//Run this in your favorite debug environment and you will notice that the
//amount of memory your program uses after Memory Manager has been
//instantiated remains the same. That means the Cached Allocator
//controls or manages all the memory for the application.
//Do forever.
while(1)
```

```
{
      //allocate the messages somewhere
      ACE_DEBUG((LM_DEBUG,"\n\n\nAllocating Messages\n"));
      for(int i=0; i<n_blocks;i++){
            ACE_OS::sprintf(message,"Message %d: Hi There",i);
            mm.allocate_msg(message);
      }

      //show the messages
      ACE_DEBUG((LM_DEBUG,"Displaying the messages\n"));
      ACE_OS::sleep(2);
      mm.display_all_msg();

      //free up the memory for the messages.
      ACE_DEBUG((LM_DEBUG,"Releasing Messages\n"));
      ACE_OS::sleep(2);
      mm.free_all_msg();
}

return 0;
}
```

这个简单的例子包含了一个消息管理器，它对缓存式分配器进行实例化。该分配器随即用于无休止地分配、显示和释放消息。但是，该应用的内存使用并没有变化。你可以通过你喜欢的调试工具来检查这一点。

3.2 ACE_Malloc

如前面所提到的，Malloc类集使用模板类ACE_Malloc来提供内存管理。如图3-2所示，ACE_Malloc模板需要两个参数（一个是内存池，一个是池锁），以产生我们的分配器类。

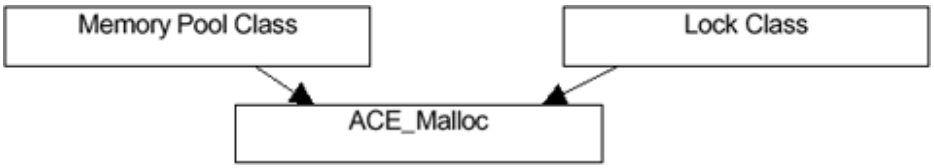

图3-2 ACE_Malloc模板参数示意图

3.2.1 ACE_Malloc工作原理

ACE_Malloc从传入的内存池中“获取”内存，应用随即通过ACE_Malloc类接口来分配（malloc()）内存。由底层内存池返回的内存又在ACE的“chunk”（大块）中被返回给ACE_Malloc类。ACE_Malloc类使用这些内存chunk来给应用开发者分配较小的内存block（块）。如图3-3所示：

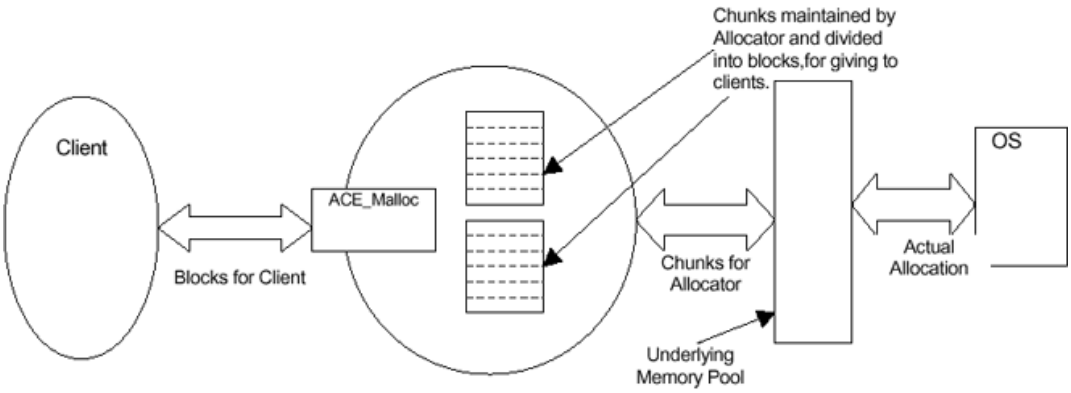

图3-3 ACE_Malloc的工作原理

当应用请求内存block时，ACE_Malloc类会检查在它从内存池中获取的chunk中，是否有足够的空间来分配所需的block。如果未能发现有足够空间的chunk，它就会要求底层内存池返回一个更大的chunk，以满足应用对内存block的请求。当应用发出free()调用时，ACE_Malloc不会把所释放的内存返还给内存池，而是由它自己的空闲表进行管理。当ACE_Malloc收到后续的内存请求时，它会使用空闲表来查找可返回的空block。因而，在使用ACE_Malloc时，如果只发出简单的malloc()和free()调用，从OS分配的内存数量将只会增加，不会减少。ACE_Malloc类还含有一个remove()方法，用于发出请求给内存池，将内存返还给OS。该方法还将锁也返还给OS。

3.2.2 使用ACE_Malloc

ACE_Malloc类的使用很简单。首先，用你选择的内存池和锁定机制实例化ACE_Malloc，以创建分配器类。随后用该分配器类实例化一个对象，这也就是你的应用将要使用的分配器。当你实例化分配器对象时，传给构造器的第一个参数是一个字符串，它是你想要分配器对象使用的底层内存池的“名字”。将正确的名字传递给构造器**非常**重要，特别是如果你在使用共享内存的话。否则，分配器将会为你创建一个新的内存池。如果你在使用共享内存池，这当然不是你想要的，因为你根本没有获得共享。

为了方便底层内存池的共享（重复一次，如果你在使用共享内存的话，这是很重要的），ACE_Malloc类还拥有一个映射（map）类型接口：可被给每个被分配的内存block一个名字，从而使它们可以很容易地被在内存池中查找的另一个进程找到。该接口含有bind()和find()调用。bind()调用用于给由malloc()调用返回给ACE_Malloc的block命名。find()调用，如你可能想到的那样，用于查找与某个名字相关联的内存。

在实例化ACE_Malloc模板类时，有若干不同的内存池类可用（如表3-2所示）。这些类不仅可用于分配在进程内使用的内存，也可以用于分配在进程间共享的内存池。这也使得为何ACE_Malloc模板需要通过锁定机制来实例化显得更清楚了。当多个进程访问共享内存池时，该锁保证它们不会因此而崩溃。注意即使是多个线程在使用分配器，也同样需要提供锁定机制。

表3-2列出了各种可用的内存池：

池名	宏	描述
ACE_MMAP_Memory_Pool	ACE_MMAP_MEMORY_POOL	使用<mmap(2)>创建内存池。这样内存就可在进程间共享了。每次更新时，内存都被更新到后备存储(backing store)。
ACE_Lite_MMAP_Memory_Pool	ACE_LITE_MMAP_MEMORY_POOL	使用<mmap(2)>创建内存池。不像前面的映射，它不做后备存储更新。代价是较低可靠性。
ACE_Sbrk_Memory_Pool	ACE_SBRK_MEMORY_POOL	使用<sbrk(2)>调用创建内存池。
ACE_Shared_Memory_Pool	ACE_SHARED_MEMORY_POOL	使用系统V<shmget(2)>调用创建内存池。
Memory_Pool		内存可在进程间共享。
ACE_Local_Memory_Pool	ACE_LOCAL_MEMORY_POOL	通过C++的new和delete操作符创建局部内存池。该池不能在进程间共享。

表3-2 可用的内存池

下面的例子通过共享内存池使用ACE_Malloc类（该例使用ACE_SHARED_MEMORY_POOL来做演示，但表3-2中的任何支持内存共享的内存池都可以被使用）。

该例创建服务器进程，该进程创建内存池，再从池中分配内存。然后服务器使用从池中分配的内存来创建它想要客户进程“拾取”的消息。其次，它将名字*绑定*（bind）到这些消息，以使客户能使用相应的find操作来查找服务器插入池中的消息。

客户在开始运行后创建它自己的分配器，但是使用的是**同一个**内存池。这是通过将同一个**名字**传送到分配器的构造器来完成的，然后客户使用find()调用来查找服务器插入的消息，并将它们打印给用户看。

例3-2

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

#include "ace/Malloc.h"

#include "ace/Malloc_T.h"

#define DATA_SIZE 100
```

```
#define MESSAGE1 "Hiya over there client process"
#define MESSAGE2 "Did you hear me the first time?"
LPCTSTR poolname="My_Pool";

typedef ACE_Malloc<ACE_SHARED_MEMORY_POOL,ACE_Null_Mutex> Malloc_Allocator;

static void server (void)
{
//Create the memory allocator passing it the shared memory
//pool that you want to use
Malloc_Allocator shm_allocator(poolname);

//Create a message, allocate memory for it and bind it with
//a name so that the client can the find it in the memory
//pool
char* Message1=(char*)shm_allocator.malloc(strlen(MESSAGE1));
ACE_OS::strcpy(Message1,MESSAGE1);
shm_allocator.bind("FirstMessage",Message1);
ACE_DEBUG((LM_DEBUG,"<<%s\n",Message1));

//How about a second message
char* Message2=(char*)shm_allocator.malloc(strlen(MESSAGE2));
ACE_OS::strcpy(Message2,MESSAGE2);
shm_allocator.bind("SecondMessage",Message2);
ACE_DEBUG((LM_DEBUG,"<<%s\n",Message2));

//Set the Server to go to sleep for a while so that the client has
//a chance to do its stuff
ACE_DEBUG((LM_DEBUG,
     "Server done writing.. going to sleep zzz..\n\n\n"));
ACE_OS::sleep(2);

//Get rid of all resources allocated by the server. In other
//words get rid of the shared memory pool that had been
//previously allocated
shm_allocator.remove();
}

static void client(void)
{
//Create a memory allocator. Be sure that the client passes
```

```
// in the "right" name here so that both the client and the
//server use the same memory pool. We wouldn’t want them to

// BOTH create different underlying pools.
Malloc_Allocator shm_allocator(poolname);

//Get that first message. Notice that the find is looking up the
//memory based on the "name" that was bound to it by the server.
void *Message1;
if(shm_allocator.find("FirstMessage",Message1)==-1)
{
      ACE_ERROR((LM_ERROR,
            "Client: Problem cant find data that server has sent\n"));
      ACE_OS::exit(1);
}

ACE_OS::printf(">>%s\n",(char*) Message1);
ACE_OS::fflush(stdout);

//Lets get that second message now.
void *Message2;
if(shm_allocator.find("SecondMessage",Message2)==-1)
{
      ACE_ERROR((LM_ERROR,
            "Client: Problem cant find data that server has sent\n"));
      ACE_OS::exit(1);
}

ACE_OS::printf(">>%s\n",(char*)Message2);
ACE_OS::fflush(stdout);
ACE_DEBUG((LM_DEBUG,"Client done reading! BYE NOW\n"));
ACE_OS::fflush(stdout);
}

int main (int, char *[])
{
switch (ACE_OS::fork ())
{
case -1:
      ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "fork"), 1);
```

```
case 0:
      // Make sure the server starts up first.
      ACE_OS::sleep (1);
      client ();
      break;

default:
      server ();
      break;
}

return 0;
}
```

3.2.3 通过分配器接口使用Malloc类

大多数ACE中的容器类都可以接受分配器对象作为参数，以用于容器内的内存管理。因为某些内存分配方案只能用于ACE_Malloc类集，ACE含有一个适配器模板类ACE_Allocator_Adapter，它将ACE_Malloc类适配到ACE_Allocator接口。也就是说，在实例化这个模板之后创建的新类可用于替换任何ACE_Allocator。例如：

```
typedef ACE_Allocator_Adapter<ACE_Malloc<ACE_SHARED_MEMORY_POOL,ACE_Null_Mutex>> Allocator;
```

这个新创建的Allocator类可用在任何需要分配器接口的地方，但它使用的却是采用ACE_Shared_Memory_Pool的ACE_Malloc的底层功能。这样该适配器就将Malloc类“适配”到了分配器（Allocator）类。

这样的适配允许我们使用与ACE_Malloc类集相关联的功能，同时具有ACE_Allocator的动态绑定灵活性。但重要的是要记住，这样的灵活性是以牺牲部分性能为代价的。