目录

ACE 的陷阱

坦白说，使用这个标题无非是希望能够吸引你的眼球,这篇文章的目的仅仅是为了揭示一些 ACE 缺陷的。文章适合的读者是对 ACE（ADAPTIVE Communication Environment）有一定研究，或者正在使用 ACE 从事项目开发的人士参考。如果你对 C++还是新手，甚至包括 ACE 知识初学者，（但你想飞的更高），建议你收藏这篇文档以后阅读。

秉承陷阱系列文章的传统，我只是通过一些辩证的角度去看 ACE 的一些不足，对于 ACE 的强大和优美我就不再作赞美。从 2000 年，到现在，ACE 在中国已经从星星之火，开始有燎原之势。这一方面说明 ACE 的优美和实力已经逐步得到大家的认可（我所知道的 Adobe reader 的使用 ACE，估计是为了跨平台，国内的大量电信的网管，计费，智能网软件也使用 ACE），一方面要感谢的是的马维达这位国内少有的职业作家，国内的 ACE 的中文资料（包括大量免费资料）都出自这位老兄。

但 ACE 无疑是复杂的,能够畅快的遨游在其中的绝对不是泛泛之辈。没有对网络，设计模式，操作系统有一定的底蕴，想痛快的驾驭 ACE 无疑是较难的。另外，由于 ACE 仍然处在逐步发展的过程中。他的很多问题仍然有待进一步完善。重要的是一些文案的不足，受众面狭小，导致许多 ACE 的使用者在使用 ACE 的时候会碰上很多问题。这篇文案就是用于彻底揭示部分这些问题。希望大家能在更加顺捷的使用它。

另外，请注意我使用的陷阱这个术语，而不是原罪。（C Trap and Pitfalls 倒有很多应该是 Original sin）ACE 还在不停的发展中。很多问题可能会在以后的版本中间改进。所以在我认为的的确是问题的章节后面，我会附上知道错误的版本号。

1 我将什么列为陷阱

1.1 低效的模块

作为一个代码级的中间件。ACE 无疑是高效的，但是坦白说 ACE 的代码不是非常完美的。ACE 的很多地方提供的是一个框架解决方案，为了保证框架的可移植和通用，代码中大量使用了 virtual 函数，Bridge 模式，多线程下的锁操作，甚至有相当的 new 操作……，这些东西都限制 ACE 的性能。所以个人谨慎的将 ACE 的效率定义为中上。

个人认为，一般情况下，如果你使用 ACE 的 API 代替系统 API，速度应该降低 0.01%以下，主要导致这些差役在于 ACE 的再次封装，而函数栈的调用成本应该可以几乎不计。ACE 的优势在高性能的系统架构，而不是绝对的函数性能，如果你要再考虑在加入系统框架的其它功能呢，（举一个例子，当你想把定时器优美的合入你的代码时），ACE 就有足够的优势让你选择他。【注】

在此啰嗦一句，同样也有很多人质疑 STL 的性能。所有好的类库一样，他带来优势的同时也会有一定的遗憾，比如少量性能降低。但是如果说他们的性能不好，那是无稽之谈。（不信，把你认为性能差的代码给我写写看。）建议固步自封的程序员不要再干买椟还珠的事情，先去读读那些优美的代码。

但是和所有的框架一样，ACE 也有不少的地方的地方是性能的暗礁，你最好绕开。当然一般而言 ACE 会提供多条道路，重要的是你能选择正确。

1.2 设计缺陷

ACE 的有多个层次，侧记缺陷这类错误往往出现在 ACE 的高阶封装中。同时由于 ACE 是一个跨平台的中间件。所以为了平台的兼容性，ACE 做了很多折中和弥补，有些是很漂亮的，但有些却不是非常理想。

1.3 使用不便的地方

所有的代码都是不完美的，特别是 ACE 这种要让无数人在无数环境下使用的软件。很多使用不便的问题都是来自我个人的一些习惯，这些算是苛责了。

1.4 容易误解或者误用的地方

由于 ACE 的庞大性，很多时候大家会错误的理解使用 ACE 的某些代码实现某些特性。在此将写一些曾经让我们栽跟头的阴沟写出来。另一方面，ACE 的文档的某些介绍也存在含混，会误导大家的理解，错误的地方。

2 ACE 的链接 Link 错误

很多人在 Windows 使用 ACE 的时候往往会出现以下的 Link 错误。

Why do I get errors while using 'TryEnterCriticalSection'?

/ace/OS.i(2384) : error C2039:

'TryEnterCriticalSection': is not a member of ''`global namespace''

其实这个错误不是由于 ACE 导致的，只是编译器把这个赃栽倒了 ACE 上。出现这个错误的原因主要是因为一些关键宏定义冲突，一般是_WIN32_WINNT，'TryEnterCriticalSection' 这个函数是 NT4.0 后才出现的函数，如果这个宏被定义的小于 0x0400 或者没有定义，那么就会出现这个错误。

所以最简单的处理方法是在自己的预定义头文件中加入一行。

```
#if !defined (_WIN32_WINNT)

# define _WIN32_WINNT 0x0400
```

```
#endif
```

其实 ACE 自己对于宏的处理是比较严谨的，ACE 的 config-win32-common.h 中间就有这行定义，所以在一般而言，可以将 ACE 的头文件包含定义放在在顶部，这样也可以避免这个编译错误。

预定义头文件是一个良好的编程习惯，你可以将自己的大部分宏定义，include 包含的本工程以外的外部.h 文件。简言之就是预定义头文件中使用#include<>，表示包含工程以外文件，自己工程内部只使用#include””，表示包含当前工程目录下的文件。大部分 C/C++的程序员都有过链接和一些预定义冲突错误消耗大量的时间，原来我也是如此，但是在掌握预定义头文件方法后，我几乎没有为这个问题折磨过。其实 Virsual C++ 在生产 MFC 工程的时候，会自动帮你自动生产一个预定义头文件 stdafx.h，只是我们不善利用而已。

其实对于很多编译器，使用预定义头文件还可以加快编译速度。Virusal C++的预定义会生产一个 pch 文件，基本可以提高编译速度一倍。Virusal C++的工程中间有专门的预定义头文件设置。C++ Builder 采用可以采用的编译宏（好像是专用的）加快编译速度。大致的原理是编译器会在对预定义头文件中包含的文件进行与处理，在外部文件没有发生改动的时候，编译器可以使用编译这些文件生成的中间文件加快编译速度。

3　不要使用 ACE_Timer_Hash

ACE 有一个非常优美的定时器队列模型，他提供了 4 种定时器 Queue 让大家使用：ACE_Timer_Heap，ACE_Timer_Wheel，ACE_High_Res_Timer，ACE_Timer_Hash。在《C++ Network Programming Volume 2 - Systematic Reuse with ACE and Frameworks》中间有相应的说明，其中按照说明最诱人的的是：

> ACE_Timer_Hash, which uses a hash table to manage the queue. Like the timing wheel implementation, the average-case time required to schedule, cancel, and expire timers is O(1) and its worst-case is O(n).

但是遗憾的是，ACE_Timer_Hash 其实是性能最差的。几乎不值得使用。我曾经也被诱惑过，但是在测试中间发现，文档中所述根本不属实，在一个大规模定时

器的程序中，我使用 ACE_Timer_Hash 发现性能非常不理想，检查后发现 ACE 的源代码如下：

```
template <class TYPE, class FUNCTOR, class ACE_LOCK, class BUCKET> int

ACE_Timer_Hash_T<TYPE, FUNCTOR, ACE_LOCK, BUCKET>::expire (const ACE_Time_Value &cur_time)

{

  // table_size_为 Hash 的桶尺寸，如果要避免冲突，桶的数量应该尽量大，

    //每个桶可以理解为一个 Hash 开链的链表

  // Go through the table and expire anything that can be expired

  //遍历所有的桶

  for (size_t i = 0;

       i < this->table_size_;

       ++i)

    {

      //在每个桶中检查是否有要进行超时处理的元素

      while (!this->table_[i]->is_empty ()

             && this->table_[i]->earliest_time () <= cur_time)

        {

          .....
```

简单说明一下上面的代码，ACE_Timer_Hash_T 采用开链的 Hash 方式，每个桶就是一个链表，在超时检查时所有的桶中是由有要进行超时处理的元素。所以在超时处理中 ACE 采用了遍历所有元素的方法。但悖论是如果你希望 Hash 的冲突不

大，你就必须将桶的个数调整的尽量多。我在测试中将上述的程序的 Time_Queue 替换为标准的的 ACE_Timer_Heap，发现性能提高数百倍。

冷静下来思考一下，这也是正常的。对于一个 Hash 的实现，保证查询的速度，也就是通过定时器 ID 进行操作的速度是足够快的。但是实际上对于定时器操作，最大的成本应该是寻找要超时的定时器，对于 Hash 这种数据结构，只能采用迭代遍历的方式……， 所以采用 Hash 的低效是正常的。而原文应该改为 schedule, cancel, 的最好时间复杂度是 O(1)，最差是 O(n)，而 expire 的时间复杂度始终是 O(n)。

这个问题在 ACE 自己的文档《Design, Performance, and Optimization of Timer Strategies for Real-time ORBs》中间也有较为正确的描述。

这个问题至少倒 5.6.1 的版本还是存在的。我个人估计也不会得到解决。Hash 的特性摆在那儿呢，除非 ACE 采用更加复杂的数据结构。

4 Reactor 定时器的精度取决于实现

由于 Reactor 在各个平台的默认实现都取决于平台的实现，比如在 Windows 下默认的 Reactor 是 WFMO_REACTOR，而在 Linux 和 UNIX 平台，默认的 Reactor 是 Select_Reactor, 而 Reactor 的实现往往取决于使用的反应器底层实现，而这些反应器的时间精度就决定了你的定时器的时间精度。下表大致反馈了一些常用的定时器的实现。

表 1 常用 Raactor 的实现

Reactor	反应器的底层实现	时间精度
ACE_Select_Reactor	select 函数	使用 struct timeval 结构进行超时处理; timeval 结构可以精确倒微秒。
Dev_Poll_Reactor	poll 或者而 epoll	timeout 参数的单位是毫秒。

ACE_WFMO_REACTOR	WaitForMultipleObjects	dwMilliseconds 的参数单位是毫秒

不过作为服务器的开发，我倒想不出什么地方需要精确到 0.1s 定时器的地方，了解一下差异性就足够了。

5 WFMO_Reactor 的与众不同

WFMO_Reactor 是 ACE_Reactor 在 Windows 下的默认实现（为什么不选择 ACE_Select_Reactor 作为默认实现，可能是基于效率和强大性的考虑）,WFMO_Reactor 的低层使用的函数是 WaitForMultipleObjects 和 WSAEventSelect，WSAEnumNetworkEvents。其中 WaitForMultipleObjects 函数用于处理线程，互斥量，信号灯，事件，定时器等事件，而 WSAEventSelect 用于处理网络 IO 事件。

由于 Windows API 和操作系统的特性不一样，WFMO_Reactor 在很多地方的表现和其他平台不一致。 【注】

【注】其实这两个问题在《C++ Network Programming Volume 2 - Systematic Reuse with ACE and Frameworks》中 4.4 The ACE_WFMO_Reactor Class 有说明。这儿算是借花献佛。

5.1 WFMO_Reactor 只能处理 62 个句柄

由于 WaitForMultipleObjects 不是一个处理大量事件的函数，其最多处理 64 个事件句柄，而 WFMO_Reactor 自身为了处理使用了 2 个句柄，所以一个 WFMO_Rector 对象只能处理。

如果你想做大规模的网络接入，62 个事件句柄显然是不够的，特别是要同时处理 IO 事件时，导致这个不足的应该是 WFMO_Reactor 的设计者的一个选择。在赋予 WFMO_Reactor 强大的特性的同时，WFMO_Reactor 的设计者只能让网络 IO 事件的数量委屈一下了。

5.2 WRITE_MASK 触发机制

WFMO_Reactor 选择的是 Windows 的 WSAEventSelect 函数作为网络的 IO 的反应器。但是 WSAEventSelect 函数的 FD_WRITE 的事件处理和传统的 IO 反应器（select）不同。下面是 MSDN 的描述。

> The FD_WRITE network event is handled slightly differently. An FD_WRITE network event is recorded when a socket is first connected with connect/WSAConnect or accepted with accept/WSAAccept, and then after a send fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD_WRITE network event setting and lasting until a send returns WSAEWOULDBLOCK. After such a failure, the application will find out that sends are again possible when an FD_WRITE network event is recorded and the associated event object is set.

简单翻译就是，只有在三种条件下，WSAEventSelect 才会发出 FD_WRITE 通知，一是使用 connect 或 WSAConnect，一个套接字成功建立连接后；二是使用 accept 或 WSAAccept，套接字被接受以后；三是若 send、WSASend、sendto 或 WSASendTo 函数返回失败，而且错误是 WSAEWOULDBLOCK 错误后，缓冲区的空间再次变得可用时。【注】

【注】这种触发方式在 IO 反应器或者说 IO 多路复用模型中应该被称为边缘触发方式。select 函数好像没有这种触发方式而是水平触发方式， Epoll 是支持这种方式的，但是默认还是水平触发，这种方式可能有更高的效率，但是代码更加难写。

可以这么理解，WSAEventSelect 认为套接字基本都是可写状态，它认为你应该大胆 send。只有 send 出现 WSAEWOULDBLOCK 失败后，你才需要使用 WSAEventSelect 反应器。【注】

所以对于 WFMO_Reactor 的，你不可能依靠注册（或者是唤醒）IO 句柄进行写操作，WMFO_Reactor 很有可能不会去回调你的 handle_output 函数。

【注】对于网络套接字，只要缓冲区还有空间就可以直接发送，除非缓冲区没有空间了，才可能出现阻塞错误，所以直接 send 失败的可能性很小，另外反复调用注册 IO 句柄一类的操作其实是比较耗时的。其实先 send，如果 send 失败再注册 IO 句柄到反应器的方式应该是一种更加高效的方式，高压力的通讯服务器应该选择这个编写方式。

我自己的通信服务器通过这个改造，提高的性能在 15%左右（CPU 占用率下降）。

由于 WFMO_Reactor 的这些特点，其实很大的限制了 Reactor 的可移植性。其实个人感觉如果你对系统特性没有那么多要求，在 Windows 下选择 Select_Reactor 替换 WFMO_Reactor 是更好的选择。

6　尽量使用 ID 取消 ACE_Event_Handler 定时器

ACE 的 Reactor　提供了两种方式取消定时器：

```
virtual int cancel_timer (ACE_Event_Handler *event_handler,
                          int dont_call_handle_close = 1);
virtual int cancel_timer (long timer_id,
                          const void **arg = 0,
                          int dont_call_handle_close = 1);
```

一种是使用定时器 ID 取消定时器，这个 ID 是定时器是的返回值，一种是采用相应的 ACE_Event_Handler 指针取消定时器。一般情况下使用 ACE_Event_Handler 的指针取消定时器无疑是最简单的方法，但是这个方法却不是一个高效的实现。所以如果您的程序有大规模的定时器设置取消操作，建议尽量使用 ID 取消定时器。我们用 ACE_Timer_Heap 和 ACE_Timer_Has 两个 Timer_Queue 剖析一下。

6.1 ACE_Timer_Heap 如何根据 Event_handler 取消

先选择最常用的 Time_Queue ACE_Timer_Heap 举例，其使用 ACE_Event_Handler 关闭定时器的代码是：

```
template <class TYPE, class FUNCTOR, class ACE_LOCK> int
ACE_Timer_Heap_T<TYPE, FUNCTOR, ACE_LOCK>::cancel (const TYPE &type,
                                                   int dont_call)
{
  // Try to locate the ACE_Timer_Node that matches the timer_id.
  //循环比较所有的的 ACE_Event_Handler 的指针是否相同
  for (size_t i = 0; i < this->cur_size_; )
    {
      if (this->heap_[i]->get_type () == type)
        {
          ...................
        }
    }
```

而使用 TIMER_ID 关闭的代码如下，它是通过数组下标进行的定位操作。

```
template <class TYPE, class FUNCTOR, class ACE_LOCK> int
ACE_Timer_Heap_T<TYPE, FUNCTOR, ACE_LOCK>::cancel (long timer_id,
                                                   const void **act,
```

```
                                int dont_call)
{
  //通过数组下标操作，速度当然奇快无比。
  ssize_t timer_node_slot = this->timer_ids_[timer_id];
  ……
  //跟进数组 ID 进行操作
  else
  {
    ACE_Timer_Node_T<TYPE> *temp =
      this->remove (timer_node_slot);
  }
}
```

对于 ACE_Timer_Heap，采用 ACE_Event_Handler 指针取消定时器的方式的平均时间复杂度应该就是 O(N)。由于 ACE 的的一个 Event_handler 可能对应多个定时器，所以必须检查所有的才能确保取消所有的相关定时器。

6.2 ACE_Timer_Hash 如何根据 Event_handler 取消

对于 Timer_Hash，其通过 ACE_Event_Handler 关闭定时器的代码是：

```
template <class TYPE, class FUNCTOR, class ACE_LOCK, class BUCKET> int
ACE_Timer_Hash_T<TYPE, FUNCTOR, ACE_LOCK, BUCKET>::cancel (const TYPE &type,
                                int dont_call)
```

```
{
  Hash_Token<TYPE> **timer_ids = 0;
  //根据 Event Handler 有一个定时器 new 一个数组出来
  ACE_NEW_RETURN (timer_ids,
                  Hash_Token<TYPE> *[this->size_],
                  -1);
  size_t pos = 0;
  //根据定时器的个数再进行取消
  for (i = 0;
       i < this->table_size_;
       ++i)
    {
      ACE_Timer_Queue_Iterator_T<TYPE,
                                 ACE_Timer_Hash_Upcall<TYPE, FUNCTOR, ACE_LOCK>,
                                 ACE_Null_Mutex> &iter =
        this->table_[i]->iter ();
```

可以看到 Timer_Hash 的 cancel 比 ACE_Timer_Heap 的 cancel(Event_Handler)要好一点点。但是其中也有 new 和 delete 操作，这些操作也不是高效操作。

所以说在大规模的定时器使用中，推荐你还是使用定时器的 ID 取消定时器更加高效的多。

7 注意 ACE_Pipe 的实现

ACE_Pipe 是一个跨平台的管道实现。标准情况来讲，采用的实现，但是在最大的两个平台 Windows 和 Linux 上，ACE 的实现是采用的 Socket 实现。

```
int

ACE_Pipe::open (int buffer_size)

{

  ACE_TRACE ("ACE_Pipe::open");

  

#if defined (ACE_LACKS_SOCKETPAIR) || defined (__Lynx__)

  

  //绑定了一个本地端口，0.0.0.0，然后找到相应的端口，用于后面的链接

  if (acceptor.open (local_any) == -1

      || acceptor.get_local_addr (my_addr) == -1)

    result = -1;

  else

    {

      // Establish a connection within the same process.

      if (connector.connect (writer, sv_addr) == -1)

        result = -1;

  ……
```

所以很多管道特性所特有的东西，在这两个平台上是无法使用 ACE_Pipe 实现的。比如，管道的特性可以保证在暂时没有接受者的情况下使用，而 Socket 是不可能有这个特性的。你必须保证先有接受者，后有发送者的时序。

所以在这些平台上最好不用这个封装。

8 慎用 Reactor Notify 机制

在 Reactor 的模式，有一种辅助的通知机制,Notify 机制，简单说就是通过通知发起者调用 notify 函数，notify 的消息被保存在一个管道中，handle_event 的处理中会检查这个管道中是否有通知数据，如果有就根据通知的消息，会根据默认的通知消息的类型去调用 hanle_input 等函数。

从设计的角度将，这个机制无疑是非常优美的，对于 Reactor，它在 IO 驱动以外，提供了一种新的驱动方式。但是从实现角度来讲，这个机制要慎用。原因有两个。

8.1 ACE Reactor 的默认 Notify 方式采用的是 ACE_Pipe

ACE Reactor 的默认 Notify 方式采用的是 ACE_Pipe，所以 ACE_Pipe 在 Windows 和 Linux 平台上的问题，Notify 机制把 ACE_Pipe 的缺陷一个不少的继承了，而且问题更加多。

```
/**
* Contains the ACE_HANDLE the ACE_Dev_Poll_Reactor is listening
* on, as well as the ACE_HANDLE that threads wanting the attention
* of the ACE_Dev_Poll_Reactor will write to.
*/
ACE_Pipe notification_pipe_;
```

原来在调试 ACE 代码的时候，我发现只要一使用 Reactor，即使只使用定时器（除非明确不使用 Notify），防火墙都会报警有监听端口。我曾经对此大惑不解，直到读了 ACE 的这部分原代码。这样做的坏处有很多。第一个是由于采用的阻塞 IO。速度会慢很多，第二个由于是单线程的处理，如果在压力极大的情况下，可能出现死锁的问题。比如在有大规模的 Notify 的情况下，发送缓冲区很可能会被塞满（由于是单线程，这时不会有接受者），同时由于为了简化，ACE_Pipe 采用的 IO 是阻塞的，所以会导致整个程序死锁。第三就是这样的情况下 ACE_Pipe 会打开一个临时的端口，而且会绑定所有的 IP（0.0.0.0），如果对于一个安全要求严格的的场景，这个将是一个不可饶恕的错误。【注】

【注】在一个安全要求严格的环境下，这个临时端口轻则可以让你的服务器轻易陷于崩溃，重则可以让你整个网络被黑客攻陷。

不过还好的是 ACE 的开发者估计自己也意识倒了这个麻烦。所以提供了另外一种消息队列的方式。你可以通过定义 ACE_HAS_REACTOR_NOTIFICATION_QUEUE 的宏编译 ACE，这样 ACE 将不使用 ACE_Pipe 作为 Notify 消息的管道，而使用一个自己的内存队列保存 Notify 消息，这个队列是动态扩展的。而且由于是内存操作，性能方面没有太大问题。

大体位置在重复编译的卫哨后面，#include /**/ "ace/pre.h"前面。保证这个宏起到作用。

```
#ifndef ACE_CONFIG_LINUX_H

#define ACE_CONFIG_LINUX_H


//使用内存队列作为 Notify Queue

#define ACE_HAS_REACTOR_NOTIFICATION_QUEUE
```

#include /**/ "ace/pre.h"

这个问题到 5.6.1 还是存在的，估计由于历史的原因，在很长一段时间也不会得到解决。

8.2 考虑不周的 Reactor Notify 机制

同上，这也应该是一个 BUG，Reactor Notify 的代码有考虑不周的地方。Notify 机制的本质是提供了一条消息队列让大家有方法调用 Event_handler，但是存在一种可能，在你的通知消息在消息队列的时候，Event_hanlder 由于后面的处理可能已经 handle_close 了。但是 ACE 的 dispatch_notify 却没有考虑倒这一点（或者说考虑倒这一点也不好解决）。

ACE_Select_Reactor_Notify::dispatch_notify 函数的代码。

```
int
ACE_Select_Reactor_Notify::dispatch_notify (ACE_Notification_Buffer &buffer)
{
.............
 ACE_Event_Handler *event_handler =
      buffer.eh_;

    bool const requires_reference_counting =
      event_handler->reference_counting_policy ().value () ==
      ACE_Event_Handler::Reference_Counting_Policy::ENABLED;
      //如果此时这个 ACE_Event_Handler 已经被 handle_close 了，你如何是好。。。。
```

```
    switch (buffer.mask_)
    {
    case ACE_Event_Handler::READ_MASK:
    case ACE_Event_Handler::ACCEPT_MASK:
        result = event_handler->handle_input (ACE_INVALID_HANDLE);
```

这个 bug 到 5.6.1 还没有解决。我觉得这个问题是可以解决的（暂时还没有提 BUG），但是得到解决的方式却仍然是低效的方案（还记得取消定时器的那个缺陷吗）。

如果你仔细看过上面的几节，你也许会发出惊叹，啊，又是 Reactor Notify? 对，又是它。看起来我好像一直在和 ACE 的 Notify 机制在做对，但它的确让我吃了无数的苦头。这部分的设计的确有一点画蛇添足的感觉，而且由于跨平台性等原因，这个东东的实现一直不如意。其实自己使用 ACE 的实现（比如 Message_Queue）一套这样的机制应该是易如反掌的事情。不苛求了。

如果你用不到 Notify 机制，最好在 ACE_Reactor 初始化的时候彻底关闭 Notify 机制。很多 Reactor 的初始化函数都提供了关闭 notify pipe 的方式。比如 ACE_Select_Reactor_T 的 open 函数的 disable_notify_pipe 参数。当其为 1 的时候表示关闭 notify 管道。

```
//disable_notify_pipe 参数为 1 时表示关闭 NOTIFY PIPE，不使用他
template <class ACE_SELECT_REACTOR_TOKEN> int
ACE_Select_Reactor_T<ACE_SELECT_REACTOR_TOKEN>::open
  (size_t size,
   int restart,
   ACE_Sig_Handler *sh,
   ACE_Timer_Queue *tq,
```

```
    int disable_notify_pipe, /* 等于==1 表示关闭 notify 机制 */

    ACE_Reactor_Notify *notify)
```

9 ACE_Dev_Poll_Reactor 的处理优先级严重偏向定时器

不使用 POLL 和 EPOLL【注】的人，估计不太知道这个 ACE_Dev_Poll_Reactor，但实际上。特别是 Linux 下的 EPOLL（一个 IO 多路服用模型），这是 Linux 大规模接入的重要法宝，从目前的表现来看，其他平台上还没有可以超越 EPOLL 的东西，Windows 下的异步 IO 的性能也还远远逊于 EPOLL。

如果要使用 EPOLL 而不是 POLL，要使用宏 ACE_HAS_EVENT_POLL 编译 ACE，大体位置在重复编译的卫哨后面，#include /**/ "ace/pre.h"前面。保证起到作用。

#ifndef ACE_CONFIG_LINUX_H

#define ACE_CONFIG_LINUX_H

// ACE_HAS_EVENT_POLL 宏用于定义使用 EPOLL 模块，同时注意不同 LINUX 平台下编译可能有少量

//不同。我曾经使用过的一个内核 2.4 的 Slackware 平台，要在编译 ACE 的时候加入 –lepoll，可能是由于

//其是打补丁增加的功能

#define ACE_HAS_EVENT_POLL

#include /**/ "ace/pre.h"

但也许是由于这个东西过新还是由于设计者是一个定于时间要求很敏感的人。的设计明显的是定时器优先。但是了解 EPOLL 和 POLL 的人都知道，UNIX 和 Linux 设计这两个咚咚的目的就是解决大规模 IO 复用。不是为了保证定时器优先，所以我对这个设计很是不解，郁闷。其大体思路为，

1.） 先检查定时器超时的队列，计算最小的超时时间，用于 IO 等待。

2.） 触发 IO 事件

3.） 处理超时的 Handler，如果有超时的事件，返回（1）。这点我看得最郁闷。

4.） 再分发处理 IO 事件

可以看到在处理超时句柄的时候，ACE_Dev_Poll_Reactor 发现有超时的事件会返回到检查超时队列。所以如果在 Reactor 同时有定时处理，IO 的优先级会很低。

其实这个的设计者也知道这个问题。他在代码中间做了如下的记录。

```
int

ACE_Dev_Poll_Reactor::dispatch (Token_Guard &guard)

{

……

  // Handle timers early since they may have higher latency

  // constraints than I/O handlers.  Ideally, the order of

  // dispatching should be a strategy...

  if ((result = this->dispatch_timer_handler (guard)) != 0)

    return result;
```

由于 EPOLL 的特性，使用它大部分都是为了处理大规模的 IO 请求，定时器其实只有少量的需求，不是我们需求的重点。

这个问题到最近的 5.6.1 版本没有得到解决。

我曾经反馈过这个问题。但是得到没有明确的解答。解决这个问题的方法其实也很简单，自己重载这个类，然后自己实现相应的函数。触发 IO 事件后立即分发 IO 事件，而且加入了一个 IO 的优先级别。在多次 IO 处理的循环后在进入时间事件处理。保证时间处理的粒度在 1s 以内基本就可以了。

10 Event_Handler 在程序退出前应该自己关闭

在程序退出的【注】，我们往往不会自己关闭 Event_Handler，而寄希望 Reactor 的清理。但是实际情况会复杂很多。使用的时候必须当心。

【注】是否要在退出的时候清理所有分配的内存？在普通的操作系统中，程序的退出会回收所有的分配内存。所以很多人会逃避在最后阶段的清理分配的内存。但是这实在不是一个良好的喜欢。一方面对于很多 OS（比如嵌入系统）不会回收内存资源，一些内核资源（UNIX）也不会在进程退出后释放，编程就应该要养成清理的好习惯，更何况不进行释放在内存检查的软件一般会报错，如果不清理会干扰我们对于内存泄露的定位。

10.1 Reactor 的 close 可能不会关闭 Event_Handler

理论上讲，ACE_Reactor 提供了一个 close 函数，所有的 Event_Handler 应该统一在这个函数进行关闭。

ACE_Reactor 采用的是模式，封装了不同 Reactor 的实现。这些实现的 close 函数未存在一定的差异性。就我的阅读和尝试来看，Select_Reactor 在 close 函数关闭了所有的 IO 句柄相关的 Event_Handler，而 Dev_Poll_Reactor 的 close 实现就没有关闭。

Select_Reactor 的 close 代码。

```
template <class ACE_SELECT_REACTOR_TOKEN> int

ACE_Select_Reactor_T<ACE_SELECT_REACTOR_TOKEN>::close (void)

{

  ……

  //在 handler_rep 的 close 函数会关闭所有的 register 的句柄的 handler，调用他们的

  //handle_close 函数

  this->handler_rep_.close ();
```

Dev_Poll_Reactor 的 close 的调用了函数 ACE_Dev_Poll_Reactor_Handler_Repository::close，而后有逐步调用了 unbind_all，remove_reference。

```
//close 会经过多级调用到 ACE_Dev_Poll_Reactor_Handler_Repository:: unbind_all

//unbind 被 unbind_all 函数调用 decr_refcnt == true

int

ACE_Dev_Poll_Reactor_Handler_Repository::unbind (ACE_HANDLE handle,

                                                 bool decr_refcnt)

{……

  // remove_reference 函数没有调用 handle_close，而是减去了引用计数

   if (decr_refcnt)

   this->handlers_[handle].event_handler->remove_reference ();

……

 }
```

```
ACE_Event_Handler::Reference_Count

ACE_Event_Handler::remove_reference (void)

{

  //如果打开了引用计数，则使用应用计数方式管理方式。但是代码默认不采用应用计数模式

  //所以下面的代码都无法执行

  if (reference_counting_required)

  {

    //减去引用计数

    Reference_Count result =

      --this->reference_count_;

    //如果已经没用引用个数了，删除自己。

    if (result == 0)

      delete this;

}
```

可以看到 ACE_Event_Handler 的代码默认不采用应用计数模式，（eference_counting_required 默认为 DISABLED）而 Dev_Poll_Reactor 却非要使用引用计数模式去清理 Event_Handler。

我对 Dev_Poll_Reactor 为什么要设计成这样表示不解。也对 Dev_Poll_Reactor 提交过 BUG，但是 Dev_Poll_Reactor 的开发者不认为这样有什么不妥，本人 E 文羞涩，无法说服具体的开发人员，不过在提交 BUG 时，居然得到了 Douglas 反馈（他开始时认同我的看法），对于他们的执着和认真还是表示敬仰。

10.2 可能会导致重复释放引发 Coredump

这个问题是在工作中调试一个 BUG 出现的。

在测试一个服务器的时候发现 Coredump 发生 kill 进程，让其退出在之后，会出现 Coredump 文件。Coredump 显示出现问题的地方在。

```
#1  0x0805bc7b in ~ACE_Timer_Heap_T (this=0x82d3ec8) at
/usr/local/ACE_wrappers/ace/Timer_Queue_T.cpp:442

#2  0x0805b86d in ~ACE_Singleton (this=0x82cca70) at egg_application.cpp:52

#3  0x08056785 in ACE_Singleton<EggSvrdAppliction, ACE_Null_Mutex>::cleanup
(this=0x82dfb90)
```

由于希望改变 ACE_Time_Queue 的特性（数量），我替换 Reactor 的默认 Time_Queue，所以必须自己销毁自己管理的 TimeQueue。而在外部最后销毁的时候出现 Coredump。由于和 Time_Queue 相关，我检查了所有的 Timer 相关的 Event_handler，发现有一个 Event_handler 没有自己主动调用 handler_close 释放，这个 Event_handler 只有定时器，没有注册任何 IO 事件。修改代码为主动释放后，再次测试就发现 Coredump 的问题得到解决。

我检查了一下原有代码堆栈的调用顺序，找到了问题原因。

(1)ACE_Reactor::close，实际调用 ACE_Select_Reactor::close

(2) Select_Reactor::close　尝试关闭所有的 IO 句柄相关的 Event_handler，但由于 Time_Queue 是外部传入的参数，所以不清理 Time_Queue。

(3)Time_Queue 清理，Time_Queue 的析构函数被调用，Time_Queue 的析构函数会释放所有的定时器相关的 Event_handler。而他的释放还会调用 hanlder_close。但是这是 Reactor 对象已经销毁了。所以造成了 Coredump。

注意由于 Reactor 的封装了 Event_handler 定时器，IO 句柄，Notify 机制等回调接口。所以 Event_handler 可能只关联到 IO 句柄，也可能只关联定时器，同时 Reactor 的模型决定了他的内部管理是复杂的。而在释放的过程中很可能会发生交错的问题，而，像上面问题的 Event_handler 就只关联的定时器，所以在 Reactor 的 close 的时候没有关闭。从而导致在后面的清理工作中产生时序问题。

最简单的方式还是自己在程序退出前清理释放所有的 Event_handler. 再调用 Reactor 的 close。

11 调整系统时钟导致 ACE 定时器丢失

由于我们采用的服务器一般都是靠纽扣电池作为能源驱动和记录时钟，一般在运行一段时间后都会出现时间误差。所以很多大规模的分布系统都有校时操作，特别是一些对时钟要求精确的分布式系统（比如计费等），往往都会有一个主机提供精确时钟服务（其可能采用 GPS 校时），其他服务器通过这台服务器校时，校时操作一般都是直接改变系统时钟。

ACE 的定时器都是采用 Event_Handler 进行处理，而 Event_Handler 一般而言都是采用绝对时间作为记录超时的时间戳，但是绝对时间的方式在系统时钟被调整的时候，会导致“丢失”部分定时器的处理，导致一些问题。

在设置定时器时，schedule_timer 函数通过 gettimeofday 得到定时器时间点的时间。

```
template <class ACE_SELECT_REACTOR_TOKEN> long

ACE_Select_Reactor_T<ACE_SELECT_REACTOR_TOKEN>::schedule_timer

  (ACE_Event_Handler *handler,

   const void *arg,

   const ACE_Time_Value &delay_time,

   const ACE_Time_Value &interval)
```

```
{
  // schedule_timer 记录的是系统时间,
  if (0 != this->timer_queue_)
    return this->timer_queue_->schedule
      (handler,
       arg,
       timer_queue_->gettimeofday () + delay_time,
       interval);
}
```

在派发定时器的过程中也是调用 gettimeofday 函数。

```
template <class TYPE, class FUNCTOR, class ACE_LOCK> ACE_INLINE int
ACE_Timer_Queue_T<TYPE, FUNCTOR, ACE_LOCK>::expire (void)
{
  if (!this->is_empty ())
    return this->expire (this->gettimeofday () + timer_skew_);
  else
    return 0;
}
```

可以看出，如果在 schedule_timer 后，将系统时钟向前调节（调慢）以后，原有的定时器将要经过更多的时间才能触发。从而导致这段时间内定时器无法触发。从而造成定时器丢失。

这个问题的解决方法有 2 个，简单方法是将系统时钟校准的频度提高，保证每次校准的时候，系统的时钟出现的偏差都不会影响时钟的定时器触发。

另外一种是 ACE 的 Timer_Queue 自己提供的方法，通过上面的代码我们可以发现，其实 ACE_Timer_Queue_T::gettimeofday 是一个调用的是一个函数指针。默认使用 ACE_OS:: gettimeofday 函数，这个函数可以替换的。

```
void gettimeofday (ACE_Time_Value (*gettimeofday)(void));
```

ACE 提供一个依赖于操作系统的高解析定时器，ACE_High_Res_Timer，这个类是通过 OS 的 TICK 数量来得到更加精确的时钟的【注】。

【注】OS 在启动后，都会有一个 TICK 在不断的计数，这个 TICK 就像一个打点计数器，每次增加 1.一般计数周期就是一个 CPU 周期。

由于 CPU 的 TICK 不会随着你调整系统时钟而调整。所以可以看做是一个相对值。ACE_High_Res_Timer 可以根据相对值计算得到非常精确的程序运行时钟，。直接使用 ACE_High_Res_Timer:: gettimeofday_hr 函数作为 ACE_Timer_Queue_T::gettimeofday 函数指针。并且在程序的开始部分使用函数，ACE_High_Res_Timer::global_scale_factor ()，用于激活高精度定时器。【注】

【注】这个方法得益于原来公司的两位同事 zhangtianhu 和 liaobincai 的一个终结。在此怀念一下和他们共事的日子。另外，我没有仔细研究过这个方法，由于获取 CPU 的 TICK 的获取很有可能是一个内核操作，效率可能不高。

采用上述的两个方法基本可以避免这个问题。

12 ACE 的 CDR 中的字节对齐问题

大家应该都知道计算机中间都有字节对齐问题。CPU 访问内存的时候，如果从特定的地址开始访问一般可以加快速度，比如在 32 位机器上，如果一个 32 位的整数被放在能被 32 模除等于 0 的地址上，只需要访问一次，而如果不在，可能要访问两次。但是这样就要求一些数据从特定的地址开始，而不是顺序排放（中间会有一些空余的地址），这就是字节对齐。

而 ACE CDR 的估计也是为了加快速度，从而在 CDR 编码上默认也使用了字节对齐。所以在 ACE 的 CDR 编解码过程中，传入的参数地址最好是能符合字节对齐规则，否则可能会编解码错误。

ACE_OutputCDR 构造函数会调用一个函数 mb_align 调整传入的地址参数成为地址对齐地址。但是其的调整函数 ACE_ptr_align_binary 不知处于什么考虑，不是按照机器的对齐长度而是采用的 ACE_CDR::MAX_ALIGNMENT（64bit，长度为 8BYTPES）作为参数地址。那么 ACE_OutputCDR 的内部地址是按照 8 字节作为对齐的，但是 ACE_InputCDR 却没有将内部地址调整为模除 64 等于 0 的地址上，而只是调整为模除 32（在 32 位机器上）等于 0 的地址。

```
void

ACE_CDR::mb_align (ACE_Message_Block *mb)

{

#if !defined (ACE_CDR_IGNORE_ALIGNMENT)

  //如果使用字节对齐方式，使用最大的对齐方式调整内存。调整为模除 64 等于 0 的地址上。

  char * const start = ACE_ptr_align_binary (mb->base (),

                                             ACE_CDR::MAX_ALIGNMENT);

#else

……

}
```

使用一段简单的代码可以测试发现这个问题。

```
char *tmp_buffer = new char [2048];
//使用一个无法对齐的参数作为 ACE_InputCDR，ACE_OutputCDR 的参数地址，
char *tmp_data = tmp_buffer +1;

// output_cdr 调整了对齐的起始地址为 8 字节的默认
ACE_OutputCDR output_cdr(tmp_data,512);
ACE_InputCDR input_cdr(tmp_data,512);

ACE_CDR::ULong cdr_long = 123;
bool bret =false;
//
bret = output_cdr.write_ulong(cdr_long);
// cdr_long 不等于 123，而是一个错误无效数据。
bret = input_cdr.read_ulong(cdr_long);
```

其实如果编解码的 BUFF 都采用相同的对齐方式，那么理论上也不应该出现问题，最多是出现为了对齐而进行填补的空隙，但是这样能带来 CPU 的效率提升，也是好事。但是由于 ACE_OutputCDR 的一个地址调整。却可能导致编解码的 BUFFER 不一致，我不能肯定这到底是一个错误还是作者有他自己的考虑。

这个问题到 5.6.1 还存在。我已经提交了问题报告。

当然有一个方法解决这个问题。就是定义宏 ACE_CDR_IGNORE_ALIGNMENT【注】，只要定义了这个宏，ACE 就不会使用字节对齐处理 CDR 编码。使用这个方法的，编码占用空间会压缩一些，但效率上可能低一点（其实未必，因为为了字节对齐还要耗费一些计算时间），

【注】ACE 不知道为什么在代码中使用两个不使用字节对齐的宏，一个是在 CDR_Base.h CDR_Base.cpp 文件中使用的是 ACE_CDR_IGNORE_ALIGNMENT，在 CDR_Stream.cpp 和 CDR_Stream.h 文件上使用的宏 ACE_LACKS_CDR_ALIGNMENT。

我一般将两个宏都定义上。

13 尽量使用 STL 而不是 ACE 的容器

这个纯属个人感觉（偏见）。我有如下理由不使用 ACE 的容器：

- 一些实现不符合大家对于容器的认识，比如 ACE_DLList，在其中存放的居然是对象的指针而不是拷贝。你还必须记住去释放 ACE_DLList 内部管理的指针。
- ACE 容器的迭代器不符合 STL 的要求，从而造成 ACE 的容器无法使用 STL 的各种模板算法和函数。总不能因为 ACE 容器失去 STL 算法这片森林吧。
- 现在的编译器上已经非常普遍实现了 STL，想找一个还不支持 STL 的编译器应该都不容易了。
- ACE 的容器中间有大量指针，所以 ACE 的容器也不可能用在共享内存中。其的应用场景和 STL 没有本质区别。

ACE 的文档《The.ACE.Programmers.Guide》中间也说过：

> That being said, the standard C++ containers are recommended for application development when you are using ACE.

所以在可以使用 STL 的情况下，还是优先使用 STL。

14 ACE 的日志的不如意

ACE 的日志部分是一个非常漂亮的实现,在多线程和多进程模型下都能较好的效率和安全使用。但是却又少量的不足，让人意犹未尽。

14.1 无法替换的时间戳格式

ACE 日志对于时间戳的格式是固定的，采用的是格式，这个格式在西方人看起来估计还比较顺眼，在东方人眼中却不如人意。更好的方式当然是时间戳的函数可以重载。或者用函数对象（指针）作为参数传入。

虽然这部分代码可以重载解决这个问题,但是要大动干戈只修正这个问题感觉却又不值得的。

14.2 日志策略的初始化方式别扭

ACE 提供了一个日志策略类 ACE_Logging_Strategy 辅助大家定义日志策略。但是他的初始化参数却是命令行参数，而不是变量参数。

```
int

ACE_Logging_Strategy::init (int argc, ACE_TCHAR *argv[])
```

你必须使用这样的命令行去初始化日志策略模块。

```
-m1024 -N10 -fSTDERR|OSTREAM -s../log/c4ad.log
```

试问有几个服务器的开发人员会将这些日志策略的初始化放到命令行参数上去。

14.3 没有按天（时间）分割日志文件的方式

ACE_Logging_Strategy 的日志文件的分割策略采用的是按照文件大小分割文件，文件的序号采用滚动的，但这种日志分割方式无法根据文件时间了解日志内容，（由于文件序号要滚动，序号文件的最后修改时间都一样），你只能 grep 所有的日志寻找你要的内容。

而在我看来，最好日志分割方式肯定是按照日期进行分割日志文件。每天创建一个新的日志文件，可以方便分割日志。清理和管理的工作量大大降低。

14.4 日志槽的方式

ACE_Logging_Strategy 采用的是日志槽的方式 Enable 或者 Disable 某些级别的日志。但是感觉多少有点不自然的，ACE 自己的日志级别本身就是分级的。个人感觉应该是如果日志输出的日志级别大于定义的级别就能输出应该是一个更好的选择。

解决 ACE_Logging_Strategy 的问题最好的办法还是扩展这个类。实现自己的日志策略类。

15 ACE_Time_Value 的赋值效率

ACE_Time_Value 是使用 ACE 会大量使用类。但是他的部分函数没有高效的实现。比如构造函数：

```
ACE_INLINE

ACE_Time_Value::ACE_Time_Value (time_t sec, suseconds_t usec)
```

和 set 函数

```
ACE_INLINE void

ACE_Time_Value::set (time_t sec, suseconds_t usec)
```

为了规范用户的赋值，在这些函数的最后都会调用 normalize 函数。

```
void ACE_Time_Value::normalize (void)
```

但如果你的赋值的微秒数值不合适（过大）时，normalize 却不是一个高效实现。下面简单摘取 normalize 的一段代码。

```
void
```

```
ACE_Time_Value::normalize (void)

{

 //如果赋值的大于微秒数值大于 1s。

 if (this->tv_.tv_usec >= ACE_ONE_SECOND_IN_USECS)

  {

   /*! /todo This loop needs some optimization.  */

   //作者都认为这个代码要优化

   //那么进入循环，每次减去 1000000 的微秒单位，在秒的单位+1，上帝呀。

      do

   {

    ++this->tv_.tv_sec;

    this->tv_.tv_usec -= ACE_ONE_SECOND_IN_USECS;

   }

   while (this->tv_.tv_usec >= ACE_ONE_SECOND_IN_USECS);

  }

 ……

}
```

很不理解为什么会写成如此的低效。为什么不直接使用除法呢，我很不理解。所以如果你在代码的主循环中如果使用了 ACE_Time_Value，使用上面的那些函数就可能掉入陷阱。

解决方法是尽量使用函数 sec 和 usec 赋值，这些函数不会调用 normalize，这两个函数会直接赋值。如果非要使用上面的那些函数方式，也一定不要使用过大的（错误的）时间参数。

这个问题到 5.6.1 还没有得到修正。

16 非阻塞网络函数封装不一致

ACE 的非阻塞网络函数参数设计有不合理的地方。ACE_SOCK_Stream 和 ACE_SOCK_Connector 在非阻塞的的调用的接口对于 ACE_Time_Value *timeout 参数的使用不一致，一个要使用 NULL，一个却要使用 ACE_Time_Value::zero。

ACE_SOCK_Stream，非阻塞调用 send 函数的时候【注】，timeout 参数必须填写为 NULL。它最后调用的是 ACE::send。将 ACE_Time_Value 填写为 ACE_Time_Value::zero （0,0）是不行的。如果填写 ACE_Time_Value::zero，会大大降低这个非阻塞调用的性能。

```
ssize_t

ACE::send (ACE_HANDLE handle,

        const void *buf,

        size_t n,

        int flags,

        const ACE_Time_Value *timeout)

{

  if (timeout == 0)

    return ACE_OS::send (handle, (const char *) buf, n, flags);

  else

    {
```

```
    ……
  }
}
  timeout);
```

注意使用非阻塞的的 IO 要调用 recv，send 函数，而不要调用 recv_n,send_n 这些函数接口，这些函数接口如果 timeout 参数传递 NULL，表示阻塞。

另外非阻塞 IO 还是要自己设置 Socket 的选项。

但是 ACE_SOCK_Connector 却采用另外一个封装方式，其是传入一个 NULL 表示阻塞，而传入 ACE_Time_Value::zero （0,0）表示进行非阻塞链接操作。

```
  * @param timeout    Pointer to an @c ACE_Time_Value object with amount
  *                   of time to wait to connect. If the pointer is 0
  *                   then the call blocks until the connection attempt
  *                   is complete, whether it succeeds or fails.  If
  *                   *timeout == {0, 0} then the connection is done
  *                   using nonblocking mode.  In this case, if the
  *                   connection can't be made immediately, this method
  *                   returns -1 and errno == EWOULDBLOCK.
  int connect (ACE_SOCK_Stream &new_stream,
               const ACE_Addr &remote_sap,
```

```
        const ACE_Time_Value *timeout = 0,
        const ACE_Addr &local_sap = ACE_Addr::sap_any,
        int reuse_addr = 0,
        int flags = 0,
        int perms = 0,
        int protocol = 0);
```

大家在处理这些 IO 时务必当心。

17 过于前卫的 Makefile 方式

这个”陷阱”的说法有点吹毛求疵，ACE 提供了一种很前卫的 Makefile 方式，他定义了 Makefile 的基础变量，以及包括规则。如果使用他来辅助 Makefile 的书写,特别是在跨平台开发中，你可以大大节省 Makefile 开发时间。

```
BIN   = hello_ace
BUILD = $(VBIN)
SRC = $(addsuffix .cpp,$(BIN))
LIBS = -lMyOtherLib
LDFLAGS = -L$(PROJ_ROOT)/lib
#----------------------------------------------
#Include macros and targets
#----------------------------------------------
include $(ACE_ROOT)/include/makeinclude/wrapper_macros.GNU
```

```
include $(ACE_ROOT)/include/makeinclude/macros.GNU

include $(ACE_ROOT)/include/makeinclude/rules.common.GNU

include $(ACE_ROOT)/include/makeinclude/rules.nonested.GNU

include $(ACE_ROOT)/include/makeinclude/rules.bin.GNU

include $(ACE_ROOT)/include/makeinclude/rules.local.GNU
```

但是麻烦就在于 ACE 的这些 Makefile 方法几乎没有一个文档帮助说明，我一直无法理解$VBIN 到底是什么。这也许，另外，定义到规则这一层也大大限制了大家对 Makefile 的扩展能力。这就有一点点高不成低不就的味道了，Makefile 的新手几乎不可能了解 ACE 的 Makefile,老手又会因为特殊的需求得不到满足而踌躇。而我个人一般只使用 ACE 定义的 Makefile 变量。这些变量大部分在 wrapper_macros.GNU，platform_macros.GNU

表 2 ACE Mafile 的变量定义

变量	描述
AR	ar　命令的名字
ARFLAGS	ar　的参数
CC	C 编译器的命令的
CXX	C++编译器的命令
RC	资源编译器命令的名字
COMPILE.c	编译 C 文件的命令行，　　一般为:$(CC) $(CFLAGS) $(CPPFLAGS) -c
COMPILE.cc	编译 C++文件的命令行,一般为:$(CXX) $(CCFLAGS) $(CPPFLAGS) $(PTDIRS) -c
COMPILEESO.cc	$(CXX) $(CCFLAGS) $(CPPFLAGS) $(PTDIRS),没太搞明白，不知道为什么和 SO 有关，好像是为了修正错误增加的。不理也罢
CPPFLAGS	C,C++语言编译的预标志，比如 DEFINDE 等. CPPFLAGS += $(DEFFLAGS) $(INCLDIRS)
CFLAGS	C 语言编译选项

CCFLAGS	C++语言编译选项
DCFLAGS	Debugging 程序的 C 语言编译选项,一般在有 debug=1 变量时有效
DCCFLAGS	Debugging 程序的 C++语言编译选项,一般在有 debug=1 变量时有效
DEFFLAGS	C++ 预处理的 DEFINE 部分
DLD	dynamic linker 动态库 link 命令的名字,
LD	linker 命令的名字
IDL	CORBA IDL compiler 命令的名字
INCLDIRS	INCLUDE 的头文件
LDFLAGS	ld linker flags
LINK.c	链接 C 文件的命令行
LINK.cc	链接 C++文件的命令行,一般为:$(PURELINK) $(PRELINK) $(LD) $(CCFLAGS) $(CPPFLAGS) $(PTDIRS)
MAKEFLAGS	Flags that are passed into the compilation from the commandline
OCFLAGS	Optimizing 程序的 C 语言编译选项
OCCFLAGS	Optimizing 程序的 C++语言编译选项
PIC	PIC 就是 position independent code
PCFLAGS	profiling 程序的 C 语言编译选项 profiling 是什么不要问我。
PCCFLAGS	profiling 程序的 C++语言编译选项
PRELINK	LINK 之前执行的命令
PURELINK	purify 执行的命令，purify 是什么不要问我。
PWD	得到当前目录的命令

PTDIRS	模板文件的路径定义
RM	删除工具的命令
ACE_MKDIR	递归创建的目录
SOFLAGS	生成.so 库时候的参数
SOLINK.cc	生成.so 库时候的命令行
VAR	Variant identifier suffix
VDIR	Directory for object code .obj/
VSHDIR	Directory for shared object code .shobj/

看起来变量很多，其实要记住和使用的可以很少，你需要留意的主要是.cc 结尾的变量就可以了。我们可以使用 ACE MakreFile 的变量，方便我们的 Makefile 开发。比如：

我的 Makefile，就使用了$(LINK.cc)， $(COMPILE.cc)两个宏。

```
#使用 ACE 的 wrapper_macros.GNU 的定义变量

include $(ACE_ROOT)/include/makeinclude/wrapper_macros.GNU

#得到 C,CPP 文件的列表

SRC_FILE = $(wildcard ./*.cpp )

#通过.C,.CPP 文件名称得到.O 文件名称，.o 文件放在../../object/exampleexe/目录下

O_FILE = $(patsubst ./%.cpp, ../../object/exampleexe/%.o, $(CPP_FILE))
```

```
#输出文件 exe_file

OUTFILE = ../../bin/exampleexe

# LIB_ALL 为 –l 文件和-L 目录的定义

$(OUTFILE): $(O_FILE)

    $(LINK.cc) -o$(OUTFILE) $(O_FILE)  $(LIB_ALL)

#.o 输出文件放在../../object/目录下

../../object/exampleexe/%.o : ./%.cpp

    $(COMPILE.cc) $(INC_ALL) $< -o $@

clean:

    $RM -f $(OUTFILE) $(O_FILE)
```

是不是也很酷，轻松实现 Makefile 的跨越平台移植。

18 共享内存的与位置无关分配？

在文档《ACE Programmer's Guide, The: Practical Design Patterns for Network and Systems Programming》中介绍了一种与位置无关的共享内存分配，但是实际上这种方式并不是太理想。按照文章中的介绍的方式，其实主要是采用 ALWAYS_FIXED 参数，使用制定的基地址作为共享内存的地址。同时使用辅助类保证 2 个进程使用相对地址使用共享内存。

```
    ACE_MMAP_Memory_Pool_Options options

      (ACE_DEFAULT_BASE_ADDR,

       ACE_MMAP_Memory_Pool_Options::ALWAYS_FIXED);

    ACE_NEW_RETURN (g_allocator,
```

```
                ALLOCATOR (BACKING_STORE,

                           BACKING_STORE,

                           &options),

                -1);

    ACE_DEBUG ((LM_DEBUG,

                ACE_TEXT ("Mapped to base address %@/n"),

                g_allocator->base_addr ()));

    

    showRecords ();
```

但是，首先要求大家能使用相同的基地址，按照 ACE 给出的例子。其给出默认基地址一个宏 ACE_DEFAULT_BASE_ADDR(在 Linux 下是 0x80000000)。因为地址空间管理都是操作系统的负责的事情，所以要求使用同一块共享内存的 2 个进程分配的基地址是一样的是很不靠谱的事情。采用这种方式可能有 2 个后果，第一如果你要使用多个共享内存，你要自己计算管理进程空间，第二你程序可移植性很低，甚至会出现在一台机器上可以运行，在另外 1 台机器无法运行。所以大家慎用这个特性比较好。把程序的可靠运行寄托于运气好，这不应该是一个程序员的作风。

所以对于共享内存，如果希望实现与位置无关的分配，我个人的忠告如下：

- 一开始分配足够的空间，不要再进行扩展【注】。因为扩展共享内存可能意味着原来所有的共享内存相关指针会失效。
- 各自进程管理自己的地址空间，共享内存内部不要保存任何指针（特别不要在共享内存内保存指针），所有的地址都使用相对值。这样才能保证重入，和基础地址变化下不出现问题。

《ACE Programmer's Guide, The: Practical Design Patterns for Network and Systems Programming》中间还提出过处理共享内存池封装，但考虑到涉及所有的共享内存地址的都要调整。不是太认可这种方式。

另外由于 ACE 的容器都使用了指针，不建议在共享内存中使用 ACE 的容器。

19 自己初始化 Timer_Queue 的尺寸

如果你的应用有大量的定时器，你最好自己控制 Timer_Queue 的尺寸。原因如下。默认的 ACE 的 Timer_Queue 初始化的尺寸不大，一般只有 44 个。而原有的尺寸不能满足你的要求的时候，Timer_Queue 会自动增长，以 Timer_Heap 为例，增长的方式是扩大一倍空间。在性能要求严格环境下，多次增长队列的尺寸对性能会造成一定的冲击。下面是空间调整函数 grow_heap 的部分代码剖析。

```
template <class TYPE, class FUNCTOR, class ACE_LOCK> void

ACE_Timer_Heap_T<TYPE, FUNCTOR, ACE_LOCK>::grow_heap (void)

{

  //调整为最大尺寸的两倍

  size_t new_size = this->max_size_ * 2;

   ACE_Timer_Node_T<TYPE> **new_heap = 0;

  //NEW 新的空间，将原有的空间的数据拷贝回来。

  ACE_NEW (new_heap,

           ACE_Timer_Node_T<TYPE> *[new_size]);

  ACE_OS::memcpy (new_heap,

                  this->heap_,

                  this->max_size_ * sizeof *new_heap);
```

```
    delete [] this->heap_;

    this->heap_ = new_heap;

    //后面还有多个空间要扩展和调整

    ……

    this->max_size_ = new_size;

}
```

其实这和 std::vector 一样，如果你知道要使用多少空间，先调用 reserve 预分配空间会大大加快后面的执行速度。如果你知道要使用多少个定时器，告知底层，它会帮你提前分配好空间，否则他会采用他认为合理的方式和尺寸。

所以最好的方法是你先估算你大致需要使用的 Timer 数量，在初始化是告诉 Timer_Queue。但是 Reactor 没有办法通过使用参数调整 Time_Queue 的大小，你必须自己进行替换 Time_Queue 来实现目的。方法大致如下：

```
    ACE_Timer_Queue *timer_queue_=NULL;

    //根据自己的需要调整 Time_Queue 的尺寸

    timer_queue_ = new ACE_Timer_Heap(maxaccept + maxconnect + 16);

     ACE_Reactor::instance(new ACE_Reactor(new
ACE_Select_Reactor(NULL,timer_queue_,1),1),1);
```

这样你就替换了 Reactor 的 Timer_Queue，同时你要记住在程序运行退出前自己释放的你申请的 timer_queue_;

20 杂项

这一节列一些 ACE 使用中要注意的一些问题。

20.1 ACE_Reactor 的初始化应尽量提前

由于为了一些自己需要的特性，我一般会自己初始化 ACE_Reactor，而不是让系统默认初始化。要注意必须在程序的最开始就初始化 ACE_Reactor。

由于 ACE 的很多代码都会使用 ACE_Reactor，包括日志的策略类。所以 ACE_Reactor 必须在这些代码前面，否则会出现奇怪的错误，比如无法响应某些 IO，我至少掉到这个陷阱里面 5 次。

20.2 ACE_SOCK_Stream 不会在析构关闭

有 OO 基础的程序都会放资源的释放放入析构中间去。所以我看到 ACE_SOCK_Stream 也以为他的在析构中关闭 Socket 的句柄，但是事实是 ACE_SOCK_Stream 必须自己显式调用 close 函数关闭 Socket 句柄。

当然，这倒不是 ACE 的设计缺陷，而是 ACE 的 ACE_SOCK_Stream 是一个可以出现在堆栈，可以作为参数传递，进行赋值的类，如果在析构中关闭，就无法实现这些功能了。

实现决定设计。辨证呀。

20.3 handle_events 函数的 ACE_Time_Value 参数

Reactor 的 handle_events 参数里面的有一个 ACE_Time_Value 参数，注意这个参数是一个传入传出参数。

```
virtual int handle_events (ACE_Time_Value &max_wait_time);
```

由于 Reactor 内部同时要管理定时器和 IO 句柄，所以 ACE 很可能不能等待你制定的时间长度，所以他会在传出参数告诉你剩余的等待时间。这时你可以让 ACE 继续等待剩余时间。但在主循环处理中，你不能这样做，因为经过多次调用后，ACE_Time_Value 参数会变成 0（ACE_Time_Value::zero）。这是会导致 hanlde_events 空转，会导致 CPU 占用率很高。

对于大部分主循环的程序，都不需要这样做，而应该重新制定一个等待时间。

20.4 正确理解 ACE_Singleton 的加锁

ACE_Singleton 的模板参数是可以带一个锁参数的。

```
template <class TYPE, class ACE_LOCK>

class ACE_Singleton : public ACE_Cleanup
```

但你可能会错误理解这个锁参数的用途。

```
typedef ACE_Singleton<Manager, ACE_Thread_Mutex> MANAGER;

MANAGER::instance()->ProcessFunA();
```

初学者可能会疑惑加锁的是不是 ProcessFunA，的处理被加锁了。但是实际上 ACE_Singleton 的锁只保护 ACE_Singleton 内部的指针分配和销毁不出现重入。也就是保护 instance 函数内部的指针分配和释放部分。代码剖析如下：

```
template <class TYPE, class ACE_LOCK> TYPE *

ACE_Singleton<TYPE, ACE_LOCK>::instance (void)

{

    //加锁部分的代码，使用 GUARD 方式保护 new

    ACE_GUARD_RETURN (ACE_LOCK, ace_mon, *lock, 0);

    if (singleton == 0)

    {

        ACE_NEW_RETURN (singleton, (ACE_Singleton<TYPE, ACE_LOCK>), 0);

    }

  ……

  return &singleton->instance_;

}
```

其实理解函数栈调用的兄弟应该很容易理解这个问题，ProcessFunA 函数入栈的时候 instance 函数已经出栈了。instance 函数内部加（解）的锁无法影响后续的调用。

20.5 ACE_DEBUG 的两层括号

这儿只是分析（猜测）一下 ACE_DEBUG 两层括号的来由。用习惯了 Windows 下面跟踪宏 TRACE 的人开始用 ACE 的调试宏 ACE_DEBUG 的宏都会有点不习惯，因为你必须写两层括号。

```
#if defined (ACE_NLOGGING)

#define ACE_DEBUG(X) do {} while (0)   /*注意 ACE 定义的是(X)*/

#else

#define ACE_DEBUG(X) /

  do { /

    ACE_Log_Msg *ace___ = ACE_Log_Msg::instance (); /

    ace___->log X; /          /*注意这儿，这个奇怪的写法*/

  } while (0)

#endif

//使用实例，

ACE_DEBUG((LM_ERROR,"i=%d./n",i++));
```

比较起来，对于 Windows 下的 TRACE 宏的定义如下：

```
#ifdef _DEBUG

#define TRACE ATLTRACE
```

```
#else

#define TRACE   __noop     /* MSVC 特有的一个标识符，用于忽视后面的参数 */

#endif
```

而 ACE_DEBUG 的定义比 TRACE 的定义是多一层(X)的，所以你必须写两层括号，ACE 实际上将内层括号的内容全部作为宏参数使用了。

我曾经对这两层括号疑惑了很久。因为我觉得可以采用其他方法绕开两个括号，（你可以写一个日志类尝试一下）

```
#if defined (ACE_NLOGGING)

// 直接定义为一个函数的名字，当然这儿还要改写其他的很多代码

#define Z_DEBUG  ACE_Log_Msg::instance ()->log

#else

#define Z_DEBUG

#endif
```

这样的在没有定义 ACE_NLOGGING 的时候，Z_DEBUG(LM_ERROR,"i=%d./n",i++);会被替换成，(LM_ERROR,"i=%d./n",i++)，这样也不会有任何输出效果。

直到有一次发现 GCC2.9 的环境下编译类似代码，GCC 会对这样的代码会产生告警，我大致明白了 ACE_DEBUG 设计者的苦衷。只有双层括号的方法才能彻底让这行代码不起任何告警。

另外使用两层括号也有性能上的好处，大家注意代码被替换成(LM_ERROR,"i=%d./n",i++)后，i++的代码还是要执行，在我自己测试中，即使是在 GCC 的 O3 级别的优化编译中，这样的代码也不会被优化掉。而如果采用 ACE_DEBUG 的设计，统一替换为 do {} while (0)，这行代码则必然将被优化掉。而对于 MSVC 的编译器，他提供一个特别的标识符__noop 帮助编译器优化。

21 总结和如何用好 ACE

21.1 实践，不断尝试

大学毕业生中能成为好的程序员绝对不是纯粹考试得高分死记公式拿奖学金的同学 ，而是那些熬夜写代码的狂人，哈哈。

计算机是一门实践科学，你只有不断尝试才能进步。

21.2 阅读的 ACE 代码

好像是 Linus（虽然他好像有点抵触 C++，哈哈），好像是 Linus Torvalds 在回答一个提问者时说：“请去阅读我的代码”。了解一个实现，发现问题的最好方式还是阅读源代码。代码面前，了无秘密。

当然 ACE 的代码阅读起来不是一件那么舒心的事情。开发者们采用的是一些非常传统的 UNIX 习惯，比如对齐方式采用 2 个空格缩进，单行 if 语句不用{}包含，稍显奇特的 inc 文件方式，另外，为了支持跨平台特性，ACE 的代码用了大量的宏。这都无疑增加了阅读的难度。不过总体说了，ACE 的代码比较起 Linux 内核代码和很多其他类库的代码还是好的多，至少注释很清晰，而且 Doxgen 生产的文档很酷，也够用。

21.3 了解操作系统和平台特性

由于 ACE 是一个跨平台实现。如果你了解平台的实现。不光你阅读代码的速度会快很多，也会让你对实现的困惑就会越少，让你的代码避开效率的陷阱，你的实现就会越高效。

21.4 好好学习 C++

不需要 OO 的封装，不用美妙的设计模式，没有对效率的执着追求，没有惊艳的范化设计，用 C++干什么？但没有这些信仰，也就不会有 ACE，而且没有这些信仰要程序员做什么？

21.5 慎用高阶特性

在 ACE 的使用过程中，发现 ACE 的主要问题出在一些高阶实现上。所以如果你要使用高阶特性最好能了解背后的实现。

21.6 为 ACE 作出贡献

多用 ACE，将发现的问题反馈给 ACE 的开发者和 ACE 社区。

22 后记

22.1 作者介绍

笔名：雁渡寒潭(insailer@gmail.com)

曾星　　腾讯公司互动娱乐后台开发程序员，目前从事游戏后台设计开发

个人兴趣范围：大规模分布系统的架构设计，高容量，大压力的服务器设计；跨平台开发；数据库的设计，原理和调优；多核（CPU）环境下的程序设计；OO 和设计模式；C++和 STL 以及模板，ACE。欢迎大家交流。

22.2 参考文档

表 3 参考的文档

参考书目	作者/译者	说明
《C++ Network Programming Volume 1_Mastering Complexity With ACE and Patterns》	Douglas C. Schmidt, Stephen D. Huston	很多问题在这本书的副栏都有描述，如果你看的很认真，也许不会想我这样碰暗礁。
《C++网络编程卷 1:运用 ACE 和模式消除复杂性》	於春景	
《C++ Network Programming Volume 2 - Systematic Reuse with ACE and Frameworks》	Douglas C. Schmidt, Stephen D. Huston	很多问题在这本书的副栏都有描述，如果你看的很认真，也许不会想我这样碰暗礁。
《C++网络编程，卷 2，基于 ACE 和框	马维达	

架的系统化复用》		
《The.ACE.Programmers.Guide》	Stephen D. Huston, James CE Johnson, Umar Syyid	
《ACE 程序员指南》	马维达	
《ACE 自适配通信环境中文技术文档》	马维达	
ACE html	ACE 用 Doxgen 自动生成的文档	

22.3 文章说明和版权声明

此文档是耗费两年时间总结一些自己在使用 ACE 的 7 年中发现的一些问题，在凑够了 20 个标题后才进行发布。后面也许会根据自己的一些新的发现修正补充一下文档，也许。

本着自由的精神，阅读者可以无须授权就可以自由的转载这个文档，我只保留作者的署名权利，也就是说，你转载只需保留这段说明和文档的完整性（但你不能修改这个文档，谢谢）。

这篇文档也是为了回馈一下这些年来为自由软件奋斗的人，也谢谢周围陪我一起玩 ACE 的 Rong，Sonicmao，Awayfang 等兄弟们。最后感谢一下 Annie，她忍受了我整理文档而不陪她看电视。