基于 ACE 的共享内存的开发与研究

Development and Research of Share Memory Based on ACE

蓝炳雄 张 丽 （中央财经大学 100081）

摘要：ACE(Adaptive Communication Environment)是可以自由使用、开放源码的面向对象(OO)框架，ACE 提供了一组包括信号处理、进程间通信、共享内存管理、并发执行和同步等组件，本文主要介绍如何通过 ACE 的共享内存管理类实现进程间的通信。

关键词：ACE 内存管理 共享内存

1 ACE 简介

ACE(Adaptive Communication Environment)是可自由使用、开放源码的面向对象框架。框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。

1.1 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。

1.2 ACE_Malloc

Malloc 类集使用模板类 ACE_Malloc 来提供内存管理。如图 1 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。

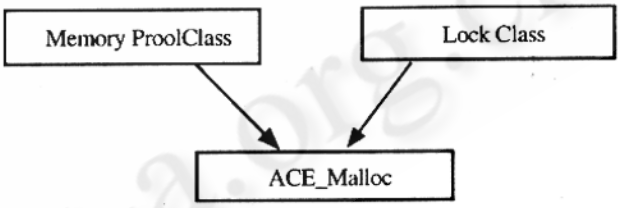

图 1 ACE_Malloc 模板参数示意图

(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 2 所示。

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。

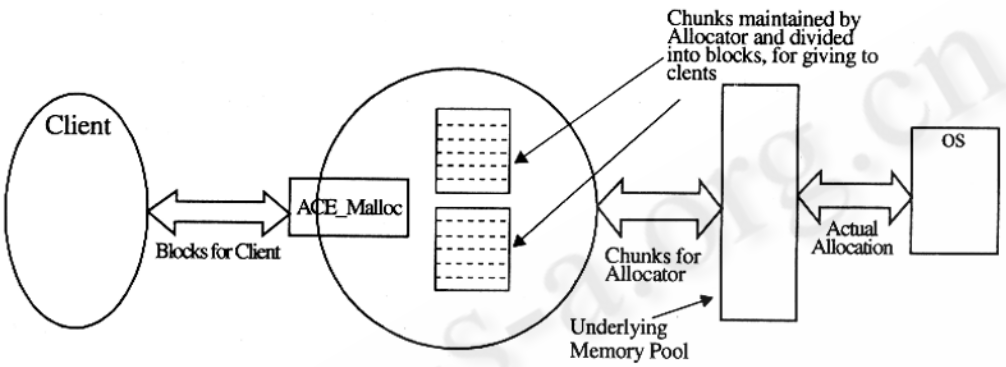

图 2　ACE_Malloc 的工作原理

（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 1 列出了各种可用的内存池。

（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。

2　基于 ACE 的共享内存开发的简单例子

下面是利用分配器接口使用 Malloc 类的实现共享内存的例子，由于篇幅有限，有些变量说明已省略。

头文件：memMgr. h

```
#ifndef _MEMMGR_H_
#define _MEMMGR_H_
……
#define POOL_NAME " mem_pool"  //内存池名
typedef ACE_Allocator_Adapter < ACE_Malloc_T < ACE_
```

```
MMAP_MEMORY_POOL,
    ACE_Null_Mutex,ACE_Control_Block > > Malloc_
Allocator; //通过分配器接口使用 Malloc 类
```

表 1 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。
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。

```
    char bindName[MAX_STR_LEN];//保存内存的绑
定名
    void * pbindData; // 指向由 Malloc 分配绑定在
bindName 的指针
};
class CMemMgr
{
public:
    CMemMgr(char * poolname);
    ~CMemMgr();
    int FindData(char * name,void * &point);//查找
内存
    void ReleaseData(void * &pPoint);//释放内存
    int BindData(char * name,void * &point);//绑定
内存
    void * CallocMem(size_t n_elem,size_t elem_
size);//分配内存
private:
    Malloc_Allocator shm; //共享内存池的说明
    CMgrData * pMgrhead; // 指向内存管理的头指
针;
};
#endif
```

类实现文件:memMgr.cpp

```
CMemMgr::CMemMgr(char * poolname):shm(pool-
name)//定义共享内存池
{
    char strname[80];
    ACE_OS::memset(strname,'\0',sizeof(str-
name));
    ACE_OS::sprintf(strname," Mgrhead% s",pool-
name);
    if( shm.find(strname,(void * )pMgrhead) = =
-1)
    {
      pMgrhead = NULL;
      pMgrhead = (CMgrData * )shm.calloc(MAX_
NODE_NBR,sizeof(CMgrData));
      shm.bind(strname,pMgrhead);
    }
}
CMemMgr::~CMemMgr()
{
    shm.remove();
}
int CMemMgr::FindData(char * name,void * &point)
{
    int nRet;
    nRet = shm.find(name,point);
    return nRet;
}
int CMemMgr::BindData(char * name,void * &point)
{
    int nCount;
```

```
.....
    ACE_OS::strcpy(pMgrhead[nCount].bindName,
name);
    pMgrhead[nCount].pbindData = point;
    return shm.bind(name,point);
}
void *CMemMgr::CallocMem(size_t n_elem,size_t
elem_size)
{
    return shm.calloc(n_elem,elem_size);
}
void CMemMgr::ReleaseData(void *&pPoint)
{
    for(int i=0;i<MAX_NODE_NBR;i++)
    {
      if(pMgrhead[i].pbindData==pPoint)
      {
      shm.unbind(pMgrhead[i].bindName);
        shm.free(pMgrhead[i].pbindData);
        break;
      }
    }
}
```

主文件 1:svr.cpp

```
...
#include "memMgr.h"
main()
{
    char *pName;
    CMemMgr mgr("my_pool"); //定义名为 my_
pool 的共享内存池
    pName = mgr.CallocMem(1,1024);//分配一块
1024 字节的内存
    strcpy(pName,"This is example");//给内存赋值
    mgr.BindData("name",pName);//把内存绑定
到名字 name 上
    printf("set name: %sto my_pool\n",pName);
}
```

主文件 2:cli.cpp

```
...
#include "memMgr.h"
main()
{
    char *pName;
    CMemMgr mgr("my_pool"); 定义名为 my_pool
的共享内存池
    mgr.FindData("name",pName);查找名为 name
的内存
    printf("find the name is : %s\n",pName);
}
```

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。

3 总结

ACE 可用于 Win32,以及 Unix 的大多数版本(如 SunOS4.x,HPUX 等)和 POSIX 系统(VxWorks 和 MVS OpenEdition)。ACE 提供了 C++和 Java 两个实现版本。在 ACE 组件的帮助下,很容易在一种 OS 平台上编写并发网络应用,然后快速地将它们移植到各种其他的 OS 平台上。而且,因为 ACE 是开放源码的自由软件,你无需担心被锁定在特定的操作系统平台或编译器上。ACE 的设计使用了许多可提高软件质量的关键模式,这些质量因素包括通信软件灵活性、可扩展性、可复用性和模块性。

参考文献

1 Douglas. C. Schmidt, Stephen. D. Huston C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns 2001.10.

2 Douglas. C. Schmidt, Stephen. D. Huston C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks 2002.10.