ACE Tutorial [翻译]

注:文章整理自 jnn 的 blog,为方便阅读,整理成单独的 word 文档
整理时间:2008.11.29 23 时 6 分 5 秒 by Blade
blog 地址: http://jnn.blogbus.com/tag/ACE_TAO/

- **ACE 教程 [翻译] （前言）**
  2004-10-24

  Tag：ACE_TAO

写在前面的话

很早以前就有写一个 ACE 教程的冲动，其实当初的目的很简单，就是想让自己能够更好的学习 ACE，理解 ACE，应用 ACE。遗憾的是一直都没有找到一个合适的入手点。

其实目前世面上的有个 ACE 介绍的书很多，其中有：

C＋＋ Network Programing V1

C＋＋ Network Programing V2

ACE Programmer's Guide

这些书的中文译本有些已经出版了，有些也要马上出版。

不知道大家留心 ACE 发行包中的 doc 目录下面还有一个大概是 2000 年左右写的教程，我找了一下目前还没有看到有中文的译本，于是就想把这个教程整理一下翻译成为中文。也算是 ACE 这个开源项目在国内的推广尽微薄力了。

也许你会问，2004 年的现在，有这么多讲述 ACE 的书了，为什么还要翻译这个教程呢？

我觉得首要的一点是因为这个教程是为零起点的用户写的，通过对有关 ACE 最基本的概念的介绍，以及简单的例子程序，让读者能够迅速入手。另外本教程最大特点就是对示例代码有特别详细的注释，通过阅读这些注释，能够比较对代码的来龙去脉有比较清楚的了解，即刻将所学的东西运用与实际的编程中。最后，通过翻译这个教程，能够再一次熟悉 ACE 的各部分功能，我将努力把自己对 ACE 的理解和大家分享。

- **ACE Tutorial [翻译]01－page01**

2004-10-24

Tag：ACE_TAO

http://jnn.blogbus.com/logs/459148.html

初学者如何实现 ACE 工具包

本教程的目的是向你展示如何创建一个简单的能够同时响应多个客户连接的服务器程序。于“传统”的服务器端应用不同，本服务程序只在一个进程中响应所有的客户请求。有个多进程或者多线程方面的问题将在本教程的后续部分讨论。

创建一个服务器需要做那些事情？

1. 需要接受客户端的请求
2. 建立连接后相关处理
3. 需要一个主程序来循环处理上面内容

ACE 中的 Acceptor 为我们服务器程序需求提供了一个很好的解决方案。这个类通过给定的 TCP/IP 端口号监听客户发来的连接请求。

当 acceptor 接收到的连接请求后，它将创建一个新的对象（the handler）来处理客户的请求，同时 acceptor 返回并监听其他的连接。

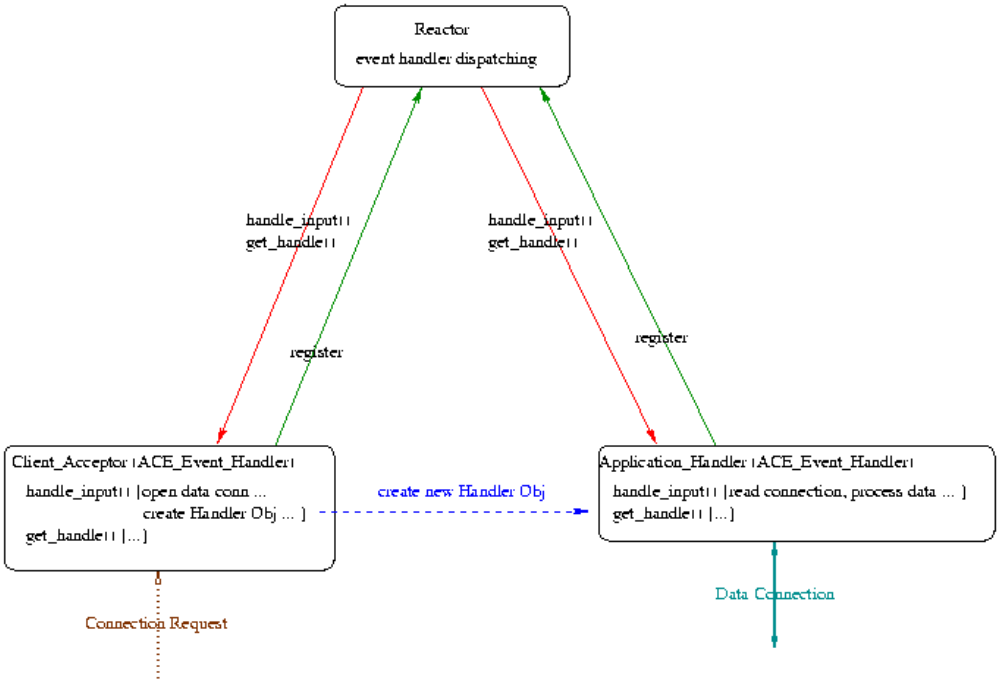
ACE 中的 EventHandler 可以满足我们的第二个需求。虽然这样看上去不太明显，但是随着教程的深入，大家将会逐渐认识到 EventHandler 的重要性。

最后，通过一个简单的 main() 方法来实现的我们的循环处理程序。循环处理程序的作用就是在所有的初始化操作完毕之后，进入一个死循环，在该循环中负责调用 Acceptor 处理客户连接请求或者调用 EventHandler 处理数据“事件”。

在我们继续下面教程之前，需要向大家介绍一下 ACE 中的另一个重要概念， Reactor （反应器）

在现在的阶段我不希望大家对什么是 reactor，reactor 是做什么的，以及 reactor 是如何实现这些细节很了解，但是你需要能够理解 reactor 最基本的功能，因为在下面的第一段代码中，将会出现 reactor。（负责注册事件处理句柄以及分发网络事件）

下图展示的 Reactor，Acceptor 以及应用处理句柄之间的相互关系。



简单来说： reactor 是一个负责对发生在其他对象的事情作出响应的对象。这些事情被称之为事件。其他对象是你向 reactor 注册的通讯对象。在向 reactor 注册的过程时，你可以向这些对象指定你所感兴趣的事件。当你注册的对象感兴趣的事件发生时，操作系统会将这些事件向 reactor 转发。Reactor 通过调用注册对象的成员方法来处理这些事件。注意 reactor 并不关心事件是如何发生的。它只是负责正确的处理事件。Reactor 只是简单地向注册对象转发事件。

为什么使用 reactor?

这将随着教程的深入，而逐渐清晰。现在，一个比较简单的回答是，它能够允许多个客户的同时连接能够在一个单线程服务器中有效地被处理。

传统地服务器一般都会每一个所服务的客户创建一个独立的线程或者进程。对于一个服务量比较大的服务（例如 telnet 和 ftp）这样的策略是很正确的。但是对于一个轻量级的服务俩说，创建处理进程所代来的系统负担已经超出的实际工作的负担。所以大家开始使用线程来替代进程来处理客户的请求。这是个比较好的解决方案，但是在一些情况下，也会对系统带来很大的负担。相比之下，为什么不使用单

线程来响应多个客户的请求，或者是使用比一个线程/进程一个客户更加智能的负载均衡方法来实现。

Caveat：在一个进程中的一个线程里面处理所有的请求只在这个请求可以立刻被处理完的情况下适用。

这就是 reactor 最强大的地方，同时也是最方便扩展的地方。开发者可以创建一个简单的，单线程的应用，在后面的教程里，可以进一步扩展到 线程-每-用户，进程-每-用户或者是线程池的解决方案。

如果是上面的胡言乱语使你认为 ACE 很难理解，不必担心。我们将通过代码细节再向你详细讲述。所以你可以把上面的内容先放在一边，当你在代码中遇到的时候在回顾一下。

- **ACE Tutorial [翻译] 01－page02**

2004-10-24

Tag：ACE_TAO

从现在开始，我们将把目光集中在主循环程序中。虽然这段代码是我们最后的实现产品，但是由于这段代码很简单，因此是一个比较好的入手点。因为大量的时间工作是由的派生类来实现的，所以主程序的确特别简单。

Kirthika Parameswaran **提供了对教程 1 的简要描述：**

这是一个简单的日志服务器例子。Reactor 采用单线程而不是一个客户一个线程方式，实现了的多个客户的请求处理。Reactor 负责响应事件，并通过“callback”技术将事件转发到注册了该事件的 Event_Handler。Reactor 在一个死循环中运行，用以响应所有的到来的事件。

Logging_Acceptor 监听一个叫做 SERVER PORT 的网络地址，并被动地接收到来的请求。这个 Acceptor 也是一个注册到 Reactor 上的 Event_Handler。对于 Reactor 来说，这只是又一个简单的 Event_Handler，不需要对其进行特别的处理。一旦客户连接事件发生，Acceptor 会接受客户连接并且为此建立一个连接。这里需要一个监听处理句柄会向 Reactor 进行注册，并且需要使用一个 ACE_Event_Handler::ACCEPT_MA

SK 掩码.

Logging_Client 是用以处理客户请求的一个 Event_Handler，它通过 handle_input()方法来处理与用户的交互。它使用 ACE_Event_Handler::READ_MASK 向 Reactor 进行注册。Event_Handler 可以通过 handle_close()方法或者显示调用 remove_handler()向 Reactor 注销。编译完服务应用，并运行，程序就循环等待客户请求的到达。

FYI (from Doug):

ACCEPT_MASK 是在 ACE_Event_Handler 类中所定义的掩码。它负责向 Reactor 通知，和它一起注册的事件处理句柄希望被动的接收连接请求。一般情况下，ACE_Acceptor 使用这个掩码。

READ_MASK 也是 ACE_Event_Handler 类中所定义的掩码。它负责向 Reactor 通知，和它一起注册的事件处理句柄希望从一个建立好的连接中读取数据。

```
// page02.html,v 1.14 2000/03/19 20:09:19 jcej Exp
/* 包含 client acceptor 所定义的头文件 */
#include "ace/Reactor.h"
/* For simplicity, we create our reactor in the global address space. In later tutorials we will do something more clever and appropriate. However, the purpose of this tutorial is to introduce a connection acceptance and handling, not the full capabilities of a reactor.
简单来说，在全局空间里面创建了一个 reactor。在后面的教程中，讲介绍更加合适和聪明的方法，来创建 reactor。由于本教程的目标是为了介绍如何建立连接和处理连接的，所以没有涉及更多的有关 reactor 部分的内容。*/
ACE_Reactor *g_reactor;
/* Include the header where we define our acceptor object.  An acceptor is an abstraction that allows a server to "accept" connections from clients.  */
/*包含了我们的 acceptor 对象的头文件，acceptor 对象是一个能够让服务器“接收”客户连接请求抽象类*/
#include "acceptor.h"
```

/* A TCP/IP server can listen to only one port for connectio
n requests. Well-known services can always be found at the
same address. Lesser-known services are generally told where
 to listen by a configuration file or command-line parameter.
 For this example,we're satisfied with simply hard-coding a
random but known value. */
/* 一个 TCP/IP 服务器只能监听一个连接请求端口。著名的服务所监
听的端口都是一样的 。而其他的服务一般通过读取配置文件或者是命
令行的方式来实现对端口的监听。在这个例子中我们使用直接在程序
中指定的方法来实现（产生随机指定的端口）*/
static const u_short PORT = ACE_DEFAULT_SERVER_PORT;
int main (int, char *[])
{
/* Create a Reactor instance. Again, a global pointer isn't
 exactly the best way to handle this but for the simple exam
ple here, it will be OK. We'll get cute with it later. Not
e how we use the ACE_NEW_RETURN macro, which returns 1 if op
erator new fails.
创建一个 Reactor 实例。再次声明，全局指针不是一个创建 Reactor
最好的办法。教程里面是为了简化操作，才这么写是的。在后面的教
程里面我们将把这段代码删除。注意我们在这使用了 ACE_NEW_RETURN
，当 new 操作失败，这个宏会返回 1*/
        ACE_NEW_RETURN (g_reactor, ACE_Reactor,1);
/* Like the Reactor, I'm skimming over the details of the AD
DR object. What it provides is an abstraction for addressing
 services in the network. All we need to know at this point
is that we are creating an address object which specifies th
e TCP/IP port on
which the server will listen for new connection requests.
和 Reactor 一样，我们将跳过 ADDR 对象的细节。它提供了一个对网络
地址的抽象。目前只需要知道我们创建的是一个指定 TCP/IP 端口的地
址对象。我们的服务器通过这个端口用来监听新的连接请求。*/

```
        ACE_INET_Addr addr (PORT);
        Logging_Acceptor *peer_acceptor;
/* We now create an acceptor object.  No connections will ye
t be established because the object isn't "open for business
" at this time. Which brings us to the next line...
```
我们现在来创建 acceptor 对象。 由于它现在还没有“开张”，因此还不能建立任何连接。有关“开张”的代码将会在下一行出现*/
```
ACE_NEW_RETURN (peer_acceptor, Logging_Acceptor, 1);
/* where the acceptor object is opened.  You'll find that mo
st ACE objects have to be open()ed before they're of any use
 to you.  On this open() call, we're telling the acceptor wh
ere to listen for connections via the 'addr' object.  We're
also telling it that we want it to be registered with our 'g
_reactor' instance.
```
这段代码调用了 acceptor 对象的 open 方法。你会发现大多数的 ACE 对象在正式使用之前，都会调用 open（）方法。在 open 方法中，我们将告知 acceptor 去监听那个 “addr”对象。我们同时告知 acceptor 将会被注册到我们的“g_reactor”实例上*/
```
        if (peer_acceptor->open (addr, g_reactor) == -1 )
                ACE_ERROR_RETURN ((LM_ERROR,"Opening Acceptor
\n"),-1);
        ACE_DEBUG ((LM_DEBUG,"(%P|%t) starting up server logg
ing daemon\n"));
/* The reactor's handle_events member function is responsibl
e for looking at all registered objects and invoking an appr
opriate member function when anything of interest occurs. Wh
en an event is processed, the handle_events function returns.
 In order to get all
events, we embed this in an infinite loop. Since we put ours
elves into an infinite loop, you'll need to CTRL-C to exit t
he program.
```
Reactor 的 handle_events 方法是负责查找所有的注册对象，并且当这

些对象所感兴趣的事件发生时，调用这些对象的确定的方法。当事件处理完毕，handle_event 方法返回。为了能够处理所有的事件，handle_event 方法被放置在一个死循环中虽然程序是处于死循环的状态中，但是你可以通过 CTRL-C 来退出程序。*/

```
        for (;;)
                g_reactor->handle_events ();
        return 0;
}
```

正如前面说讲，这个主程序的确很简单

创建一个 address 对象，指定的我们所希望监听的端口。

创建一个 acceptor，用来监听指定的 address。

向 Reactor 注册 acceptor，来响应连接请求。

进入一个死循环，让 reactor 来负责响应事件。

在下一页中，我们将详细介绍 acceptor，看一下它是如何响应新的连接请求的。

- **ACE Tutorial [翻译] 01－page03**

2004-10-25

Tag：ACE_TAO

现在我们来看一下 acceptor 对象

Kirthika 有这样一个类比

在办公室中

Reactor：接待员 （前台）

Event_Handlers： 满足不同需求的不同部门

SERVER_PORT： 大门

Acceptor： 门卫

当一个客户（client）进入被门卫（等待客户请求的 accptor ）看管的大门（port），前台（reactor）负责将这个人接待至合适的部门（event_handler）,由这些部门来满足他的需求。

```
// page03.html,v 1.13 2000/03/19 20:09:19 jcej Exp


#ifndef _CLIENT_ACCEPTOR_H

#define _CLIENT_ACCEPTOR_H


/* A SOCK_Acceptor knows how to accept socket connections.  We'll use

  one of those at the heart of our Logging_Acceptor.  */

/* SOCK_Acceptor 知道如何接受 socket 连接。我们将在 Logging_Acceptor 内部使用*/

#include "ace/SOCK_Acceptor.h"


#if !defined (ACE_LACKS_PRAGMA_ONCE)

# pragma once

#endif /* ACE_LACKS_PRAGMA_ONCE */


/* An Event_Handler is what you register with ACE_Reactor.  When

  events occur, the reactor will callback on the Event_Handler.  More

  on that in a few lines.  */
```

```
/* Event_Handler 是我们向 ACE_Reactor 注册的类。当事件发生，
Reacotr 会回调响应的 Event_Handler。这将会在下面的代码中展示*/
#include "ace/Event_Handler.h"

/* When a client connects, we'll create a Logging_Handler to deal with
   the connection.  Here, we bring in that declaration.  */
/*当客户连接，我们需要创建一个 Logging_Handler 来处理连接。
   在这里加上相关的定义*/
#include "logger.h"

/* Our Logging_Acceptor is derived from ACE_Event_Handler.  That lets
   the reactor treat our acceptor just like every other handler.  */
/* 我们的 Logging_Acceptor 是一个 ACE_Event_Handler 的派生类。
这使得 Reactor 可以像其他的 handler 一样处理我们定义的 acceptor */
class Logging_Acceptor : public ACE_Event_Handler
{
public:

  /* For this simple case we won't bother with either constructor or
     destructor.  In a real application you would certainly have them.  */
```

```
/* 为了简化内容，我们并没有考虑有关构建和析构的内容。
 在一个真实的应用中，我们需要考虑到这些。*/


/* Here's the open() method we called from main().  We have two
  things to accomplish here: (1) Open the acceptor so that we can
  hear client requests and (2) register ourselves with the reactor
   so that we can respond to those requests.  */
/* 这就是我们在 main()中调用的 open()方法。在这里我们主要做了两件事情
  (1)打开 acceptor 监听客户连接请求
  (2)向 reactor 注册，使得 acceptor 能够响应这些请求
  */
int open (const ACE_INET_Addr &addr,
          ACE_Reactor *reactor)
{
  /* Perform the open() on the acceptor.  We pass through the
     address at which main() wants us to listen.  The second
     parameter tells the acceptor it is OK to reuse the address.
     This is necessary sometimes to get around closed connections
      that haven't timed out.  */
  /* 在 open()方法中的实现。第一个参数是在 main()方法中所期望监听的端口。
```

```
第二个参数通知 acceptor 允许复用地址。这样就可以不必等待连接关闭的超时*/
if (this->peer_acceptor_.open (addr, 1) == -1)
  return -1;

/* Remember the reactor we're using.  We'll need it later when we
   create a client connection handler.  */
/*传入我们所使用的 reactor。在创建客户连接处理句柄的时候会使用到。*/
reactor_ = reactor;

/* Now we can register with the reactor we were given.  Since the
  reactor pointer is global, we could have just used that but it's
  gross enough already.  Notice that we can pass 'this' right into
  the registration since we're derived from ACE_Event_Handler.  We
  also provide ACCEPT_MASK to tell the reactor that we want to
   know about accept requests from clients.  */
/*现在我们可以向 reactor 注册，由于 reactor 指针是一个全局变量，
并且已经初始化了，我们可以直接使用这个变量。注意，向 reactor 注册是，
传入的是“this”，这是由于 acceptor 是派生于 ACE_Event_Handler。
ACCEPT_MASK 是为了告诉 reactor，我们关心的事件是客户请求事件。*/
return reactor->register_handler (this,
```

```
                                        ACE_Event_Handler::ACCEPT_MASK);
  }

private:

  /* To provide multi-OS abstraction, ACE uses the concept of
     "handles" for connection endpoints.  In Unix, this is a
     traditional file descriptor (or integer).  On other OS's, it may
     be something else.  The reactor will need to get the handle (file
     descriptor) to satisfy it's own internal needs.  Our relevant
     handle is the handle of the acceptor object, so that's what we
      provide.  */
  /* 为了保证多操作系统的移植性，ACE 使用了“句柄（handles）”这个概念
  来描述一个连接端点。在 Unix 中，连接端点是用一个传统的文件描述符
  （或者整型数）。在其他的操作系统中，这能用其他形式来描述。Reactor
  会获取句柄（文件描述符）来满足自己内部的需要。在 acceptor 中也要提供
  相关的句柄 */
  ACE_HANDLE get_handle (void) const
  {
    return this->peer_acceptor_.get_handle ();
```

```
}

/* When an accept request arrives, the reactor will invoke the
  handle_input() callback.  This is where we deal with the
   connection request.  */
/* 当一个连接器请求到达，reactor 将会调用 hanlde_input()。
    在这个方法中我们处理有关连接的请求。*/
virtual int handle_input (ACE_HANDLE handle)
{
  /* The handle provided to us by the reactor is the one that
    triggered our up-call.  In some advanced situations, you might
    actually register a single handler for multiple connections.
    The _handle parameter is a way to sort 'em out.  Since we don't
    use that here, we simply ignore the parameter with the
     ACE_UNUSED_ARG() macro.  */
     /* handle 是 reactor 所提供的可以提供向上的回调接口。在一些情况下，
     你可以为多个连接注册一个处理句柄。 这个 handle 参数可以用来区分这些
     连接句柄。由于下面的代码并不使用这一参数，所有我们使用 ACE_UNUSED_ARG()
     宏来忽略这一参数*/
  ACE_UNUSED_ARG (handle);
```

```
Logging_Handler *svc_handler;



/* In response to the connection request, we create a new
   Logging_Handler.  This new object will be used to interact with
   the client until it disconnects.  Note how we use the
    ACE_NEW_RETURN macro, which returns -1 if operator new fails. */
/* 为了响应连接请求，我们创建了一个 Logging_Handler。
这个对象负责与客户段进行交互，直到连接断开。注意使用
ACE_NEW_RETURN 宏，当 new 操作失败时返回－1*/
ACE_NEW_RETURN (svc_handler,
                Logging_Handler,
                -1);



/* To complete the connection, we invoke the accept() method call
   on the acceptor object and provide it with the connection
   handler instance.  This transfers "ownership" of the connection
    from the acceptor to the connection handler.  */
/* 为了完成连接，我们调用 acceptor 对象的 accept()方法来实现有关连接的
处理。在这里将 acceptor 的监听到的连接所有权转交给连接操作处理模块。
```

```
【这里的 accept 方法只能够接受 ACE_SOCK_Stream 类型的参数，
  但是为什么可以直接将 svc_handler 传递过去？】
   */
if (this->peer_acceptor_.accept (*svc_handler) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "%p",
                     "accept failed"),
                    -1);

/* Again, most objects need to be open()ed before they are useful.
  We'll give the handler our reactor pointer so that it can
  register for events as well.  If the open fails, we'll force a
   close().  */
/* 大多少的对象需要调用 open 方法来进行初始化操作。我们通过向处理句柄
传递 reactor 参数，来实现事件注册。如果 open 调用失败，我们将调用
close()注销注册*/
if (svc_handler->open (reactor_) == -1)
  svc_handler->close ();

return 0;
```

```
  }


protected:


  /* Our acceptor object instance */

  /* 我们的 accetpor 对象实例*?

  ACE_SOCK_Acceptor peer_acceptor_;


  /* A place to remember our reactor pointer */

  /*放置 reactor 指针参数 */

  ACE_Reactor *reactor_;

};


#endif /* _CLIENT_ACCEPTOR_H */
```

这里需要说明的是，我们在开放这个对象的时候，几乎没有写有关应用的代码。实事上，如果我们继续深入，就可以发现，我们向 accept 方法传入了一个新创建的 Logging_Handler 对象。你也许会问，为什么这里没有任何 C++模板。实事上 ACE 工具包中可以这样实例化你的 Acceptor 模板。

typedef ACE_Acceptor <*YourHandlerClass*, ACE_SOCK_ACCEPTOR> *YourAcceptorClass*;

我们可以这么做

typedef ACE_Acceptor <Logging_Handler, ACE_SOCK_ACCEPTOR> Client_Acceptor;

这将创建一个和我们上面相类似的代码。其中最大的不同是 handle_input 方法是由模板所创建并且不向 reactor 进行注册。从长远方面来讲，这样做会代来很大的好处。这是因为我们可以向 open 方法注入 Logging_Handler 的逻辑，来实现一个更具通用化的 acceptor。（范型编程的思想？）

现在我们知道如何来接受一个连接请求了，下面我们讲介绍如何对建立好的连接进行处理。虽然我们刚刚介绍了够酷的模板，但是我们还将继续我们的“手工”acceptor 的上面开发。

正如上面提到的，他们两者的区别是在连接处理的 open 方法上面。

- **ACE Tutorial [翻译] 01－page04**

2004-10-25

Tag：ACE_TAO

现在我们开始着手研究 logger object

```
// page04.html,v 1.14 2000/03/19 20:09:20 jcej Exp



#ifndef _CLIENT_HANDLER_H
#define _CLIENT_HANDLER_H
```

```
/* A connection handler will also be derived from ACE_Event_
Handler so that we can register with a reactor.
由于连接处理句柄继承于ACE_Event_Handler,因此可以很方便的向Re
actor进行注册 */
#include "ace/Event_Handler.h"
#if !defined (ACE_LACKS_PRAGMA_ONCE)
# pragma  once
#endif  /* ACE_LACKS_PRAGMA_ONCE */
#include "ace/INET_Addr.h"
/* Since we're doing TCP/IP, we'll need a SOCK_Stream for th
e connection.
由于我们使用TCP/IP，所以我们使用SOCK_Stream来处理连接*/
#include "ace/SOCK_Stream.h"
class Logging_Handler : public ACE_Event_Handler
```

```
{
public:
/* Like the acceptor, we're simple enough to avoid construct
or and destructor.
和 acceptor 一样，为了简化，这里我们忽略了构造和析构*/
/* To open the client handler, we have to register ourselves
 with the reactor.  Notice that we don't have to "open" our
ACE_SOCK_Stream member variable.  Why?  Because the call to
the acceptor's accept() method took care of those details fo
r us.
为了初始化客户处理句柄，我们需要向 reactor 进行注册。 注意我们
并不需要“打开”我们的 ACE_SOCK_Stream 成员变量。这是为什么？
这是由于调用 acceptor 的 accept()方法来帮我们实现相关的操作*/
int open (ACE_Reactor *reactor)
{
/* Remember our reactor...*/
reactor_ = reactor;
/* In this case we're using the READ_MASK.  Like the accepto
r, handle_input() will be called due to this mask but it's a
 nice piece of bookkeeping to have separate masks for the se
parate types of activity.
在这里我们使用 READ_MASK. 使用这个掩码虽然 acceptor 一样，hand
le_input()将会被调用，但是为了保证一个好的登记信息的 ，在这里
为不同的活动使用了不同的掩码*/
       if (reactor->register_handler (this, ACE_Event_Handle
r::READ_MASK) == -1)
              ACE_ERROR_RETURN ((LM_ERROR,"(%P|%t) can't reg
ister with reactor\n"),-1);
       return 0;
}
/* If we're explicitly closed we'll close our "file handle
".  The net result is to close the connection to the client
```

and remove ourselves from the reactor if we're registered
如果你没有显示的关闭，在这里我们将关闭我们的“文件描述符” 。
这里只是单纯的关闭到客户端的连接，如果向 reactor 注册过，就进行注销的操作*/

```
int close (void)
{
        return this->handle_close(ACE_INVALID_HANDLE,ACE_Event_Handler::RWE_MASK);
}
```

/* This is a bit of magic... When we call the accept() method of the acceptor object, it wants to do work on an ACE_SOCK_Stream.We have one of those as our connection to the client but it would be gross to provide a method to access that object. It's much cooler if the acceptor can just treat the Logging_Handler as an ACE_SOCK_Stream. Providing this cast operator lets that happen cleanly. */

/* 这是一个小魔法… 当我们调用 acceptor 对象的 accept()方法时，我们希望
对 ACE_SOCK_Stream 对象进行一些操作。在类内部，有这个对象，但是为了传递这个对象，需要再提供一个类方法来实现。通过下面的函数，acceptor 就可以直接将 Logging_Handler 看作时 ACE_SOCK_Stream。*/

```
operator ACE_SOCK_Stream &()
{
        return this->cli_stream_;
}
protected:
```

/* Again, like the acceptor, we need to provide the connection
handle to the reactor.
和 acceptor 一样，我们在这里需要提供一个 get_handle 方法，来处

理连接*/

```
ACE_HANDLE get_handle (void) const
{
        return this->cli_stream_.get_handle ();
}
```

/* And here's the handle_input(). This is really the workho rse of the application.

下面的 handle_input()，是我们应用真正处理的部分*/

```
virtual int handle_input (ACE_HANDLE)
{
```

/* Create and initialize a small receive buffer. The extra byte is there to allow us to have a null-terminated string w hen it's over.

创建一个小的接收缓冲区，多余的一位是为了当缓冲区满是，能够让我们处理 null 结尾的字符串*/

```
char buf[BUFSIZ + 1];
```

/* Invoke the recv() method of the ACE_SOCK_Stream to get so me data. It will return -1 if there is an error. Otherwise, it will return the number of bytes read. Of course, if it read zero bytes then the connection must be gone. How do I know that? Because handle_input() would not be called by th e reactor if there wasn't *some* kind of activity and a clos ed connection looks like a read request to the reactor. But when you readf rom a closed connection you'll read zero byt es.

通过调用 ACE_SOCK_Stream 中的 recv()方法来获取数据。当出现错误的时候，它会直接返回－1。否则将会返回实际读取到的字节数。当然，如果它读到零个字节的话，连接应该中断了？我应该怎么处理这样的情况呢？因为如果没有发生某种事件或者是关闭连接这类向 reactor 发出读取请求的事件，handle_input()是不会被 reactor 调用的。 这样，你从一个已经断开的连接上，将会只读取到零个字节。

Notice that in the error case or closed case we return -

1. That tells the reactor to call our handle_close() where we'll take care of shutting down cleanly.
注意当错误或者关闭发生时，我们返回－1。这样就是告诉 reactor 调用我们的 handle_close()来实现相关的关闭操作。
Although we don't make use of them, there are additional parameters you can use with the recv() call. One of these is an ACE_Time_Value that allows you to limit the amount of time blocking on the recv(). You would use that if you weren't sure if data was available. Since we only get to handle_input() when data is ready, that would be redundant. On the other hand, if you use recv_n() to read *exactly* a number of bytes then limiting the time you wait for those bytes might be good. The other paramter that may come in handy is an integer *flags*. This is passed directly to the underlying OS recv() call. See the man page recv(2) and the header sys/socket.h for the gory details.
在调用传统的 recv 方法的时候，还需要传入一些参数。一个是 ACE_Time_Value 类型的参数，可以定义阻塞接收的最大时间。如果你不能确定什么时候数据到达。当数剧到达时，我们使用 handle_input() 来处理数据，这样做会增大数据负担。 另一方面，如果你使用 recv_n() 在指定的时间内来读取指定的字节的数据也许是更加合适的作法。还有一个整型参数 falgs。这个参数是直接传输到底层操作系统中的 recv 调用的。大家可以通过查找 recv(2)以及 sys/socket.h 来更翔实的细节。
*/

```
ssize_t retval;
switch (retval = this->cli_stream_.recv (buf, BUFSIZ))
{
case -1:
ACE_ERROR_RETURN ((LM_ERROR,"(%P|%t) %p bad read\n","client
logger"),-1);
case 0:
```

```
ACE_ERROR_RETURN ((LM_ERROR,"(%P|%t) closing log daemon (fd
=%d)\n",this->get_handle ()),-1);
default:
buf[retval] = '';
ACE_DEBUG ((LM_DEBUG,"(%P|%t) from client: %s",buf));
}
return 0;
}
/* When handle_input() returns -1, we'll end up here.  There
 are a few housekeeping chores to handle.  */
int handle_close (ACE_HANDLE,ACE_Reactor_Mask _mask)
{
/* Remove ourselves from the reactor.  We have to include th
eDONT_CALL in the mask so that it won't call handle_close()
on us again!  */
reactor_->remove_handler (this,_mask | ACE_Event_Handler::DO
NT_CALL);
/* Close the socket that we're connected to the client wit
h.  */
cli_stream_.close ();
/* Since we know we were dynamically allocated by the accept
or,now is a good time to get rid of ourselves.  */
delete this;
return 0;
}
protected:
/* Our peer connection.  */
ACE_SOCK_Stream cli_stream_;
/* Our reactor (and our acceptor's reactor).  */
ACE_Reactor *reactor_;
};
#endif  /* _CLIENT_HANDLER_H */
```

在注释中我们关注的重点时 handle_input(),其他部分的内容只是简单的管理附属模块。在以后的教程中，我们将学习封装了管理附属功能的 ACE_Svc_Handler<>。

- **ACE Tutorial [翻译] 02—page01**

2004-10-28

Tag：ACE_TAO

创建一个更好的服务器

在这个教程中，我们将在第一教程的基础上做少量扩展。在结束本教程时，我们可以获得一个更简单，可维护性更好的服务器对象。

首先，我们还需要搞清楚外面在教程 001 中所涉及到的几个问题

创建服务器程序所需要做的事情？

1. 接收客户端的连接请求
2. 在建立的连接之上做处理。
3. 在主程序中循环处理上面的内容。

先前，我们针对上面的问题分别给出的解决方案。在这个解决方案的最后，我们发现我们的应用层代码都封闭在 *Handler* 模块中。同时我们也提到了可以将 acceptor 中的手写代码进行简化。现在就来看看我们时怎么做的吧。

- **ACE Tutorial [翻译] 02—page02**

2004-10-28

Tag：ACE_TAO

和教程 1 一样，这也是一个很小的程序。在这里我将添加一下新的概念，做为补偿，我将吧 acceptor 进行大量简化。

Kirthika's 给出的概述

通过 ACE 所提供的现货组件以及类，使得我们的服务例子变得更加简单。

在这里，Logging_Acceptor 继承于 ACE_Acceptor，并且和 Logging_Handler

以及 ACE_SOCK_ACCEPTOR 相关联。这样通过向 reactor 实例进行注册，获得客户端得连接请求并进行处理。

我们将通过 ACE_SigAction 以及 ACE_SignalHandler,来实现信号（signal）处理抓取 CTRL_C (产生 SIGINT 信号).通过这个信号来终止 reactor 处理事件。

接下来，让 reactor 接收到^C 时，结束死循环的执行，并完成 reactor 和 acceptor 的销毁工作。

Logging_Handler 这次是派生于 ACE_Svc_Handler 而不是 Event_Handler，这是因为 Svc_Handler 包含了 SOCK_Stream 并且提供了所有 reactor 需要的方法。Svc_Handler 具备响应事件，并通过底层的数据流和远端的任务进行交互。

在 Reactor 中的定时器可根据需要运行需要定时执行的任务，其中事件间隔是通过 ACE_TimeValue 来实现。

同时对象的析构方法进行优化。（Also, optimisations have been made in the form of a separate function for destroying the objects used.）

在这里我们展示了如何用 ACE 的组件构建一个简单的服务器程序。

下面让我们看一下主程序

```
// page02.html,v 1.10 2000/03/19 20:09:20 jcej Exp

/* As before, we need a few ACE objects as well as our Logging_Handler
  declaration.  */
/* 和以前 Logging_Handler 一样，我们在这里需要引入一下 ACE 所提供的对象*/
```

```
#include "ace/Acceptor.h"
#include "ace/SOCK_Acceptor.h"
#include "ace/Reactor.h"
#include "handler.h"

/* We'll still use the global reactor pointer.  There's a snappy way
   around this that shows up in later server tutorials.  */
/* 我们还是使用一个全局的 reactor 指针。在后面的教程中我们将会展示一个
    更加简单的方式*/
ACE_Reactor *g_reactor;

/* This was hinted at in Tutorial 1.  Remember the hand-coded acceptor
  that we created there?  This template does all of that and more and
  better.  If you find yourself creating code that doesn't feel like a
  part of your application, there's a good chance that ACE has a
  template or framework component to do it for you.  */
/* 在教程 1 中，我们曾提到过这部分内容。 还记得我们为 acceptor
所写的代码吗？ 通过下面的模板，可以多快好省地实现我们上面提到功能。
如果你决定不写与应用层不相关的代码，那么使用 ACE 所提供的模板将是最佳选择*
/
typedef ACE_Acceptor<Logging_Handler, ACE_SOCK_ACCEPTOR> Logging_Accept
or;

/* One of the new things will be a signal handler so that we can exit
  the application somewhat cleanly.  The 'finished' flag is used
  instead of the previous infninite loop and the 'handler' will set
  that flag in respose to SIGINT (CTRL-C).
  The invocation of ACE_Reactor::notify() will cause the
  handle_events() to return so that we can see the new value of 'finishe
d'.
*/
/*这里我们为了能够让应用能够比较清晰地退出，将接触一个新的概念信号处理。
```

```
  通过 finished 标记来取代死循环，并通过 handler 方法来响应 SIGINT(CTRL_C)
  并改变 finished 标记的值。
  通过调用 ACE_Reactor::notify()方法使得 handle_events()方法返回，并使得
  'finished'的值生效*/
static sig_atomic_t finished = 0;
extern "C" void handler (int)
{
  finished = 1;
  g_reactor->notify();
}

static const u_short PORT = ACE_DEFAULT_SERVER_PORT;

int
main (int, char **)
{
  // Create the reactor we'll register our event handler derivatives wit
h.
  // 创建 reactor，这样就可以把我的事件句柄向其注册了
  ACE_NEW_RETURN (g_reactor,
                  ACE_Reactor,
                  1);

  // Create the acceptor that will listen for client connetions
  // 创建 acceptor，来监听客户的连接请求
  Logging_Acceptor peer_acceptor;

  /* Notice how similar this is to the open() call in Tutorial 1.  I
    read ahead when I created that one so that it would come out this
    way...  */
  /* 注意这里的 open 方法是和教程 1 中提到的内容一致。*/
  if (peer_acceptor.open (ACE_INET_Addr (PORT),
```

```
                        g_reactor) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "%p\n",
                     "open"),
                    -1);

/* Here's the easiest way to respond to signals in your application.
   Simply construct an ACE_Sig_Action object with a "C" function and
   the signal you want to capture.  As you might expect, there is
   also a way to register signal handlers with a reactor but we take
   the easy-out here.  */
   /*这里给出了一让应用响应信号的简单实现方式。
   创建 ACE_Sig_Action 对象，并传入一个"C"语言的方法指针，
   以及你所要捕获的信号。这样正如你所期望的，通过这个对象就实现了
   向 reactor 注册信号处理句柄的工作。*/
ACE_Sig_Action sa ((ACE_SignalHandler) handler, SIGINT);

ACE_DEBUG ((LM_DEBUG,
            "(%P|%t) starting up server logging daemon\n"));

// Perform logging service until the signal handler receives SIGINT.
// 在接收到 SIGINT 之前一直提供日志服务
while (!finished)
  g_reactor->handle_events ();

// Close the acceptor so that no more clients will be taken in.
// 关闭 acceptor，不接收客户端的请求
peer_acceptor.close();

// Free up the memory allocated for the reactor.
// 释放 reactor 所占有的空间
delete g_reactor;
```

```
  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) shutting down server logging daemon\n"));
  return 0;
}

#if defined (ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION)
template class ACE_Acceptor <Logging_Handler, ACE_SOCK_ACCEPTOR>;
template class ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>;
#elif defined (ACE_HAS_TEMPLATE_INSTANTIATION_PRAGMA)
#pragma instantiate ACE_Acceptor <Logging_Handler, ACE_SOCK_ACCEPTOR>
#pragma instantiate ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
#endif /* ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION */
```

- ## ACE Tutorial [翻译] 02－page03

2004-10-28

Tag：ACE_TAO

现在看一下我们新的日志服务程序

```
// page03.html,v 1.11 2000/03/19 20:09:20 jcej Exp

#ifndef LOGGING_HANDLER_H
#define LOGGING_HANDLER_H

#include "ace/INET_Addr.h"

#if !defined (ACE_LACKS_PRAGMA_ONCE)
# pragma once
```

```
#endif /* ACE_LACKS_PRAGMA_ONCE */

#include "ace/SOCK_Stream.h"
#include "ace/Reactor.h"

/* Since we used the template to create the acceptor, we don't know if
   there is a way to get to the reactor it uses.  We'll take the easy
   way out and grab the global pointer.  (There is a way to get back to
   the acceptor's reactor that we'll see later on.)  */
/*我们采用模板创建了 acceptor，但是我们还不知道是否有一个更加简便使用
reactor 的方法。我们还是采用最简单的全局指针的方法来使用 reactor。
（再后面的教程中，我们将介绍一种在 acceptor 中获得 reactor 的方法）*/
extern ACE_Reactor *g_reactor;

/* This time we're deriving from ACE_Svc_Handler instead of
   ACE_Event_Handler.  The big reason for this is because it already
   knows how to contain a SOCK_Stream and provides all of the method
   calls needed by the reactor.  The second template parameter is for
   some advanced stuff we'll do with later servers.  For now, just use
   it as is...  */
/*这次我们选择了由 ACE_Svc_Handler 而不是 ACE_Event_Handler 来派生
Logging_Handler。其中最主要的原因是 ACE_Svc_Handler 已经包含了一个
SOCK_Stream，并且提供 reactor 所需要的所有方法。在第二个模板参数中，定义
了一些比较高级一点的材料，这将后面进行介绍。 现在姑且这么用一下。*/
class Logging_Handler : public ACE_Svc_Handler <ACE_SOCK_STREAM, ACE_NUL
L_SYNCH>
{
public:

  /* The Acceptor<> template will open() us when there is a new client
     connection.  */
  /* 当由新的客户连接是，Acceptor 模板会调用这个 open 方法*/
```

```
virtual int open (void *)
{
  ACE_INET_Addr addr;

  /* Ask the peer() (held in our baseclass) to tell us the address
     of the cient which has connected.  There may be valid reasons
     for this to fail where we wouldn't want to drop the connection
     but I can't think of one. */
  /* 通过访问 peer()(这是在父类中所拥有的对象)来获知连接的客户端的地址。
  执行这一操作的过程中，如果出现失败，我们将释放连接。*/
  if (this->peer ().get_remote_addr (addr) == -1)
    return -1;

  /* The Acceptor<> won't register us with it's reactor, so we have
     to do so ourselves.  This is where we have to grab that global
     pointer.  Notice that we again use the READ_MASK so that
     handle_input() will be called when the client does something.  */
  /* Acceptor<>不会将 handler 向 reactor 进行注册,所以我们只能自力更生了。
  在这里就是我们需要获得全局指针的地方。主要在这里我们还是使用 READ_MAS
K,
  这样当客户端向我们发送数据的时候,reactor 就会调用我们的 handle_input()
*/
  if (g_reactor->register_handler (this,
                                   ACE_Event_Handler::READ_MASK) == -
1)
    ACE_ERROR_RETURN ((LM_ERROR,
                       "(%P|%t) can't register with reactor\n"),
                      -1);

  /* Here's another new treat.  We schedule a timer event.  This
     particular one will fire in two seconds and then every three
     seconds after that.  It doesn't serve any useful purpose in our
```

```
    application other than to show you how it is done.  */
  /*这里有个新要处理的情况。我们需要设置定时器。 这个定时器将在 2 秒之后
  触发一次，之后每三秒再触发一次。这个定时器只是用来实例，并不产生实际
的作用*/
  else if (g_reactor->schedule_timer (this,
                                      0,
                                      ACE_Time_Value (2),
                                      ACE_Time_Value (3)) == -1)
    ACE_ERROR_RETURN ((LM_ERROR,
                       "can'(%P|%t) t register with reactor\n"),
                      -1);

  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) connected with %s\n",
              addr.get_host_name ()));

  return 0;
}

/* This is a matter of style & maybe taste.  Instead of putting all
  of this stuff into a destructor, we put it here and request that
  everyone call destroy() instead of 'delete'.  */
/* 这是编程习惯的问题。 我们把所有清除的操作都放在 destroy 方法中而不是
  放在析构函数中。*/
virtual void destroy (void)
{
  /* Remove ourselves from the reactor
    向 reactor 注销操作 ACE_Event_Handler::DONT_CALL */
  g_reactor->remove_handler
    (this,
     ACE_Event_Handler::READ_MASK | ACE_Event_Handler::DONT_CALL);
```

```
  /* Cancel that timer we scheduled in open()
      取消在 open()方法中定义的 timer */
  g_reactor->cancel_timer (this);

  /* Shut down the connection to the client.
  关闭客户端的连接 */
  this->peer ().close ();

  /* Free our memory.
  释放自己占用的内存 */
  delete this;
}

/* If somebody doesn't like us, they will close() us.  Actually, if
  our open() method returns -1, the Acceptor<> will invoke close()
  on us for cleanup.  */
/* 向外部提供 close()的方法。 如果 open()方法返回-1,Acceptor<>将会调用
  close()进行相关清理的操作*/
virtual int close (u_long flags = 0)
{
  /* The ACE_Svc_Handler baseclass requires the <flags> parameter.
    We don't use it here though, so we mark it as UNUSED.  You can
    accomplish the same thing with a signature like handle_input's
    below.  */
  /* ACE_Svc_Handler 的基类需要 <flags>参数。 在这里我们不使用它，所有
  我们使用 UNUSED 宏。 对于下面的 handle_input 方法，你也可以这样做*/
  ACE_UNUSED_ARG (flags);

  /*
    Clean up and go away.
    清理现场并退出
    */
```

```
    this->destroy ();
    return 0;
  }

protected:

  /* Respond to input just like Tutorial 1.  */
  /* 像教程 1 一样响应输入*/
  virtual int handle_input (ACE_HANDLE)
  {
    char buf[128];
    ACE_OS::memset (buf, 0, sizeof (buf));

    switch (this->peer ().recv (buf,
                                sizeof buf))
      {
      case -1:
        ACE_ERROR_RETURN ((LM_ERROR,
                           "(%P|%t) %p bad read\n",
                           "client logger"),
                          -1);
      case 0:
        ACE_ERROR_RETURN ((LM_ERROR,
                           "(%P|%t) closing log daemon (fd = %d)\n",
                           this->get_handle ()),
                          -1);
      default:
        ACE_DEBUG ((LM_DEBUG,
                    "(%P|%t) from client: %s",
                    buf));
      }
```

```
  return 0;
}

/* When the timer expires, handle_timeout() will be called.  The
  'arg' is the value passed after 'this' in the schedule_timer()
  call.  You can pass in anything there that you can cast to a
  void*.  */
/* 当定时器超时时，handle_timeout()就会被调用。 在这里 arg 是
'this' 向 schedule_timer()传入的参数。只要把你要传的参数转换成为 void*
就可以了传递任意类型的参数。 */
virtual int handle_timeout (const ACE_Time_Value &tv,
                            const void *arg)
{
  ACE_UNUSED_ARG(tv);
  ACE_UNUSED_ARG(arg);
  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) handling timeout from this = %u\n",
              this));
  return 0;
}

/*
  Clean ourselves up when handle_input() (or handle_timer()) returns -
1
  当 handle_input()（或者 handler_timer()）返回-1 时，进行清理操作
  */
virtual int handle_close (ACE_HANDLE,
                          ACE_Reactor_Mask)
{
  this->destroy ();
  return 0;
}
```

```
};

#endif /* LOGGING_HANDLER_H */
```

- **ACE Tutorial [翻译] 03**

2004-10-29

Tag：ACE_TAO

http://jnn.blogbus.com/logs/465772.html

创建一个简单的客户端

我们已经知道如何创建一个服务器，在这里我们将介绍一下如何创建一个客户端。由于这部分比较简单，所有我只打算用一页的篇幅来进行介绍。

Kirthika 说，“这里有一小段概述来描述这个客户端应用”。

在 server 中有一个 ACE_SOCK_Stream 类型的流对象。ACE_Sock_Connector 的职责就是向监听的 Server 创建连接。它使用的是存储在 ACE_INET_Addr 对象中的 server_host_address 和端口号来建立连接的。一旦连接建立，客户端将回集中与服务器进行交互，并且不断地向服务器发送消息。 注意：在这里调用了 send_n()方法，该方法提供了网络缓冲以及将数据稳定发送到服务器的功能。当然，为了提供容错功能这里采用了超时处理机制，保证传输事务能在服务结束之前完成。在服务器端如果接收到了零个字节将在 Event_Handler::handle_input() 中调用 close()方法进行结束操作。这样就给客户端断开连接的机会。之后服务器端中止。

```
// page01.html,v 1.12 2000/11/27 17:56:42 othman Exp

/* To establish a socket connection to a server, we'll need an
  ACE_SOCK_Connector.
  为了获得与服务器的 socket 连接，我们需要使用 ACE_SOCK_Connector。
*/
```

```
#include "ace/SOCK_Connector.h"
#include "ace/Log_Msg.h"

/* Unlike the previous two tutorials, we're going to allow the user to
   provide command line options this time.  Still, we need defaults in
   case that isn't done.
   与前面两个教程不同，我们在这里打算为用户提供命令行的参数选择。不过之前
   我们还是需要为这些参数设置缺省值。
*/
static u_short SERVER_PORT = ACE_DEFAULT_SERVER_PORT;
static const char *const SERVER_HOST = ACE_DEFAULT_SERVER_HOST;
static const int MAX_ITERATIONS = 4;

int
main (int argc, char *argv[])
{
  /* Accept the users's choice of hosts or use the default.  Then do
     the same for the TCP/IP port at which the server is listening as
     well as the number of iterations to perform.
     选择是用户定义的主机还是选择缺省值。同时也需要选择 TCP/IP 服务器
     进程所监听的端口号，以及循环的迭代次数。
     */
  const char *server_host = argc > 1 ? argv[1] : SERVER_HOST;
  u_short server_port = argc > 2 ? ACE_OS::atoi (argv[2]) : SERVER_PORT;
  int max_iterations = argc > 3 ? ACE_OS::atoi (argv[3]) : MAX_ITERATION
S;

  /* Build ourselves a Stream socket. This is a connected socket that
     provides reliable end-to-end communications. We will use the
     server object to send data to the server we connect to.
     建立我们自己的 socket 流。这是个能够保证可靠的端到端通讯功能的
     可连接的 socket。我们将采用 server 对象来负责向服务器传输数据。
```

```
 */
ACE_SOCK_Stream server;

/* And we need a connector object to establish that connection. The
   ACE_SOCK_Connector object provides all of the tools we need to
   establish a connection once we know the server's network
   address...
   现在我们需要一个 connector 对象来负责建立连接。ACE_SOCK_Connector
   对象向我们提供了一个向指定的服务器地址建立连接的工具*/
ACE_SOCK_Connector connector;

/* Which we create with an ACE_INET_Addr object. This object is
   given the TCP/IP port and hostname of the server we want to
   connect to.
   创建一个 ACE_INET_Addr 对象。这个对象需要指定我们想要连接的服务器主机
名
   以及 TCP/IP 端口号。
*/
ACE_INET_Addr addr (server_port,
                    server_host);

/* So, we feed the Addr object and the Stream object to the
   connector's connect() member function. Given this information, it
   will establish the network connection to the server and attach
   that connection to the server object.
   现在我们想 connetor 的 connect 方法中传入地址对象以及流对象。它根据
   这些给定的信息向指定的服务器建立连接并且与服务器对象建立连接。
*/
if (connector.connect (server, addr) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "%p\n",
                     "open"),
```

```
                    -1);

 /* Just for grins, we'll send the server several messages.
    呵呵，现在我们可以向服务器发送几条消息了。
*/
 for (int i = 0; i < max_iterations; i++)
   {
     char buf[BUFSIZ];

     /* Create our message with the message number
       创建消息号
     */
     ACE_OS::sprintf (buf,
                      "message = %d\n",
                      i + 1);
     /* Send the message to the server.  We use the server object's
       send_n() function to send all of the data at once. There is
       also a send() function but it may not send all of the
       data. That is due to network buffer availability and such. If
       the send() doesn't send all of the data, it is up to you to
       program things such that it will keep trying until all of the
       data is sent or simply give up. The send_n() function already
       does the "keep trying" option for us, so we use it.
        向服务器发送消息。我们通过调用 server 的 send_n()方法一次发送
       所有的数据。这里我们可以选择使用 send()方法，但是这个方法并不
       一次发送所有的数据，你需要负责进行不断尝试发送数据或者执行放弃
       操作的内容进行编程。而 send_n()方法已经为我们提供了这样的不断
       尝试发送的选择，所以我们使用这个方法。

       Like the send() method used in the servers we've seen, there
       are two additional parameters you can use on the send() and
       send_n() method calls.  The timeout parameter limits the
```

```
      amount of time the system will attempt to send the data to the
      peer.  The flags parameter is passed directly to the OS send()
      system call.  See send(2) for the valid flags values.

      和我们以前常见的 send()方法使用一样，在 send_n()方法中，我们需要
      使用两个传统的参数。在这里省略了系统尝试向对方发送数据的超时参数。
      这个标识参数是直接被定义到 OS 的 send()系统调用中的。你可以通过查找
      send(2)获取更详细的信息。
    */
    if (server.send_n (buf,
                       ACE_OS::strlen (buf)) == -1)
      ACE_ERROR_RETURN ((LM_ERROR,
                         "%p\n",
                         "send"),
                        -1);
    else
      /* Pause for a second.
      暂停一秒 */
      ACE_OS::sleep (1);
  }

/* Close the connection to the server.  The servers we've created so
  far all are based on the ACE_Reactor.  When we close(), the
  server's reactor will see activity for the registered event
  handler and invoke handle_input().  That, in turn, will try to
  read from the socket but get back zero bytes.  At that point, the
  server will know that we've closed from our side.
  关闭与服务器之间的连接。 由于我们创建的服务器都是建立在 ACE_Reactor
  基础之上的。当我们调用 close()方法,服务器的 reactor 将会激发注册的句柄,
  并调用 handle_input()方法。这样，在处理方法中将会从指定的 socket 中获取
  零个字节。至此，服务器端就能够检测到我们关掉连接了。
  */
```

```
  if (server.close () == -1)
    ACE_ERROR_RETURN ((LM_ERROR,
                       "%p\n",
                       "close"),
                      -1);
  return 0;
}
```

这是不是很简单啊。如果将所有与连接相关的内容封装在对象中，通过重载这些简单的操作方法以降低其对网络的依赖，就能使的这部分程序更加简单了。这部分的内容将在另一个教程出现。

- **ACE Tutorial [翻译] 04**

2004-10-29

Tag：ACE_TAO

一个更加聪明的客户端。

在上以个教程中，我们学会 了如何创建一个简单的能够传输整块数据的客户端。在这里我们将介绍一个更酷的方法，就是重载 C++中的<<操作符，来传输数据。我们将在本教程汇总详细介绍这部分内容。（这个教程是 ACE_IOStream 的原创地哦）

Kirthika 的介绍

够酷的客户端中酷的事情是我们使用了 C++中有关流数据的处理中<<操作符技巧。同时将有关连接部分的操作封装在了 open()方法中，需要传入服务器的主机名和端口，这样我们就可以构建的一个更加简洁的客户端，来实现服务其的连接建立。

```
// page01.html,v 1.12 2000/11/27 17:56:42 othman Exp
```

```
/* We need the connector object & we also bring in a simple string  class.
  我们需要 connector 对象已经一些有关*/
#include "ace/Log_Msg.h"
#include "ace/SOCK_Connector.h"
#include "ace/SString.h"
/* In this tutorial, we extend SOCK_Stream by adding a few wrappers  around the send_n() method.
 在并教程中，我们将通过添加一些在 send_n 之上的方法扩展 SOCK_Stream */
class Client : public ACE_SOCK_Stream
{
public:  // Basic constructor 基本的构造函数
  Client (void);

  /* Construct and open() in one call.  This isn't generally a good
  idea because you don't have a clean way to inform the caller when
     open() fails.  (Unless you use C++ exceptions.)
    构造和 open()方法的调用。这样做不是一个比较好的方法，这是由于当调用
    open()方法失败的话，我们没有做相关清理的方法。(除非你使用 C++的异常)
   */
  Client (const char *server,
  u_short port);

  /* Open the connection to the server.  Notice that this mirrors the
  use of ACE_SOCK_Connector.  By providing our own open(), we can
     hide the connector from our caller & make it's interaction easier.
     打开与服务器的连接。注意在这里需要使用 ACE_SOCK_Connector.通过我们
     自己定义的 open()方法，我们可以将与 connector 相关的调用进行隐藏，使
     得交互操作更加简单。
      */
  int open (const char *server,
  u_short port);

  /* These are necessary if you're going to use the constructor that
```

```
    invokes open().
    如果你打算使用构造函数来调用 open 方法的话，这样做是很有必要的。
  */
  int initialized (void) { return initialized_; }
  int error (void) { return error_; }

  /* This is where the coolness lies.  Most C++ folks are familiar
  with "cout << some-data."  It's a very handy and easy way to toss
  data around.  By adding these method calls, we're able to do the
    same thing with a socket connection.
    这些就是最酷的代码了。大多数的 C++用户都会熟悉"cout<<数据" 这是一个
    非常便捷和简单的传递数据的方法。通过添加这些方法调用，我们可以同样
    实现对一个 socket 连接的处理。
    */
  Client &operator<< (ACE_SString &str);
  Client &operator<< (char *str);
  Client &operator<< (int n);

protected:
  u_char initialized_;
  u_char error_;
};

/* The basic constructor just sets our flags to reasonable values.
  这是一个基本的构造方法，用来为变量设置初始值
 */
Client::Client(void)
{
  initialized_ = 0;
  error_ = 0;
}
```

```
/* This constructor also sets the flags but then calls open().  If the
  open() fails, the flags will be set appropriately.  Use the two
  inline method calls initialized() and error() to check the object
  state after using this constructor.
  这个构造函数执行同样设置变量以及调用 open()方法的操作。
   如果 open 调用失败，标识的值也会发生响应的改变。在执行完构造之后通过
   调用两个内联方法 initialized()以及 error()方法来确定对象的状态。
 */
Client::Client (const char *server,
  u_short port)
{
  initialized_ = 0;
  error_ = 0;
  this->open (server, port);
}

/* Open a connection to the server.  This hides the use of
  ACE_SOCK_Connector from our caller.  Since our caller probably
  doesn't care *how* we connect, this is a good thing.
  打开与服务器的连接。 这样就可以将 ACE_SOCK_Connector 向我们的调用者进行
   隐藏。因为我们的调用者并不关心我们是“如何”进行连接，这是一个好事情。
*/
int
Client::open (const char *server,
  u_short port)
{
  /* This is right out of Tutorial 3.  The only thing we've added is
     to set the initialized_ member variable on success.
     这与教程 3 中的内容很类似，唯一的不同是当连接成功我们需要设置 inialized_
     成员变量的值。
     */
```

```
  ACE_SOCK_Connector connector;
  ACE_INET_Addr addr (port, server);

  if (connector.connect (*this, addr) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
  "%p\n",
  "open"),
  -1);
  initialized_ = 1;
  return 0;
}

/* The first of our put operators sends a simple string object to the
  peer.
  我们需要做的简单操作就是向对方发送一个简单的字符对象
*/
Client &
Client::operator<< (ACE_SString &str)
{
  /* We have to be able to allow: server << foo << bar << stuff;
  我们必须允许： sever<<<
  To accomplish that, every << operator must check that the object
     is in a valid state before doing work.
    为了实现这一点，所有的<<操作在开始之前都需要检查对象是否处于工作状态
   */

  if (initialized () && !error ())
  {
        /* Get the actual data held in the string object
          获得 string 对象中中说包含的实际数据
          */
  const char *cp = str.fast_rep ();
```

```
  /* Send that data to the peer using send_n() as before.  If we
  have a problem, we'll set error_ so that subsequent <<
  operations won't try to use a broken stream.
          向以前一样使用 send_n()方法发送数据。如果发送错误，我们将设置
           error_值，这样下一个<<操作就不会试图在这个断开的流上传输数据了。
        */
  if (this->send_n (cp,
  ACE_OS::strlen (cp)) == -1)
  error_ = 1;
  }
  else
  /* Be sure that error_ is set if somebody tries to use us when
  we're not initialized.
  当我们没有初始化时，任何人在试图调用我们时，确保 error_值被设置，
     */
  error_ = 1;
       /* We have to return a reference to ourselves to allow chaining of
put operations (eg -- "server << foo << bar").  Without the
  reference, you would have to do each put operation as a statement.
  That's OK but doesn't have the same feel as standard C++iostreams.
在这里我们需要返回自己来保证这里链式的调用(例如—"server<<")。
如果不返回这个应用，调用者就必须单独调用输出操作。这样做是可以的，但是
感觉不像标准 C＋＋ 输入输出流的操作。*/
  return *this ;
}

/* How do you put a char*?  We'll take an easy way out and construct
an ACE_SString from the char* and then put that.  It would have been
more efficient to implement this with the body of the
operator<<(ACE_SString&) method and then express that method in terms
of this one.  There's always more than one way to do things!
```

```
你传入一个 char* 参数嘛。 通过 ACE_Sstring 构造函数，我们可以将 char*的
数据放置其中。这样做是一个比较高效的作法，这样我们就可以复用
operator<<(ACE_Sstring&)中的方法。当然这里还有其他的实现方式。
*/

Client &
Client::operator<< (char *str)
{
  ACE_SString newStr (str);

  *this << newStr;

  return *this ;

  /* Notice that we could have been really clever and done:
  注意，我们可以使用更加简洁的写法
    return *this << ACE_SString (str);

     That kind of thing just makes debugging a pain though!
     这样做的坏处就是除错的过程中会很难受。
      */
}

/* ACE_SString and char* are both about the same thing.  What do you
  do about different datatypes though?
  ACE_Sstring 和 char* 都做了同样的事情。 对于其他不同的类型，你将
   会如何处理?

  Do the same thing we did with char* and convert it to ACE_SString
  where we already have a << operator defined.
  我们将向对象 char*以及进行处理，将数据转换成为 ACE_Sstring 对象，这样
   我们就可以用已有的<<操作符来进行处理了。
```

```
*/
Client &
Client::operator<< (int n)
{
  /* Create a character buffer large enough for the largest number.
     That's a tough call but BUFSIZ should be quite enough.
     创建一个足够大的字符串缓冲区，通过 BUFSIZ 来定义大小。
     */
  char buf[BUFSIZ];

  /* Put the number into our buffer...
  把数字放到我们的缓冲区中
   */
  ACE_OS::sprintf (buf,
  "(%d)\n",
  n);

  /* And create the ACE_SString that we know how to put.
       通过上面的方法来将其放置在 ACE_Sstring 对象中
     */
  ACE_SString newStr (buf);

  /* Send it and...
  将其发送出去
   */
  *this << newStr;

  /* return ourselves as usual.
       和往常一样返回
     */
  return *this;
}
```

```
/* Now we pull it all together.  Like Tutorial 3, we'll allow command
   line options.
   现在把所有的内容都放置好，和教程 3 一样，我们采用的命令行参数
*/
int
main (int argc, char *argv[])
{
  const char *server_host = argc > 1 ? argv[1] : ACE_DEFAULT_SERVER_HOST;
  u_short server_port = argc > 2 ? ACE_OS::atoi (argv[2]) : ACE_DEFAULT_SERVER_PORT;
  int max_iterations = argc > 3 ? ACE_OS::atoi (argv[3]) : 4;

  /* Use the basic constructor since the other isn't really very safe.
  使用基本的构造函数，这样比较安全
   */
  Client peer;

  /* Open the server connection.  Notice how this is simpler than
  Tutorial 3 since we only have to provide a host name and port
     value.
       建立与服务器的连接。注意，这里比教程 3 中的内容更加简单，因为
     我们只需要传入主机名和端口号。
  */
  if (peer.open (server_host,
    server_port) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
  "%p\n",
  "open"),
  -1);

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

```
  /* Tell the server which iteration we're on.   No more mucking
  around with sprintf at this level!  It's all hidden from us.
        告诉服务器我们现在迭代的次数,这里没有使用 sprintf 一级的代码。
      */
  peer << "message = " << i+1;

        /* Everything OK?
          一切正常?
         */
  if (peer.error ())
  ACE_ERROR_RETURN ((LM_ERROR,
  "%p\n",
  "send"),
    -1);
  else
  ACE_OS::sleep (1);
  }

  if (peer.close () == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
  "%p\n",
  "close"),
  -1);
  return 0;
}
```

哦，终于完成了。 正如你所见，创建一个对象可以使用比调用 send()或者 send_n()更加"自然"的方法来传送数据。你可以创建一个对象使用类似于 C++流模板的方式来传输数据。（我们将在后面的内容中涉及。）当然要想写一个完全可以和标准 C++输出流互操作的流，是非常困难的。还有这里的客户端还有需要优化的工作可以做。

作为对读者的扩展练习（你不会讨厌的了!）你可以把服务器端的代码修改成这样。如果你准备开始这样做，你需要好好看一下跟随 ACE 发行的 IOStream_Test 中的代码。

- **ACE Tutorial [翻译] 05 －page01**

2004-11-02

Tag：ACE_TAO

http://jnn.blogbus.com/logs/474165.html

迈向多线程服务器

在教程中，我们将简单回顾一下我们前面创建的那个简单的服务器。我们创建的简单服务器，是在一个线程中完成了所有的工作。当我们对这些内容有清晰的认识之后，我们将把目光转向下一个介绍并发概念的教程。

在本教程中有四个源文件：server.cpp，client_acceptor.h，client_handler.h 和 client_handler.cpp。和以往得方式一下我对这几个文件逐一进行注释。作为补充，我将简单介绍一下 Makefile 以及我新加的简单得 perl 脚本程序。

- **ACE Tutorial [翻译] 05 －page02**

2004-11-02

Tag：ACE_TAO

http://jnn.blogbus.com/logs/474169.html

我们先从 server.cpp 开始

Kirthika 所做得概述

这个教程是对客户－服务连接教程的翻新操作，使得代码更加简洁（例如：使用 destroy()方法来删除对象，并且 process()来负责从客户端获取响应的数据）。

这里我们还是使用 ACE_Reactor 来处理实际喜讯那，这些都在一个线程中进行处理。

这个教程是迈向多线程服务模式的基石。

——本教程内容和教程 2 很类似

```
// page02.html,v 1.12 2000/03/19 20:09:22 jcej Exp


/* We try to keep main() very simple.  One of the ways we do that is
   to push much of the complicated stuff into worker objects.  In this
   case, we only need to include the acceptor header in our main
   source file.  We let it worry about the "real work".
   我们尽量是 main()保持简单。 一个方法是你把复杂的操作都放在工作对象中。
   在这里，我们只需要在我们的主程序文件中，包含 acceptor 的头文件。让这个
   对象来完成真正的工作。
*/


#include "client_acceptor.h"


/* As before, we create a simple signal handler that will set our
   finished flag.  There are, of course, more elegant ways to handle
   program shutdown requests but that isn't really our focus right
```

now, so we'll just do the easiest thing.

和以往一样，我们创建了一个简单的信号量处理函数，用来设置 finished 标志位。

当然这里还有更加精细的关闭程序的方法（我也想知道），不过现在我们的目光是

让程序正确运行，所以我们在这里做了最简单的事情。

*/

```
static sig_atomic_t finished = 0;
extern "C" void handler (int)
{
  finished = 1;
}
```

/* A server has to listen for clients at a known TCP/IP port. The

default ACE port is 10002 (at least on my system) and that's good

enough for what we want to do here. Obviously, a more robust

application would take a command line parameter or read from a

configuration file or do some other clever thing. Just like the

signal handler above, though, that's not what we want to focus on,

so we're taking the easy way out.

服务器在此监听一个知名的 TCP/IP 端口。ACE 中缺省的端口号是 10002

（至少我的系统）是这样的，目前我们就需要知道这样。显然，一个更加健壮的应用需要通过命令行参数或者是读取配置文件的方式，或者是更加聪明的方法来实现。 正如对对上面的信号处理句柄一样，我们现在关心的重点简单的实现需求。

*/

```
static const u_short PORT = ACE_DEFAULT_SERVER_PORT;
```

/* Finally, we get to main. Some C++ compilers will complain loudly if your function signature doesn't match the prototype. Even though we're not going to use the parameters, we still have to specify them.

现在我们研究一下 main 函数是怎么写的。如果你的函数声明和原形不一致的话许多编译都会向你大声抱怨。 因此，即使我们不使用这些参数，我们也需要把把他们列出来。

*/

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

```
ACE_UNUSED_ARG(argc);

ACE_UNUSED_ARG(argv);



/* In our earlier servers, we used a global pointer to get to the
   reactor. I've never really liked that idea, so I've moved it into
   main() this time. When we get to the Client_Handler object you'll
    see how we manage to get a pointer back to this reactor.
    在早前的版本的服务器中，我们使用全局指针来获得 reactor。现在我不打算
    这么做了。在 Client_Handler 中，你将会看到我们是如何获得这个 reactor
    指针的。
*/

ACE_Reactor reactor;



/* The acceptor will take care of letting clients connect to us.  It
  will also arrange for a Client_Handler to be created for each new
  client.  Since we're only going to listen at one TCP/IP port, we
  only need one acceptor.  If we wanted, though, we could create
  several of these and listen at several ports.  (That's what we
   would do if we wanted to rewrite inetd for instance.)
   acceptor 负责处理客户的连接。对了每个新连接的客户，它都会为其创建一个
```

Client_Handler。

*/

Client_Acceptor peer_acceptor;

/* Create an ACE_INET_Addr that represents our endpoint of a

connection. We then open our acceptor object with that Addr.

Doing so tells the acceptor where to listen for connections.

Servers generally listen at "well known" addresses. If not, there

must be some mechanism by which the client is informed of the

server's address.

创建一个 ACE_INET_Addr 来标识我们连接的端点。我们在打开 acceptor 对象

的时候传入 Addr。这样就可以让 acceptor 开始通过指定的地址监听连接请求。

如果不这样，就需要其他的机制让客户端获得服务器端的地址。

Note how ACE_ERROR_RETURN is used if we fail to open the acceptor.

This technique is used over and over again in our tutorials.

注意 当我们的调用 acceptor 的 open 方法失败是，ACE_ERROR_RETURN 是如何

工作的。 这在我们的教程中大量使用。

*/

```
if (peer_acceptor.open (ACE_INET_Addr (PORT),
                        &reactor) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "%p\n",
                     "open"),
                    -1);

/* Here, we know that the open was successful.  If it had failed, we
   would have exited above.  A nice side-effect of the open() is that
   we're already registered with the reactor we provided it.
   这里我们的 open 操作已经成功，如果 open 操作失败，我们将会退出。open()
   的一个好的副作用就是执行该调用时，我们已经向 reactor 进行注册了。
   */

/* Install our signal handler.  You can actually register signal
   handlers with the reactor.  You might do that when the signal
   handler is responsible for performing "real" work.  Our simple
   flag-setter doesn't justify deriving from ACE_Event_Handler and
   providing a callback function though.
   安装我们的信号处理句柄。你也可以直接向 reactor 进行注册。这样做的目的
```

```
   时让信号处理句柄做一下现实一点的工作。这里的 handler 只是提供一个简单
   的回调方法，负责设置一下标志位，没有必要继承 ACE_Event_Handler。

   */
ACE_Sig_Action sa ((ACE_SignalHandler) handler, SIGINT);

/* Like ACE_ERROR_RETURN, the ACE_DEBUG macro gets used quite a bit.
   It's a handy way to generate uniform debug output from your
   program.
   和 ACE_ERROR_RETURN 一样，ACE_DEBUG 宏也被大量使用。
   这个宏的作用是在程序中生成一个统一的调试输出信息。
*/
ACE_DEBUG ((LM_DEBUG,
            "(%P|%t) starting up server daemon\n"));

/* This will loop "forever" invoking the handle_events() method of
   our reactor. handle_events() watches for activity on any
   registered handlers and invokes their appropriate callbacks when
   necessary.  Callback-driven programming is a big thing in ACE, you
   should get used to it. If the signal handler catches something,
```

```
    the finished flag will be set and we'll exit.  Conveniently
    enough, handle_events() is also interrupted by signals and will
    exit back to the while() loop.  (If you want your event loop to
    not be interrupted by signals, checkout the 'restart' flag on the
    open() method of ACE_Reactor if you're interested.)
    这里 reactor 采用的死循环的方式来调用 handle_event()方法。Handle_event()
    会照看那些注册的句柄，并在合适的时候，回调这些句柄。在 ACE 中大量用到回调
    驱动编程方法，你需要适应这一点。如果信号句柄被抓住，finished 标识会被设置，
    这样我们就可以退出了。handle_evnents()也会被信号中断，同时进入 while 判断中。
    (如果你想让事件循环不被信号量中断，你可以在 ACE_Reactor 的 open 方法中设置
     restart 标记。)
    */
  while (!finished)
    reactor.handle_events ();

  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) shutting down server daemon\n"));

  return 0;
}
```

```
#if defined (ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION)

template class ACE_Acceptor <Client_Handler, ACE_SOCK_ACCEPTOR>;

template class ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>;

#elif defined (ACE_HAS_TEMPLATE_INSTANTIATION_PRAGMA)

#pragma instantiate ACE_Acceptor <Client_Handler, ACE_SOCK_ACCEPTOR>

#pragma instantiate ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>

#endif /* ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION */
```

- **ACE Tutorial [翻译] 05 －page03**

2004-11-02

Tag：ACE_TAO

http://jnn.blogbus.com/logs/474176.html

现在让我们看一下 client_acceptor.h。 在这里我们将看到它是如何接收客户的的连接的，所有它应该会复杂吧！事实上，没有这么复杂。

随着你深入使用 ACE，你会发现你的已经需要了解更多 ACE 的细节。从接收客户端的连接请求来说: 应该不会有其他的方法来实现。ACE 团队选择采用 C++模板来帮你实现这些操作。你所需要做的只是提供接收新的连接请求的对象类型。

```
// page03.html,v 1.11 2000/03/19 20:09:22 jcej Exp
```

```
#ifndef CLIENT_ACCEPTOR_H

#define CLIENT_ACCEPTOR_H


/* The ACE_Acceptor<> template lives in the ace/Acceptor.h h
eader

   file. You'll find a very consitent naming convention betw
een the

   ACE objects and the headers where they can be found.  In
general,

   the ACE object ACE_Foobar will be found in ace/Foobar.h.

   ACE_Acceptor<>模板是在 ace/Acceptor.h 头文件中定义。你需要
根据 ACE

    对象定位其所在头文件。 通常情况下，ACE 对象 ACE_Foobar 是
在

    ace/Foobar.h 中定义

 */


#include "ace/Acceptor.h"


#if !defined (ACE_LACKS_PRAGMA_ONCE)
```

```
# pragma once

#endif /* ACE_LACKS_PRAGMA_ONCE */



/* Since we want to work with sockets, we'll need a SOCK_Acc
eptor to

   allow the clients to connect to us.

   由于我们需要喝 sokcets 打交道，我们需要 SOCK_Acceptor 接收客
户的连接请求。

*/

#include "ace/SOCK_Acceptor.h"



/* The Client_Handler object we develop will be used to hand
le clients

   once they're connected.  The ACE_Acceptor<> template's fi
rst

   parameter requires such an object.  In some cases, you ca
n get by

   with just a forward declaration on the class, in others y
ou have to

   have the whole thing.

   我们开发的 Client_Handler 对象负责处理 client 连接完成之后的
数据处理。
```

ACE_Acceptor<>模板的第一个参数就是这样的一个对象。在一些情况下，你

可以把这个类通过预定义的方式获得，有时候，你得给出这个类的完全定义。

*/

```
#include "client_handler.h"


/* Parameterize the ACE_Acceptor<> such that it will listen for socket
   connection attempts and create Client_Handler objects when they
   happen. In Tutorial 001, we wrote the basic acceptor logic on our
   own before we realized that ACE_Acceptor<> was available. You'll
   get spoiled using the ACE templates because they take away a lot of
   the tedious details!
```

向 ACE_Acceptor<>中传入的参数使得其可以监听 socket 的连接请求，并创建

Client_Handler 对象处理请求。 在教程 001 中，在选用 ACE_Acceptor<>之前，

我们实现了一个简单的 acceptor 逻辑。 使用 ACE 模板类是一个很暇逸的事情，

因为它们屏蔽了很多单调乏味的东西。

*/

```
typedef ACE_Acceptor <Client_Handler, ACE_SOCK_ACCEPTOR> Client_Acceptor;


#endif /* CLIENT_ACCEPTOR_H */
```

- **ACE Tutorial [翻译] 05 －page04**

2004-11-02

Tag：ACE_TAO

http://jnn.blogbus.com/logs/474180.html

好，现在我们进入 main 循环，看一下 acceptor 是如何初始化，同时感受它是如何简单地创建 acceptor 对象的。尽管做了这么多工作，但是我们几乎没有写什么代码。现在看一下要改变那些东西…

首先，我们看一下 Client_Handler.h 中所声明的 Client_Handler 对象。 接着我们将看一下与应用相关的定义是如何实现的。

```
// page04.html,v 1.13 2000/03/19 20:09:22 jcej Exp


#ifndef CLIENT_HANDLER_H
```

```
#define  CLIENT_HANDLER_H



/* Our client handler must exist somewhere in the ACE_Event_Handler
   object hierarchy.  This is a requirement of the ACE_Reactor because
   it maintains ACE_Event_Handler pointers for each registered event
   handler.  You could derive our Client_Handler directly from
   ACE_Event_Handler but you still have to have an ACE_SOCK_Stream for
   the actual connection.  With a direct derivative of
   ACE_Event_Handler, you'll have to contain and maintain an
   ACE_SOCK_Stream instance yourself.  With ACE_Svc_Handler (which is
   a derivative of ACE_Event_Handler) some of those details are
   handled for you.
   我们的客户处理句柄必须要处于 ACE_Event_Handler 的继承树中。
    这是由于 ACE_Reactor 需要为每个注册的事件句柄维护 ACE_Event_Handler 指针。
    你将你的 Client_Handler 派生于 ACE_Event_Handler,但是你还需要为实际的连接
    包含 ACE_SOCK_Stream 成员。如果直接派生于 ACE_Event_Handler,你自己需要包含
    和维护一个 ACE_SOCK_Stream 实例。如果使用 ACE_Svc_Handler（ACE_Event_Handler
    的子类），你就不需要考虑这么多的细节了。
  */
```

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


#if !defined (ACE_LACKS_PRAGMA_ONCE)
# pragma once
#endif /* ACE_LACKS_PRAGMA_ONCE */


#include "ace/SOCK_Stream.h"


/* Another feature of ACE_Svc_Handler is it's ability to present the
   ACE_Task<> interface as well.  That's what the ACE_NULL_SYNCH
   parameter below is all about.  That's beyond our scope here but
   we'll come back to it in the next tutorial when we start looking at
   concurrency options.
   ACE_Svc_Handler 还提供了另外一个功能，就是它也支持 ACE_Task<>接口。
   这就是为什么还要传入 ACE_NULL_SYNCH 参数了。为什么要使用 ACE_NULL_SYNCH
   已经超出了我们目前讨论的范围，具体的内容我们会再下面有关讨论同步选项的
   教程中有所涉及。
*/
class Client_Handler : public ACE_Svc_Handler <ACE_SOCK_STREAM, ACE_NULL_SYN
CH>
```

```
{
public:
  // Constructor... 构造函数
  Client_Handler  (void);

  /* The destroy() method is our preferred method of destruction.  We
     could have overloaded the delete operator but that is neither easy
     nor intuitive (at least to me).  Instead, we provide a new method
     of destruction and we make our destructor protected so that only
     ourselves, our derivatives and our friends can delete us. It's a
     nice compromise.
     Destroy 方法实现的是有关析构的操作。我们可以通过重载析构函数来实现，
     但这对于我们来说不够简单也不够直观。 因此我们选择了另外一个新的析构
     的丰富，将我们的析构函数设置为保护只能供我们调用。这是一个比较好的
     折中方案。
     */
  void destroy (void);

  /* Most ACE objects have an open() method.  That's how you make them
     ready to do work.  ACE_Event_Handler has a virtual open() method
```

which allows us to create an override. ACE_Acceptor<> will invoke

this method after creating a new Client_Handler when a client

connects. Notice that the parameter to open() is a void*. It just

so happens that the pointer points to the acceptor which created

us. You would like for the parameter to be an ACE_Acceptor<>* but

since ACE_Event_Handler is generic, that would tie it too closely

to the ACE_Acceptor<> set of objects. In our definition of open()

you'll see how we get around that.

需要的 ACE 对象都提供了 open 方法，通过这个方法我们来初始化对象。

ACE_Event_Hander 定义了一个虚拟的 open 方法允许我们进行重载。ACE_Acceptor

将会在客户连接，创建新的 Client_Handler 的时候调用这个方法。注意 oepn 方法

的参数是 void*. 通过这个参数我们将 acceptor 传递到 open 方法中。你也许会

觉得采用 ACE_Acceptor<>* 更恰当一下，但是这样的话，会使得程序对

ACE_Acceptor<>耦合度提高。

*/

int open (void *acceptor);

/* When there is activity on a registered handler, the

handle_input() method of the handler will be invoked. If that

method returns an error code (eg -- -1) then the reactor will

invoke handle_close() to allow the object to clean itself

up. Since an event handler can be registered for more than one

type of callback, the callback mask is provided to inform

handle_close() exactly which method failed. That way, you don't

have to maintain state information between your handle_* method

calls. The <handle> parameter is explained below... As a

side-effect, the reactor will also invoke remove_handler() for the

object on the mask that caused the -1 return. This means that we

don't have to do that ourselves!

当注册的句柄所对应的事件发生是，这个句柄的 handle_input()就会被调用。

如果这个方法返回错误（如—— -1），reactor 会调用 handle_close（）方法，

来让对象执行一些有关清理的操作。由于事件句柄可以注册不只一种函数，

在调用 handle_close()时，需要指定回调掩码，通过这个掩码指定时那个回调

方法调用失败。这样你就不需要为你的 handle_*方法维护一个状态信息了。

<handle>参数，将在下面进行介绍。作为副作用，当掩码所对应的对象出现

错误时，reactor 也会调用 remove_handler()。这意味着我们不需要处理这些内容。

*/

int handle_close (ACE_HANDLE handle,

ACE_Reactor_Mask mask);

protected:

/* When we register with the reactor, we're going to tell it that we want to be notified of READ events. When the reactor sees that there is read activity for us, our handle_input() will be invoked. The _handle provided is the handle (file descriptor in Unix) of the actual connection causing the activity. Since we're derived from ACE_Svc_Handler<> and it maintains its own peer (ACE_SOCK_Stream) object, this is redundant for us. However, if we had been derived directly from ACE_Event_Handler, we may have chosen not to contain the peer. In that case, the <handle> would be important to us for reading the client's data.

当我们向 reactor 进行注册时，我们需要向它通知我们只关系 READ 事件。当 reactor 获知这些读活动时，我们的 handle_input（）就会被调用。 Handle 参数，是一个能够处理连接的句柄（在 unix 中就是一个文件描述符）。 由于我们是派生于 ACE_Svc_Handler<> ,它维护了一个自己的端对象（ACE_SOCK_Stream）对象，这样就会加重我们的负担。然而，如果我们需要自己从 ACE_Event_Handler，直接派生的话，我们也许会选择不包含端对象。在这个例子中，handle 参数对于我们获取客户端的数据非常重要。

*/

```
int handle_input (ACE_HANDLE handle);

/* This has nothing at all to do with ACE.  I've added this here as
   a worker function which I will call from handle_input().  That
   allows me to introduce concurrency in later tutorials with no
   changes to the worker function.  You can think of process() as
   application-level code and everything else as
   application-framework code.
   这与 ACE 没有任何关系。添加这个方法是为了能在 handle_input()时，调用
   一个工作方法。这样可以在我们以后的介绍有关同步的教程中，不需要改变
   相关的工作方法。 你可以认为 process()是一个应用一级的代码，或者应用
   框架部分的代码。
   */
int process (char *rdbuf, int rdbuf_len);

/* We don't really do anything in our destructor but we've declared
   it to be protected to prevent casual deletion of this object.  As
   I said above, I really would prefer that everyone goes through the
   destroy() method to get rid of us.
   在析构函数中，我们不需要做什么事情。但是我们这样声明，是为了将其作为
```

```
    保护方法，意外防止删除对象。正如我在上面所说，我们希望将大家通过 destroy
    方法来释放我们。
     */
  ~Client_Handler   (void);
};


#endif  /* CLIENT_HANDLER_H */
```

- **ACE Tutorial [翻译] 05 －page05**

2004-11-02

Tag：ACE_TAO

现在我们开始关注 Client_handler.cpp,在这里主要完成大部分与应用相关的代码。

```
// page05.html,v  1.13  2000/03/19  20:09:22  jcej  Exp


/* In client_handler.h  I  alluded  to  the fact  that we'll  mess  around
    with a  Client_Acceptor pointer.  To do  so,  we  need  the
    Client_Acceptor  object  declaration.
    在 Client_handler.h 中我提到了我们需要和 Client_Acceptor 指针进行周旋。
```

为了这么做，我们需要使用 Client_Acceptor 对象的声明。

We know that including client_handler.h is redundant because client_acceptor.h includes it. Still, the sentry prevents double-inclusion from causing problems and it's sometimes good to be explicit about what we're using.

我们知道包含 client_Handler.h 会使得 client_acceptor.h 重复包含。

实际上通过#ifdef，我们可以防止 include 文件的二次包含。

On the other hand, we don't directly include any ACE header files here.

另一方面，我们这并不直接包含 ACE 头文件，

*/

```
#include "client_acceptor.h"
#include "client_handler.h"

/* Our constructor doesn't do anything. That's generally a good idea.
    Unless you want to start throwing exceptions, there isn't a really
    good way to indicate that a constructor has failed. If I had my
    way, I'd have a boolean return code from it that would cause new to
```

```
    return 0 if I failed.  Oh well...
    我们的构造函数并不执行任何操作。这里有一些比较好的点子，除非你想
     抛出异常，在这里指定构造函数失败是不明智的。如果我来做，我会设置
     一个返回值，当创建失败时，返回 0。
*/
Client_Handler::Client_Handler  (void)
{
}

/* Our destructor doesn't do anything either.  That is also by design.
    Remember, we really want folks to use destroy() to get rid of us.
    If that's so, then there's nothing left to do when the destructor
    gets invoked.
    我们的析构函数也不做什么事情。这是的操作是设计时指定的。记着我们只
   希望使用 destroy()方法来执行退出的操作。这样，在析构函数中，就不需要
   再执行其他的操作了。
*/
Client_Handler::~Client_Handler  (void)
{
  // Make sure that our peer closes when we're deleted.  This
```

```
  // will probably happened when the peer is deleted but it
  // doesn't hurt to be explicit.
  // 保证在对象析构的时候，会关闭对方的连接。
  // 这一般发生这在对方连接被删除，但是这不会产生直接的影响。
  【这句话还是有点问题】
  this->peer ().close ();
}

/* The much talked about destroy() method!  The reason I keep going on
   about this is because it's just a Bad Idea (TM) to do real work
   inside of a destructor.  Although this method is void, it really
   should return int so that it can tell the caller there was a
   problem.  Even as void you could at least throw an exception which
   you would never want to do in a destructor.
   这里要开始大篇幅讨论 destroy()方法。 我打算这么做的目的是，在把析构函数
   中做大量的现实工作是一个非常坏的主意。虽然这个方法没有返回值，它确实应该
   返回 int 值，这样当调用出错的时候，可以通知调用这。虽然对于 void 你可以采用
   扔异常的方式来通知错误，但是我们在析构函数中从来不这么做。
*/
void
```

```
Client_Handler::destroy (void)
{
  /* Tell the reactor to forget all about us.  Notice that we use the
     same args here that we use in the open() method to register
     ourselves.  In addition, we use the DONT_CALL flag to prevent
     handle_close() being called.  Since we likely got here due to
     handle_close(), that could cause a bit of nasty recursion!
     通知 reactor 将我们遗忘（分手了！）。注意我们需要和负责注册 open 方法
     传人的相同的参数。还有，我们使用了 DONT_CALL 表示来防止调用 handle_close。
     这是由于我们是通过调用 handle_close()进入到这条语句的，所以需要防止
     比较麻烦的递归的调用。
  */
  this->reactor ()->remove_handler (this,
                                    ACE_Event_Handler:: READ_MASK | ACE_Event_
Handler::DONT_CALL);

  /* This is how we're able to tell folks not to use delete.  By
     deleting our own instance, we take care of memory leaks after
     ensuring that the object is shut down correctly.
     这里我们给大家介绍了不使用 delete 的原因。只要保证对象被正确关闭，
```

```
    通过删除我们自己的实例，就可以防止内存泄漏。
  */
  delete this;
}

/* As mentioned before, the open() method is called by the
   Client_Acceptor when a new client connection has been accepted.
   The Client_Acceptor instance pointer is cast to a void* and given
   to us here.  We'll use that to avoid some global data...
  正如前面所提到的，open 方法可以在 Client_Acceptor 接收新的客户连接是被调用。
  Client_Acceptor 的实例指针可以被转换成为 void*,并传递给我们。这样我们就可以
  减少全局数据的使用。
*/
int
Client_Handler::open (void *_acceptor)
{
  /* Convert the void* to a Client_Acceptor*.  You should probably use
     those fancy ACE_*_cast macros but I can never remember how/when
     to do so.  Since you can cast just about anything around a void*
     without compiler warnings be very sure of what you're doing when
```

you do this kind of thing. That's where the new-style cast

operators can save you.

将 void* 参数转换为 Client_Acceptor*。 在这里你可能需要使用 ACE_*_cast 宏

但是我还没有搞清楚什么时候，如何用这些东西。因为你可以把 void*和任何类型

数据进行互相转换，而且这样做编译器并不会报任何警告，所以你需要对你所做的

事情特别清楚。这是新风格的转换符所能代给你最有魅力的地方。

*/

Client_Acceptor *acceptor = (Client_Acceptor *) _acceptor;

/* Our reactor reference will be set when we register ourselves but

I decided to go ahead and set it here. No good reason really...

当我们向 reacotr 注册的时候，我们的 reactor 引用是会被初始化的，但是我

还是决定在这就把事情给先干了。 这没有什么特别的原因...

*/

this->reactor (acceptor->reactor ());

/* We need this to store the address of the client that we are now

connected to. We'll use it later to display a debug message.

我们需要把连接我们的客户端的地址记录下来。我们需要在调试信息中，

把这些内容输出。

```
*/
  ACE_INET_Addr addr;


 /* Our ACE_Svc_Handler baseclass gives us the peer() method as a way
     to access our underlying ACE_SOCK_Stream.  On that object, we can
     invoke the get_remote_addr() method to get an ACE_INET_Addr
     having our client's address information.  As with most ACE
     methods, we'll get back (and return) a -1 if there was any kind
     of error.  Once we have the ACE_INET_Addr, we can query it to
     find out the clien's host name, TCP/IP address, TCP/IP port value
     and so forth.  One word of warning: the get_host_name() method of
     ACE_INET_Addr may return you an empty string if your name server
     can't resolve it.  On the other hand, get_host_addr() will always
     give you the dotted-decimal string representing the TCP/IP
     address.
     我们的 ACE_Svc_Handler 基类为我们通过的 peer()方法,通过这个方法，我们可以
     访问 ACE_SOCK_Stream 成果。对于这个对象，我们可以通过调用 get_remote_addr()
     方法来获得 ACE_INET_Addr，并且获得客户端的地址信息。正如大多数的 ACE 方法一
样，
     我们通过返回值-1 来说明调用过程中所发生的错误。一旦我们拥有了 ACE_INET_Addr。
```

我们可以通过调用它来获得客户端主机的名字，TCP/IP 地址，TCP/IP 端口号，或者其他

什么的。在这里需要提醒的是，ACE_INET_Addr 所提供的，get_host_name()方法当你的

域名服务器无法解析的时候，可能会返回空值。 另外，get_host_addr()会返回给你带

点和数字组成的 TCP/IP 地址串。

*/

```
if (this->peer ().get_remote_addr (addr) == -1)
  return -1;
```

/* If we managed to get the client's address then we're connected to

a real and valid client. I suppose that in some cases, the

client may connect and disconnect so quickly that it is invalid

by the time we get here. In any case, the test above should

always be done to ensure that the connection is worth keeping.

如果我们能获得客户端地址，那么我们就于一个真实的客户端建立连接了。

在这里可能还会出现其他的情况，客户可能连接上了，但是很快的断开，

这样当我们走到这一步的时候，它就无效了。 无论如何，我们就需要

测试一下，以保证获得到的连接是值的保存的。

Now, register ourselves with a reactor and tell that reactor that

```
    we want to be notified when there is something to read.
    Remember, we took our reactor value from the acceptor which
    created us in the first place.  Since we're exploring a
    single-threaded implementation, this is the correct thing to do.
    现在向 reactor 进行注册，并且告诉 reactor 我们希望对什么事件感兴趣。
    注意，我们是通过前面的 acceptor 获得的 reactor 值。由于我们现在阶段
    还是单线程的实现，所以下面的写法是正确的。
*/
if (this->reactor ()->register_handler (this,
                                        ACE_Event_Handler::READ_MASK) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "(%P|%t) can't register with reactor\n"),
                    -1);

/* Here, we use the ACE_INET_Addr object to print a message with the
    name of the client we're connected to.  Again, it is possible
    that you'll get an empty string for the host name if your DNS
    isn't configured correctly or if there is some other reason that
    a TCP/IP addreess cannot be converted into a host name.
    现在我们使用 ACE_INET_Addr 对象来打印连接到我们的客户端的名字。
```

```
        注意，如果你的 DNS 没有配置正确，或者是其他的什么原因影响到了
        TCP/IP 地址到主机名的转换，那么你会获得一个空串。
    */
    ACE_DEBUG ((LM_DEBUG,
                "(%P|%t) connected with %s\n",
                addr.get_host_name ()));

    /* Always return zero on success.
     在这里老是返回零 */
    return 0;
}

/* In the open() method, we registered with the reactor and requested
    to be notified when there is data to be read. When the reactor
    sees that activity it will invoke this handle_input() method on us.
    As I mentioned, the _handle parameter isn't useful to us but it
    narrows the list of methods the reactor has to worry about and the
    list of possible virtual functions we would have to override.
    在 open 方法中，我们向 reactor 注册并且请求获知我们所关心的读取数据事件。
    正如我前面讲的，_handle 参数对于我们没有用处，但是它减少了 reactor 需要
```

```
    关心的方法以及我们需要照看的虚函数。

*/

int

Client_Handler::handle_input  (ACE_HANDLE  handle)

{

  /* Some  compilers  don't  like  it  when  you  fail  to  use  a  parameter.

      This macro  will  keep  'em  quiet  for  you.

    一些编译器会警告你还还有未使用的参数，使用这个宏，可以帮你屏蔽

    这个警告。

*/

  ACE_UNUSED_ARG  (handle);



  /* Now,  we  create  and  initialize  a  buffer  for  receiving  the  data.

      Since  this  is  just  a  simple  test app, we'll  use  a  small  buffer

      size.

     现在，我们为接收的数据初始化一块那查。由于我们是一个简单的测试应用，

     我们使用的一个比较小的缓冲区。

 */

  char  buf[BUFSIZ];
```

```
  /* Invoke the process() method with a pointer to our data area.
     We'll let that method worry about interfacing with the data.  You
     might choose to go ahead and read the data and then pass the
     result to process().  However, application logic may require that
     you read a few bytes to determine what else to read...  It's best
     if we push that all into the application-logic level.
     调用 process()方法来处理我们的数据。我们把和数据交互的内容都交给那个方法。
     你可以选择在这里直接读取数据，并把结果值传递给 process。然而，应用的逻辑需要
     我们读取一段数据来决定下面需要读取的内容... 所以把所有这些都放到应用层是
     一个好主意。
  */
  return this->process (buf, sizeof (buf));
}

/* If we return -1 out of handle_input() or if the reactor sees other
   problems with us then handle_close() will be called.  The reactor
   framework will take care of removing us (due to the -1), so we
   don't need to use the destroy() method.  Instead, we just delete
   ourselves directly.
   如果我们在 handle_input()返回-1 或者 reactor 在执行过程中其他问题，那么
```

```
    handle_close()就会被调用。Reactor 框架会负责删除我们（因为返回值-1）。
    所以我们不必使用 destroy()方法。作为替代，我们会直接删除我们自己。
*/
int
Client_Handler::handle_close (ACE_HANDLE handle,
                              ACE_Reactor_Mask mask)
{
  ACE_UNUSED_ARG (handle);
  ACE_UNUSED_ARG (mask);

  delete this;
  return 0;
}

/* And, at last, we get to the application-logic level.  Out of
    everything we've done so far, this is the only thing that really
    has anything to do with what your application will do.  In this
    method we will read and process the client's data.  In a real
    appliation, you will probably have a bit more in main() to deal
    with command line options but after that point, all of the action
```

```
    takes place here.
    现在我们终于进入应用逻辑层了。与我们做了这么多外围的事情相比，这里
    只包含了我们的应用需要做的事情。在这个方法中，我们需要读取客户的数据。
    对于一个真正的应用来说，你需要在 main()函数中处理命令行参数之后，将
    目光集中在这。
 */
int
Client_Handler::process (char *rdbuf,
                         int rdbuf_len)
{
    ssize_t bytes_read = -1;

  /* Using the buffer provided for us, we read the data from the
     client. If there is a read error (eg -- recv() returns -1) then
     it's a pretty good bet that the connection is gone. Likewise, if
     we read zero bytes then something wrong has happened. The
     reactor wouldn't have called us if there wasn't some kind of read
     activity but there wouldn't be activity if there were no bytes to
     read...
     使用提供给我的缓冲区，我们从客户端读取数据。如果这里发生读取错误
```

（例如–recv()返回－1）那么多半是连接断开了。同样，如果你读取到 0 个

字节，也说明连接断口了。Reactor 只在发生读取数据事件时通知我们的，但是

对于没有数据获取时，Reactor 是不会调用我们的。

【读取到 0 字节其实也是接收了数据，只不过应用层的数据长度为 0】

On the other hand, if we got some data then we can display it in

a debug message for everyone to see.

另一方面，如果我们接收到数据，那么我们打印出一条调试信息。

*/

```
switch ( (bytes_read = this->peer ().recv (rdbuf, rdbuf_len)) )
{
case -1: // Complain and leave
  ACE_ERROR_RETURN ((LM_ERROR,
                     "(%P|%t) %p bad read\n",
                     "client"),
                    -1);
case 0: // Complain and leave
  ACE_ERROR_RETURN ((LM_ERROR,
                     "(%P|%t) closing daemon (fd = %d)\n",
                     this->get_handle ()),
```

```
                                -1);

      default:  // Show  the data

          // NULL-terminate  the  string  before  printing it.

        rdbuf[bytes_read] =  0;

        ACE_DEBUG  ((LM_DEBUG,

                     "(%P|%t) from  client:  %s",

                     rdbuf));

      }



    /* It's  also  worth mentioning  that recv() has  a  cousin: recv_n().

       recv_n()  will receive  exactly  the number  of  bytes you provide it.

       This is  very  good  when  you know exactly  how  much  you expect  to
```

- **ACE Tutorial [翻译] 06 －page01**

2004-11-21

Tag：ACE_TAO

http://jnn.blogbus.com/logs/508334.html

在这个教程中，我们在教程 5 的基础上进行扩展，创建一线程每连接的服务器。这个实现将为每一个连接到我们的客户端创建一个新的线程。ACE_Reactor 在这里还是被使用，但目前只负责处理接收新的连接。Client_Handler 对象将不向 reactor 进行注册，而是只负责直接监听他们的 peer()。

概述：

这里我们将创建一个连接每线程的简单服务器。这是一个单线程服务器向多线程服务器转变的第一步。

在这里我们将使用 Strategy 模式。ACE_Acceptor 继承与 ACE_Acceptor_Base 类，这样就可以根据服务器使单线程还是每个连接一个线程，来实现不同的并发策略。这样也允许我们能够将来扩展服务器功能，在以实现不同的策略。

这些连接信息都是向 Client_Handler 传递的（还记得 ACE_Acceptor< Client_Handler, ACE_SOCK_ACCEPTOR>吗？）。那个 Client_Handler 是一个派生于 Event_Handler 的 ACE_Svc_Handler，并且 Client_Handler 还与 ACE_Sock_Stream 相关联。同时它（Client_Handler）也派生于 ACE_Task 类，这样来允许我们实现一线程每连接的工作方式。

我们在 svc 方法中实现数据的交互处理，svc 方法被一线程每连接服务器的线程所调用。

注意这里所有的 Client_Handler 对象都没有向 reactor 进行注册。Reactor 在这里只负责接收客户的连接。

一旦为连接建立线程，Client Handler 对象就需要为客户连接负责，并且取代 reactor 的工作，处理后面的事件。

* Abstract by Kirthika as always

- **ACE Tutorial [翻译] 06 －page02**

2004-11-21

Tag：ACE_TAO

现在我们还是从 server.cpp 开始。 如果你仔细观察就可以发现，这里和教程 5 实现的差别就是一个简单的注释。

```
// page02.html,v 1.10 2000/03/19 20:09:23 jcej Exp

/* We try to keep main() very simple.  One of the ways we do that is
   to push much of the complicated stuff into worker objects.  In this
   case, we only need to include the acceptor header in our main
   source file.  We let it worry about the "real work".
   我们尽量是 main()保持简单。一个方法是你把复杂的操作都放在工作对象中。
   在这里，我们只需要在我们的主程序文件中，包含 acceptor 的头文件。让这个
   对象来完成真正的工作。

*/

#include "client_acceptor.h"

/* As before, we create a simple signal handler that will set our
   finished flag.  There are, of course, more elegant ways to handle
   program shutdown requests but that isn't really our focus right
   now, so we'll just do the easiest thing.
   和以往一样，我们创建了一个简单的信号量处理函数，用来设置 finished 标志位。
```

当然这里还有更加精细的关闭程序的方法（我也想知道），不过现在我们的目光是

让程序正确运行，所以我们在这里做了最简单的事情。

*/

```
static sig_atomic_t finished = 0;
extern "C" void handler (int)
{
  finished = 1;
}
```

/* A server has to listen for clients at a known TCP/IP port. The

default ACE port is 10002 (at least on my system) and that's good

enough for what we want to do here. Obviously, a more robust

application would take a command line parameter or read from a

configuration file or do some other clever thing. Just like the

signal handler above, though, that's what we want to focus on, so

we're taking the easy way out.

服务器在此监听一个知名的 TCP/IP 端口。ACE 中缺省的端口号是 10002

（至少我的系统）是这样的，目前我们就需要知道这样。显然，一个更加

健壮的应用需要通过命令行参数或者是读取配置文件的方式，或者是更加

聪明的方法来实现。 正如对对上面的信号处理句柄一样，我们现在关心的

重点简单的实现需求。

*/

static const u_short PORT = ACE_DEFAULT_SERVER_PORT;

/* Finally, we get to main. Some C++ compilers will complain loudly

if your function signature doesn't match the prototype. Even

though we're not going to use the parameters, we still have to

specify them.

现在我们研究一下 main 函数是怎么写的。如果你的函数声明和原形不一致的话

许多编译都会向你大声抱怨。 因此，即使我们不使用这些参数，我们也需要把

把他们列出来。

*/

int

main (int argc, char *argv[])

{

```
ACE_UNUSED_ARG(argc);

ACE_UNUSED_ARG(argv);



/* In our earlier servers, we used a global pointer to get to the

   reactor. I've never really liked that idea, so I've moved it into

   main() this time. When we get to the Client_Handler object you'll

    see how we manage to get a pointer back to this reactor.

      在早前的版本的服务器中，我们使用全局指针来获得 reactor。现在我不打算

      这么做了。在 Client_Handler 中，你将会看到我们是如何获得这个 reactor

      指针的。

    */

ACE_Reactor reactor;



/* The acceptor will take care of letting clients connect to us.  It

   will also arrange for a Client_Handler to be created for each new

   client.  Since we're only going to listen at one TCP/IP port, we

   only need one acceptor.  If we wanted, though, we could create

   several of these and listen at several ports.  (That's what we

    would do if we wanted to rewrite inetd for instance.)

    acceptor 负责处理客户的连接。对了每个新连接的客户，它都会为其创建一个
```

Client_Handler。由于我们只是打算监听一个 TCP/IP 端口，我们只需要使用

一个 acceptor。如果我们愿意，我们可以创建许多这样的监听端口。（这样

我们可以重新实现 inetd）

*/

Client_Acceptor peer_acceptor;

/* Create an ACE_INET_Addr that represents our endpoint of a

connection. We then open our acceptor object with that Addr.

Doing so tells the acceptor where to listen for connections.

Servers generally listen at "well known" addresses. If not, there

must be some mechanism by which the client is informed of the

server's address.

创建一个 ACE_INET_Addr 来标识我们连接的端点。我们在打开 acceptor 对象

的时候传入 Addr。这样就可以让 acceptor 开始通过指定的地址监听连接请求。

如果不这样，就需要其他的机制让客户端获得服务器端的地址。

Note how ACE_ERROR_RETURN is used if we fail to open the acceptor.

This technique is used over and over again in our tutorials.

注意 当我们的调用 acceptor 的 open 方法失败是，ACE_ERROR_RETURN 是如何

```
    工作的。 这在我们的教程中大量使用。

     */

if (peer_acceptor.open (ACE_INET_Addr (PORT),

                                &reactor) == -1)

  ACE_ERROR_RETURN ((LM_ERROR,

                     "%p\n",

                     "open"),

                    -1);



/* As with Tutorial 5, we know that we're now registered with our

   reactor so we don't have to mess with that step.

   和教程 5 一样，现在我们向 reactor 进行注册，目前我们还不需要做什么改变。

   */



/* Install our signal handler.  You can actually register signal

   handlers with the reactor.  You might do that when the signal

   handler is responsible for performing "real" work.  Our simple

   flag-setter doesn't justify deriving from ACE_Event_Handler and

   providing a callback function though.

   安装我们的信号处理句柄。你也可以直接向 reactor 进行注册。这样做的目的
```

```
    时让信号处理句柄做一下现实一点的工作。这里的 handler 只是提供一个简单
    的回调方法，负责设置一下标志位，没有必要继承 ACE_Event_Handler。
    */
ACE_Sig_Action sa ((ACE_SignalHandler) handler, SIGINT);

/* Like ACE_ERROR_RETURN, the ACE_DEBUG macro gets used quite a bit.
   It's a handy way to generate uniform debug output from your
    program.
    和 ACE_ERROR_RETURN 一样，ACE_DEBUG 宏也被大量使用。
    这个宏的作用是在程序中生成一个统一的调试输出信息。
    */
ACE_DEBUG ((LM_DEBUG,
            "(%P|%t) starting up server daemon\n"));

/* This will loop "forever" invoking the handle_events() method of
   our reactor. handle_events() watches for activity on any
   registered handlers and invokes their appropriate callbacks when
   necessary. Callback-driven programming is a big thing in ACE, you
   should get used to it. If the signal handler catches something,
   the finished flag will be set and we'll exit. Conveniently
```

enough, handle_events() is also interrupted by signals and will

exit back to the while() loop. (If you want your event loop to

not be interrupted by signals, checkout the restart flag on

the open() method of ACE_Reactor if you're interested.)

这里 reactor 采用的死循环的方式来调用 handle_event()方法。Handle_event()

会照看那些注册的句柄，并在合适的时候，回调这些句柄。在 ACE 中大量用到回调

驱动编程方法，你需要适应这一点。如果信号句柄被抓住，finished 标识会被设置，

这样我们就可以退出了。handle_evnents()也会被信号中断，同时进入 while 判断中。

(如果你想让事件循环不被信号量中断，你可以在 ACE_Reactor 的 open 方法中设置

restart 标记。）

*/

```
while (!finished)
  reactor.handle_events ();

ACE_DEBUG ((LM_DEBUG,
            "(%P|%t) shutting down server daemon\n"));

return 0;
}
```

```
#if !defined(ACE_HAS_GNU_REPO)

#if defined (ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION)

template class ACE_Acceptor <Client_Handler, ACE_SOCK_ACCEPTOR>;

template class ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>;

#elif defined (ACE_HAS_TEMPLATE_INSTANTIATION_PRAGMA)

#pragma instantiate ACE_Acceptor <Client_Handler, ACE_SOCK_ACCEPTOR>

#pragma instantiate ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>

#endif /* ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION */

#endif /* ACE_HAS_GNU_REPO */
```

我们继续教程，看一下 Client_Accpetor 所发生的内容。

• ACE Tutorial [翻译] 06 －page03

2004-11-21

Tag：ACE_TAO

在 client_acceptor.h，我们需要对我们的对象进行扩展。这主要的原因是允许我们选择单线程的实现或者采用一个线程每连接的实现。 Client_Acceptor 自己不需要知道这些现在，但是 Client_Handler 对象在创建的时候，需要了解这些内容。如果我们想使用一个单线程策略实现，我们就不需要对教程 5 中的这个文件进行修改。

```
// page03.html,v 1.10 2000/03/19 20:09:23 jcej Exp


#ifndef CLIENT_ACCEPTOR_H

#define CLIENT_ACCEPTOR_H


/* The ACE_Acceptor<> template lives in the ace/Acceptor.h header
   file. You'll find a very consistent naming convention between the
   ACE objects and the headers where they can be found. In general,
   the ACE object ACE_Foobar will be found in ace/Foobar.h.
   ACE_Acceptor<>模板是在 ace/Acceptor.h 头文件中定义。你需要根据 ACE
    对象定位其所在头文件。 通常情况下，ACE 对象 ACE_Foobar 是在
    ace/Foobar.h 中定义
*/


#include "ace/Acceptor.h"


#if !defined (ACE_LACKS_PRAGMA_ONCE)

# pragma once

#endif /* ACE_LACKS_PRAGMA_ONCE */
```

```
/* Since we want to work with sockets, we'll need a SOCK_Acceptor to
   allow the clients to connect to us.
   由于我们需要喝 sokcets 打交道，我们需要 SOCK_Acceptor 接收客户的连接请求。
 */
#include "ace/SOCK_Acceptor.h"

/* The Client_Handler object we develop will be used to handle clients
   once they're connected.  The ACE_Acceptor<> template's first
   parameter requires such an object.  In some cases, you can get by
   with just a forward declaration on the class, in others you have to
   have the whole thing.
   我们开发的 Client_Handler 对象负责处理 client 连接完成之后的数据处理。
   ACE_Acceptor<>模板的第一个参数就是这样的一个对象。在一些情况下，你
  可以把这个类通过预定义的方式获得，有时候，你得给出这个类的完全定义。
 */
#include "client_handler.h"

/* Parameterize the ACE_Acceptor<> such that it will listen for socket
   connection attempts and create Client_Handler objects when they
   happen. In Tutorial 001, we wrote the basic acceptor logic on our
```

own before we realized that ACE_Acceptor<> was available. You'll get spoiled using the ACE templates because they take away a lot of the tedious details!

向 ACE_Acceptor<>中传入的参数使得其可以监听 socket 的连接请求，并创建 Client_Handler 对象处理请求。 在教程 001 中，在选用 ACE_Acceptor<>之前，我们实现了一个简单的 acceptor 逻辑。 使用 ACE 模板类是一个很暇逸的事情，因为它们屏蔽了很多单调乏味的东西。

*/

```
typedef ACE_Acceptor <Client_Handler, ACE_SOCK_ACCEPTOR> Client_Acceptor_Base;
```

/* Here, we use the parameterized ACE_Acceptor<> as a baseclass for our customized Client_Acceptor object. I've done this so that we can provide it with our choice of concurrency strategies when the object is created. Each Client_Handler it creates will use this information to determine how to act. If we were going to create a system that was always thread-per-connection, we would not have bothered to extend Client_Acceptor.

在这里，我们实现参数化的 ACE_Acceptor<>作为我们客户化的 Client_Acceptor 对象。我这么做这样我们可以在我们创建并发策略的时候有选择的余地。每个 Client_Handler 根据这些信息创建并觉得如何工作。 如果我们打算创建一个一线程每连接的系统，我们

就不需要再对 Client_Acceptor 进行扩展了。

*/

class Client_Acceptor : public Client_Acceptor_Base

{

public:

/*

This is always a good idea. If nothing else, it makes your code more

orthogonal no matter what baseclasses your objects have.

这（通过创建派生类的方法来扩展）是一个好主意。即使没有做什么事情，这也使得

你的代码可以和基类的对象正交。

*/

typedef Client_Acceptor_Base inherited;

/*

Construct the object with the concurrency strategy. Since this tutorial

is focused on thread-per-connection, we make that the default. We could

have chosen to omitt the default and populate it in main() instead.

通过同步策略创建对象。由于这个教程集中在线程每连接，我们把这个参数设置成为

缺省参数。我们可以选择将这个缺省参数删除，选择在 main()中进行创建。

*/

```
  Client_Acceptor (int thread_per_connection = 1)
    : thread_per_connection_ (thread_per_connection)
  {
  }

  /* Return the value of our strategy flag.  This is used by the
     Client_Handler to decide how to act.  If 'true' then the handler
     will behave in a thread-per-connection manner.
     返回我们策略标识。这个标识用来帮助 Client_Handler 定义动作。如果返回值是
     'true'那么句柄就使用一线程每连接的方式。
     */
  int thread_per_connection (void)
  {
    return this->thread_per_connection_;
  }

protected:
  int thread_per_connection_;
};
```

```
#endif  /*  CLIENT_ACCEPTOR_H */
```

好了，到此我们还没有涉及到很多有关同步策略的内容。我们继续看一下 Client_Handler,看它是否需要进行大幅修改。

• ACE Tutorial [翻译] 06 －page04

2004-11-21

Tag：ACE_TAO

client_handler.h 比以往的代码有较大的改变。在这里最主要的变化是添加的 svc()方法，实现我们连接线程的退出。

```
//  page04.html,v  1.11  2000/03/19  20:09:23  jcej  Exp



#ifndef  CLIENT_HANDLER_H

#define  CLIENT_HANDLER_H



/*  Our  client  handler  must  exist  somewhere  in  the  ACE_Event_Handler

    object  hierarchy.  This  is  a  requirement  of  the  ACE_Reactor  because

    it  maintains  ACE_Event_Handler  pointers  for  each  registered  event

    handler.  You  could  derive  our  Client_Handler  directly  from
```

ACE_Event_Handler but you still have to have an ACE_SOCK_Stream for the actually connection. With a direct derivative of ACE_Event_Handler, you'll have to contain and maintain an ACE_SOCK_Stream instance yourself. With ACE_Svc_Handler (which is a derivative of ACE_Event_Handler) some of those details are handled for you.

我们的客户处理句柄必须要处于 ACE_Event_Handler 的继承树中。

这是由于 ACE_Reactor 需要为每个注册的事件句柄维护 ACE_Event_Handler 指针。

你将你的 Client_Handler 派生于 ACE_Event_Handler,但是你还需要为实际的连接包含 ACE_SOCK_Stream 成员。如果直接派生于 ACE_Event_Handler,你自己需要包含和维护一个 ACE_SOCK_Stream 实例。如果使用 ACE_Svc_Handler（ACE_Event_Handler 的子类），你就不需要考虑这么多的细节了。

*/

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

#if !defined (ACE_LACKS_PRAGMA_ONCE)
# pragma once
#endif /* ACE_LACKS_PRAGMA_ONCE */
```

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


/* Another feature of ACE_Svc_Handler is it's ability to present the
   ACE_Task<> interface as well.  That's what the ACE_NULL_SYNCH
   parameter below is all about.  If our Client_Acceptor has chosen
   thread-per-connection then our open() method will activate us into
   a thread.  At that point, our svc() method will execute.  We still
   don't take advantage of the things ACE_NULL_SYNCH exists for but
   stick around for Tutorial 7 and pay special attention to the
   Thread_Pool object there for an explanation.
   ACE_Svc_Handler 还提供了另外一个功能，就是它也支持 ACE_Task<>接口。
   这就是为什么还要传入 ACE_NULL_SYNCH 参数了。如果我们的 Client_Acceptor 选择
   那么我们就需要在我们的 open()方法中激活一个线程。从这一点开始，我们的 svc()
   方法会被执行。在这里我们还不需要利用 ACE_NULL_SYNCH，把这些内容留到教程 7 中，
    在线程池对象中进行特别地介绍。
*/

class Client_Handler : public ACE_Svc_Handler <ACE_SOCK_STREAM, ACE_NULL_SYN
CH>

{

public:
```

```
typedef ACE_Svc_Handler <ACE_SOCK_STREAM, ACE_NULL_SYNCH> inherited;

// Constructor...构造函数
Client_Handler (void);

/* The destroy() method is our preferred method of destruction.  We
   could have overloaded the delete operator but that is neither easy
   nor intuitive (at least to me).  Instead, we provide a new method
   of destruction and we make our destructor protected so that only
   ourselves, our derivatives and our friends can delete us. It's a
   nice compromise.
   Destroy 方法实现的是有关析构的操作。我们可以通过重载析构函数来实现，
   但这对于我们来说不够简单也不够直观。 因此我们选择了另外一个新的析构
   的丰富，将我们的析构函数设置为保护只能供我们调用。这是一个比较好的
   折中方案。
   */
void destroy (void);

/* Most ACE objects have an open() method.  That's how you make them
   ready to do work.  ACE_Event_Handler has a virtual open() method
```

which allows us to create this overrride. ACE_Acceptor<> will

invoke this method after creating a new Client_Handler when a

client connects. Notice that the parameter to open() is a void*.

It just so happens that the pointer points to the acceptor which

created us. You would like for the parameter to be an

ACE_Acceptor<>* but since ACE_Event_Handler is generic, that would

tie it too closely to the ACE_Acceptor<> set of objects. In our

definition of open() you'll see how we get around that.

需要的 ACE 对象都提供了 open 方法，通过这个方法我们来初始化对象。

ACE_Event_Hander 定义了一个虚拟的 open 方法允许我们进行重载。ACE_Acceptor

将会在客户连接，创建新的 Client_Handler 的时候调用这个方法。注意 oepn 方法

的参数是 void*. 通过这个参数我们将 acceptor 传递到 open 方法中。你也许会

觉得采用 ACE_Acceptor<>* 更恰当一下，但是这样的话，会使得程序对

ACE_Acceptor<>耦合度提高。

*/

int open (void *acceptor);

/* When an ACE_Task<> object falls out of the svc() method, the

framework will call the close() method. That's where we want to

cleanup ourselves if we're running in either thread-per-connection

```
    or thread-pool mode.
    当一个 ACE_Task<>对象放弃 svc()方法，那么框架就会调用 close 方法。
    这是我们如果允许在线程每连接或者是线程池模式下，需要清理我们自己。
    */
int close (u_long flags = 0);


/* When there is activity on a registered handler, the
   handle_input() method of the handler will be invoked. If that
   method returns an error code (eg -- -1) then the reactor will
   invoke handle_close() to allow the object to clean itself
   up. Since an event handler can be registered for more than one
   type of callback, the callback mask is provided to inform
   handle_close() exactly which method failed. That way, you don't
   have to maintain state information between your handle_* method
   calls. The <handle> parameter is explained below... As a
   side-effect, the reactor will also invoke remove_handler() for the
   object on the mask that caused the -1 return. This means that we
    don't have to do that ourselves!
    当注册的句柄所对应的事件发生是，这个句柄的 handle_input()就会被调用。
    如果这个方法返回错误（如 —— -1），reactor 会调用 handle_close（）方法，
```

来让对象执行一些有关清理的操作。由于事件句柄可以注册不只一种函数，

在调用 handle_close()时，需要指定回调掩码，通过这个掩码指定时那个回调

方法调用失败。这样你就不需要为你的 handle_*方法维护一个状态信息了。

<handle>参数，将在下面进行介绍。作为副作用，当掩码所对应的对象出现

错误时，reactor 也会调用 remove_handler()。这意味着我们不需要处理这些内容。

*/

virtual int handle_close (ACE_HANDLE handle = ACE_INVALID_HANDLE,

ACE_Reactor_Mask mask = ACE_Event_Handler::ALL_EVENTS_MASK);

protected:

/* If the Client_Acceptor which created us has chosen a

thread-per-connection strategy then our open() method will

activate us into a dedicate thread. The svc() method will then

execute in that thread performing some of the functions we used to

leave up to the reactor.

如果我们选择线程每连接策略创建 Client_Acceptor，那么我们的 open()方法

将激活一个线程。Svc 方法将会这线程中允许，并执行一下操作，让我们脱离

reactor 的控制。（通过句柄的转移来实现？）

```
    */
int svc (void);

/* When we register with the reactor, we're going to tell it that we
   want to be notified of READ events.  When the reactor sees that
   there is read activity for us, our handle_input() will be
   invoked. The _handleg provided is the handle (file descriptor in
   Unix) of the actual connection causing the activity.  Since we're
   derived from ACE_Svc_Handler<> and it maintains it's own peer
   (ACE_SOCK_Stream) object, this is redundant for us.  However, if
   we had been derived directly from ACE_Event_Handler, we may have
   chosen not to contain the peer.  In that case, the <handle> would
   be important to us for reading the client's data.
   当我们向 reactor 进行注册时，我们需要向它通知我们只关系 READ 事件。当
   reactor 获知这些读活动时，我们的 handle_input（）就会被调用。 Handle 参数，
   是一个能够处理连接的句柄（在 unix 中就是一个文件描述符）。 由于我们是
   派生于 ACE_Svc_Handler<> ,它维护了一个自己的端对象（ACE_SOCK_Stream）
   对象，这样就会加重我们的负担。然而，如果我们需要自己从 ACE_Event_Handler，
   直接派生的话，我们也许会选择不包含端对象。在这个例子中，handle 参数
   对于我们获取客户端的数据非常重要。
```

```
    */
int handle_input (ACE_HANDLE handle);

/* This has nothing at all to do with ACE.  I've added this here as
   a worker function which I will call from handle_input().  As
   promised in Tutorial 5 I will use this now to make it easier to
   switch between our two possible concurrency strategies.
   这与 ACE 没有任何关系。添加这个方法是为了能在 handle_input()时，调用
   这一工作方法。正如教程 5 所保证的，采用这个方法来确保我们可以很方便得
   在两种可能的同步策略直接进行转换。
    */
int process (char *rdbuf, int rdbuf_len);

/* We don't really do anything in our destructor but we've declared
   it to be protected to prevent casual deletion of this object.  As
   I said above, I really would prefer that everyone goes through the
   destroy() method to get rid of us.
       在析构函数中，我们不需要做什么事情。但是我们这样声明，是为了将其作为
   保护方法，意外防止删除对象。正如我在上面所说，我们希望将大家通过 destroy
   方法来释放我们。
```

```
    */

  ~Client_Handler  (void);

};



#endif  /*  CLIENT_HANDLER_H  */
```

我们添加了的 svc()方法，并且提到了有关 open()方法的改变。我们在下面的章节我们跳过对象的定义，看一下有关实现的内容。

- **ACE Tutorial［翻译］06 －page05**

  2004-11-21

  Tag：ACE_TAO

  client_handler.cpp 展示了我刚才所提到的内容。在这里需要特别主要 open()方法中的决定，以及 svc()的小技巧。

```
//  page05.html,v  1.14  2000/03/19  20:09:23  jcej  Exp



/*  In  client_handler.h  I  alluded  to  the  fact  that  we'll  mess  around

    with  a  Client_Acceptor  pointer.  To  do  so,  we  need  the
```

Client_Acceptor object declaration.

在 Client_handler.h 中我提到了我们需要和 Client_Acceptor 指针进行周旋。

为了这么做，我们需要使用 Client_Acceptor 对象的声明。

We know that including client_handler.h is redundant because

client_acceptor.h includes it. Still, the sentry prevents

double-inclusion from causing problems and it's sometimes good to

be explicit about what we're using.

我们知道包含 client_Handler.h 会使得 client_acceptor.h 重复包含。

实际上通过#ifdef，我们可以防止 include 文件的二次包含。

On the other hand, we don't directly include any ACE header files

here.

另一方面，我们这并不直接包含 ACE 头文件，

*/

```
#include "client_acceptor.h"
#include "client_handler.h"
```

/* Our constructor doesn't do anything. That's generally a good idea.

Unless you want to start throwing exceptions, there isn't a really

good way to indicate that a constructor has failed. If I had my way, I'd have a boolean return code from it that would cause new to return 0 if I failed. Oh well...

我们的构造函数并不执行任何操作。这里有一些比较好的点子，除非你想抛出异常，在这里指定构造函数失败是不明智的。如果我来做，我会设置一个返回值，当创建失败时，返回 0。

*/

```
Client_Handler::Client_Handler (void)
{
}
```

/* Our destructor doesn't do anything either. That is also by design.

Remember, we really want folks to use destroy() to get rid of us.

If that's so, then there's nothing left to do when the destructor gets invoked.

我们的析构函数也不做什么事情。这是的操作是设计时指定的。记着我们只希望使用 destroy()方法来执行退出的操作。这样，在析构函数中，就不需要再执行其他的操作了。

```
*/

Client_Handler::~Client_Handler (void)

{

}



/* The much talked about destroy() method!  The reason I keep going on

    about this is because it's just a Bad Idea (TM) to do real work

    inside of a destructor.  Although this method is void, it really

    should return int so that it can tell the caller there was a

    problem.  Even as void you could at least throw an exception which

    you would never want to do in a destructor.

    这里要开始大篇幅讨论 destroy()方法。我打算这么做的目的是，在把析构函数

中做大量的现实工作是一个非常坏的主意。虽然这个方法没有返回值，它确实应该

返回 int 值，这样当调用出错的时候，可以通知调用这。虽然对于 void 你可以采用

扔异常的方式来通知错误，但是我们在析构函数中从来不这么做。

 */

void

Client_Handler::destroy (void)

{

  /* Tell the reactor to forget all about us.  Notice that we use the
```

```
    same args here that we use in the open() method to register
    ourselves.  In addition, we use the DONT_CALL flag to prevent
    handle_close() being called.  Since we likely got here due to
    handle_close(), that could cause a bit of nasty recursion!
    通知 reactor 将我们遗忘（分手了！）。注意我们需要和负责注册 open 方法
    传入的相同的参数。还有，我们使用了 DONT_CALL 表示来防止调用 handle_close。
    这是由于我们是通过调用 handle_close()进入到这条语句的，所以需要防止
    比较麻烦的递归的调用。*/
  this->reactor ()->remove_handler (this,
                                    ACE_Event_Handler::READ_MASK
                                    | ACE_Event_Handler::DONT_CALL);

  /* This is how we're able to tell folks not to use delete.  By
     deleting our own instance, we take care of memory leaks after
     ensuring that the object is shut down correctly.
     这里我们给大家介绍了不使用 delete 的原因。只要保证对象被正确关闭，
     通过删除我们自己的实例，就可以防止内存泄漏。
      */
  delete this;
}
```

```
/* As mentioned before, the open() method is called by the
   Client_Acceptor when a new client connection has been accepted.
   The Client_Acceptor instance pointer is cast to a void* and given
   to us here.  We'll use that to avoid some global data...
   正如前面所提到的，open 方法可以在 Client_Acceptor 接收新的客户连接是被调用。
  Client_Acceptor 的实例指针可以被转换成为 void*,并传递给我们。这样我们就可以
  减少全局数据的使用。
*/
int
Client_Handler::open (void *void_acceptor)
{
  /* We need this to store the address of the client that we are now
     connected to.  We'll use it later to display a debug message.
     我们需要存储客户端的地址，这样我们可以知道谁现在连接上了服务器。
     我们将再后面把客户的地址信息显示在调试信息中。
   */
  ACE_INET_Addr addr;

  /* Our ACE_Svc_Handler baseclass gives us the peer() method as a way
```

to access our underlying ACE_SOCK_Stream. On that object, we can invoke the get_remote_addr() method to get get an ACE_INET_Addr having our client's address information. As with most ACE methods, we'll get back (and return) a -1 if there was any kind of error.

Once we have the ACE_INET_Addr, we can query it to find out the client's host name, TCP/IP address, TCP/IP port value and so forth. One word of warning: the get_host_name() method of ACE_INET_Addr may return you an empty string if your name server can't resolve it. On the other hand, get_host_addr() will always give you the dotted-decimal string representing the TCP/IP address.

我们的 ACE_Svc_Handler 基类为我们通过的 peer()方法,通过这个方法，我们可以访问 ACE_SOCK_Stream 成果。对于这个对象，我们可以通过调用 get_remote_addr() 方法来获得 ACE_INET_Addr，并且获得客户端的地址信息。正如大多数的 ACE 方法一样，我们通过返回值-1 来说明调用过程中所发生的错误。一旦我们拥有了 ACE_INET_Addr。我们可以通过调用它来获得客户端主机的名字，TCP/IP 地址，TCP/IP 端口号，或者其他什么的。在这里需要提醒的是，ACE_INET_Addr 所提供的，get_host_name()方法当你的域名服务器无法解析的时候，可能会返回空值。 另外，get_host_addr()会返回给你带点和数字组成的 TCP/IP 地址串。

```
    */
  if (this->peer ().get_remote_addr (addr) == -1)
    return -1;

  /* Convert the void* to a Client_Acceptor*.  You should probably use
     those fancy ACE_*_cast macros but I can never remember how/when to
     do so.  Since you can cast just about anything around a void*
     without compiler warnings be very sure of what you're doing when
     you do this kind of thing.  That's where the new-style cast
     operators can save you.
     将 void* 参数转换为 Client_Acceptor*。 在这里你可能需要使用 ACE_*_cast 宏
     但是我还没有搞清楚什么时候，如何用这些东西。因为你可以把 void*和任何类型
     数据进行互相转换，而且这样做编译器并不会报任何警告，所以你需要对你所做的
     事情特别清楚。这是新风格的转换符所能代给你最有魅力的地方。
  */
  Client_Acceptor *acceptor = (Client_Acceptor *) void_acceptor;

  /* Our Client_Acceptor is constructed with a concurrency strategy.
     Here, we go back to it to find out what that strategy was.  If
```

thread-per-connection was selected then we simply activate a

thread for ourselves and exit. Our svc() method will then begin

executing in that thread.

我们的 Client_Acceptor 在创建时指定了一个并发策略。这里我们需要把这一

并发策略找出来。如果是选择的是线程每连接方式，我们就简单的激活一个线程，

然后退出。这样我们的 svc()方法就会在这个新创建的线程中开始执行。

If we are told to use the single-threaded strategy, there is no

difference between this and the Tutorial 5 implementation.

如果我们被告知是采用单线程策略，这里就教程 5 的实现没有什么区别了。

Note that if we're in thread-per-connection mode, open() is exited

at this point. Furthermore, thread-per-connection mode does not

use the reactor which means that handle_input() and it's fellows

are not invoked.

注意，如果我们在线程每连接模式中。open()方法在这一点推出

（后面的语句将不会执行） 。此外，线程每连接模式并不使用 reactor，这就意味着

handle_input(),以及相关的 handle 函数是不会被 reactor 所调用的。

*/

if (acceptor->thread_per_connection ())

return this->activate (THR_DETACHED);

```
// ************************************************************************

// From here on, we're doing the traditional reactor thing.  If

// you're operating in thread-per-connection mode, this code does

// not apply.

// 从这里开始，我们将做一些传统的调用 reactor 的事情，如果你是工作在

// 线程每连接模式下，下面这些代码将不再有效。

// ************************************************************************



/* Our reactor reference will be set when we register ourselves but

   I decided to go ahead and set it here.  No good reason really...

   当我们注册自己的时候，我们的 reactor 引用将会被赋值，但是我不打算这么做，

   所以我直接在这里进行赋值了。这么做其实也没有什么好的理由...

*/

this->reactor (acceptor->reactor ());



/* If we managed to get the client's address then we're connected to

   a real and valid client.  I suppose that in some cases, the client

   may connect and disconnect so quickly that it is invalid by the

   time we get here. In any case, the test above should always be

   done to ensure that the connection is worth keeping.
```

如果我们设法拿到了客户端地址那么我们就已经连接上了一个真实有效的客户端。

在一些场合下，客户端会连接断开非常迅速，以至于我们拿到它地址引用的时候，

客户端地址已经无效了。无论如何，我们就需要测试一下，以保证获得到的连接

是值得保存的。

Now, regiser ourselves with a reactor and tell that reactor that

we want to be notified when there is something to read. Remember,

we took our reactor value from the acceptor which created us in

the first place. Since we're exploring a single-threaded

implementation, this is the correct thing to do.

现在向 reactor 进行注册，并且告诉 reactor 我们希望对什么事件感兴趣。

注意，我们是通过前面的 acceptor 获得的 reactor 值。由于我们现在阶段

还是单线程的实现，所以下面的写法是正确的。

*/

```
if (this->reactor ()->register_handler (this,
                                        ACE_Event_Handler::READ_MASK) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "(%P|%t) can't register with reactor\n"),
                    -1);
```

```
  /* Here, we use the ACE_INET_Addr object to print a message with the
     name of the client we're connected to.  Again, it is possible that
     you'll get an empty string for the host name if your DNS isn't
     configured correctly or if there is some other reason that a
     TCP/IP addreess cannot be converted into a host name.
      现在我们使用 ACE_INET_Addr 对象来打印连接到我们的客户端的名字。
      注意，如果你的 DNS 没有配置正确，或者是其他的什么原因影响到了
      TCP/IP 地址到主机名的转换，那么你会获得一个空串。

*/
  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) connected with %s\n", addr.get_host_name ()));

  /* Always return zero on success.
          在成功的情况下，这里是返回零
*/
  return 0;
}

/* As mentioned in the header, the typical way to close an object in a
```

```
    threaded context is to invoke it's close() method.  Since we
    already have a handle_close() method built to cleanup after us,
    we'll just forward the request on to that object.
    正如前面所提到的，在一个线程上下文环境中释放一个对象的典型方法就是调用
    它的 close()方法。由于我们已经使用了一个 handle_close()方法来负责对象的
    清理工作。我们只需要将这个请求转发到对象即可。
*/
int
Client_Handler::close(u_long  flags)
{
  ACE_UNUSED_ARG (flags);

  /* We use the destroy() method to clean up after ourselves.  That
     will take care of removing us from the reactor and then freeing
     our memory.
     我们使用 destroy()方法来释放我们自己。这个反复会将我们从 reactor 中删除，
     并释放我们所占有的内存。
   */
  this->destroy ();
```

```
    /* Don't forward the close() to the baseclass!  handle_close() above
       has already taken care of delete'ing.  Forwarding close() would
       cause that to happen again and things would get really ugly at
       that point!
    不要调用基类的 close()方法！上面的 handle_close()方法会执行删除操作的。
    直接调用 close()方法会使得删除方法再次被调用，这样会非常难看的。
     */
    return 0;
}

/* In the open() method, we registered with the reactor and requested
    to be notified when there is data to be read.  When the reactor
    sees that activity it will invoke this handle_input() method on us.
    As I mentioned, the _handle parameter isn't useful to us but it
    narrows the list of methods the reactor has to worry about and the
    list of possible virtual functions we would have to override.
    在 open 方法中，我们向 reactor 注册并且请求获知我们所关心的读取数据事件。
    正如我前面讲的，_handle 参数对于我们没有用处，但是它减少了 reactor 需要
     关心的方法以及我们需要照看的虚函数。
    Again, this is not used if we're in thread-per-connection mode.
```

```
    注意，如果我们在线程每连接模式下本函数并不使用。
    */
int
Client_Handler::handle_input  (ACE_HANDLE  handle)
{
  /* Some  compilers  don't  like  it  when  you  fail  to  use  a  parameter.
     This macro  will  keep  'em  quiet  for  you.
     一些编译器会警告你还还有未使用的参数，使用这个宏，可以帮你屏蔽
     这个警告。
     */
  ACE_UNUSED_ARG  (handle);

  /* Now,  we  create  and  initialize  a  buffer  for  receiving  the  data.
     Since  this  is  just  a  simple  test app, we'll  use  a  small  buffer
     size.
     现在，我们为接收的数据初始化一块内存。由于我们是一个简单的测试应用，
     我们使用的一个比较小的缓冲区。
     */
  char  buf[BUFSIZ];
```

```
  /* Invoke the process() method with a pointer to our data area.
     We'll let that method worry about interfacing with the data.  You
     might choose to go ahead and read the data and then pass the
     result to process().  However, application logic may require that
     you read a few bytes to determine what else to read...  It's best
     if we push that all into the application-logic level.
     调用 process()方法来处理我们的数据。我们把和数据交互的内容都交给那个方法。
     你可以选择在这里直接读取数据，并把结果值传递给 process。然而，应用的逻辑需要
     我们读取一段数据来决定下面需要读取的内容… 所以把所有这些都放到应用层是
     一个好主意。
     */
  return this->process (buf, sizeof (buf));
}

/* If we return -1 out of handle_input() or if the reactor sees other
   problems with us then handle_close() will be called.  The reactor
   framework will take care of removing us (due to the -1), so we
   don't need to use the destroy() method.  Instead, we just delete
   ourselves directly.
   如果我们在 handle_input()返回-1 或者 reactor 在执行过程中其他问题，那么
```

```
    handle_close()就会被调用。Reactor 框架会负责删除我们（因为返回值-1）。
    所以我们不必使用 destroy()方法。作为替代，我们会直接删除我们自己。
*/
int
Client_Handler::handle_close (ACE_HANDLE handle,
                              ACE_Reactor_Mask mask)
{
  ACE_UNUSED_ARG (handle);
  ACE_UNUSED_ARG (mask);

  this->destroy ();
  return 0;
}

/* The ACE_Svc_Handler<> is ultimately derived from ACE_Task<>.  If
    you want to create a multi-threaded application, these are your
    tools!  Simply override the svc() method in your derivative and
    arrange for your activate() method to be called.  The svc() method
    then executes in the new thread.
    ACE_Svc_Handler<>是直接继承于 ACE_Task<>。如果你想写一个多线程的应用，
```

ACE_Task<>是一个很好的工具类。你只需要在你地派生类中简单地重载 svc（）

方法并且在你的 activate()方法中调用它。 这个 svc()方法就会在一个新的线程</

- **ACE Tutorial [翻译] 08－page01**

2004-11-29

Tag：ACE_TAO

在大多少得 IPC 程序中，客户端都知道服务器在什么地方。例如一个 mail 客户端会通过一个包含 mail 服务器地址的配置文件获得主机所在位置。你的 Web 浏览器也会根据你所输入的地址信息来”定位”主机位置。

如果你想写一个服务端应用程序，同时你想把这个应用允许在你的网络中的好几个系统之上。从客户端来看这些实例（应用运行产生的）或多或少是一样的，所以你不想在客户端把所有的服务程序都配置一遍。同样的，如果你希望能够在任何时候添加删除服务器地址，你就不能简单地给客户端发一串地址就能完事的。

这样，如何让客户端知道服务器的地址呢?

数据报文可以满足这样的需求。你可以向网络发送一个数据报文，这样任何一个在监听这个端口的服务器都会接收到这个报文。像我们前面所见到的 ACE_SOCK_Stream，你可以通过广播搞获取到对方的地址信息。这样，服务器就可以转发回应消息给客户端。对于客户端来说，也可以提出出对方的地址，这样就能知道现在服务器在哪里了。

在这个教程中，我们将写三个应用：服务器监听数据报，一个向已知的主机发送数据的客户端，一个向整个（子）网发送消息的客户端。在下 一个教程中，我们将介绍如何将服务器写得更加正规。

Kirthika's abstract:

在这里，我们将使用 datagram socket 来实现客户端的服务发现机制。数据报文是采用 UDP 来进行发送的，这是一个不可靠并且是非面向连接的协议。数据报文一般来说都是比较小，并且不是用来处理服务器与客户端之间非常重要的交互。

服务器在一个固定的端口等待数据报的道理。在一般情况下，客户端很少将数据报发送到一个指定的服务器端，这样就需要客户端能够发现服务器并且需要客户端能够在她所在的子网通过广播的方式发送数据报。然后，所有在指定接口监听的服务器将接收到相关的数据。同时有合适的服务器来处理消息。请记住，这里没有建立任何可靠的连接。服务器通过 recv()方法获得远程客户端的地址，然后与客户端进行通讯。

这样我们就可以简单的了解一下通过数据报文进行通讯的另一种方式。

* 事实上，服务器只是可能够获得数据报的。数据报是一个不太可靠的报文。（我所知道的一下操作系统是这样的。）然而，如果网络拥塞不是很厉害，这些数据报就是能够被传递的。你的客户端可能会发送更多的请求，如果在一定的时间上没有任何响应的话。

• ACE Tutorial [翻译] 08－page02

2004-11-29

Tag：ACE_TAO

首先，我们看一下 server.cpp 是如何实现的。 这是一个特别简单的应用，该应用只负责监听指定端口的数据报文，并且将收到的消息转发过去。为了能够真实地展现"服务发现"的机制，服务器需要在确定反馈消息部分做一些特别地工作。这将在我们地下面地教程中详细介绍。

```
// page02.html,v 1.13 2000/11/27 17:56:43 othman Exp
```

```
/* Our datagram server will, of course, need to create a datagram.
   We'll also need an address object so that we know where to listen.
   我们的数据报文服务器，显而易见，需要创建一个数据报。
   我们需要一个地址对象来知道我们希望监听的端口
*/
#include "ace/Log_Msg.h"
#include "ace/SOCK_Dgram.h"
#include "ace/INET_Addr.h"

/* Use the typical TCP/IP port address for receiving datagrams.
   使用一个典型的 TCP/IP 端口来接收数据。
*/
static const u_short PORT = ACE_DEFAULT_SERVER_PORT;

int
main (int, char**)
{
  /* This is where we'll listen for datagrams coming from the clients.
     We'll give this address to the open() method below to enable the
     listener.
     这里我是我们接收客户端所发送过来的数据报。我们使用这个地址来打开
     我们所希望的监听。
     */
  ACE_INET_Addr local (PORT);

  /* A simply constructed datagram that we'll listen with.
     简单构造函数来创建我们所希望监听的数据报
  */
  ACE_SOCK_Dgram dgram;

  /* Like most ACE objects, the datagram has to be opened before it
     can be uses.  Of course, -1 on failure.
```

和其他的 ACE 对象一样，数据报在使用的之前需要调用 open 方法。当然，如果调用失败返回值为－1。

A datagram will fail to open if there is already a datagram listening at the port we've chosen. It *is* OK to open a datagram at a port where there is an ACE_SOCK_Stream though. This is because datagrams are UDP and SOCK_Stream is TCP and the two don't cross paths.

如果在我们所选定的端口已经在监听数据报了，那么数据报接收器在创建的时候会出现错误。但是它在打开一个 ACE_SOCK_Stream 所占用的端口是没有问题的。这是由于 UDP 的数据报文和 TCP SOCK_Stream 的端口是不相关的。

```
*/
if (dgram.open (local) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "%p\n",
                     "open"),
                    -1);

/* Create a simple buffer to receive the data.  You generally need
  to provide a buffer big enough for the largest datagram you expect
  to receive.  Some platforms will let you read a little and then
  some more later but other platforms will throw out whatever part
  of the datagram you don't get with the first read.  (This is on a
  per-datagram basis BTW.)  The theoretical limit on a datagram is
  about 64k.  The realistic limit (because of routers & such) is
  much smaller.  Choose your buffer size based on your application's
  needs.
```

为接收到的数据创建一个缓冲区。你需要设置一个比你所期望接收的数据报更大一些的缓冲区。一些系统平台运行你读取一小部分数据，过一段时间之后在继续读剩余的内容，但是另一下系统平台在你第一次读取部分数据之后，就将数据报的剩余内容丢弃。（这是一个面向单数据报的模式。）一个数据报的理论最大值是 64K。在现实中的最大值（由于路由或者其他）比这个值要小

```
    很多。你需要根据应用需要选择你的缓冲大小。
    */
  char buf[BUFSIZ];

  /* Unlike ACE_SOCK_Stream, datagrams are unconnected.  That is,
     there is no "virtual circuit" between server and client.  Because
     of this, the server has to provide a placeholder for the OS to
     fill in the source (client) address information on the recv.  You
     can initialize this INET_Addr to anything, it will be overwritten
     when the data arrives.
     与 ACE_SOCK_Stream 不同，数据报是非面向连接的。这就是说，在服务器和客户端
     之间是没有"虚拟链路"的。正是因为这样，服务器需要提供存储空间来为存放操作
     系统接收到的（客户端）的地址信息。你可以给 INET_Addr 初始化任何一个值，当
     接收到数据报时，这个值将会被覆盖。
     */
  ACE_INET_Addr remote;

  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) starting up server daemon\n"));

  /* Receive datagrams as long as we're able.
     一直接收我们所能接收到的数据报
  */
  while (dgram.recv (buf,
                     sizeof (buf),
                     remote) != -1)
    {
      /* Display a brief message about our progress.  Notice how we
         use the 'remote' object to display the address of the client.
```

```
  With an ACE_SOCK_Stream we used get_remote_addr() to get the
  address the socket is connected to.  Because datagrams are
  unconnected, we use the addr object provided to recv().
  显示现在进展情况的简要信息。注意我们怎样使用 'remot' 对象来显示
  客户端的地址。在 ACE_SOCK_Stream 中，我们使用 get_remote_addr()来
  获取连接到我们的 socket 地址。由于数据报时非面向连接的，我们使用
  recv()方法所提供的地址对象来获取。
*/
ACE_DEBUG ((LM_DEBUG,
            "(%P|%t) Data (%s) from client (%s)\n",
            buf,
            remote.get_host_name ()));

/* To respond to the client's query, we have to become a client
  ourselves.  To do so, we need an anonymous local address from
  which we'll send the response and a datagram in which to send
  it.  (An anonymous address is simply one where we let the OS
  choose a port for us.  We really don't care what it is.
  为了响应客户端的请求。我们需要创建一个客户端对象。为了这样做，
  我们需要一个匿名的本地地址，这样我们可以通过这个地址向发送响应
  的数据报。（一个匿名地址包含操作系统提供为我们选定的一个端口号，
  在这里，我们并不关心其详细内容）
*/
ACE_INET_Addr local ((u_short) 0);
ACE_SOCK_Dgram client;

/* Open up our response datagram as always.
   初始化我们的响应报文
 */
if (client.open (local) == -1)
  {
    ACE_ERROR_RETURN ((LM_ERROR,
```

```
                        "%p\n",
                        "client open"),
                       -1);
    return 0;
  }

/* Build a witty response...
   创建一个诙谐的响应
*/
sprintf (buf,
         "I am here");

/* and send it to the client.  Notice the symmetry with the
  recv() method.  Again, the unconnected nature of datagrams
  forces us to specify an address object with each read/write
  operation.  In the case of read (recv()) that's where the OS
  stuffs the address of the datagram sender.  In the case of
  write (send()) that we're doing here, the address is where we
  want the network to deliver the data.
  将信息回传到客户端。注意这是一个和recv()方法对应的函数。由于数
  据报时非面向连接的，这就要求我们在进行读写操作时，需要指明地址
  对象。在读取（recv()）情况下，我们从操作系统获得的地址，就是发送
  数据的源地址在写数据（send()）时，这个地址就是我们希望发送的目的
  地址。

  Of course, we're assuming that the client will be listening
  for our reply...
  当然，我们需要假设客户端已经在监听我们的回应了…
  */
if (client.send (buf,
                 ACE_OS::strlen (buf) + 1,
                 remote) == -1)
```

```
        ACE_ERROR_RETURN ((LM_ERROR,
                           "%p\n",
                           "send"),
                          -1);
    }

  return 0;
}
```

这就是我们涉及到的内容。显然这里还有很多可以扩展的空间。这里最值得争议的是在接收报文时设置小一点的缓冲区。现在我还不能给一个明确的设置数据报大小的回答。理论上来说，缓冲区有一个 64K 大小的限制，所以你需要自己进行关分包的操作。一些文章指出 8K 是一般比较合适的大小，其他这说更小一点的数据包比较合适。我认为最好的保证数据尽可能小（例如——小于 8K）并且进行大量测试。如果你发现你的路由器已经对你的大数据报进行了分包的操作，你需要减小你的数据包的大小。当然，如果你需要发送 100K 并且一次只能发送 1K，你需要考虑有关重传或者从组的问题。从这一点来说，你也许会考虑采用 TCP。

注意：数据包是不可靠的。

• ACE Tutorial [翻译] 08－page03

2004-11-29

Tag：ACE_TAO

在 directed_client.cpp 我们创建一个客户端来向指定的服务器发送数据报。如果你知道服务器在什么地方，这是一个比较好的建立连接的方法。例如在 Unix 中的 talk 服务，就是采用这样的方式来实现的。

```
// page03.html,v 1.13 2000/11/27 17:56:43 othman Exp
```

```
#include "ace/Log_Msg.h"
#include "ace/SOCK_Dgram.h"
#include "ace/INET_Addr.h"

/* Once again, we use the default server port.  In a "real" system,
   the server's port (or ports) would be published in some way so that
   clients would know where to "look".  We could even add entries to
   the operating system's services file and use a service name instead
   of a number.  We'll come back to that in some other tutorial
   though.  For now, let's stay simple.
   这里，我们还是选定缺省的服务器端口好。在实际系统中，服务器的端口号
   （端口号组）可以通过某种方式来发布，这样客户端就知道如何来查找服务器。
   我们也可以通过向操作系统的服务文件添加项目，通过项目来取代相应的数字。
   如果感兴趣的化，可以通过其他的教程来学习相关的内容。
 */
static const u_short PORT = ACE_DEFAULT_SERVER_PORT;

/* Our goal here is to develop a client that can send a datagram to a
   server running on a known host.  We'll use a command-line argument
   to specify the hostname instead of hard-coding it.
   我们目前要实现的功能就是要开发一个客户端，能够向指定的服务器发送数据
报。
   我们使用命令行参数来指定相关的主机，而不是写进我们到我们的代码里。
*/
int
main (int argc, char *argv[])
{
  /* All datagrams must have a point of origin.  Since we intend to
     transmit instead of receive, we initialize an address with zero
     and let the OS choose a port for us.  We could have chosen our own
     value between 1025 and 65535 as long as it isn't already in use.
     所有的数据报都需要有一个源点。由于我们是发送数据而不是接收数据，我们
```

```
    将地址初始化伟 0，并且让 OS 来为我们选择一个端口。我们也可以选择 1025
到
    65535 之间的没有被使用的任意端口。

    The biggest difference between client and server when datagrams
    are used is the fact that servers tend to have a known/fixed
    address at which they listen and clients tend to have arbitrary
    addresses assigned by the OS.
    客户端与服务器端的最大的不同是服务器端的数据报是监听一个众所周知/
    固定的地址,而客户端倾向于使用 OS 分配的地址。
    */
  ACE_INET_Addr local((u_short) 0);

  /* And here is our datagram object.
     这是我们的数据报对象
  */
  ACE_SOCK_Dgram dgram;

  /* Notice that this looks a lot like the server application.
    There's no difference in creating server datagrams an client
    datagrams.  You can even use a zero-constructed address for your
    server datagram as long as you tell the client where you're
    listening (eg -- by writting into a file or some such).
    注意这和服务器端的应用有很多相同的地方。在客户端创建数据报是和服务
    器端创建数据报没有什么不同。只要你能告诉客户端你监听的端口（例如
    把端口号写到文件里面），你就可以向服务器端的数据包传入一个空值的地址。

  */
  if (dgram.open (local) == -1)
    ACE_ERROR_RETURN ((LM_ERROR,
                       "%p\n",
                       "datagram open"),
```

```
                        -1);

  /* Yep.  We've seen this before too...
     这和我们以前看到的一样
  */
  char buf[BUFSIZ];

  /* Ok, now we're doing something different.
    现在我们做一些不同的事情
  */
  sprintf (buf, "Hello World!");

  /* Just like sending a telegram, we have to address our datagram.
    Here, we create an address object at the desired port on the
    chosen host.  To keep us from crashing, we'll provide a default
    host name if we aren't given one.
    和发送电报一样，我们也需要指定我们的发送地址。这里我们为我们选定的
    主机指定的端口创建一个地址对象。为了防止程序崩溃，如果没有选定主机地
址
    我们需要提供一个缺省的主机，
    */
  ACE_INET_Addr remote (PORT,
                        argc > 1 ? argv[1] : "localhost");

  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) Sending (%s) to the server.\n",
              buf));
  /* Now we send our buffer of stuff to the remote address.  This is
    just exactly what the server did after receiving a client message.
    Datagrams are rather orthogonal that way: they don't generally
    make much of a fuss about being either client or server.
    现在我们把 buffer 的数据发送到远端的地址去。这和服务器接收到客户端的
```

```
  发送的数据一样的处理方法，对于数据报来说，服务器和客户端基本没有区别。
  */
if (dgram.send (buf,
                ACE_OS::strlen (buf) + 1,
                remote) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "%p\n",
                     "send"),
                    -1);

/* Now we've turned around and put ourselves into "server mode" by
  invoking the recv() method.  We know our server is going to send
  us something, so we hang out here and wait for it.  Because we
  know datagrams are unreliable, there is a chance that the server
  will respond but we won't hear.  You might consider providing a
  timeout on the recv() in that case.  If recv() fails due to
  timeout it will return -1 and you can then resend your query and
  attempt the recv() again.
  现在通过调用 recv()方法，我们需要转变角色到“服务模式”。我们知道
  服务器会给我们发送消息，所有我们阻塞在这里，等待服务器发送的消息。
  因为我们知道数据报是不可靠的，服务器发送过来的消息，我们有可能会
  接收不到。你也许会考虑让 recv()提供一个一个超时参数。 如果 recv()
  由于超时返回失败，他将会返回一个－1，你可以可以重新发送一个你的请求，
  再尝试接收一次。
  Like the server application, we have to give the recv() an
  uninitialized addr object so that we can find out who is talking
  back to us.
  和服务应用一样，我们需要给 recv()指定一个没有初始化的地址对象，这样
  我们就可以知道谁回应我们的消息了。
   */
if (dgram.recv (buf,
                sizeof (buf),
```

```
                    remote) == -1)
    ACE_ERROR_RETURN ((LM_ERROR,
                       "%p\n",
                       "recv"),
                      -1);

  /* Find out what the server had to say.
     显示服务器返回的消息
  */
  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) The server said:  %s\n",
              buf));

  /* Using the "remote" object instance, find out where the server
     lives.  We could then save this address and use directed datagrams
     to chat with the server for a while.
     使用 remote 对象实例，显示服务器的地址。 接下来我们可以将这个地址保存，
     然后通过数据报直接和服务器进行对话。
  */
  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) The server can be found at:  (%s:%d)\n",
              remote.get_host_name(),
              PORT));

  return 0;
}
```

这就这么简单明了，但是现阶段我们所做只是和我们知道的服务器进行对话，但是没能发现我们未知的服务器。现在你可以之间发送数据报到你网络中任何主机，但是这不是最佳的解决方案。再下一页，我们将向你展示正确的做法。

- **ACE Tutorial [翻译] 08－page04**

2004-11-29

Tag：ACE_TAO

在 broadcast_client.cpp 中，我们将展示如何向子网内的所有的主机发送一个单一的数据报。这这里我们说子网是因为广播报文是不能够通过路由器转发过去的。所以，如果你的网络管理员已经将你的网络划分成为子网，你所发送的广播报文只能在你所在的子网中传送。

这里我只注释和 directed_client 不同的部分。

```
// page04.html,v 1.13 2000/11/27 17:56:43 othman Exp

#include "ace/Log_Msg.h"
#include "ace/SOCK_Dgram_Bcast.h"
#include "ace/INET_Addr.h"

static const u_short PORT = ACE_DEFAULT_SERVER_PORT;

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

  ACE_INET_Addr local ((u_short) 0);

  /* Instead of creating the ACE_SOCK_Dgram we created last time,
    we'll create an ACE_SOCK_Dgram_Bcast.  "Bcast" means, of course,
    "Broadcast".  This ACE object is clever enough to go out to the OS
```

and find all of the network interfaces. When you send() on a Dgram_Bcast, it will send the datagram out on all of those interfaces. This is quiet handy if you do it on a multi-homed host that plays router...

与我们上次创建的 ACE_SOCK_Dgram 不同，我们创建了一个 ACE_SOCK_Dgram_Bcast.

“Bcast”的意思是"Broadcast 广播"。这个 ACE 的对象是非常机灵，它将会通过 OS

遍历所以的网络接口。当你通过调用 Dgram_Bcast 的 send()方法，它将会向所以的

网络接口发送数据报。如果你是在一个起路由功能的多地址主机这么做将非常有效。

*/

```
ACE_SOCK_Dgram_Bcast dgram;

if (dgram.open (local) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "%p\n",
                     "datagram open"),
                    -1);

char buf[BUFSIZ];

sprintf (buf, "Hello World!");

/* The only other difference between us and the directed client is
  that we don't specify a host to receive the datagram.  Instead, we
  use the magic value "INADDR_BROADCAST".  All hosts are obliged to
  respond to datagrams directed to this address the same as they
  would to datagrams sent to their hostname.
```

这和直接连接的客户端的最大的阿不同是，我们不需要指定接收数据报的主机。

这样我们使用了一个魔数“INADDR_BROADCAST”.和用主机地址直接发送一样，

这些主机都必须对接收的到的数据报进行响应。

Remember, the Dgram_Bcast will send a datagram to all interfaces on the host. That's true even if the address is for a specific host (and the host address makes sense for the interface). The real power is in using an INADDR_BROADCAST addressed datagram against all interfaces.

需要提醒的是 Dgram_Bcast 将会通过主机的所有接口向外发送数据。即使这个地址指定的是一个特定的主机（并且主机地址是对这个接口有效的）。使用 INADDR_BROADCAST 地址的主要目的是需要防止这样的情况出现。

*/

```
ACE_INET_Addr remote (PORT,
                      INADDR_BROADCAST);

ACE_DEBUG ((LM_DEBUG,
            "(%P|%t) Sending (%s) to the server.\n",
            buf));

if (dgram.send (buf,
                ACE_OS::strlen (buf) + 1,
                remote) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "%p\n",
                     "send"),
                    -1);

if (dgram.recv (buf,
                sizeof (buf),
                remote) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "%p\n",
```

```
                          "recv"),
                         -1);

  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) The server said:  %s\n",
              buf));

  /* Using the "remote" object instance, find out where the server
     lives.  We could then save this address and use directed datagrams
     to chat with the server for a while.
     使用“remote”对象实例，来获得服务器的地址。接下来我们可以将这个地址保存，
     然后通过数据报直接和服务器进行对话。
     */
  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) The server can be found at:  (%s:%d)\n",
              remote.get_host_name(),
              PORT));

  return 0;
}
```

About that subnet thing:

有关子网的内容：

> 如果你的客户端运行的机器有好几个网络接口，那么广播报将向这些接口的所有子网进行发送。如果你需要将数据包通过路由器转发，所需要做的是什么？我的建议是写一个服务程序运行在你的路由器两端。当在路由器一端的服务程序接收到了广播包，它可以将这毅数据包通过路由器直接转发到对等一段的服务

器。对等端的服务器可以在将原有的数据报在原有的子网内广播。 这是一个比较经济有效的做法。

需要提醒的内容：

当你创建广播报文的时候，你可能会看到这样的错误信息： *ACE_SOCK_Dgram_Bcast::mk_broadcast: Broadcast is not enable for this interface.: Unknown error.*， 是因为一些接口如（ppp，slip）都不支持广播报文。

再次需要注意的地方

如果当你调用客户端是，你的网络里面这时有多个服务器在运行，那回应可以是从任何一个机器返回的。
