



ACE 开发指南(初级)

文档信息

作者	郑明智
创建日期	2006-12-19
版本	1.0
部门名称	开发部

修订文档历史记录

[illegible]

目录

1. 介绍.....	1
1.1 目的.....	1
1.2 文档协定.....	1
1.3 阅读者建议.....	1
1.4 术语说明.....	1
1.5 翻译约定.....	2
1.6 相关资料.....	2
1.7 参考文献.....	2
1.8 补充说明.....	2
2. ACE 简介及环境搭建.....	3
2.1 ACE 简介.....	3
2.2 本指南的主要内容.....	3
2.3 获取 ACE.....	4
2.4 编译 ACE.....	4
2.4.1 为什么要编译 ACE.....	4
2.4.2 在 Window 上编译.....	4
2.4.3 在 Linux 上编译.....	5
2.5 前行的路标.....	6
3. ACE REACTOR 框架.....	6
3.1 Reactor（反应器）框架.....	6
3.1.1 ACE_Event_Handler(事件处理器).....	7
3.1.2 ACE_Reactor.....	10
3.2 Acceptor(接受器)-Connector(连接器)框架.....	12
3.2.1 ACE_Svc_Handler(服务处理器).....	13
3.2.2 ACE_Acceptor.....	14
3.2.3 ACE_Connector.....	15
3.3 ACE Reactor Server (Demo).....	17
3.3.1 需求.....	17
3.3.2 实现.....	17
3.3.3 ACE 工具类.....	26
3.3.4 Server 改进.....	27

3.4	ACE Reactor Client (Demo)	30
3.4.1	需求	30
3.4.2	实现 I	30
3.4.3	使用超时机制发送消息	34
3.4.4	实现 II	34
3.5	前行的路标	38
4.	ACE PROACTOR 框架	39
4.1	Proactor (前摄器) 框架	39
4.1.1	异步 I/O 工厂类	41
4.1.2	ACE_Handler(完成处理器)	42
4.1.3	ACE_Message_Block	43
4.1.4	ACE_Proactor	44
4.2	前摄式 Acceptor-Connector 框架	46
4.2.1	ACE_Service_Handler	46
4.2.2	ACE_Asynch_Acceptor	47
4.2.3	ACE_Asynch_Connector	47
4.3	既生 Proactor, 何生 Reactor (二者的应用范围)	47
4.4	ACE Proactor Server (Demo)	48
4.4.1	需求	48
4.4.2	实现	48
4.5	前行的路标	55
5.	ACE TASK 框架	56
5.1	我们的新需求	56
5.2	Task(任务)框架	56
5.3	ACE_Message_Queue	57
5.4	ACE_Task	60
5.5	Demo(Reactor Client 的改写)	62
5.5.1	需求	62
5.5.2	实现	63
5.6	基本的线程安全性	72
5.6.1	互斥体(Mutex)	73
5.6.2	守卫(Guard)	77
5.7	前行的路标	78
6.	总结	78
7.	常见问题	79

1. 介绍

1.1 目的

本指南作为使用 ACE 框架开发应用程序的参考，以期能够对使用 ACE 框架的同事有所帮助。

1.2 文档协定

本文档的书写遵循公司定义的文档规范。

本指南写作时，ACE 最新的稳定版本为 5.5 版。本指南中观点和代码并不保证适用于后续的 ACE 版本。

本指南旨在帮助新手入门，如果您已对 ACE 有一定使用经验并想更深入地了解 ACE，建议您阅读 ACE 的相关书籍。

1.3 阅读者建议

本指南假定阅读者有 C++ 的开发经验和通信程序的开发经验，文档中 C++ 及 Socket 等开发知识及相关概念不再赘述。

1.4 术语说明

C/S Client/Server 客户端/服务器架构

Client 客户端

Server 服务器

ACE 自适应通信环境 (Adaptive Communication Environment)

Reactor 反应器，高效的事件多路分离和分派提供可扩展的面向对象框架

Proactor 前摄器

aio 异步 I/O

Asynchronous I/O 异步 I/O

Epoll Linux 从 2.6 开始支持的异步事件 I/O 技术

1.5 翻译约定

Method/Function 方法

Nested Class 内部类

Callback 回调

Hook Method 挂钩方法

Handle 句柄

Template 模板

Daemon 守护

Override/ Overwrite 覆盖

Overload 重载

1.6 相关资料

ACE-5.5.zip

Magic C++ 3.5

1.7 参考文献

- [1] ACE 程序员指南 - 网络与系统编程的实用设计模式
- [2] C++网络编程(卷 1) 运用 ACE 和模式消除复杂性
- [3] C++网络编程(卷 2) 基于 ACE 和框架的系统化复用
- [4] ACE 自适应通信环境中文技术文档 - 中篇 ACE 程序员教程
- [5] ACE API 列表
- [6] ACE 源代码

1.8 补充说明

[1] 本文档中提到的资料，工具，代码都放在了 相关资料放在<\\192.168.0.20\\upload\\开发部\\技术组\\ACE> 目录下。

[2] 如果您发现本指南有错误或者疏漏，请通知[作者](#)修改，以使本指南能够更好的帮助大家。

2. ACE 简介及环境搭建

2.1 ACE 简介

ACE 自适应通信环境 (Adaptive Communication Environment) 是面向对象的框架和工具包，它为通信软件实现了核心的并发和分布式模式。ACE 包含的多种组件可以帮助通信软件的开发获得更好的灵活性、效率、可靠性和可移植性。ACE 中的组件可用于以下几种目的：

- ✓ 并发和同步
- ✓ 进程间通信 (IPC)
- ✓ 内存管理
- ✓ 定时器
- ✓ 信号
- ✓ 文件系统管理
- ✓ 线程管理
- ✓ 事件多路分离和处理器分派
- ✓ 连接建立和服务初始化
- ✓ 软件的静态和动态配置、重配置
- ✓ 分层协议构建和流式框架
- ✓ 分布式通信服务：名字、日志、时间同步、事件路由和网络锁定，等等。

上面是比较官方的介绍。总之，如果你想用 C++ 实现一个通信程序，而你又不想纠缠于 Windows/Linux 等各平台不同的 Socket 细节，那就可以考虑 ACE 框架。ACE 经过 10 余年的开发，已经不仅仅是通信的包，还实现了很多其它功能比如内存管理，文件系统管理等，可以满足您大多数开发的需要。而且 ACE 是被全世界很多项目采用的可以称得上是电信级别的框架。

ACE 是跨平台的，完全可以在 Windows 上开发，运行在 Linux 上（当然，与 Java 不完全相同的是您需要重新编译链接。而且如果您的程序使用了 GUI 的包，这么做是行不通的）。

ACE 的简介就到这里。当我们使用 ACE 开发了一些程序，对 ACE 有了一定的认识之后，再回过头来进一步认识 ACE。

2.2 本指南的主要内容

本指南的目标是 ACE 零基础 到 可以用 ACE 开发一些简单的 C/S 程序。因此本指南的主要内容有：

1. ACE 的获取编译

首先介绍如何搭建 ACE 开发环境的问题。

2. ACE Reactor/Proactor 框架

接下来本文并不打算像其它 ACE 的书籍从 ACE 历史介绍，基础机制讲起，而是直接进入 Reactor/Proactor 框架的介绍及使用这 2 个框架来开发程序，给初学者最想要的东西。

3. ACE 其他机制

最后对 ACE 的其他框架机制等做一概要介绍，比如 ACE_Task 机制，线程管理，同步机制等。

2.3 获取 ACE

从服务器上获取 ACE-5.5.zip。或者从http://download.dre.vanderbilt.edu/previous_versions/ 下载你需要的 ACE 版本。

2.4 编译 ACE

2.4.1 为什么要编译 ACE

应用程序在链接及运行时需要使用 ACE 的库和 dll 文件(Windows)，而你下载的包里没有这些文件或者不适合你的平台。这时候就需要自己编译 ACE。编译一般来说需要花很长时间。

注 1：项目组的其他人或者公司里的同事可能已经有编译好的库文件，如果你不想花时间编译 ACE，可以向其他人索取。

服务器上有 ace5.3 版本在 gcc2.96/3.23 编译通过的 rpm 文件，如果适合你的平台，直接安装即可。

注 2：下面的在 Windows/Linux 编译部分取自陈昕发的贴，本文做了适当修改。文中目录等请注意修改成符合你的环境的目录等。陈昕原贴在

<http://192.168.0.18/bbs/viewthread.php?tid=1085&fpage=1>

2.4.2 在 Window 上编译

1. 解压缩到本地目录

例：解压缩到 C:\ACE_wrappers

2. 查看安装向导

ACE 有自带的安装向导文件 C:\ACE_wrappers\ACE-INSTALL.html

查看 ACE-INSTALL.html 文件中的 Building and Installing ACE on Windows with Microsoft Visual C++ 章节, 按照上面所写即可, 下面是对此过程的总结

3. 确定编译环境

本文使用的是 C++.net framework 1.1, 以下为该编译环境下的步骤

4. 新建 config.h 文件

由于 ACE 支持很多操作系统, 因此必须建立一个 config.h 文件, 在该文件包含相应的操作系统的头文件, 从而让 ACE 知道将在什么操作系统下进行编译

进入 C:\ACE_wrappers\ace 目录, 新建 config.h 文件, 在 config.h 文件中添加

```
#include "ace/config-win32.h"
```

如果系统为 Windows98/Me, 那么需要添加

```
#define ACE_HAS_WINNT4 0
```

如果需要使用标准 C++ 的类库 (例 iostream 等), 那么需要添加

```
#define ACE_HAS_STANDARD_CPP_LIBRARY 1
```

如果需要使用 MFC (不推荐), 那么需要添加

```
#define ACE_HAS_MFC 1
```

5. 设置环境变量

开始->运行->cmd

```
set ACE_ROOT=C:\ACE_wrappers
```

6. 使用 .net 环境编译

打开 ACE.sln, 可进行 debug/release 编译

7. 生成 lib.dll 文件

生成的 lib 和 dll 文件在 C:\ACE_wrappers\lib。

2.4.3 在 Linux 上编译

注: 该步骤使用 ACE-5.4.7 测试成功

1. 解压缩到本地目录

例:

```
#pwd
/home/xchen
#tar -xzf ACE.tar.gz
```

2. 设置环境变量

```
#su -
#vi /etc/profile
```

在 export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE INPUTRC 下加入以下两行

```
export ACE_ROOT=/usr/local/ACE_wrappers
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ACE_ROOT/ace
#. /etc/profile
```

3. 建立编译环境

```
#cd /home/xchen/ACE_wrappers
#mkdir build
#cd build
#../configure
```

4. 配置 config.h 文件

../configure 之后, 系统会在/home/xchen/ACE_wrappers/build 下建立 ace/conig.h 文件
在// ACE configuration header file 下行添加

```
#include "ace/config-linux.h"
```

5. 编译

```
#cd /home/xchen/ACE_wrappers/build
#make
```

6. 生成 .so 文件

生成的 so 文件在/home/xchen/ACE_wrappers/build/ace/.libs 目录下。

2.5 前行的路标

对 ACE 感兴趣, 可以访问 ACE 网站: <http://www.cs.wustl.edu/%7Eschmidt/ACE.html>

ACE 的 Yahoo Group: <http://tech.groups.yahoo.com/group/ace-users/>

ACE 开发者论坛(中文): <http://www.acejoy.com/Index.html>

3. ACE Reactor 框架

3.1 Reactor (反应器) 框架

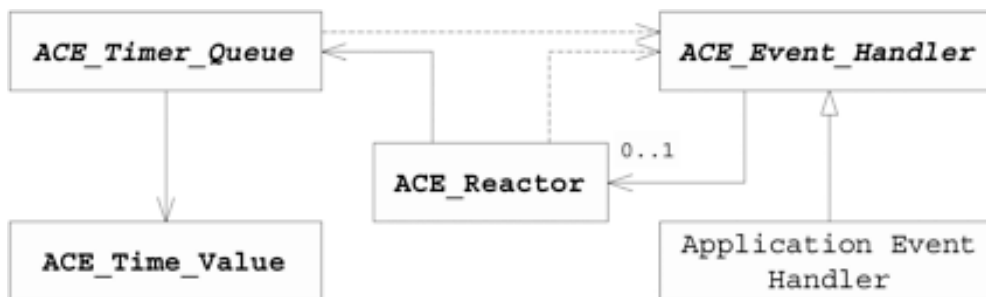
在编写通讯程序时, 如果服务器需要处理多个客户端的连接, 传统做法是为每个客户端创建一个进程或者线程。在大多数情况下, 这种方法是可以解决问题的。但是, 在某些情况下, 进程线程创建维护并发的开销是无法让人接受的。

而 ACE Reactor 框架通过事件多路分离器, 只需要用一个进程或者线程就可以处理多个客户端的连接的事件。而且用户程序使用 Reactor 框架也非常的简单, 只需要做 3 件事情:

- 1) 从 ACE_Event_Handler 派生子类，并 Overwrite 特定的事件虚回调方法。（比如 handle_input, handle_timeout）。
- 2) 向 ACE_Reactor 类登记你刚才创建的 ACE_Event_Handler 子类。
- 3) 运行 ACE_Reactor 事件循环。

Reactor 的含义—被动反应，可以让我们理解它的精髓：它是事件驱动的，就像 Windows 的事件驱动一样。

下面是 ACE Reactor 框架的类图和描述。



ACE 类	描述
ACE_Time_Value	提供时间和持续时间(Duration)的可移植，规范化的表示
ACE_Event_Handler	抽象类，其定义的挂钩(hook)方法是 ACE_Reactor 回调的目标。大多数通过 ACE 开发的应用事件处理器都是 ACE_Event_Handler 的后代。
ACE_Timer_Queue	抽象类，定义定时器队列的能力和接口。ACE 含有多种派生自 ACE_Timer_Queue 的类，为不同的定时机制提供了灵活的支持。
ACE_Reactor	提供了一个接口，用来在 ACE 框架中管理事件处理器登记，并执行事件循环来驱动事件检测，多路分离和分派。

我们这里只关心 ACE_Event_Handler 和 ACE_Reactor 类。

3.1.1 ACE_Event_Handler(事件处理器)

ACE_Event_Handler 是 ACE 中所有反应式事件处理器的基类。其接口如下所示：

方法	描述
----	----

<code>ACE_Event_Handler()</code>	指派与事件处理器相关联的 <code>ACE_Reactor</code> 指针。
<code>~ACE_Event_Handler()</code>	将其自身从反应器的通知机制中移出。
<code>handle_input()</code>	输入事件(如 连接或数据事件)发生时的挂钩(hook)方法。
<code>handle_output()</code>	输出事件成为可能时(如 流控制缓和或非阻塞式连接完成时)被调用的挂钩方法。
<code>handle_exception()</code>	异常事件(如 <code>TCP</code> 紧急数据的到达)发生时被调用的挂钩方法。
<code>handle_timeout()</code>	在定时器到期时被调用的挂钩方法。
<code>handle_signal()</code>	<code>OS</code> 发出信号时(或是 <code>POSIX</code> 信号, 或是 <code>Windows</code> 同步对象迁移到激发(Signaled)状态时)被调用的挂钩方法。
<code>handle_close()</code>	在其他 <code>handle_*()</code> 挂钩方法返回-1 或者当 <code>ACE_Reactor::remove_handler()</code> 被显式调用来解除事件处理器的登记时, 执行用户定义的中止活动的挂钩方法。
<code>get_handle()</code>	返回底层的 I/O 句柄。如果事件处理器只处理时间驱动的时间, 该方法可实现为 <code>no-op</code> (空操作)
<code>reactor()</code>	访问器, 用于 获取/设置可与 <code>ACE_Event_Handler</code> 相关联的 <code>ACE_Reactor</code> 指针。
<code>priority()</code>	访问器, 用于 获取/设置 事件处理器的优先级。

在开发一个普通的 C/S 应用程序时, 只需 Overwrite `handle_input()` 和 `handle_close()` 这两个回调方法也就足够了。

可以在 `Reactor` 的定时器队列设置定时器(参考下面的 `Reactor` 接口), 当超时事件发生时 `handle_timeout()` 会来处理超时时应执行的操作。

下面讨论一下使用 `ACE_Event_Handler` 时注意的 3 个方面:

1. 事件类型:

在向 `Reactor` 登记事件处理器的时候, 必须指定事件类型。事件类型定义在 `ACE_Event_Handler` 中, 包括下面几种:

事件类型	描述
<code>READ_MASK</code>	指定输入事件(如数据出现在 <code>socket</code> 或文件句柄上)。反应器分派 <code>handle_input()</code> 来处理输入事件。

WRITE_MASK	指定输出事件（如在流控制缓和时）。反应器分派 <code>handle_output()</code> 来处理输出事件。
EXCEPT_MASK	指定异常事件（如紧急数据出现在 <code>socket</code> 上）。反应器分派 <code>handle_exception()</code> 来处理异常事件。
ACCEPT_MASK	指定被动模式的连接事件。反应器分派 <code>handle_input()</code> 来处理连接事件。
CONNECT_MASK	指定非阻塞式连接的完成。反应器分派 <code>handle_output()</code> 来处理非阻塞式连接的完成事件。

可以对这些事件进行组合（|）来高效的定义一组事件。这组事件被填充在

`ACE_Reactor::register_handler()` 方法的 `ACE_Reactor_Mask` 参数。

2. 挂钩方法的返回值

登记的事件发生时，反应器会分派相应的 `handle_*` 来处理。而这些方法的返回值需注意：

返回值	描述
0	指示反应器应继续为此事件处理器检测和分派所登记的事件。对于需要重复处理一个事件的事件处理器（比如当有数据可读时从 <code>Socket</code> 读取数据），这种行为是常见的。
> 0	也指示反应器应继续为此事件处理器检测和分派所登记的事件。此外，如果在处理了某个 I/O 事件后返回值大于 0，反应器将在阻塞它的事件多路分离器之前，再次在句柄上分派此事件处理器。
-1	指示反应器停止为此事件处理器检测已登记的事件。在反应器移除此事件处理器之前，它调用事件处理器的 <code>handle_close()</code> 方法，传给该方法正在被解除登记的事件的 <code>ACE_Reactor_Mask</code> 值。

所以，我们在使用 Reactor 框架来开发 C/S 程序时，一般应在 `handle_*` 方法里返回 0。

3. 清理事件处理器

`handle_close()` 方法会在某个挂钩方法决定要进行清理时被调用。`handle_close()` 方法可随即进行用户定义的关闭活动，如释放内存，关闭 IPC 对象等。Reactor 框架会忽略 `handle_close()` 方法的返回值。

如上所述，Reactor 只会在挂钩方法返回负值，或事件处理器被显式移除时调用 `handle_close()`，而不会在 IPC 机制到达文件末尾或 I/O 句柄被本地或远程关闭时调用 `handle_close()`。因此，应用必须检测 I/O 句柄何时关闭，并采取措施。比如，当 `recv()` 或 `read()` 调用返回 0 时，事件处理器应从 `handle_*` 方法里返回 -1，或调用 `ACE_Reactor::remove_handler()`。

3.1.2 ACE_Reactor

`ACE_Reactor` 是 Reactor 模式的核心类，Reactor 模式的注册事件处理器，运行事件循环等功能都要通过这个类来进行。

`ACE_Reactor` 实现了 Façade 模式，为应用程序提供了各种访问 Reactor 框架特性的接口。

1. 初始化和析构方法：

方法	描述
<code>ACE_Reactor()</code> <code>open()</code>	这两个方法创建并初始化反应器的实例。
<code>~ACE_Reactor()</code> <code>close()</code>	这两个方法清理反应器在初始化时分配的资源。

2. 事件处理器管理方法

方法	描述
<code>register_handler()</code>	为基于 I/O 和信号的事件登记事件处理器
<code>remove_handler()</code>	移除某事件处理器，使其不再参加基于 I/O 和信号的事件分派
<code>suspend_handler()</code>	暂时停止分派事件给某事件处理器
<code>resume_handler()</code>	恢复为先前挂起的事件处理器分配事件
<code>mask_ops()</code>	获取，设置，增加或者清除与某事件处理器相关的事件类型及掩码
<code>schedule_wakeup()</code>	将指定的掩码增加到某事件管理器的条目中，该处理器在之前必须已通过 <code>register_handler()</code> 登记过。
<code>cancel_wakeup()</code>	从某事件处理器的条目中清除指定的掩码，但并不移除该事件处理器

3. 事件循环(Event-loop)方法

方法	描述
<code>handle_events()</code>	等待事件发生，并随即分派与之相关联的事件处理器。可通过超时参数限制花费在等待事件上的时间。
<code>run_reactor_event_loop()</code>	反复调用 <code>handle_events()</code> 方法，直至其失败，或者 <code>reactor_event_loop_done()</code> 返回 1，或者超时（可选）。

<code>end_reactor_event_loop()</code>	指示反应器关闭其事件循环。
<code>reactor_event_loop_done()</code>	在反应器的事件循环被 <code>end_reactor_event_loop()</code> 调用结束时返回 1

4. 定时器管理方法

方法	描述
<code>schedule_timer()</code>	登记一个事件处理器，它将在用户规定的时间后被执行。
<code>cancel_timer()</code>	取消一个或多个之前登记的定时器。

我们可以使用这两个方法来使代码具有超时特性，当超时发生时，事件处理器的 `handle_timeout()` 方法会被调用。

5. 通知方法

反应器拥有一种通知机制，应用可将其用于把事件和事件处理器插入反应器的分派引擎中。

方法	描述
<code>notify()</code>	将事件（及可选的事件处理器）插入反应器的事件检测器中，从而使其在反应器下次等待事件时被处理
<code>max_notify_iterations()</code>	设置反应器将在其通知机制中分派的处理器器的最大数目。
<code>purge_pending_notifications()</code>	从反应器的通知机制中清除指定的事件处理器或所有事件处理器

6. 实用方法(Utility Method)

方法	描述
<code>instance()</code>	静态方法，返回指向单体（Singleton） <code>ACE_Reactor</code> 的指针。这是通过 Singleton 模式和 Double-Checked Locking Optimization 模式来创建和管理的。
<code>owner()</code>	使某个线程“拥有”反应器的事件循环。

因为大部分的应用程序都需要实现 Server 或者 Client 端的内容，所以下面我们将分别实现一个 Reactor 的 Server 和 Client。ACE 也提供了实现 Server 和 Client 的半成品，Acceptor 和 Connector 框架。

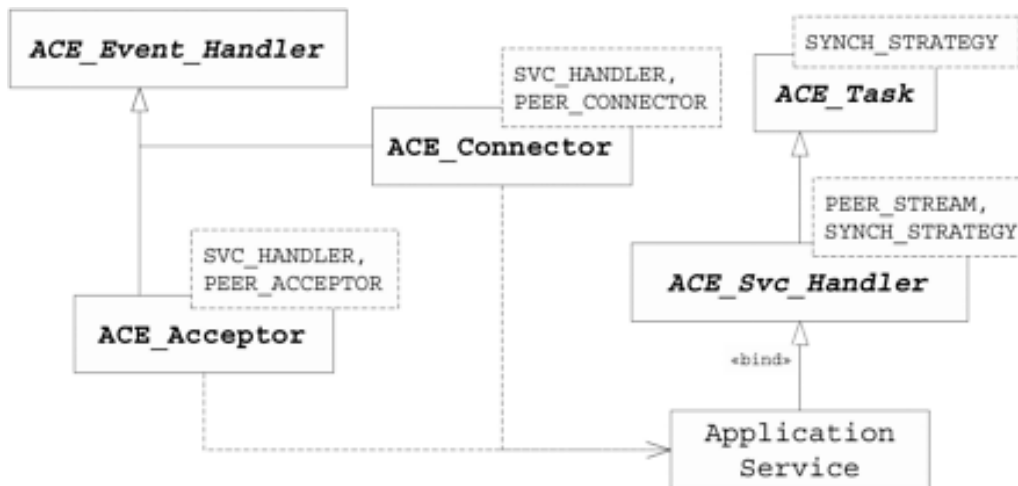
3.2 Acceptor(接受器)-Connector(连接器)框架

下面是 `Acceptor-Connector` 框架类的类图。

图中我们可以看到 `ACE_Acceptor`, `ACE_Connector` 继承自我们前面提到的 `ACE_Event_Handler`。

然后 `ACE_Svc_Handler` 是继承自 `ACE_Task` 的, 而 `ACE_Task` 其实也是 `ACE_Event_Handler` 的后代。根据前面提到的 Reactor 开发 3 步骤, 我们只需要将 `ACE_Svc_Handler` 登记到 Reactor 即可进行事件处理。

对我们开发应用程序而言, 我们只需要关心 `ACE_Acceptor`, `ACE_Connector`, `ACE_Svc_Handler` 这 3 个类。



ACE 类	描述
<code>ACE_Svc_Handler</code>	表示某个已连接服务的本地端, 其中含有一个用于与相连对端通信的 IPC 端点。
<code>ACE_Acceptor</code>	该工厂被动的等待接受连接, 并随即初始化一个 <code>ACE_Svc_Handler</code> 来响应来自对端的主动连接请求。
<code>ACE_Connector</code>	该工厂主动的连接到对端接受器, 并随即初始化一个 <code>ACE_Svc_Handler</code> 与其相连对端通信。

通俗的说, `ACE_Acceptor` 就是我们要实现的 Server, `ACE_Connector` 就是我们要实现的 Client。当 Client 连接到 Server 时, 会创建一个 `ACE_Svc_Handler` 来处理事件 (比如接收到数据, 连接被断开等); 而 Server 端也会为每个 Client 创建一个 `ACE_Svc_Handler` 来处理事件。

3.2.1 ACE_Svc_Handler(服务处理器)

如上所述，`ACE_Svc_Handler` 就是用来处理事件，那么主要需要处理那些事件呢？

1. 服务创建和激活方法

方法	描述
<code>ACE_Svc_Handler()</code>	由接受器或连接器在创建服务处理器时调用的构造方法
<code>open()</code>	由接受器或连接器自动调用来初始化服务处理器的挂钩方法。

`ACE_Svc_Handler` 已经在 `open` 方法里，把自己登记到 `Reactor` 事件循环，登记的事件类型为 `READ_MASK`。所以我们的代码将变得更加简单。

2. 服务处理方法

方法	描述
<code>svc()</code>	<code>ACE_Svc_Handler</code> 从 <code>ACE_Task</code> 继承来的 <code>svc()</code> 挂钩方法。在服务处理器的 <code>activate()</code> 方法被调用后，其大部分后续处理都可在 <code>svc()</code> 挂钩方法中执行。
<code>handle_*()</code>	<code>ACE_Svc_Handler</code> 从 <code>ACE_Event_Handler</code> 继承的 <code>handle_*()</code> 方法。因此，服务处理器可以向反应器登记自身，以在它感兴趣的各種事件发生时接收像 <code>handle_input()</code> 这样的回调。
<code>peer()</code>	返回指向底层 <code>PEER_STREAM</code> 的引用。服务处理器的 <code>PEER_STREAM</code> 是在其 <code>open()</code> 挂钩方法被调用时就绪的。任何处理方法都可使用这个访问器来获取指向已连接的 <code>IPC</code> 机制的引用。

3. 服务关闭方法

方法	描述
<code>destroy()</code>	可以用来直接关闭服务处理器
<code>handle_close()</code>	通过来自反应器的回调调用 <code>destroy()</code>
<code>close()</code>	从服务处理器退出时调用 <code>handle_close()</code>

应用可以直接调用 `destroy()` 来关闭服务处理器。该方法执行了以下步骤：

- 1) 从反应器处移除处理器
- 2) 取消任何与此处理器相关的定时器
- 3) 关闭对端流对象，以免发生句柄泄漏
- 4) 如果对象是被动态分配的，将其删除，以免内存泄漏。

下面讨论一下如何在代码中使用 `ACE_Svc_Handler`。

1. `ACE_Svc_Handler` 是一个类模板，定义它的子类时，要指定模板的类型。

✓ `PEER_STREAM`: 底层传递数据流的类型，Socket 连接使用 `ACE SOCK_Stream`

✓ `SYNCH_STRATEGY`: 同步策略，可以使用 `ACE_NULL_SYNCH`(无同步), `ACE_MT_SYNCH`(ACE 默认提供的同步策略)。

2. 子类覆盖事件类型相应的 `handle_*()` 方法。

3.2.2 ACE_Acceptor

`ACE_Acceptor` 是一个工厂，它实现了 `Acceptor-Connector` 中的 `Acceptor` 角色。也就是说起到了 `Server` 的作用。

首先我们看一下 `ACE_Acceptor` 提供了那些接口：

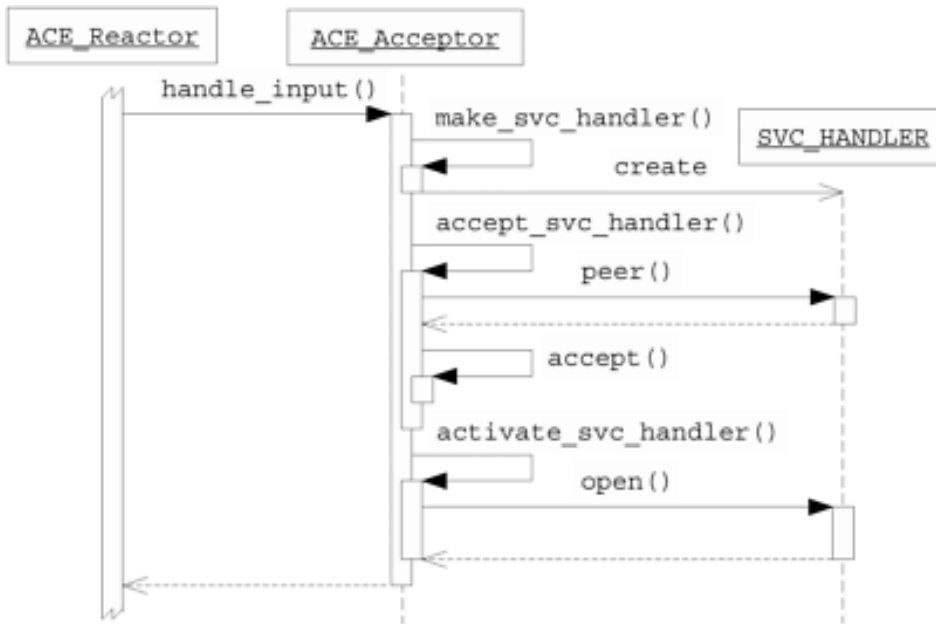
1. 初始化，析构及访问器方法

方法	描述
<code>ACE_Acceptor()</code> <code>open()</code>	将接受器的被动 IPC 端点绑定到特定地址，比如 TCP 端口或者 IPC 主机地址，然后对连接请求的到达进行监听
<code>~ACE_Acceptor()</code> <code>close()</code>	关闭接受器的 IPC 端点，并释放资源。
<code>acceptor()</code>	返回指向底层 <code>PEER_ACCEPTOR</code> 的引用。

2. 连接建立和服务处理器初始化方法

方法	描述
<code>handle_input()</code>	当连接请求从对端连接器到达时，反应器会调用此模板方法。它可以使用在下面概述的 3 个方法来使用“被动地连接 IPC 端点并创建和激活与其相关联的服务处理器”所必需的各步骤自动化。
<code>make_svc_handler()</code>	该工厂方法创建服务处理器来处理通过其已连接的 IPC 端点，从对端服务发出的数据请求
<code>accept_svc_handler()</code>	该挂钩方法使用接受器的被动模式 IPC 端点来创建已连接的 IPC 端点，并将此端点与服务处理器相关联的一个 I/O 句柄封装在一起
<code>activate_svc_handler()</code>	该挂钩方法调用服务处理器的 <code>open()</code> ，使服务处理器完成对自己的初始化。

下面的顺序图详细的描述了当连接请求到达时, `ACE_Acceptor` 所做的工作:



那么我们在应用程序中如何使用 `ACE_Acceptor` 呢?

1. 首先, 实现 `ACE_Svc_Handler` 子类, 然后覆盖相应的 `handle_*()` 事件回调方法。
2. 没有特殊需要的情况下, 我们不需要创建 `ACE_Acceptor` 的子类, 只须定义它的实例即可。

`ACE_Acceptor` 是一个类模板, 定义它的实例时, 要指定模板的类型。

- ✓ `SVC_Handler`: 指定 `Acceptor` 对应的 `ACE_Svc_Handler`, 当有连接请求时, `Acceptor` 会创建一个新的 `ACE_Svc_Handler` 来处理对端的事件。
- ✓ `PEER_ACCEPTOR`: 它能够被动的接受客户连接, 常被指定为 ACE 的各个 IPC Wrapper Façade。对于 Socket 来说, 可以指定为 `ACE_SOCKET_Acceptor`。

3. 使用 `Open()` 方法, 即可启动服务器, 开始监听客户机事件。对于 Socket 来说, `Open` 方法需要使用一个 `ACE_INET_Addr`。

3.2.3 ACE_Connector

同 `ACE_Acceptor` 相同, `ACE_Connector` 也是一个工厂, 它实现了 `Acceptor-Connector` 中的 `Connector` 角色。也就是说起到了 `Client` 的作用。

`ACE_Connector` 的接口有:

1. 连接器初始化, 析构方法以及访问器方法

方法	描述
----	----

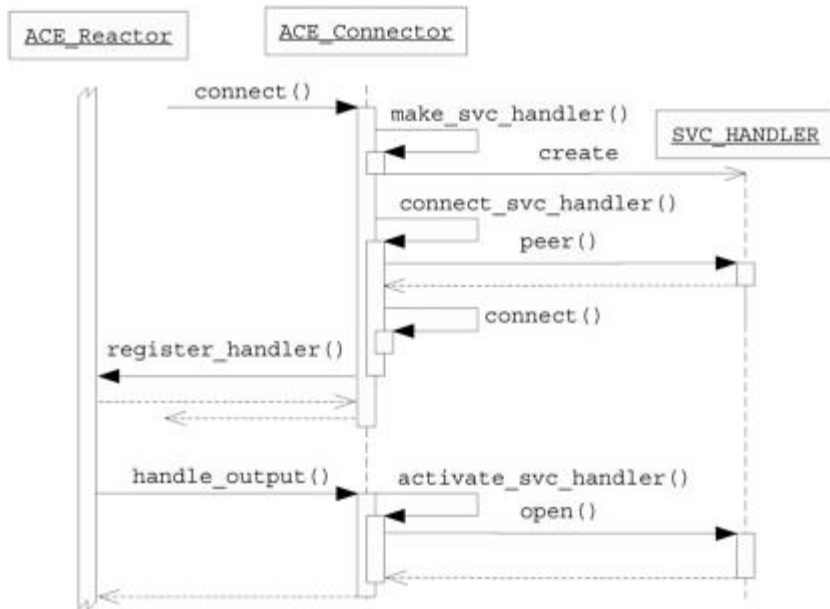
<code>ACE_Connector()</code> <code>open()</code>	用于初始化连接器的方法
<code>~ACE_Connector()</code> <code>close()</code>	释放连接器所用的资源
<code>connector()</code>	返回指向底层的 <code>PEER_CONNECTOR</code> 的引用

2. 连接建立和服务处理器初始化方法：可用于主动地建立连接，并对服务处理器进行初始化。

方法	描述
<code>connect()</code>	应用会在想要将某个服务处理器连接到监听的对端时，调用这个模板方法。它可以使用下面的 3 个方法来使“主动连接某个 IPC 端点，创建并激活与其相关联的服务处理器”所必需的各步骤自动化。
<code>make_svc_handler()</code>	该工厂方法提供服务处理器，后者会使用已连接的 IPC 端点
<code>connect_svc_handler()</code>	该挂钩方法使用服务处理器的 IPC 端点来同步或异步地主动连接端点
<code>activate_svc_handler()</code>	该挂钩方法调用服务处理器的 <code>open()</code> 挂钩方法，后者允许服务处理器在连接建立之后完成对其自身的初始化。
<code>handle_output()</code>	在异步发起的连接请求完成之后，反应器会调用这个模板方法。它调用 <code>activate_svc_handler()</code> 方法，以让服务处理器对其自身进行初始化。
<code>cancel()</code>	取消某服务处理器，其连接之前是异步发起的。调用者—不是连接器—负责关闭服务处理器。

与 `Acceptor` 有些不同的是，`Connector` 在连接时可以选择同步或异步连接。因为对于 `Acceptor` 来说，建立连接通常是即时的，而主动连接建立就要花较长的一段时间，尤其是广域网。我们这里只讨论同步连接的情形。

下面的顺序图详细的描述了异步连接建立中的各步骤：



接下来讨论一下在应用程序中如何使用 `ACE_Connector` 呢？(同 `ACE_Acceptor` 基本类似)

1. 首先，实现 `ACE_Svc_Handler` 子类，然后覆盖相应的 `handle_*()` 事件回调方法。
2. 没有特殊需要的情况下，我们也不需要创建 `ACE_Connector` 的子类，只须定义它的实例即可。

`ACE_Connector` 是一个类模板，定义它的实例时，要指定模板的类型。

- ✓ `SVC_Handler`: 指定 `Connector` 对应的 `ACE_Svc_Handler`，同服务器建立连接时，`Connector` 会创建一个新的 `ACE_Svc_Handler` 来处理对端的事件。
- ✓ `PEER_Connector`: 它能够主动的建立客户连接，常被指定为 ACE 的各个 IPC Wrapper Façade。对于 Socket 来说，可以指定为 `ACE_SOCKET_Connector`。

3. 使用 `connect()` 方法去主动连接。`connect` 方法里的参数有你刚才创建的 `ACE_Svc_Handler` 子类的指针和待连接的 `PEER_Address`(对于 Socket 来说，是 `ACE_INET_Addr`)。

3.3 ACE Reactor Server (Demo)

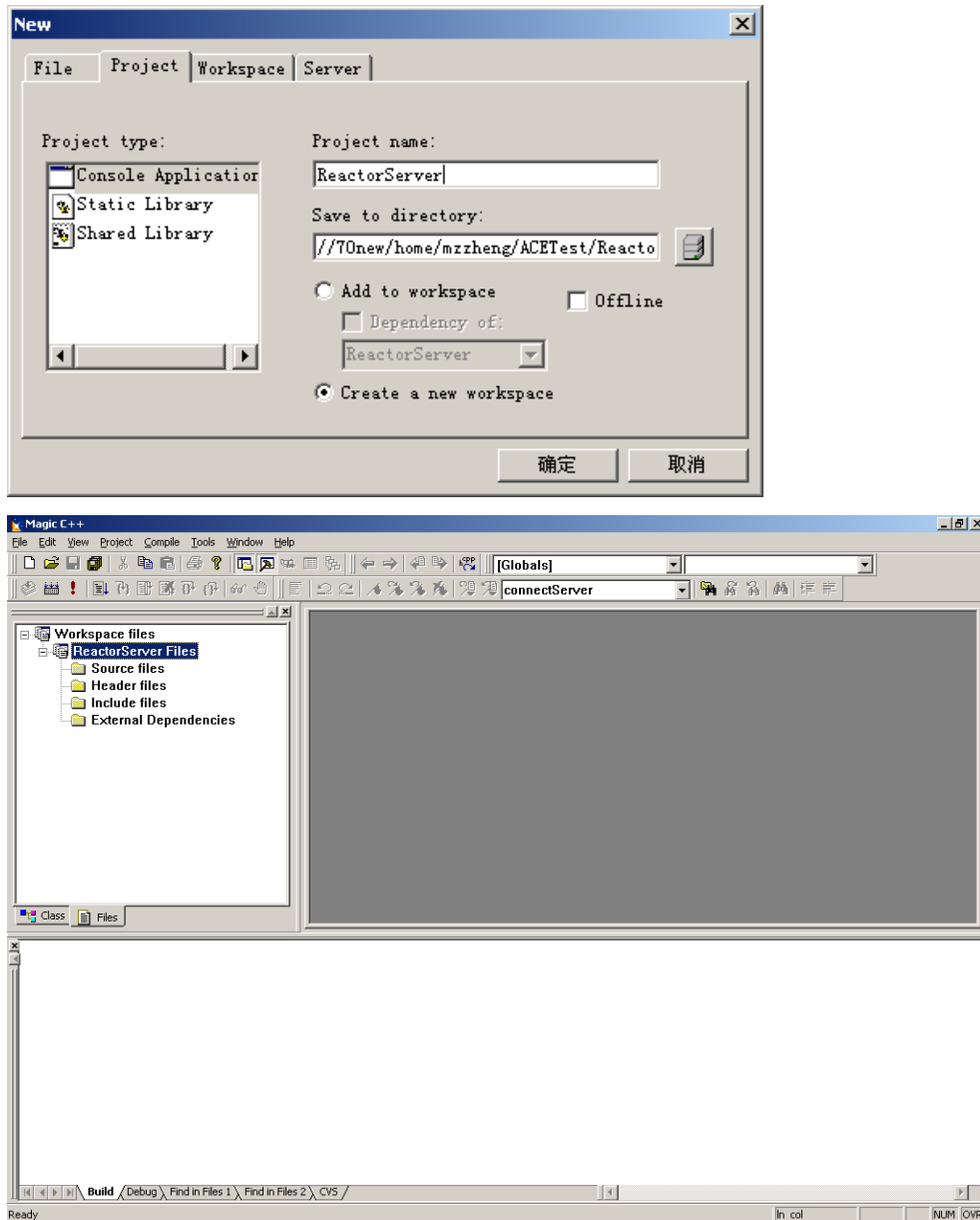
3.3.1 需求

下面我们实现一个简单的 `Server` 端，这个 `Server` 能够处理多个 `Client` 的连接，在收到 `Client` 的消息后，原封不动的将消息再发还给 `Client`。

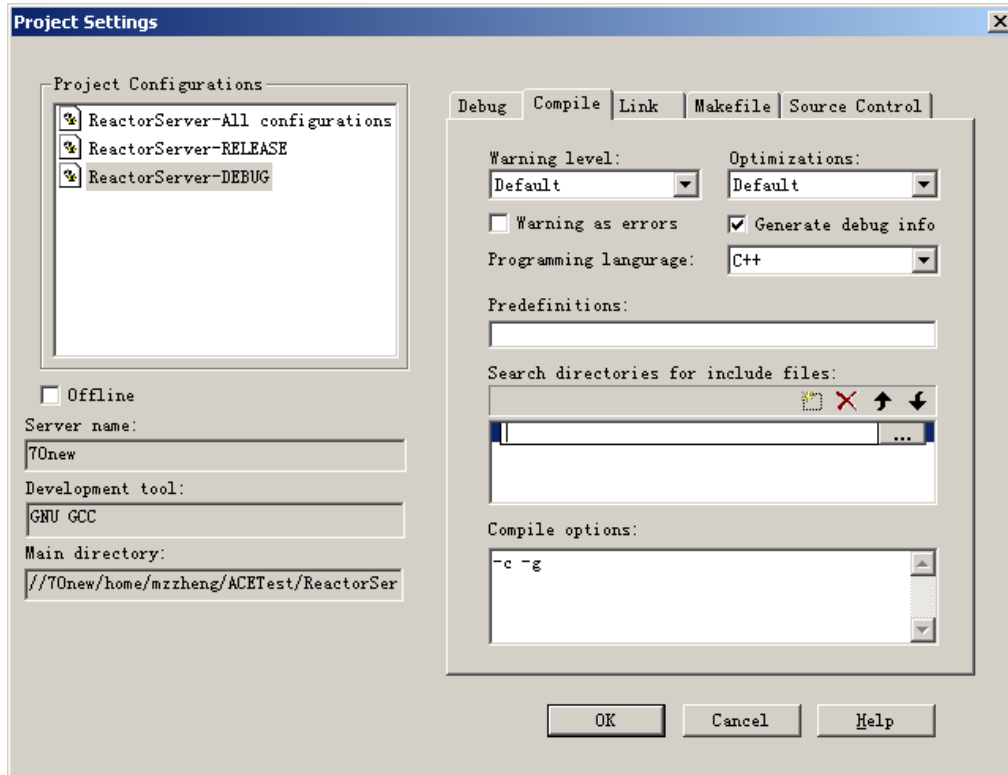
3.3.2 实现

接下来我们一步一步实现它（在 RedHat 9，使用 Magic C++开发，最终的代码可以在前面指定位置获得）：

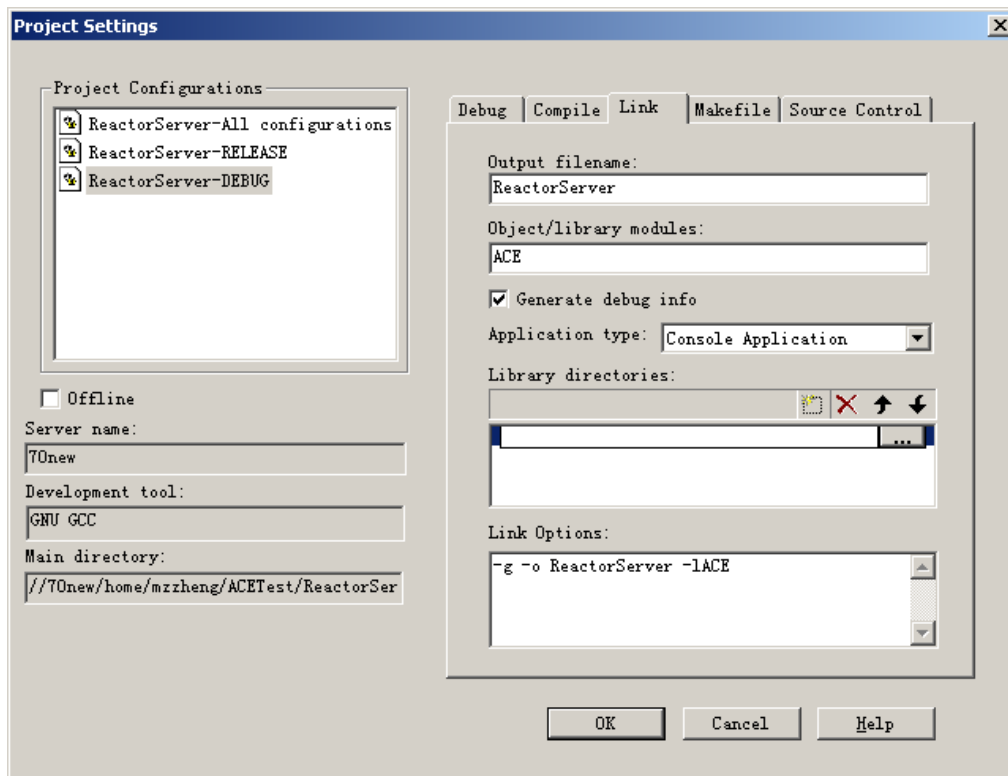
- 1. 使用 Magic C++在 Linux 服务器上建立一个新工程 ReactorServer



- 2. 如果系统不能从环境变量等找到 include 的 ACE 头文件，那么你需要在项目设定的 Compile 项中指定。
比如 /home/work/ACE_wrappers



- 3. 在项目设定的 Link 的选项卡中，库文件处加入 ACE 库的支持。同上，如果系统不能从环境变量等找到 ACE 库文件，你也需要在下面的 Library directories 中指定。



- 4. 准备工作做好之后，下面我们创建 服务处理器。创建新类 `ServerHandler`，继承自 `ACE_Svc_Handler`。修改头文件 `ServiceHandler.h`。

1) 由于我们创建的是 Socket 服务器，`ACE_Svc_Handler` 的模板里 `PEER_STREAM` 被指定为 `ACE_SOCK_STREAM`，同时我们不需要使用同步机制，模板里的 `SYNCH_STRATEGY` 被指定为 `ACE_NULL_SYNCH`。

```
class ServerHandler : public ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
```

2) 我们只关心输入事件，也就是客户端发来的消息，覆盖 `handle_input` 方法。

```
virtual int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE);
```

下面是 `ServiceHandler.h` 的完整代码

```
#ifndef SERVERHANDLER_H
#define SERVERHANDLER_H

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

class ServerHandler : public ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
{
public:
    virtual int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE);
};
#endif
```

- 5. 下面我们来实现 `ServiceHandler` 的 `handle_input` 方法。修改 cpp 文件 `ServiceHandler.cpp`。

1) 在 `handle_input` 方法里，我们定义了一个 4096 的 char 数组。并初始化该数组。

```
const int INPUT_SIZE = 4096;
char buffer[INPUT_SIZE];
memset(buffer, 0, INPUT_SIZE);
```

2) 然后使用底层的 peer 来接收收到的消息，`recv` 返回了收到的消息的 length。

```
int recv_cnt = this->peer().recv( buffer, sizeof(buffer));
```

3) 如果 `length <= 0`，说明对端的连接被断开，这时候应返回-1，指示反应器停止为此事件处理器检测已登记的事件。

```
if (recv_cnt <= 0 )
```

```

{
    printf("Connection close\n");
    return -1;
}

```

4) 然后我们再使用底层的 peer 将收到的消息原封不动的发回去。

```
this->peer().send(buffer, recv_cnt);
```

5) `handle_input` 结束时，我们应该返回 0，当下次有消息到来时，仍能被此服务处理器处理。

```
return 0;
```

下面是 ServiceHandler.cpp 的完整代码

```

#include "ServerHandler.h"

int ServerHandler::handle_input (ACE_HANDLE)
{
    const int INPUT_SIZE = 4096;
    char buffer[INPUT_SIZE];
    memset(buffer, 0, INPUT_SIZE);
    int recv_cnt = this->peer().recv( buffer, sizeof(buffer));

    if (recv_cnt <= 0 )
    {
        printf("Connection close\n");
        return -1;
    }

    this->peer().send(buffer, recv_cnt);
    return 0;
}

```

□ 6. 最后我们实现 main 方法，增加 ReactorServer.cpp 文件

1) 在 main 方法，首先定义监听的 IP 地址，对于我们的 Server 应用来说，只须指定监听的端口即可。我们在 9000 端口监听。（`ACE_INET_Addr` 后面介绍）

```
ACE_INET_Addr port_to_accept(9000);
```

2) 定义我们的 Server，`Acceptor` 模板类的 `SVC_Handler` 指定为我们刚刚实现的 `ServerHandler`，

PEER_ACCEPTOR 指定为 `ACE_SOCKET_Acceptor`。然后使用 `open` 方法开始监听。

```
ACE_Acceptor<ServerHandler, ACE_SOCKET_ACCEPTOR> server;
if(server.open(port_to_accept) == -1)
{
    return -1;
}
```

3) 接下来运行 `ACE_Reactor` 事件循环。

```
ACE_Reactor::instance()->run_reactor_event_loop();
```

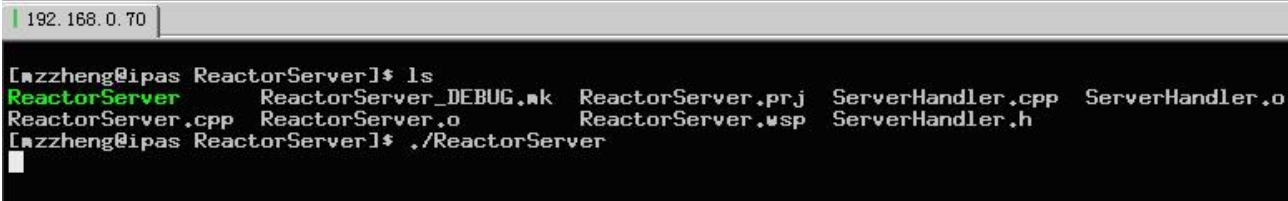
下面是 `ReactorServer.cpp` 的完整代码

```
#include "ace/Reactor.h"
#include "ace/Acceptor.h"
#include "ace/Socket_Acceptor.h"
#include "ServerHandler.h"

int main(int argc, char* argv[])
{
    ACE_INET_Addr port_to_accept(9000);
    ACE_Acceptor<ServerHandler, ACE_SOCKET_ACCEPTOR> server;
    if(server.open(port_to_accept) == -1)
    {
        return -1;
    }

    ACE_Reactor::instance()->run_reactor_event_loop();
    return 0;
}
```

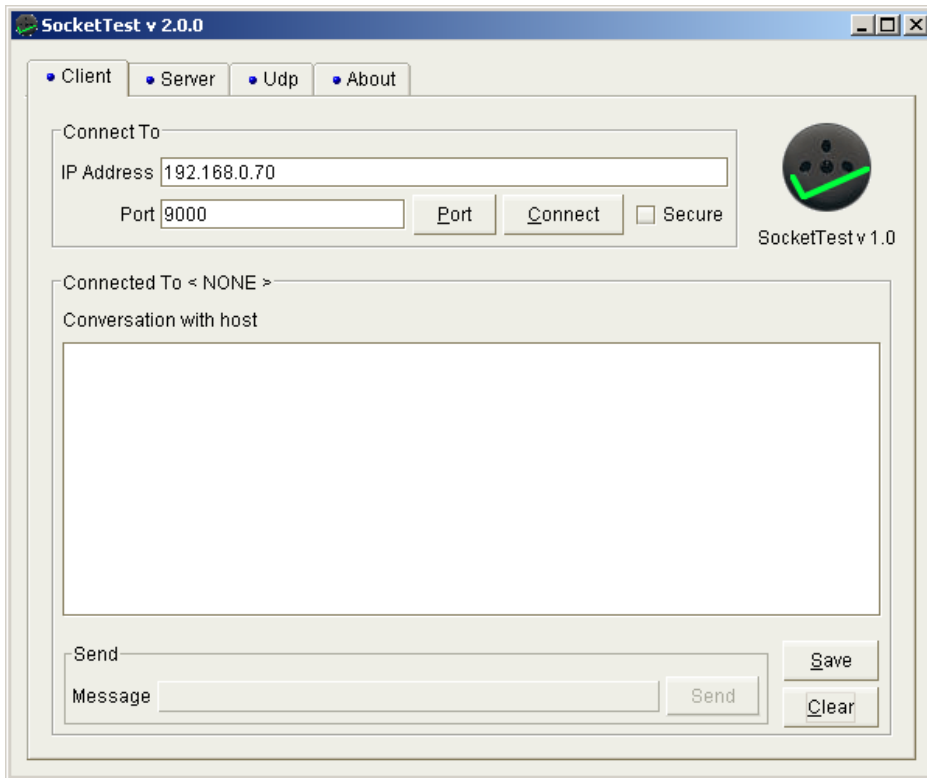
□ 7. 接下来编译，运行 `ReactorServer`



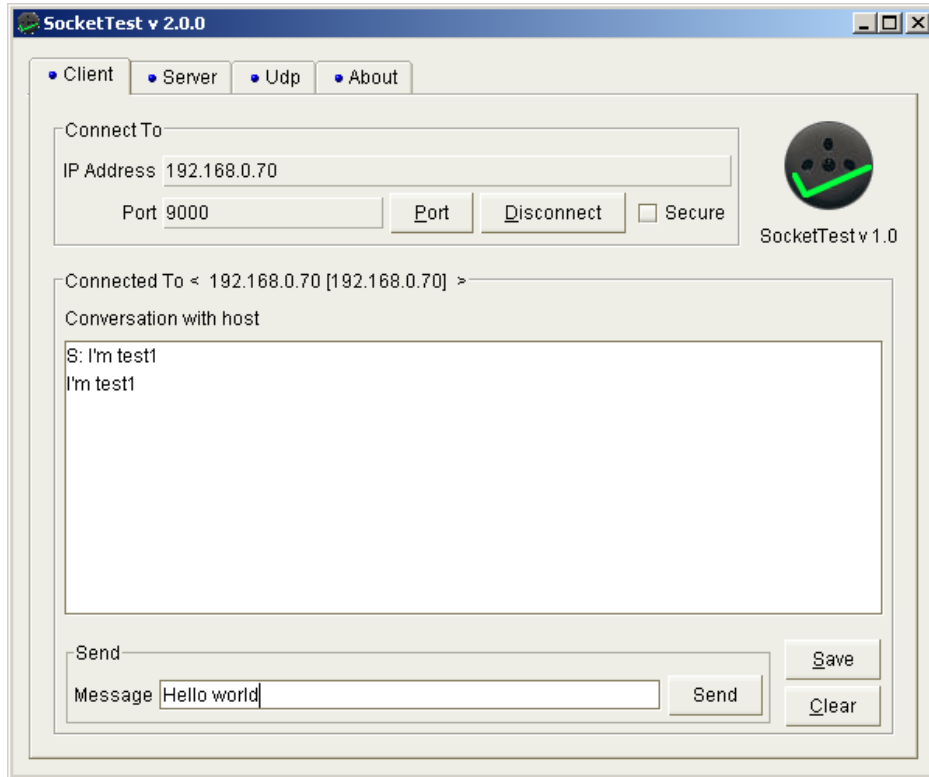
```
[mzhang@ipas ReactorServer]$ ls
ReactorServer      ReactorServer_DEBUG.mk  ReactorServer.prj  ServerHandler.cpp  ServerHandler.o
ReactorServer.cpp  ReactorServer.o         ReactorServer.usp  ServerHandler.h
[mzhang@ipas ReactorServer]$ ./ReactorServer
```

□ 8. 下面我们使用一个客户端来测试一下。这里使用的是 `SocketTest`，一个 Java 写的 `Socket` 测试工具，你可以在前面指定的位置上找到它。

打开 SocketTest，输入我们的 Server 所在的 IP 地址和端口后，点击 Connect。

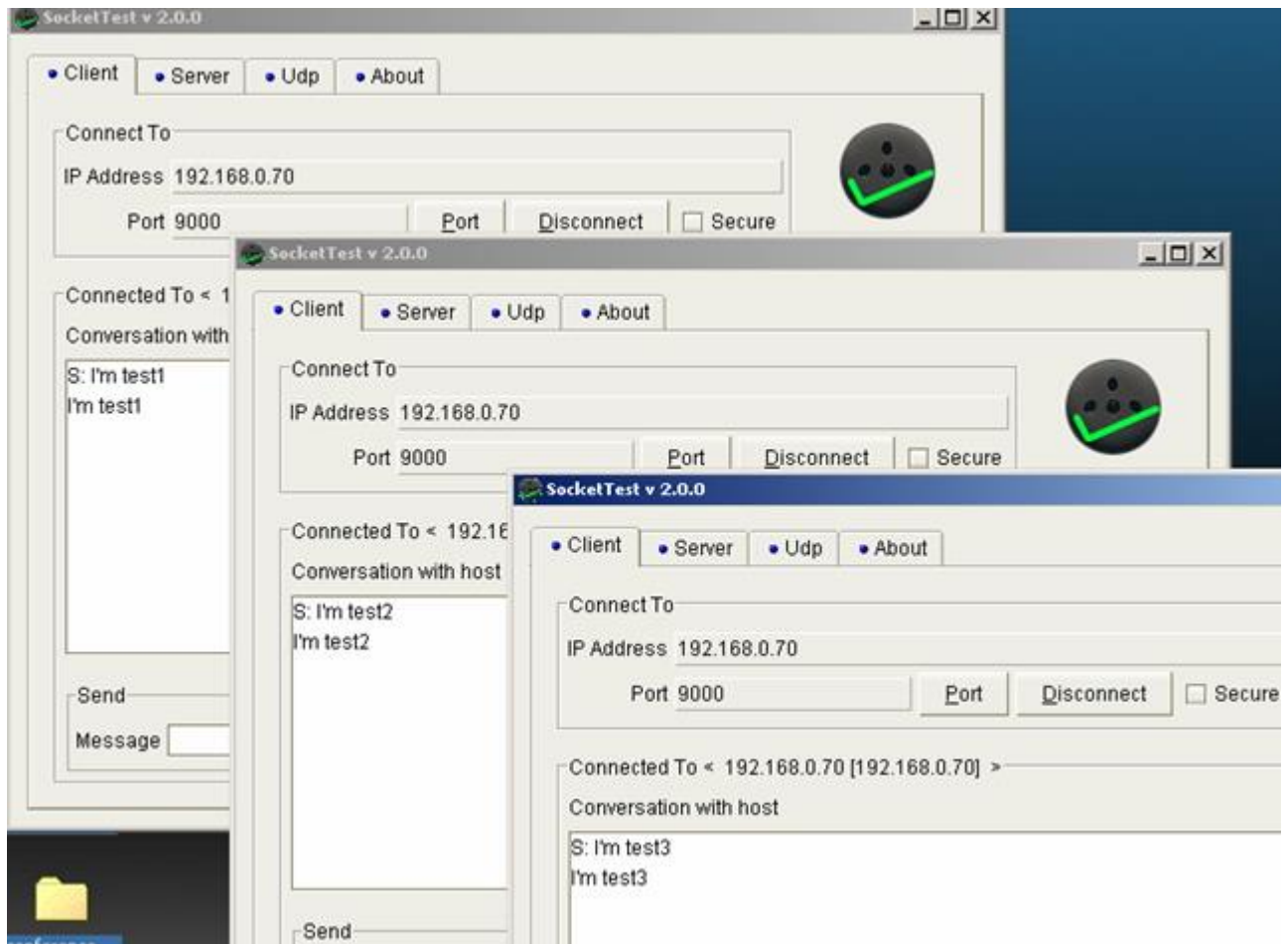


连接之后，在下面的 Message 框里输入一句话，然后点 Send 按钮，我们就会看到 Conversation with host 框里看到我们发过去的消息，以及服务器原封不动回复的消息。



- 9. 上面的测试说明我们的服务器是按照我们的预想工作的，下面我们再测试一下同时有几个客户端连接服务器的情况。

我们再打开 2 个 SocketTest，重复前面的操作。



- 10. 多个客户端同时连上去也没有关系，我们的 Server 同样能很好的运行。

这个 Server 实现了以下功能：

- ✓ 在指定端口监听
- ✓ 处理多个客户端的请求
- ✓ 将接受到的消息回复给客户端
- ✓ 这个 Server 是跨平台的，只需在不同平台分别编译运行。

而我们完成这些功能只用了：

- ✓ 一个线程
- ✓ 几十行代码
- ✓ 几分钟的时间

由此可见，使用 ACE 的好处是我们无需再关心 Socket 等连接的细节，也无需关心为每个客户端创建线程，我们只需专注业务逻辑，这极大的提高了我们的效率。

3.3.3 ACE 工具类

前面我们在编写 Server 时用到了 `ACE_INET_Addr`，其实 ACE 提供了很多的工具类（宏）。下面对这些工具类做一简单的介绍，更详细的内容请自行阅读参考资料。

作用	类（宏）	描述
地址包装类	<code>ACE_INET_Addr</code>	<code>ACE_INET_Addr</code> 类封装了 Internet 域地址族（ <code>AF_INET</code> ）。该类派生自 ACE 中的 <code>ACE_Addr</code> 。它的多种构造器可用于初始化对象，使其具有特定的 IP 地址和端口。除此而外，该类还具有若干 <code>set</code> 和 <code>get</code> 方法，并且重载了比较操作，也就是 <code>==</code> 操作符和 <code>!=</code> 操作符。
	<code>ACE_UNIX_Addr</code>	<code>ACE_UNIX_Addr</code> 类封装了 UNIX 域地址族（ <code>AF_UNIX</code> ），它也派生自 <code>ACE_Addr</code> 。该类的功能与 <code>ACE_INET_Addr</code> 类相似。
时间包装类	<code>ACE_TIME_VALUE</code>	ACE 的时间包装类，该类使用秒(<code>sec</code>)和微秒(<code>usec</code>)来描述时间。
日志宏	<code>ACE_DEBUG</code> / <code>ACE_TRACE</code>	<code>ACE_DEBUG</code> (与 <code>ACE_TRACE</code> 几乎相同)和 <code>ACE_ERROR</code> 宏用于打印调试及错误信息及记录相应的日志。它们的用法是： <code>ACE_DEBUG((severity, formatting-args))</code> <code>ACE_ERROR((severity, formatting-args))</code> <code>severity</code> 为严重级别，最常用的是 <code>LM_DEBUG</code> , <code>LM_ERROR</code> 。 <code>formatting-args</code> 使用了 <code>ACE_TEXT</code> 宏， <code>ACE_TEXT</code> 可以使用 <code>%d</code> , <code>%t</code> 等格式修饰符。就如在 <code>printf</code> 格式串中一样，下面简单介绍几个格式修饰符（区分大小写）。 <code>%P</code> 当前进程的 ID <code>%t</code> 调用线程的 ID <code>%T</code> 当前时间，格式为 <code>hour:min:sec.usec</code> <code>%s</code> 字符串
	<code>ACE_ERROR</code>	
内存分配宏	<code>ACE_NEW(p, c)</code>	使用构造器 <code>c</code> 分配内存，并把指针赋给 <code>p</code> 。失败时， <code>p</code> 被设置为 0，并执行 <code>return</code>
	<code>ACE_NEW_RETURN(p, c, r)</code>	使用构造器 <code>c</code> 分配内存，并把指针赋给 <code>p</code> 。失败时， <code>p</code> 被设置为 0，并执行 <code>return r</code>
	<code>ACE_NEW_RETURN(p, c)</code>	使用构造器 <code>c</code> 分配内存，并把指针赋给 <code>p</code> 。失败时， <code>p</code> 被设置为 0，并继续执行下

		一条语句。
--	--	-------

3.3.4 Server 改进

下面对我们刚才的 Server 进行改进，让它打印出一些调试信息来。

- 1) Client 连接时，打印出 Client 端的 IP 地址；
- 2) 收到 Client 的消息时，打印出消息的内容；
- 3) Client 断开时，打印出 Client 断开的信息。

□ 1. 打开 ServiceHandler.h 进行修改：

- 1) 为方便起见，我们创建了 super 类型定义。这样，在我们的类中调用基类方法时，可读性将大为提高。

```
typedef ACE_Svc_Handler<ACE_SOCKET_STREAM, ACE_NULL_SYNCH> super;
```

- 2) 当 Client 连接时，打印出 Client 端的 IP 地址，应该在 open() 方法里面实现，所以我们继承父类的 open 方法：

```
int open(void * = 0);
```

- 3) 将 Client 端的 IP 地址保存在类里，因此增加一个属性：

```
private:
    ACE_TCHAR peer_name[MAXHOSTNAMELEN];
```

ServiceHandler.h 修改完成，下面是改进后的 ServiceHandler.h 的完整代码

```
#ifndef SERVERHANDLER_H
#define SERVERHANDLER_H

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

class ServerHandler : public ACE_Svc_Handler<ACE_SOCKET_STREAM, ACE_NULL_SYNCH>
{
    typedef ACE_Svc_Handler<ACE_SOCKET_STREAM, ACE_NULL_SYNCH> super;

public:
    int open(void * = 0);
    virtual int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE);

private:
    ACE_TCHAR peer_name[MAXHOSTNAMELEN];
};
```



```
#endif
```

□ 2. 修改类文件 ServiceHandler.cpp。

1) 增加 open 方法，我们并不打算改变父类 open 的行为，所以首先调用父类的 open 方法。

```
int ServerHandler::open(void *p)
{
    if(super::open(p) == -1)
    {
        return -1;
    }
    return 0;
}
```

2) 接下来在 open 方法里获得 Client 的 IP 地址。使用底层 peer 的 get_remote_addr 方法。得到地址之后，我们 ACE_DEBUG 用打印出来，同时打印出时间，进程号，线程号。

```
ACE_INET_Addr peer_addr;
if (this->peer().get_remote_addr(peer_addr) == 0
    && peer_addr.addr_to_string(peer_name, MAXHOSTNAMELEN) == 0)
{
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection from %s\n"),
peer_name));
}
```

3) 修改 handle_input 方法，把原来用 printf 写的方法改成 ACE_DEBUG。再把收到的消息打印出来。

```
if (recv_cnt <= 0 ) {
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection close from %s\n"),
peer_name));
    return -1;
}

ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Receive msg from %s: %s\n"),
peer_name, buffer));
```

下面是改进后的 ServiceHandler.cpp 的完整代码

```
#include "ServerHandler.h"

int ServerHandler::open(void *p)
{
    if(super::open(p) == -1)
    {
        return -1;
    }
}
```

```

    }

    ACE_INET_Addr peer_addr;
    if (this->peer().get_remote_addr(peer_addr) == 0
        && peer_addr.addr_to_string(peer_name, MAXHOSTNAMELEN) == 0)
    {
        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection from %s\n"),
peer_name));
    }
    return 0;
}

int ServerHandler::handle_input (ACE_HANDLE){
    const int INPUT_SIZE = 4096;
    char buffer[INPUT_SIZE];
    memset(buffer, 0, INPUT_SIZE);
    int recv_cnt = this->peer().recv( buffer, sizeof(buffer));

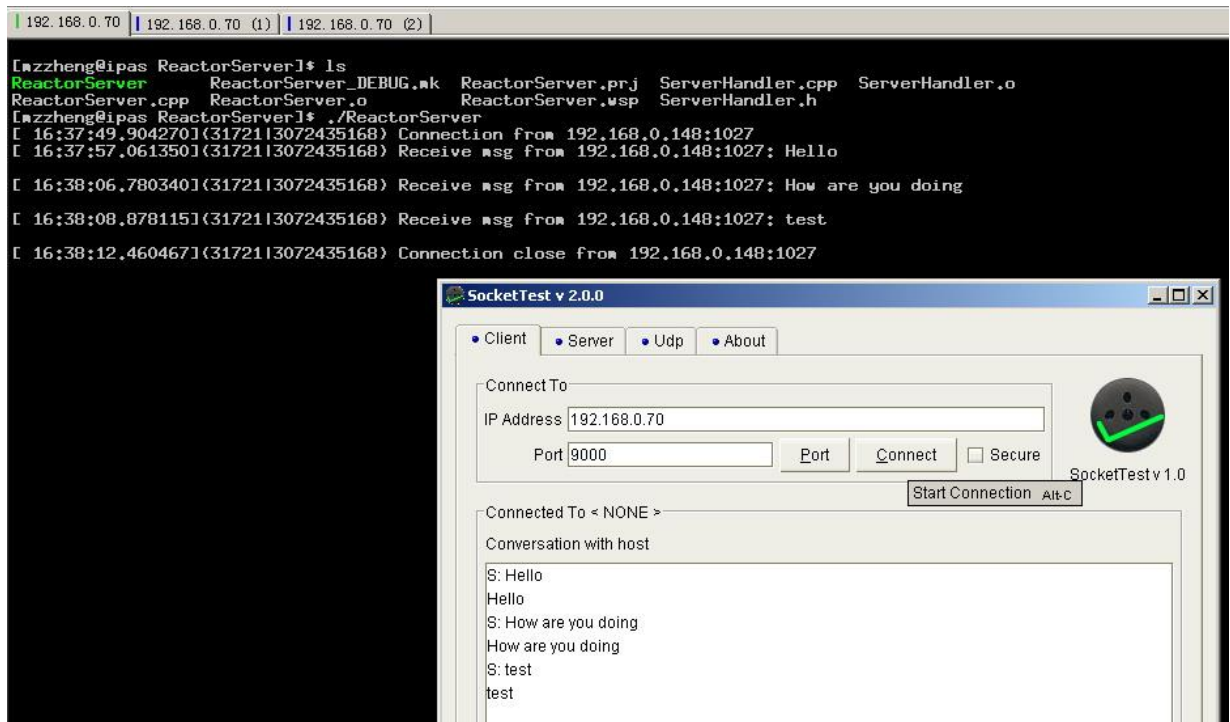
    if (recv_cnt <= 0 ) {
        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection close from %s\n"),
peer_name));
        return -1;
    }

    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Receive msg from %s: %s\n"),
peer_name, buffer));

    this->peer().send(buffer, recv_cnt);
    return 0;
}

```

- 3. 这样修改就完成了，接下来编译测试，运行 ReactorServer，然后运行 SocketTest 测试，效果类似下图：



3.4 ACE Reactor Client (Demo)

3.4.1 需求

上面 Reactor Server 的测试的客户端，我们使用了 SocketTest 这个工具。下面我们自己开发一个工具。这个工具能够：

- 1) 建立 5 个与服务器端的连接；
- 2) 每个连接每隔 2 秒钟发送一条测试消息给服务器，发送 5 次，然后断开同服务器的链接。

3.4.2 实现 I

接下来我们一步一步实现它（在 RedHat 9，使用 Magic C++开发，最终的代码可以在前面指定位置获得）：

1. 使用 Magic C++在 Linux 服务器上建立一个新工程 ReactorClient。
项目的设置等跟 ReactorServer 的设置相同
2. 准备工作做好之后，下面我们创建 服务处理器。 创建新类 ClientHandler，继承自

`ACE_Svc_Handler`。修改头文件 `ClientHandler.h` 。

1) 由于我们创建的是 Socket 服务器, `ACE_Svc_Handler` 的模板里 `PEER_STREAM` 被指定为 `ACE SOCK_STREAM`, 同时我们不需要使用同步机制, 模板里的 `SYNCH_STRATEGY` 被指定为 `ACE_NULL_SYNCH`。

```
class ClientHandler: public ACE_Svc_Handler<ACE SOCK_STREAM, ACE_NULL_SYNCH>
```

2) 覆盖 `handle_input` 方法, 以检查服务器是否将发出去的消息返回来。

```
virtual int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE);
```

下面是 `ClientHandler.h` 的完整代码

```
#ifndef CLIENTHANDLER_H
#define CLIENTHANDLER_H

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

class ClientHandler : public ACE_Svc_Handler<ACE SOCK_STREAM, ACE_NULL_SYNCH>
{
public:
    virtual int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE);
};
#endif
```

- 3. 下面我们来实现 `ClientHandler` 的 `handle_input` 方法。这个实现跟 `Server` 的几乎一样, 但只负责接收消息, 不再发送消息给服务器端。

下面是 `ClientHandler.cpp` 的完整代码

```
#include "ClientHandler.h"

int ClientHandler::handle_input (ACE_HANDLE){
    const int INPUT_SIZE = 4096;
    char buffer[INPUT_SIZE];
    memset(buffer, 0, INPUT_SIZE);
    int recv_cnt = this->peer().recv( buffer, sizeof(buffer));

    if (recv_cnt <= 0 ) {
        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection close\n")));
    }
}
```

```

        return -1;
    }

    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Receive msg from server: %s\n"),
buffer));
    return 0;
}

```

□ 4. 最后我们实现 main 方法，增加 ReactorClient.cpp 文件

1) 在 main 方法，首先定义要连接的 IP 地址，对于我们的 Client 应用来说，要连接的服务器为本机 9000 端口上的服务器。可以按照下面的方法来定义 `ACE_INET_Addr`。

```
ACE_INET_Addr port_to_connect(9000, ACE_LOCALHOST);
```

2) 定义我们的 Client，`Connector` 模板类的 `SVC_Handler` 指定为我们刚刚实现的 `ClientHandler`，`PEER_ACCEPTOR` 指定为 `ACE SOCK_Acceptor`。

```
ACE_Connector<ClientHandler, ACE SOCK_Connector> client;
```

3) 根据我们的需求，创建 5 个到服务器的连接，`connect` 方法和 `Acceptor` 的 `open` 不同，`open` 方法不需要制定 `SVC_Handler` 的指针，而 `connect` 方法需要。不过你可以传个空指针进去，`connect` 方法会自动创建一个新的 `SVC_Handler`，并且由于 `connect` 方法的参数是 `SVC_Handler` 指针的引用，所以我们能够得到 `connect` 方法里面创建的 `SVC_Handler` 的实际指针。

```

for(int i = 0; i < 5; i++)
{
    ClientHandler* handler = NULL;
    if(client.connect(handler, port_to_connect) == -1)
    {
        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection %d failed.
\n"), i));
        return -1;
    }
}

```

4) 接下来运行 `ACE_Reactor` 事件循环。

```
ACE_Reactor::instance()->run_reactor_event_loop();
```

下面是 `ReactorClient.cpp` 的完整代码

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

```

#include "ace/Connector.h"
#include "ace/SOCK_Connector.h"
#include "ClientHandler.h"

int main(int argc, char* argv[])
{
    ACE_INET_Addr port_to_connect(9000, ACE_LOCALHOST);
    ACE_Connector<ClientHandler, ACE_SOCK_Connector> client;

    for(int i = 0; i < 5; i++)
    {
        ClientHandler* handler = NULL;
        if(client.connect(handler, port_to_connect) == -1)
        {
            ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection %d failed.\n"), i));

            return -1;
        }
    }

    ACE_Reactor::instance()->run_reactor_event_loop();
    return 0;
}

```

- 5. 接下来编译。如果没有运行 ReactorServer，请首先运行 ReactorServer。然后再运行 ReactorClient。运行效果如下图：

```

192.168.0.70 Server | 192.168.0.70 Client
[mmzheng@ipas ReactorServer]$ ./ReactorServer
[ 17:09:18.892995](32132|3072435168) Connection from 127.0.0.1:32937
[ 17:09:18.893626](32132|3072435168) Connection from 127.0.0.1:32938
[ 17:09:18.893895](32132|3072435168) Connection from 127.0.0.1:32939
[ 17:09:18.894051](32132|3072435168) Connection from 127.0.0.1:32940
[ 17:09:18.894206](32132|3072435168) Connection from 127.0.0.1:32941
[mmzheng@ipas ReactorClient]$ ./ReactorClient

```

根据运行结果，我们发现，我们的测试程序成功的与服务器建立了 5 个连接，但是由于我们还没有编写发送消息的代码，所以还不能每 2 秒发送消息。

3.4.3 使用超时机制发送消息

从 ReactorServer.cpp 的 main 方法我们看到，当 5 个连接成功后，就进入到了 Reactor 的事件循环中了，而没有事件触发，是不会调用我们的服务处理器的。因此，我们的主动发送消息的代码也就没有机会执行。

解决这个问题，创造我们主动发送消息的机会，有 2 个办法：

1) 外部驱动办法：在 Reactor 线程外创建一个新的线程，这个线程每发送消息一次，sleep 2 秒钟，然后继续发送消息。这种办法我们以后在研究 ACE_Task 机制的时候再讨论。

2) 内部事件办法：这就要从 Reactor 事件机制想办法。我们可以利用 Reactor 的 Timeout 机制。在 Reactor 中设置每 2 秒钟超时一次，这样服务处理器的 Handle_timeout 方法就会被调用，这样我们就有机会主动发送消息了。

3.4.4 实现 II

下面我们采用**设置超时**的办法来实现我们的需求。

□ 1. 打开 ClientHandler.h 进行修改：

1) 为方便起见，我们创建了 super 类型定义。这样，在我们的类中调用基类方法时，可读性将大为提高。

```
typedef ACE_Svc_Handler<ACE_SOCKET_STREAM, ACE_NULL_SYNCH> super;
```

2) 在 open() 方法里面设置超时，所以我们继承父类的 open 方法：

```
int open(void * = 0);
```

3) 继承处理超时事件的 handle_timeout：

```
virtual int handle_timeout (const ACE_Time_Value &current_time, const void *act = 0);
```

4) 记录当前发送的次数和最大的发送次数：

```
private:
    enum { ITERATIONS = 5 };
    int iterations_;
```

ClientHandler.h 修改完成，下面是改进后的 ClientHandler.h 的完整代码

```
#ifndef CLIENTHANDLER_H
#define CLIENTHANDLER_H

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

```

#include "ace/SOCK_Stream.h"

class ClientHandler : public ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
{
    typedef ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH> super;

public:
    virtual int open (void * = 0);

    // Called when input is available from the client.
    virtual int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE);

    // Called when a timer expires.
    virtual int handle_timeout (const ACE_Time_Value &current_time, const void
*act = 0);

private:
    enum { ITERATIONS = 5 };
    int iterations_;
};
#endif

```

❑ 2. 修改类文件 ClientHandler.cpp。

1) 增加 open 方法，我们并不打算改变父类 open 的行为，所以首先调用父类的 open 方法。

```

int ServerHandler::open(void *p)
{
    if(super::open(p) == -1)
    {
        return -1;
    }
    return 0;
}

```

2) 接下来在 open 方法里初始化 iterations_，然后用 reactor 的 schedule_timer 设置超时。每 2 秒钟超时事件会触发一次。

```

ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection start\n")));
this->iterations_ = 0;
ACE_Time_Value iter_delay (2);    // Two seconds
return this->reactor ()->schedule_timer(this, 0, ACE_Time_Value::zero,
iter_delay);

```


3) 修改 `handle_input` 方法，当断开同服务器连接时，结束 Reactor 事件循环。

```
if (recv_cnt <= 0) {
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P| %t) Connection close\n")));
    this->reactor ()->end_reactor_event_loop ();
    return -1;
}
```

4) 实现 `handle_timeout` 方法。

当发送次数大于最大发送次数时，使用底层 `peer` 的 `close_writer` 方法关闭本端的 socket。继而服务器关闭它那一端，最后我们将在 `handle_input` 方法做一次 0 字节接收，结束 Reactor 事件循环。

否则的话，发送一条消息给服务器。

```
int ClientHandler::handle_timeout(const ACE_Time_Value &, const void *)
{
    if (++this->iterations_ > ITERATIONS)
    {
        this->peer ().close_writer ();
        return 0;
    }

    const int INPUT_SIZE = 4096;
    char buffer[INPUT_SIZE];
    memset(buffer, 0, INPUT_SIZE);
    int nbytes = ACE_OS::sprintf(buffer, "Iteration %d\n", this->iterations_);
    this->peer().send (buffer, nbytes);
    return 0;
}
```

下面是改进后的 `ClientHandler.cpp` 的完整代码

```
#include "ClientHandler.h"

int ClientHandler::open (void *p)
{
    if(super::open(p) == -1)
    {
        return -1;
    }
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P| %t) Connection start\n")));
}
```

```

        this->iterations_ = 0;
        ACE_Time_Value iter_delay (2);    // Two seconds
        return this->reactor ()->schedule_timer(this, 0, ACE_Time_Value::zero,
iter_delay);
    }

int ClientHandler::handle_input (ACE_HANDLE){
    const int INPUT_SIZE = 4096;
    char buffer[INPUT_SIZE];
    memset(buffer, 0, INPUT_SIZE);
    int recv_cnt = this->peer().recv( buffer, sizeof(buffer));

    if (recv_cnt <= 0) {
        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection close\n")));
        this->reactor ()->end_reactor_event_loop ();
        return -1;
    }

    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Receive msg from server: %s"),
buffer));
    return 0;
}

int ClientHandler::handle_timeout(const ACE_Time_Value &, const void *)
{
    if (++this->iterations_ > ITERATIONS)
    {
        this->peer ().close_writer ();
        return 0;
    }

    const int INPUT_SIZE = 4096;
    char buffer[INPUT_SIZE];
    memset(buffer, 0, INPUT_SIZE);
    int nbytes = ACE_OS::sprintf(buffer, "Iteration %d\n", this->iterations_);
    this->peer().send (buffer, nbytes);
    return 0;
}

```

- 3. 接下来编译。如果没有运行 ReactorServer，请首先运行 ReactorServer。然后再运行 ReactorClient。

运行效果如下图：

```
[mzzheng@ipas ReactorClient]$ ./ReactorClient
[ 18:52:04.565014](76713072435360) Connection start
[ 18:52:04.565400](76713072435360) Connection start
[ 18:52:04.565497](76713072435360) Connection start
[ 18:52:04.565582](76713072435360) Connection start
[ 18:52:04.565668](76713072435360) Connection start
[ 18:52:04.570446](76713072435360) Receive msg from server: Iteration 1
[ 18:52:04.570956](76713072435360) Receive msg from server: Iteration 1
[ 18:52:04.571341](76713072435360) Receive msg from server: Iteration 1
[ 18:52:04.571755](76713072435360) Receive msg from server: Iteration 1
[ 18:52:04.572160](76713072435360) Receive msg from server: Iteration 1
[ 18:52:06.575267](76713072435360) Receive msg from server: Iteration 2
[ 18:52:06.575528](76713072435360) Receive msg from server: Iteration 2
[ 18:52:06.575653](76713072435360) Receive msg from server: Iteration 2
[ 18:52:06.575770](76713072435360) Receive msg from server: Iteration 2
[ 18:52:06.575905](76713072435360) Receive msg from server: Iteration 2
[ 18:52:08.575235](76713072435360) Receive msg from server: Iteration 3
[ 18:52:08.575499](76713072435360) Receive msg from server: Iteration 3
[ 18:52:08.575633](76713072435360) Receive msg from server: Iteration 3
[ 18:52:08.575755](76713072435360) Receive msg from server: Iteration 3
[ 18:52:08.575869](76713072435360) Receive msg from server: Iteration 3
[ 18:52:10.574646](76713072435360) Receive msg from server: Iteration 4
[ 18:52:10.574708](76713072435360) Receive msg from server: Iteration 4
[ 18:52:10.574741](76713072435360) Receive msg from server: Iteration 4
[ 18:52:10.574774](76713072435360) Receive msg from server: Iteration 4
[ 18:52:10.574807](76713072435360) Receive msg from server: Iteration 4
[ 18:52:12.574648](76713072435360) Receive msg from server: Iteration 5
[ 18:52:12.574709](76713072435360) Receive msg from server: Iteration 5
[ 18:52:12.574742](76713072435360) Receive msg from server: Iteration 5
[ 18:52:12.574774](76713072435360) Receive msg from server: Iteration 5
[ 18:52:12.574807](76713072435360) Receive msg from server: Iteration 5
[ 18:52:14.574981](76713072435360) Connection close
[mzzheng@ipas ReactorClient]$
```

这样我们就完成了我们的测试程序。

3.5 前行的路标

前面我们使用了 Reactor 框架完成了一个 C/S 架构的程序，但是，从商用的角度来说，还不是完善的，还有很多工作要做。接下来请阅读相关资料，按照下面的要求改进代码。

1. 使用 ACE 消息队列来缓存待发送的消息，然后向 Reactor 登记 `WRITE_MASK`，在 `handle_output` 里面发送消息。
2. 实现你自己的 `ACE_Acceptor` 类，限制最大连接数，当超过最大连接数时，拒绝客户端的连接。另外，使用连接缓存，当客户端连接时，从缓存里获得一个空闲的服务处理器，而不是去 new 一个新的。（提示：可以覆盖 `make_svc_handler()` 方法来实现）

4. ACE Proactor 框架

4.1 Proactor（前摄器）框架

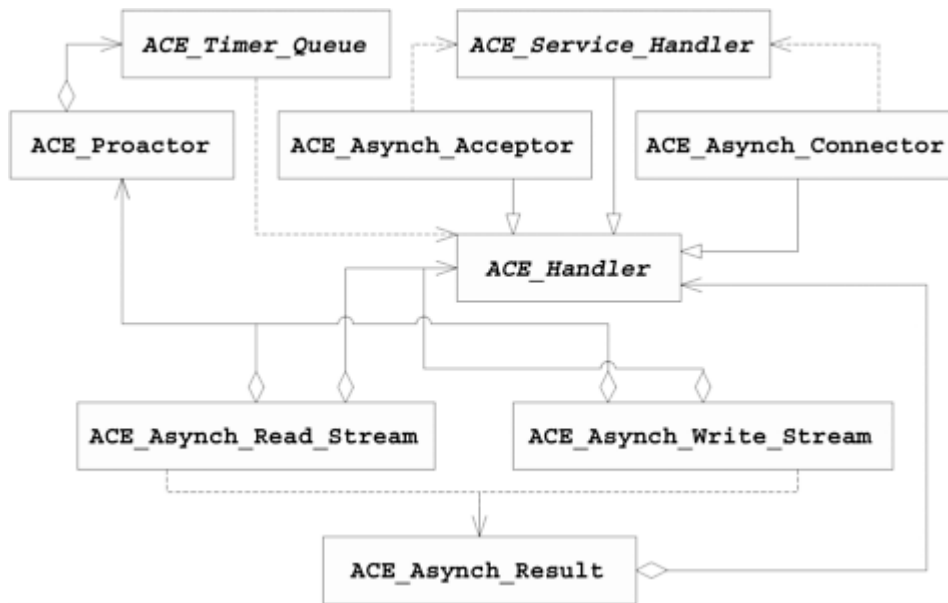
通过前面的学习，我们已经了解了 Reactor 框架的工作原理：Reactor 采用了事件处理机制，当有消息可以收或者有消息可以发送时调用相应的 `handle_input()` 和 `handle_output()` 方法。Reactor 的这种机制将我们从多线程中解放出来，只需一个线程即可处理多个客户端的连接。

但是我们注意到，在 `handle_input()` 和 `handle_output()` 方法内部，Reactor 是采用了同步的 `read` 和 `write` 的方法去收发数据的，也就是说，`read/write` 方法一直占用了 CPU 资源，直到收发完成。如果网络状况不好或者异常情况，`read/write` 方法花费了很多时间，必然会导致 Reactor 来不及处理其它事件，这在某些应用中是无法令人接受的。

于是 Proactor 框架应运而生：它同 Reactor 框架一样是事件驱动的框架，但是与 Reactor 不同的是，Proactor 的 I/O 方法都是异步的。Proactor 使用了异步的 I/O 工厂类来收发消息，`read/write` 方法都是异步的，调用完成后立即返回给 Proactor 主框架进行其它事件的处理。后台的程序去收发消息，真正收到消息或者发送消息完成时才通知 Proactor 的事件处理器去做后续处理。异步 I/O 机制使得 Proactor 的主线程更专注于我们业务逻辑的处理，而不会将时间浪费在对 I/O 的处理上。

Proactor 的中文翻译——前摄器——让人一头雾水。它的形容词 **proactive** 的英文释义为：
`Controlling a situation by causing something to happen rather than waiting to respond to it after it happens.` 从这个英文释义我们可以更好的理解 Proactor 的含义。

下面我们来看一下 Proactor 框架的核心类。



ACE 类	描述
ACE_Handler	定义用于接收异步 I/O 操作的结果和处理定时器到期的接口
ACE_Asynch_Read_Stream ACE_Asynch_Write_Stream ACE_Asynch_Result	在 I/O 流上发起异步读和异步写操作，并使每个操作与一个用于接收操作结果的 ACE_Handler 对象联系起来。
ACE_Asynch_Acceptor ACE_Asynch_Connector	Acceptor-Connector 模式的异步实现，它异步地建立新的 TCP/IP 连接。
ACE_Service_Handler	定义 ACE_Asynch_Acceptor 和 ACE_Asynch_Connector 连接工厂的目标，并提供挂钩方法来初始化通过 TCP/IP 连接的服务。
ACE_Proactor	管理定时器和异步 I/O 完成事件多路分离。这个类是 ACE Reactor 框架中的 ACE_Reactor 类的类似物。

还记得 Reactor 框架的那些核心类吗？在 Proactor 中都可以找到他们的对应的堂兄弟类（如果将 Reactor 和 Proactor 比作兄弟的话）。

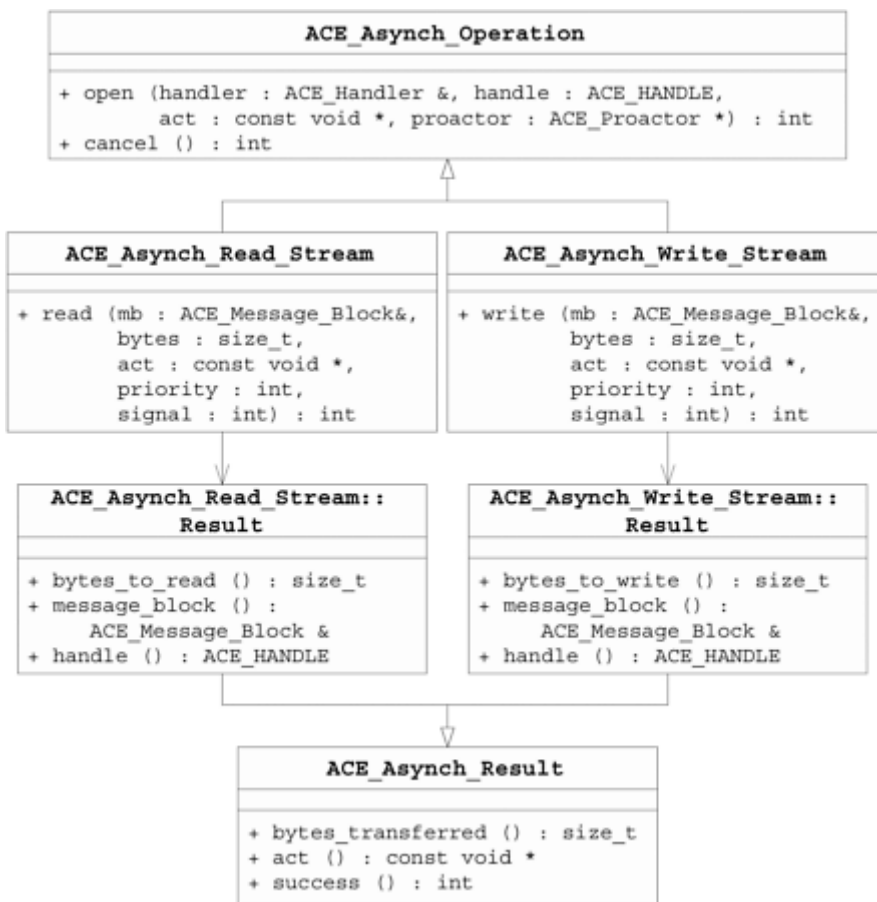
Reactor 类	Proactor 类
ACE_Event_Handler	ACE_Handler
ACE_Reactor	ACE_Proactor

ACE_Svc_Handler	ACE_Service_Handler
ACE_Acceptor	ACE_Asynch_Acceptor
ACE_Connector	ACE_Asynch_Connector
没有等价类	ACE_Asynch_Read_Stream ACE_Asynch_Write_Stream ACE_Asynch_Result

在 Proactor 中有 Reactor 所没有的对应事物--异步 I/O 工厂类，下面我们首先介绍一下它们。

4.1.1 异步 I/O 工厂类

ACE Proactor 提供了下列的异步 I/O 类：



对于我们的应用来说，主要关心 `ACE_Asynch_Read_Stream` 和 `ACE_Asynch_Write_Stream` 这两个工厂类（它们使得应用能够发起异步的 read/write 操作），以及它们分别具有的内部类 `Result`（下一节有 `Result` 主要方法的介绍）。

`ACE_Asynch_Read_Stream` 为异步读数据流，提供了下面的关键方法：

方法	描述
----	----

<code>open()</code>	初始化对象，为发起异步的 <code>read()</code> 操作做准备。
<code>cancel()</code>	尝试取消通过这个对象发起的未完成的 <code>read()</code> 操作。
<code>read()</code>	发起异步操作，从相关联的 IPC 流中读取数据。

`ACE_Asynch_Write_Stream` 为异步写数据流，提供了下面的关键方法：

方法	描述
<code>open()</code>	初始化对象，为发起异步的 <code>write()</code> 操作做准备。
<code>cancel()</code>	尝试取消通过这个对象发起的未完成的 <code>write()</code> 操作。
<code>write()</code>	发起异步操作，将数据写到相关联的 IPC 流中。

我们如何使用这 2 个数据流呢？

- 1) 在 `ACE_Handle`（下面介绍）的 `open` 方法里面调用这 2 者的 `open` 方法，初始化它们
- 2) 使用它们的 `read/write` 方法来发起异步的读写操作

4.1.2 ACE_Handler(完成处理器)

`ACE_Handler` 是 `ACE Proactor` 框架中所有异步完成处理器的基类。它在 `Proactor` 框架中的作用类似 `Reactor` 中 `ACE_Event_Handler` 的作用。

它提供了下列挂钩方法：

方法	描述
<code>handle()</code>	获取这个对象所用的句柄
<code>handle_read_stream()</code>	挂钩方法，在 <code>ACE_Asynch_Read_Stream</code> 发起的 <code>read()</code> 操作完成时调用。
<code>handle_write_stream()</code>	挂钩方法，在 <code>ACE_Asynch_Write_Stream</code> 发起的 <code>write()</code> 操作完成时调用。
<code>handle_time_out()</code>	挂钩方法，在通过 <code>ACE_Proactor</code> 调度的定时器到期时调用。

调用 `handle_read_stream()` 和 `handle_write_stream()` 挂钩方法时传入的是指向“与已经完成的异步操作相关联的 `Result` 对象”的引用。下面列出了 `Result` 的主要方法：

方法	描述
<code>success()</code>	指示异步操作是否成功
<code>handle()</code>	获取异步 I/O 操作所使用的 I/O 句柄。
<code>message_block()</code>	获取指向“在操作中使用的 <code>ACE_Message_Block</code> ”的引用
<code>bytes_transferred()</code>	指示在异步操作中实际传输了多少字节
<code>bytes_to_read()</code>	指示请求 <code>read()</code> 操作读取的字节是多少(仅 <code>ACE_Asynch_Read_Stream::Result</code> 类具有的方法)
<code>bytes_to_write()</code>	指示请求 <code>write()</code> 操作写出的字节是多少(仅 <code>ACE_Asynch_Write_Stream::Result</code> 类具有的方法)

4.1.3 ACE_Message_Block

上一章的 Forward Points 里请大家使用消息队列来缓存消息，那么在消息队列里存放的消息就是 `ACE_Message_Block`。

异步 I/O 读写的单位都是 `ACE_Message_Block`，这带来的好处是：

1. 简化了缓冲区管理：各个类都使用 `ACE_Message_Block`，高效可移植。
2. 使传输计数更新自动化：可以根据 `ACE_Message_Block` 的读写指针读写数据
3. 易于与其它 ACE 框架集成：比如同样使用了 `ACE_Message_Block` 的 ACE Task 框架等。

`ACE_Message_Block` 提供的主要方法有：

方法	描述
<code>ACE_Message_Block()</code> <code>init()</code>	对消息进行初始化
<code>msg_type()</code>	设置和获取消息的类型
<code>msg_priority()</code>	设置和获取消息的优先级
<code>clone()</code>	返回整条消息的完整 深复制 (deep copy)

<code>duplicate()</code>	返回整条消息的 浅复制 (shallow copy)，将其引用计数加 1
<code>release()</code>	将引用计数减 1，如果计数值降为 0，则释放消息的资源。
<code>set_flags()</code>	将制定的数据位(bit)同一组已有的标志(flag)执行“按位与”操作，用以指定消息的语义（如，消息释放时是否删除缓冲区等等）
<code>clr_flags()</code>	清除指定的标志位
<code>copy()</code>	从缓冲区复制 n 个字节到消息
<code>rd_ptr()</code>	设置和获取 读 指针
<code>wr_ptr()</code>	设置和获取 写 指针
<code>cont()</code>	设置和获取消息中的“连接(continuation)”字段，这个字段用于将复合消息连接在一起
<code>next()</code> <code>prev()</code>	设置和获取 “指向 ACE_Message_Queue 中的双向消息链表”的指针
<code>length()</code>	设置和获取消息的当前长度，即 <code>wr_ptr() - rd_ptr()</code>
<code>total_length()</code>	获取消息的长度，包括所有被连接的消息块
<code>size()</code>	设置和获取消息的总容量，包括分配在 <code>[rd_ptr(),wr_ptr())</code> 范围之前和之后的存储容量
<code>base()</code>	设置和获取 消息的数据

4.1.4 ACE_Proactor

`ACE_Proactor` 是 Proactor 模式的核心类，Proactor 模式的运行事件循环，定时器到期等功能都要通过这个类来进行。但是与 Reactor 框架不同的是，Proactor 并不需要各完成处理器(ACE_Handler)向其登记。异步 I/O 的读写操作时将 Proactor 和完成处理器联系在一起，读写完毕时，Proactor 通知相应的等待读写完毕的完成处理器。

`ACE_Proactor` 实现了 Façade 模式，为应用程序提供了各种访问 Proactor 框架特性的接口。

1. 生命周期管理方法

方法	描述
----	----

<code>ACE_Proactor()</code> <code>open()</code>	初始化前摄器实例
<code>~ACE_Proactor()</code> <code>close()</code>	清理前摄器被初始化时分配的资源
<code>instance()</code>	静态方法，返回指向单体 <code>ACE_Proactor</code> 的指针。这个单体前摄器是通过结合了 <code>Double-Checked Locking Optimization</code> 的 <code>Singleton</code> 模式来创建和管理的。

2. 事件循环管理方法

方法	描述
<code>handle_events()</code>	等待完成事件发生，并随即分派和其相关联的完成处理器。可以使用超时参数限制花在等待事件上的时间
<code>proactor_run_event_loop()</code>	反复调用 <code>handle_events()</code> 方法直到发生下列情况之一：该方法失败； <code>proactor_event_loop_done()</code> 返回真；发生超时（可选）。
<code>proactor_end_event_loop()</code>	指示前摄器关闭其事件循环。
<code>proactor_event_loop_done()</code>	如果前摄器的事件循环已被结束（例如，通过调用 <code>proactor_end_event_loop()</code> ），就返回 1

3. 定时器管理方法

方法	描述
<code>schedule_timer()</code>	登记一个将在用户指定的时间量后被分配的 <code>ACE_Handler</code>
<code>cancel_timer()</code>	取消一个或多个先前登记的定时器

4. I/O 操作的便利方法

方法	描述
<code>create_asynch_read_stream()</code>	创建 <code>ACE_Asynch_Read_Stream_Impl</code> 的针对特定平台的子类的实例，适用于发起异步 <code>read()</code> 操作。
<code>create_asynch_write_stream()</code>	创建 <code>ACE_Asynch_Write_Stream_Impl</code> 的针对特定平台的子类的实例，适用于发起异步 <code>write()</code> 操作。

4.2 前摄式 Acceptor-Connector 框架

下面是前摄式 Acceptor-Connector 框架的类图：



这些类所起的作用和 Reactor 中的 Acceptor-Connector 非常相似。这里不再赘述。

4.2.1 ACE_Service_Handler

ACE_Service_Handler 继承自 **ACE_Handler**，除具有其父类的方法外，还有下面的关键方法：

方法	描述
<code>open()</code>	挂钩方法，用于在新连接建立之后初始化服务
<code>addresses()</code>	挂钩方法，用于捕捉服务连接的本地和远程地址

注意：在建立连接时，如果传给异步连接工厂的 `pass_address` 参数为 1，`Proactor` 框架就会调用 `addresses` 方法。`addresses` 方法在 Windows 上是异步连接时获取远程地址的唯一方法。

4.2.2 ACE_Asynch_Acceptor

`ACE_Asynch_Acceptor` 是 `Acceptor` 角色的异步实现。提供了下面的主要方法：

方法	描述
<code>open()</code>	初始化和发出一个或多个异步 <code>accept()</code> 操作
<code>cancel()</code>	取消所有由该接受器发起的异步 <code>accept()</code> 操作。
<code>validate_connection()</code>	挂钩方法，用于在新连接打开服务之前确认对端地址
<code>make_handler()</code>	挂钩方法，用于为新连接获取服务处理器对象

4.2.3 ACE_Asynch_Connector

`ACE_Asynch_Connector` 是 `Connector` 角色的异步实现。提供了下面的主要方法：

方法	描述
<code>open()</code>	初始化用于主动连接工厂的信息
<code>connect()</code>	发起一个异步 <code>connect()</code> 操作
<code>cancel()</code>	取消所有异步 <code>connect()</code> 操作
<code>validate_connection()</code>	挂钩方法，用于了解连接部署(Disposition)，在新连接打开服务之前确认对端地址
<code>make_handler()</code>	挂钩方法，用于为新连接获取服务处理器对象

4.3 既生 Proactor，何生 Reactor (二者的应用范围)

有人可能会问，既然有了 `Proactor` 框架，还要 `Reactor` 框架干吗？直接都用 `Proactor` 框架来完成不就好了？

我们知道，ACE 和 Java 一样，都是依赖于各个操作系统的具体实现的。Proactor 虽然在理论上要比 Reactor 效率高，但是限于操作系统的支持，Proactor 的实际效率并不一定比 Reactor 好，有些操作系统不支持异步 I/O，所以那些平台上根本不能使用 Proactor。

而同步 I/O 和反应式 I/O 可在大多数现代操作系统上使用，所以 Reactor 对 Proactor 的一个优点就是兼容性好，移植性好，在编写对性能要求不高的客户端来说，使用 Reactor 是很合适的。而在某些情况，用 Reactor 开发与交换机交互消息的模块也是可以的，比如我们曾经开发过与 Excel 交换机通信的程序，由于交换机上的嵌入式 CPU 的处理能力远逊于 PC 机上的 CPU 处理速度，所以这种情况下使用 Reactor 也没什么问题。

Windows 平台通过 Overlapped I/O 和 I/O 完成端口 来支持 异步 I/O，所以在 Windows 上使用 Proactor 开发服务器端可以取得很好的性能。

在 Linux 平台上，虽然 Linux 2.6 内核将对 aio 的支持集成到内核中，但 Linux 2.6 内核 aio 对 socket 的支持却不是真正的异步的。对 socket 的异步 I/O 请求在内核中自动被转换成同步的调用。如果先对一个 socket 提交一个读操作，然后提交一个写操作，那么写操作只有在读操作完整之后才能执行。这对 Proactor 在 Linux 下的实际应用有严重的影响。而且 Proactor 在 Linux 下还有内存泄漏问题。

所以在 Linux 上，开发服务器端，可以考虑下面 2 种方法：

1. Reactor 框架，但是要替换掉默认的 ACE_Select_Reactor 实现，改用基于 Linux Epoll 技术实现的 ACE_Dev_Poll_Reactor，来提高性能。（Epoll 是 Linux 从 2.6 开始支持的异步事件 I/O 技术，更详细内容请自行阅读相关资料。）

2. 使用非官方的 Proactor 实现类，比如 **TProactor**

（<http://www.terabit.com.au/solutions.php>），据说在 Linux 上使用性能很好。

4.4 ACE Proactor Server (Demo)

4.4.1 需求

将用 Reactor 实现的 Server 改用 Proactor 实现，要求仍然是：能够处理多个 Client 的连接，在收到 Client 的消息后，原封不动的将消息再发还给 Client。

4.4.2 实现

接下来我们一步一步实现它，虽然 Linux 上的 Proactor 并不能达到很好的性能，但是作为我们的演示如何使用 Proactor 的 demo 是没有问题的，因此还是在 Linux 上进行开发（在 RedHat 9，使用 Magic C++ 开发，最终的代码可以在前面指定位置获得）：

- 1. 使用 Magic C++ 在 Linux 服务器上建立一个新工程 ProactorServer。
项目的设置等跟 ReactorServer 的设置相同
- 2. 准备工作做好之后，下面创建 服务处理器。 创建新类 ServerHandler，继承自 `ACE_Service_Handler`。修改头文件 ServerHandler.h。`ACE_Service_Handler` 没有模板。
1) 实现析构方法，在析构方法里断开与客户端的 socket 连接。

```
~ServerHandler() {
    if(this->handle() != ACE_INVALID_HANDLE) {
        ACE_OS::closesocket(this->handle());
    }
}
```

- 2) 继承父类的 `open`, `handle_read_stream`, `handle_write_stream`:

```
// Called by <ACE_Asynch_Acceptor> when a client connects.
virtual void open (ACE_HANDLE new_handle, ACE_Message_Block
&message_block);
protected:
    virtual void handle_read_stream(const ACE_Asynch_Read_Stream::Result
&result);
    virtual void handle_write_stream(const ACE_Asynch_Write_Stream::Result
&result);
```

- 3) 定义变量：输出流，输入流，对端 IP 地址

```
private:
    ACE_Asynch_Write_Stream write_;
    ACE_Asynch_Read_Stream reader_;
    ACE_TCHAR peer_name[MAXHOSTNAMELEN];
```

下面是 ServerHandler.h 的完整代码

```
#ifndef SERVERHANDLER_H
#define SERVERHANDLER_H

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

```

#include "ace/Log_Msg.h"
#include "ace/OS.h"

class ServerHandler : public ACE_Service_Handler
{
public:
    ~ServerHandler() {
        if(this->handle() != ACE_INVALID_HANDLE) {
            ACE_OS::closesocket(this->handle());
        }
    }
    // Called by <ACE_Asynch_Acceptor> when a client connects.
    virtual void open (ACE_HANDLE new_handle, ACE_Message_Block
&message_block);
protected:
    virtual void handle_read_stream(const ACE_Asynch_Read_Stream::Result
&result);
    virtual void handle_write_stream(const ACE_Asynch_Write_Stream::Result
&result);
private:
    ACE_Asynch_Write_Stream write_;
    ACE_Asynch_Read_Stream reader_;
    ACE_TCHAR peer_name[MAXHOSTNAMELEN];
};
#endif

```

- 3. 下面我们来实现 ServerHandler 类。首先修改 `open` 方法。我们在 `open` 方法里初始化输入输入流，获取客户端的 IP 地址，并开始发起异步的读操作，等待客户端的消息。

1) 定义我们每次接收的最大的消息的长度

```
#define MAX_MSG_SIZE 1024
```

2) 编写 `open` 方法，先保存本次连接的 socket 句柄：

```
void ServerHandler::open(ACE_HANDLE new_handle, ACE_Message_Block &) {
    this->handle(new_handle);
}
```

3) 初始化异步 I/O 流

```

if (this->write_.open(*this)!=0 || this->reader_.open(*this) != 0 )
{
    delete this;
    return;
}

```

4) 获取客户端的 IP 地址:

```
ACE SOCK_STREAM stream(this->handle());
ACE_INET_Addr remote_address;
if(stream.get_remote_addr(remote_address)== 0 &&
remote_address.addr_to_string(peer_name, MAXHOSTNAMELEN) == 0) {
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection from %s\n"),
peer_name));
}
```

5) 发起异步的读操作, 创建一个 `ACE_Message_Block *mb`, 将接收到的消息保存到 `mb`。

```
ACE_Message_Block *mb;
ACE_NEW_NORETURN(mb, ACE_Message_Block(MAX_MSG_SIZE));
if ( this->reader_.read(*mb, mb->space()) != 0)
{
    ACE_ERROR((LM_ERROR, ACE_TEXT("[%T] (%P|%t) Asynch Read Failed.")));
    mb->release();
    delete this;
    return;
}
```

5) `open` 方法结束。

□ 4. 实现 `ServerHandler` 的 `handle_read_stream` 方法。`handle_read_stream` 方法负责接收消息, 并返回消息。

1) 从 `result` 里拿到 `ACE_Message_Block`, 并判断接收消息是否成功, 如果失败或者收到的消息为 0, 说明连接已断开, 就释放 `ACE_Message_Block`, 并删除自身。

```
void ServerHandler::handle_read_stream(const ACE_Asynch_Read_Stream::Result &result)
{
    ACE_Message_Block &mb = result.message_block();

    if ( !result.success() || result.bytes_transferred() == 0)
    {
        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Disconnect from %s\n"),
peer_name));
        mb.release();
        delete this;
    }
}
```

2) 如果接收到的消息成功, 就再把这个 `ACE_Message_Block` 用异步输出发送出去, 然后继续发起异步读操作, 等待消息。

```
else
{
```



```

        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%)t) Receive msg from
client: %s\n"), mb.rd_ptr()));

        if (this->write_.write(mb, mb.length()) != 0)
        {
            ACE_ERROR((LM_ERROR, ACE_TEXT("[%T] (%P|%)t) Asynch Write
Failed.))));
            mb.release();
        }

        ACE_Message_Block* new_mb;
        ACE_NEW_NORETURN(new_mb, ACE_Message_Block(MAX_MSG_SIZE));
        this->reader_.read(*new_mb, new_mb->space());
    }

```

3) `handle_read_stream` 方法结束。

- 5. 实现 `ServerHandler` 的 `handle_write_stream` 方法。`handle_write_stream` 方法比较简单，释放掉 `ACE_Message_Block` 即可。

```

void ServerHandler::handle_write_stream (const ACE_Asynch_Write_Stream::Result &result)
{
    result.message_block().release();
}

```

下面是 `ServerHandler.cpp` 的完整代码

```

#include "ServerHandler.h"

#define MAX_MSG_SIZE 1024

void ServerHandler::open(ACE_HANDLE new_handle, ACE_Message_Block &) {
    this->handle(new_handle);
    if (this->write_.open(*this) != 0 || this->reader_.open(*this) != 0 )
    {
        delete this;
        return;
    }

    ACE_SOCKET_STREAM stream(this->handle());
    ACE_INET_Addr remote_address;
    if (stream.get_remote_addr(remote_address) == 0 &&
remote_address.addr_to_string(peer_name, MAXHOSTNAMELEN) == 0) {
        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%)t) Connection from %s\n"),

```

```

    peer_name));
    }

    ACE_Message_Block *mb;
    ACE_NEW_NORETURN(mb, ACE_Message_Block(MAX_MSG_SIZE));
    if ( this->reader_.read(*mb, mb->space()) != 0)
    {
        ACE_ERROR((LM_ERROR, ACE_TEXT("[%T] (%P|%t) Asynch Read Failed.")));
        mb->release();
        delete this;
        return;
    }
}

void ServerHandler::handle_read_stream(const ACE_Asynch_Read_Stream::Result &result)
{
    ACE_Message_Block &mb = result.message_block();

    if ( !result.success() || result.bytes_transferred() == 0)
    {
        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Disconnect from %s\n"),
peer_name));
        mb.release();
        delete this;
    }
    else
    {
        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Receive msg from
client: %s\n"), mb.rd_ptr()));

        if (this->write_.write(mb, mb.length()) != 0)
        {
            ACE_ERROR((LM_ERROR, ACE_TEXT("[%T] (%P|%t) Asynch Write
Failed.")));
            mb.release();
        }

        ACE_Message_Block* new_mb;
        ACE_NEW_NORETURN(new_mb, ACE_Message_Block(MAX_MSG_SIZE));
        this->reader_.read(*new_mb, new_mb->space());
    }
}

```

```

    }

    void ServerHandler::handle_write_stream (const ACE_Asynch_Write_Stream::Result &result)
    {
        result.message_block().release();
    }

```

□ 6. 最后我们实现 main 方法，增加 ProactorServer.cpp 文件

1) 在 main 方法，首先定义监听的 IP 地址，对于我们的 Server 应用来说，只须指定监听的端口即可。我们在 9000 端口监听。

```
ACE_INET_Addr port_to_accept(9000);
```

2) 定义我们的 Server，`ACE_Asynch_Acceptor` 模板类的 Handler 指定为我们刚刚实现的 `ServerHandler`。然后使用 `open` 方法开始监听。

```

ACE_Acceptor<ServerHandler, ACE_SOCKET_ACCEPTOR> server;
if(server.open(port_to_accept) == -1)
{
    return -1;
}

```

3) 接下来运行 ACE_Proactor 事件循环。

```
ACE_Proactor::instance()->proactor_run_event_loop();
```

下面是 ProactorServer.cpp 的完整代码

```

#include "ace/Proactor.h"
#include "ace/Asynch_Acceptor.h"
#include "ServerHandler.h"

int main(int argc, char* argv[])
{
    ACE_INET_Addr port_to_accept(9000);
    ACE_Asynch_Acceptor<ServerHandler> server;
    if(server.open(port_to_accept) == -1)
    {
        return -1;
    }

    ACE_Proactor::instance()->proactor_run_event_loop();
    return 0;
}

```

```

}

```

- 7. 接下来编译运行。首先运行 ProactorServer，然后运行前面实现的 ReactorClient 来测试。
运行效果如下图：

```

[mzzheng@ipas ProactorServer]$ ls
ProactorServer      ProactorServer_DEBUG.mk  ProactorServer.prj  ServerHand
ProactorServer.cpp  ProactorServer.o         ProactorServer.wsp  ServerHand
[mzzheng@ipas ProactorServer]$ ./ProactorServer
(32540 | 3072435488) ACE_POSIX_AIOCB_Proactor::Max Number of AIOs=1024
[ 10:48:30.616786](32540|3072435488) Connection from 127.0.0.1:34237
[ 10:48:30.617588](32540|3072435488) Connection from 127.0.0.1:34238
[ 10:48:30.617694](32540|3072435488) Connection from 127.0.0.1:34239
[ 10:48:30.617754](32540|3072435488) Connection from 127.0.0.1:34240
[ 10:48:30.617822](32540|3072435488) Connection from 127.0.0.1:34241
[ 10:48:30.617891](32540|3072435488) Receive msg from client: Iteration1
[ 10:48:30.618726](32540|3072435488) Receive msg from client: Iteration1
[ 10:48:30.618923](32540|3072435488) Receive msg from client: Iteration1
[ 10:48:38.624258](32540|3072435488) Receive msg from client: Iteration5
[ 10:48:38.624303](32540|3072435488) Receive msg from client: Iteration5
[ 10:48:40.624552](32540|3072435488) Disconnect from 127.0.0.1:34238
[ 10:48:40.624744](32540|3072435488) Disconnect from 127.0.0.1:34237
[ 10:48:40.624807](32540|3072435488) Disconnect from 127.0.0.1:34239
[ 10:48:40.624867](32540|3072435488) Disconnect from 127.0.0.1:34240
[ 10:48:40.624926](32540|3072435488) Disconnect from 127.0.0.1:34241
Ready

```

4.5 前行的路标

前面我们使用了 Proactor 框架完成了一个 C/S 架构的 Server，如果你感兴趣的话，接下来请：

1. 请将这个 Proactor 移植到 Windows 的 VC 上运行，并达到同样的效果。（提示：请注意 Windows 平台如何获取客户端的 IP 地址）
2. 修改 ReactorClient 工程，改用 Proactor 实现
3. 在 Windows 上，ACE 提供了集成 Reactor 和 Proactor 事件循环的机制。请在 Windows 上实现如何在一个工程里同时使用 Reactor 和 Proactor 并共用一个事件循环。

4. 实现你自己的 `ACE_Asynch_Acceptor` 类，限制最大连接数，当超过最大连接数时，拒绝客户端的连接。另外，使用连接缓存，当客户端连接时，从缓存里获得一个空闲的服务处理器，而不是去 `new` 一个新的。（提示：可以覆盖 `make_handler()` 方法来实现）

5. ACE Task 框架

5.1 我们的新需求

我们已经了解了 `Reactor/Proactor` 框架，通过这 2 个优秀的框架，我们不需要创建新线程，就可以完成对多个客户端的处理，而且我们的 `demo` 也运行的很好。但是这个世界没有万能的东西，有时候光靠 `Reactor/Proactor` 也不能完成任务，比如说我们的新需求：

从交换机收到消息，然后查询一次数据库，根据数据库的数据发送相应的消息给交换机。交换机有自己的一套 API 与之连接以及收发消息，虽然这套 API 也是 TCP/IP 协议实现的，但是没有 API 的源代码，我们不知道 API 连接之后会不会发什么认证消息或者内部通信消息什么的，所以不能用 `Acceptor-Connector` 框架来连接交换机。

出于性能考虑，项目被设计成了两个模块，数据库模块和交换机模块。

数据库模块比较好解决，直接使用 `Reactor/Proactor` 框架即可，收到消息后查询数据库，再返回数据库数据即可，只需一个线程。

那么交换机模块呢？由于交换机有自己的一套 API，因此交换机就必须有 2 个线程：一个线程使用了 `Reactor/Proactor` 框架与数据库模块打交道，另一个线程与交换机打交道。

但是两个线程又带来了新问题：它们之间要交换数据的，怎么交换？消息队列是一个好办法，收到的数据库数据可以放在消息队列里，交换机线程需要的时候可以到消息队列里面去取。

现在我们有 2 个问题要解决了：1) 创建新线程；2) 使用消息队列。怎么用 ACE 完成任务？

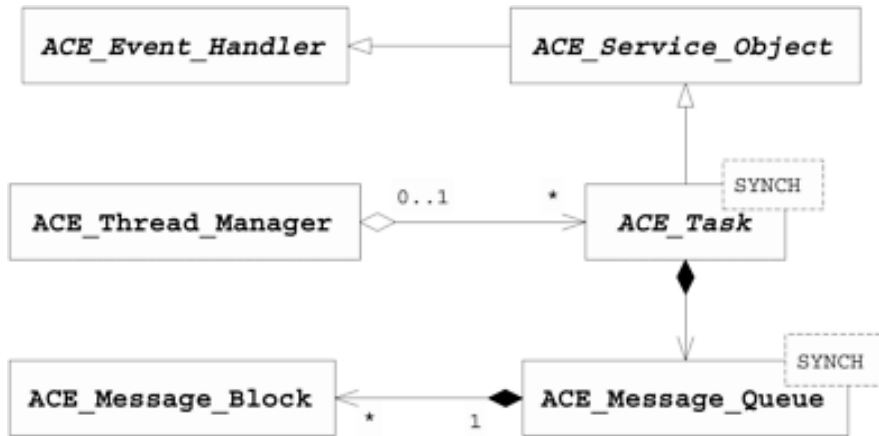
ACE 里面提供了 Task 框架，只要继承 `ACE_Task` 类即可解决这 2 个问题。（我们又不用“重新发明轮子”了。☺）

注：这个需求我们不在本指南里提供实现它的代码。因为这个代码跟交换机紧密相关，不方便学习和测试。

5.2 Task(任务)框架

ACE Task 框架提供了强大而可扩展的面向对象并发能力，提供了派生线程，在不同线程的对象间传递消息和对消息进行排队等功能。

ACE Task 框架的主要类如下图所示：



ACE 类	描述
ACE_Message_Block	实现了 Composite 模式[GoF] 以使开发者能够高效地操作定长和变长的消息。
ACE_Message_Queue	提供一种进程间的消息队列，使应用能够在进程中的线程间传递和缓冲消息
ACE_Thread_Manager	允许应用可移植地创建和管理一个或多个线程的生命期，同步以及各种属性
ACE_Task	允许应用创建主动或被动的对象，解除不同处理单元的耦合；使用消息来交流请求，响应，数据以及控制信息；并且还可以顺序地或并发地排队和处理消息。

5.3 ACE_Message_Queue

ACE 的消息队列是仿照 Unix 的消息队列设计的，所以，如果你熟悉 Unix 的消息队列的话，就很容易掌握 ACE 的消息队列。有关消息队列，生产者/消费者等概念，这里不再赘述。

ACE_Message_Queue 是一种可移植的轻量级进程内消息排队机制。它提供了下面的能力：

- ✓ 允许消息(ACE_Message_Block) 在队列的前部，后部或者基于优先级 出队/入队。
- ✓ 使用 ACE_Message_Block 提供高效的消息缓冲机制，使动态内存分配和数据复制最小化。
- ✓ 出队/入队可设置超时
- ✓ 可与 Reactor 等框架集成使用。
- ✓ 其它特性...

下面描述一下 ACE_Message_Queue 的主要方法。

1. 参数化的同步策略

ACE_Message_Queue 类的模板里可以指定同步策略：

✓ ACE_NULL_SYNCH：

可以用于队列被单线程使用的情况。当队列到达高水位写入或者为空时读出时，不会阻塞调用线程，会返回-1，并将 errno 设置为 EWOULDBLOCK。

✓ ACE_MT_SYNCCH:

当多线程访问队列时，应使用 ACE_MT_SYNCCH 类，以使多线程同步访问队列。其入队，出队方法支持阻塞，非阻塞和定时操作。

值	行为
空(NULL)的 ACE_Time_Value 指针	<p>阻塞： 入队时队列字节大于高水位（满队列）或出队时队列为空（空队列）都会导致他们阻塞，直至方法完成，队列被关闭或有信号中断调用。</p> <p>方法完成的标准是：队中字节数到达低水位时，此时认为不再认为自己是满的，可使入队方法完成；当有消息入队时，可使阻塞的出队方法完成。</p>
非 NULL ACE_Time_Value 指针，其 sec() 和 usec() 方法返回 0	<p>非阻塞： 如果入队/出队方法没有立即成功，应返回-1，并将 errno 设置为 EWOULDBLOCK。</p>
非 NULL ACE_Time_Value 指针，其 sec() 或 usec() 方法返回值 > 0	<p>定时： 指示入队/出队方法没能完成时，应等待指定的绝对时间。如果到时无法完成，应返回-1，并将 errno 设置为 EWOULDBLOCK。如果队列被关闭，或者信号中断调用，方法也会提前返回。</p>

2. 初始化和流控制方法

方法	描述
ACE_Message_Queue() open()	初始化队列，指定水位标(Watermark) 和 通知策略(可选)
high_water_mark() low_water_mark()	设置/获取确定流控制何时开始和结束的高低水位标
notification_strategy()	设置/获取通知策略

ACE 缺省的高低水位都是 16k。请注意高低水位都只对入队方法起作用。当队列入队/出队时发生阻塞时，请修改队列的高低水位标。

3. 入队/出队方法和消息缓冲

方法	描述
<code>is_empty()</code> <code>is_full()</code>	<code>is_empty()</code> 方法在队列中没有消息块时返回真。 <code>is_full()</code> 方法在队列中的字节数大于高水位标时返回真。
<code>enqueue_tail()</code>	将消息插入队列的后部
<code>enqueue_head()</code>	将消息插入队列的前部
<code>enqueue_prio()</code>	根据消息的优先级插入消息
<code>dequeue_head()</code>	移除并返回队列前部的消息
<code>dequeue_tail()</code>	移除并返回队列后部的消息

4. 关闭和消息释放方法

方法	描述
<code>deactivate()</code>	将队列状态变为 DEACTIVATED ，并唤醒所有在入队或者出队操作上等待的线程。该方法不释放任何已入队的消息。
<code>pulse()</code>	将队列状态变为 PULSED ，并唤醒所有在入队或者出队操作上等待的线程。该方法不释放任何已入队的消息。
<code>state()</code>	返回队列的状态。
<code>activate()</code>	将队列状态变为 ACTIVATED 。
<code>~ACE_Message_Queue()</code> <code>close()</code>	使队列停止活动，并立即释放所有已入队的消息
<code>flush()</code>	释放队列中的消息，但不改变其状态。

ACE_Message_Block 总是处在下面 3 种状态之一：

- **ACTIVATED** 此状态中所有操作都会正常工作（队列启动时总是处在 **ACTIVATED** 状态）

- **DEACTIVATED** 此状态中的所有入队和出队操作都立即返回-1，并将 `error` 设为 `ESHUTDOWN`，直到队列被再次激活
- **PULSED** 迁移到此状态会使等待中的入队出队操作立即返回，就好像队列被停用(Deativated)了一样。但所有在 `PULSED` 状态中发起的操作的行为和在 `ACTIVATED` 状态中发起的一样。

5.4 ACE_Task

下面我们来研究一下 Task 框架的主角：ACE_Task 类。

还记得前面讲过的 Reactor 框架中的 ACE_Svc_Handler 类吗？ACE_Svc_Handler 类就是 ACE_Task 类的子类。

ACE_Task 是 ACE 的面向对象并发框架的基础。它具有以下能力：

- ✓ 其内部封装了 ACE_Message_Block，大部分情况我们都不需要理会 ACE_Message_Block 的存在，只需调用 ACE_Task 的 `putq`，`getq` 等方法来简化队列操作。
- ✓ 使用 ACE_Thread_Manager 来激活任务，任务就会作为主动对象运行。（使用 `activate` 方法即可，你不知道 ACE_Thread_Manager 类也没关系）。
- ✓ 是 ACE_Event_Handler 的后代，所以其实例可以充当 Reactor 的事件处理器。
- ✓ ACE_Task 类也可以使用在 ACE 的另两大框架：Service Configurator 框架和 ACE Streams 框架中。（这两个框架我们将在《ACE 开发指南（高级版）》中介绍，敬请期待！）

如果你不想使用 ACE_Task 的队列能力而只想创建线程，可使用 ACE_Task 的父类 ACE_Task_Base。

ACE_Task 类具有丰富的接口：

1. 任务初始化方法：

方法	描述
<code>ACE_Task()</code>	构造器，可以指定任务所用的 <code>ACE_Message_Queue</code> 和 <code>ACE_Thread_Manager</code> 的指针
<code>open()</code>	挂钩方法，执行应用定义的初始化活动。
<code>thr_mgr()</code>	获取/设置 任务的 <code>ACE_Thread_Manager</code> 指针
<code>msg_queue()</code>	获取/设置 任务的 <code>ACE_Message_Queue</code> 指针

<code>activate()</code>	将任务转换为运行在一个或多个线程中的主动对象
-------------------------	------------------------

使用 `activate` 方法将 `ACE_Task` 转为主动对象（也就是创建一个新线程运行，线程入口方法会调用 `svc` 方法）。

使用 `activate` 创建线程时，需要使用线程创建标志：（这些标志可以用‘|’组合使用）。默认的创建标志为：`THR_NEW_LWP|THR_JOINABLE`

标志	描述
<code>THR_CANCEL_DISABLE</code>	不允许这个线程被取消
<code>THR_CANCEL_ENABLE</code>	允许这个线程被取消
<code>THR_CANCEL_DEFERRED</code>	只允许延迟的取消
<code>THR_BOUND</code>	创建一个线程，绑定到一个可由内核调度的实体
<code>THR_NEW_LWP</code>	创建一个内核级线程
<code>THR_DETACHED</code>	创建一个分离的线程
<code>THR_SUSPENDED</code>	创建一个线程，但让其处在挂起状态
<code>THR_DAEMON</code>	创建一个守护(Daemon)线程
<code>THR_JOINABLE</code>	允许新创建的线程被“会合(join)”
<code>THR_SCHED_FIFO</code>	如果可用，使用 FIFO 政策调度新创建的线程
<code>THR_SCHED_RR</code>	如果可用，使用 round-robin 方案调度新创建的线程
<code>THR_SCHED_DEFAULT</code>	使用操作系统上可用的无论哪种默认调度方案

`activate` 方法也支持一次启动多个线程，在其第二个参数指定一次要创建的线程的数目。

你可以使用 `msg_queue()` 访问 `ACE_Message_Queue` 指针来设置高低水位标。

2. 任务通信，处理和同步方法

方法	描述
<code>svc()</code>	可以实现任务的服务处理的挂钩方法。所有通过 <code>activate()</code> 方法派生的线程都会执行它。

<code>put()</code>	挂钩方法。可用于传递消息给任务。消息可被立即处理，或排队等待 <code>svc()</code> 挂钩方法随后进行处理。
<code>putq()</code> <code>getq()</code> <code>ungetq()</code>	在任务的消息队列中插入，移除和替换消息。 <code>putq()</code> 、 <code>getq()</code> 和 <code>ungetq()</code> 方法分别简化了对任务的消息队列的 <code>enqueue_tail()</code> 、 <code>dequeue_head()</code> 、 <code>enqueue_head()</code> 方法的访问。

我们需要覆盖 `svc()` 方法，实现新线程要做的事。

3. 任务析构

方法	描述
<code>~ACE_Task()</code>	删除在 <code>ACE_Task</code> 构造器中分配的资源，其中包括消息队列，如果它不是作为参数传给构造器的。
<code>close()</code>	挂钩方法，执行应用定义的关闭活动。该方法通常不应由应用直接调用，特别是如果任务是主动对象的话。
<code>flush()</code>	关闭与任务相关联的消息队列，从而释放所有在队列中的消息块，并释放所有阻塞在队列上的线程。
<code>thr_count()</code>	返回目前活动在 <code>ACE_Task</code> 中的线程的数目。
<code>wait()</code>	一个栅栏（ <code>barrier</code> ）同步体，它在返回之前等待所有运行在此任务中的 <code>joinable</code> 线程退出。

5.5 Demo(Reactor Client 的改写)

5.5.1 需求

我们在实现 `Reactor Client` 的时候，曾经提到过，除了用超时机制之外，还可以用线程的方法来解决。现在我们准备用线程而不是超时机制来解决每 2 秒发送一条消息的问题。

由于 `ACE_Svc_Handler` 是 `ACE_Task` 的子类，其线程能力和队列均继承自 `ACE_Task`，所以这里我们不再创建新的线程类，而是通过使用原 `ClientHandler` 的线程和队列功能。（这里我们虽然没有直接使用 `ACE_Task` 类，但是使用的是 `ACE_Task` 的能力）

我们准备把它改成下面的样子：

1. 每个 `ClientHandler` 实例在启动的时候，都激活一个线程成为主动对象。利用该线程每 2 秒发送一次消息。

2. 原来我们是在 `handle_timeout` 里面发的，现在我们把待发的消息缓存到队列里。当输出流可用时，从队列里取一条消息发出去。

5.5.2 实现

下面是实现过程（在 RedHat 9，使用 Magic C++开发，最终的代码可以在前面指定位置获得）：

- 1. 使用 Magic C++打开原来的 `ReactorClient`。笔者这里为了与原来的 `ReactorClient` 工程区分开来，使用了 `ReactorClient` 的代码创建了新工程 `ReactorThreadClient`，项目的设置等跟 `ReactorClient` 的设置相同。

- 2. 我们只修改 `ClientHandler` 这个类的代码即可。首先修改它的头文件：
在头文件里增加了一些方法：

1) 继承了 `handle_output` 方法，在这个方法里读取队列，发送队列里的消息。

```
virtual int handle_output (ACE_HANDLE fd = ACE_INVALID_HANDLE);
```

2) 继承 `svc` 方法，这个方法是被线程入口函数调用的入口方法。

```
virtual int svc (void);
```

3) 继承了 `close` 方法。`close` 方法是在 `svc` 结束之后被调用的。由于 `ACE_Svc_Handler` 在其类中使用 `close` 删除了自己，由于我们的 `ACE_Svc_Handler` 被注册到了 `Reactor` 中，所以不能被其本身删除。我们要修改父类的行为。

```
virtual int close (u_long) ;
```

增加了一个属性：`ACE_Reactor_Notification_Strategy` 是一个策略(Strategy)类，实现了 Strategy 模式。它允许你定制另一个类的行为，而无需改变受影响的类。

```
ACE_Reactor_Notification_Strategy notifier_;
```

在构造方法里初始化它，初始化用的参数依次是：

1) `Reactor` 指针：我们现在还不知道，赋为 0

2) `ACE_Event_Handler` 指针：设为本身， `this`

3) 注册事件类型。这里是读事件：`ACE_Event_Handler::WRITE_MASK`

```
ClientHandler():notifier_(0, this, ACE_Event_Handler::WRITE_MASK) {}
```

然后，删除了原来的 `handle_timeout` 方法。

```
virtual int handle_timeout (const ACE_Time_Value &current_time, const void *act = 0);
```

下面是 `ClientHandler.h` 修改后的完整代码

```

#ifndef CLIENTHANDLER_H
#define CLIENTHANDLER_H

#include "ace/Svc_Handler.h"
#include "ace/SOCK_Stream.h"
#include "ace/Reactor_Notification_Strategy.h"

class ClientHandler : public ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
{
    typedef ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH> super;

public:
    ClientHandler():notifier_(0, this, ACE_Event_Handler::WRITE_MASK) {}
    virtual int open (void * = 0);

    virtual int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE);
    virtual int handle_output (ACE_HANDLE fd = ACE_INVALID_HANDLE);
    virtual int svc (void);
    virtual int close (u_long) ;

private:
    enum { ITERATIONS = 5 };
    int iterations_;
    ACE_Reactor_Notification_Strategy notifier_;
};
#endif

```

□ 3. 接下来的工作是修改 `open` 方法。

1) 首先还是调用父类的行为:

```

if(super::open(p) == -1)
{
    return -1;
}

```

2) 现在我们已经知道 Reactor 的指针了, 把 Reactor 指针告诉 Strategy 类:

```

this->notifier_.reactor(this->reactor());

```

3) 为消息队列设置通知策略, 这样当有消息入队时, 会产生一个 WRITE 通知到 Reactor 消息队列中, 当输出可用时, 就会调用 `handle_output` 方法。

```

this->msg_queue()->notification_strategy(&this->notifier_);

```

4) 打印信息和初始化:

```

ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Connection start\n")));

```

```
this->iterations_ = 0;
```

5) 最后 open 方法结束时, 调用 activate() 来创建一个线程。不指定 activate 的参数, 会创建一个 THR_NEW_LWP|THR_JOINABLE 的线程。

```
return activate();
```

修改后的 open 方法如下所示:

```
int ClientHandler::open (void *p)
{
    if(super::open(p) == -1)
    {
        return -1;
    }

    this->notifier_.reactor(this->reactor());
    this->msg_queue()->notification_strategy(&this->notifier_);

    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection start\n")));
    this->iterations_ = 0;
    return activate();
}
```

- ❑ 4. 实现 close 方法。这个方法比较简单, 什么都不做, 只是 return 0。

```
int ClientHandler::close(u_long flags)
{
    return 0;
}
```

- ❑ 5. 原有的 handle_input 方法不需要做任何更改。

- ❑ 6. 接下来的工作是实现 svc 方法。这个方法里的代码是从原来 handle_timeout 里的代码修改而来。

1) 打印一条日志:

```
ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Thread starts\n")));
```

2) 起一个 while 循环, 其结束的条件是循环了超过 5 次, 超过 5 次后关闭输出流。

```
while(1) {
    if (++this->iterations_ > ITERATIONS)
    {
        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Disconnect from
server\n")));
        this->peer().close_writer();
    }
}
```

```
        break;
    }
}
```

3) while 循环的 if 语句结束后，将以前直接调用 peer 发送的代码改成将待发送的消息保存到 ACE_Message_Block，然后放在队列中。

```
    ACE_Message_Block *mb;
    ACE_NEW_RETURN (mb, ACE_Message_Block (128), -1);
    int nbytes = ACE_OS::sprintf(mb->wr_ptr (), "Iteration %d\n", this->iterations_);
    ACE_ASSERT (nbytes > 0);
    mb->wr_ptr (static_cast<size_t> (nbytes));
    this->putq (mb);
```

4) 然后线程 sleep 2 秒钟。此时 while 的代码结束。

```
    ACE_OS::sleep(2);
}
```

5) while 循环结束后，标志线程也就执行完了，打印信息后 return。

```
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Thread ends\n")));
    return 0;
}
```

svc 方法完整代码如下所示：

```
int ClientHandler::svc (void){
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Thread starts\n")));
    while(1) {
        if (++this->iterations_ > ITERATIONS)
        {
            ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Disconnect from
server\n")));
            this->peer().close_writer();
            break;
        }

        ACE_Message_Block *mb;
        ACE_NEW_RETURN (mb, ACE_Message_Block (128), -1);
        int nbytes = ACE_OS::sprintf(mb->wr_ptr (), "Iteration %d\n", this->iterations_);
        ACE_ASSERT (nbytes > 0);
        mb->wr_ptr (static_cast<size_t> (nbytes));
        this->putq (mb);
    }
}
```

```

        ACE_OS::sleep(2);
    }

    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Thread ends\n")));
    return 0;
}

```

❑ 7. 删除原有的 `handle_timeout` 方法。

❑ 8. 最后我们实现 `handle_output` 方法。

1) 从队列里取出所有的消息发送出去:

```

ACE_Message_Block *mb;
while (-1 != this->getq (mb))
{
    ssize_t send_cnt = this->peer().send (mb->rd_ptr (), mb->length ());
    if (send_cnt == -1)
    {
        ACE_ERROR ((LM_ERROR, ACE_TEXT ("%T] (%P|%t) Send msg failed\n")));
    }
    else
    {
        mb->rd_ptr (static_cast<size_t> (send_cnt));
    }
    if (mb->length () > 0)
    {
        this->ungetq (mb);
        break;
    }
    mb->release ();
}

```

2) 当队列为空时, 从反应器登记信息中移除 `WRITE_MASK`, 否则增加 `WRITE_MASK`。

```

if (this->msg_queue ()->is_empty ())
{
    this->reactor ()->cancel_wakeup(this, ACE_Event_Handler::WRITE_MASK);
}
else
{
    this->reactor ()->schedule_wakeup(this, ACE_Event_Handler::WRITE_MASK);
}

```

3) `handle_output` 方法在 `return` 语句完成后结束。


```
return 0;
```

handle_output 方法完整代码如下所示:

```
int ClientHandler::handle_output (ACE_HANDLE)
{
    ACE_Message_Block *mb;
    while (-1 != this->getq (mb))
    {
        ssize_t send_cnt = this->peer().send (mb->rd_ptr (), mb->length ());
        if (send_cnt == -1)
        {
            ACE_ERROR ((LM_ERROR, ACE_TEXT ("%T] (%P|%t) Send msg failed\n")));
        }
        else
        {
            mb->rd_ptr (static_cast<size_t> (send_cnt));
        }
        if (mb->length () > 0)
        {
            this->ungetq (mb);
            break;
        }
        mb->release ();
    }
    if (this->msg_queue ()->is_empty ())
    {
        this->reactor ()->cancel_wakeup(this, ACE_Event_Handler::WRITE_MASK);
    }
    else
    {
        this->reactor ()->schedule_wakeup(this, ACE_Event_Handler::WRITE_MASK);
    }
    return 0;
}
```

□ 9. 这样我们的代码就修改完成了。

下面是 ClientHandler.cpp 修改后的完整代码

```
#include "ClientHandler.h"
```

```
int ClientHandler::open (void *p)
```

```

{
    if(super::open(p) == -1)
    {
        return -1;
    }

    this->notifier_.reactor(this->reactor());
    this->msg_queue()->notification_strategy(&this->notifier_);

    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection start\n")));
    this->iterations_ = 0;
    return activate();
}

int ClientHandler::close(u_long flags)
{
    return 0;
}

int ClientHandler::handle_input (ACE_HANDLE)
{
    const int INPUT_SIZE = 4096;
    char buffer[INPUT_SIZE];
    memset(buffer, 0, INPUT_SIZE);
    int recv_cnt = this->peer().recv( buffer, sizeof(buffer));

    if (recv_cnt <= 0 )
    {
        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Connection close\n")));
        this->reactor ()->end_reactor_event_loop ();
        return -1;
    }

    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Receive msg from server: %s\n"),
buffer));
    return 0;
}

int ClientHandler::svc (void){
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Thread starts\n")));
    while(1) {

```

```

        if (++this->iterations_ > ITERATIONS)
        {
            ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Disconnect from
server\n")));

            this->peer().close_writer();
            break;
        }

        ACE_Message_Block *mb;
        ACE_NEW_RETURN (mb, ACE_Message_Block (128), -1);
        int nbytes = ACE_OS::sprintf(mb->wr_ptr (), "Iteration %d\n", this-
>iterations_);
        ACE_ASSERT (nbytes > 0);
        mb->wr_ptr (static_cast<size_t> (nbytes));
        this->putq (mb);

        ACE_OS::sleep(2);
    }

    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Thread ends\n")));
    return 0;
}

int ClientHandler::handle_output (ACE_HANDLE)
{
    ACE_Message_Block *mb;
    while (-1 != this->getq (mb))
    {
        ssize_t send_cnt = this->peer().send (mb->rd_ptr (), mb->length ());
        if (send_cnt == -1)
        {
            ACE_ERROR ((LM_ERROR, ACE_TEXT ("%T] (%P|%t) Send msg failed\n")));
        }
        else
        {
            mb->rd_ptr (static_cast<size_t> (send_cnt));
        }
        if (mb->length () > 0)
        {
            this->ungetq (mb);
            break;
        }
    }
}

```

```
    }  
    mb->release ();  
}  
if (this->msg_queue ()->is_empty ())  
{  
    this->reactor ()->cancel_wakeup(this, ACE_Event_Handler::WRITE_MASK);  
}  
else  
{  
    this->reactor ()->schedule_wakeup(this, ACE_Event_Handler::WRITE_MASK);  
}  
return 0;  
}
```

- ❑ 10. 接下来编译。如果没有运行 ReactorServer，请首先运行 ReactorServer。然后再运行修改后的 ReactorClient（或者新工程 ReactorThreadClient）。
运行效果如下图：

```

192.168.0.70 | 192.168.0.70 (1) |
[mzzheng@ipas ReactorThreadClient]$ ls
ClientHandler.cpp  ClientHandler.o      ReactorThreadClient.cpp  React
ClientHandler.h    ReactorThreadClient  ReactorThreadClient_DEBUG.mk  React
[mzzheng@ipas ReactorThreadClient]$ ./ReactorThreadClient
[ 17:21:56.269956](32455|3072435360) Connection start
[ 17:21:56.271667](32455|3072435360) Connection start
[ 17:21:56.272492](32455|3072435360) Connection start
[ 17:21:56.273314](32455|3072435360) Connection start
[ 17:21:56.274269](32455|3072383920) Thread starts
[ 17:21:56.274634](32455|3061894064) Thread starts
[ 17:21:56.274889](32455|3051404208) Thread starts
[ 17:21:56.275118](32455|3040914352) Thread starts
[ 17:21:56.275371](32455|3072435360) Connection start
[ 17:21:56.277074](32455|3072435360) Receive msg from server: Iteration 1
[ 17:21:56.277375](32455|3072435360) Receive msg from server: Iteration 1
[ 17:21:56.277510](32455|3072435360) Receive msg from server: Iteration 1
[ 17:21:56.277636](32455|3072435360) Receive msg from server: Iteration 1
[ 17:21:56.277823](32455|3030424496) Thread starts
[ 17:21:56.278450](32455|3072435360) Receive msg from server: Iteration 1
[ 17:22:04.314954](32455|3072435360) Receive msg from server: Iteration 5
[ 17:22:04.314989](32455|3072435360) Receive msg from server: Iteration 5
[ 17:22:06.323748](32455|3072383920) Disconnect from server
[ 17:22:06.323908](32455|3072383920) Thread ends
[ 17:22:06.324058](32455|3061894064) Disconnect from server
[ 17:22:06.324110](32455|3061894064) Thread ends
[ 17:22:06.324161](32455|3051404208) Disconnect from server
[ 17:22:06.324208](32455|3051404208) Thread ends
[ 17:22:06.324254](32455|3040914352) Disconnect from server
[ 17:22:06.324301](32455|3040914352) Thread ends
[ 17:22:06.324348](32455|3030424496) Disconnect from server
[ 17:22:06.324397](32455|3030424496) Thread ends
[ 17:22:06.325593](32455|3072435360) Connection close
[mzzheng@ipas ReactorThreadClient]$
Ready

```

与原来使用超时机制的结果基本一致。

5.6 基本的线程安全性

前面我们使用多线程来改写了客户端程序。多线程的好处是显而易见的。

你一定经历过这种情况：在 Windows 的某个 GUI 程序执行非常耗时的操作时，你转到其它窗口，再转回到那个 GUI 窗口时，发现程序主窗体一片空白，那就是因为程序的主线程被阻塞了，不能去调用它

的 Draw 方法来绘制自己的主界面了。这对用户来说是很不友好的。但是如果你使用了线程，就可以派生一个线程去执行耗时操作，这样就不会出现前面的尴尬情况。

但是多线程带来了便利的同时，也带来了很多问题。其中最困难的问题之一就是线程的安全和同步问题：多线程有可能同时去写一个变量的值，多线程先后执行的顺序是不可预知的。尤其是 CPU 多核技术的到来，更要重视线程的安全和同步问题：单核时代多线程本质上还是串行的，因为同时只可能有一个线程获得 CPU，但是多核技术使得多线程有可能实现真正的并行。

所以怎么保证数据的安全性和一致性是多线程程序一定要考虑和克服的问题。

ACE 提供了一组丰富的线程安全和同步的原语。这里介绍最简单的两个：Mutex 和 Guard。

（Mutex，令牌等操作系统基础知识这里不介绍）

5.6.1 互斥体(Mutex)

互斥体是 ACE 中最简单的保护原语，你只需要在需要保护的代码之前调用 acquire()接口，保护代码执行完毕后执行 release()接口即可。如果线程成功获取(acquire)了互斥体，它将继续执行，否则就阻塞，直到该互斥体的持有者释放(release)了互斥体为止。

ACE 提供了若干互斥体类，包括轻量级的线程同步原语，也包括重量级的跨进程同步原语。

对于线程的同步来说，使用 ACE_Thread_Mutex 即可。

下面演示 ACE_Thread_Mutex 如何使用。

需求：有个设备仓库，仓库里的设备只能同时有一个线程访问。

- 1. 使用 Magic C++创建了新工程 ThreadMutex，项目的设置等跟 ReactorClient 的设置相同。
- 2. 创建仓库类 DeviceRepository。这个类拥有 updateDevice 方法和一个互斥体。保证同时只能有一个线程访问。

下面是 DeviceRepository.h 的完整代码

```
#ifndef DEVICEREPOSITORY_H
#define DEVICEREPOSITORY_H

#include "ace/Synch.h"

class DeviceRepository
{
public:
    DeviceRepository() {}

    void updateDevice(int deviceId);
};
```

```
private:
    ACE_Thread_Mutex mutex_;
};
#endif
```

□ 3. 接下来的工作是实现 `updateDevice` 方法。

1) 首先是获取互斥体:

```
ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Try to acquire the mutex...\n")));
mutex_.acquire();
ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Got the mutex.\n")));
```

2) 然后访问设备之后 sleep 1 秒钟, 这里简单起见, 只是打印访问语句。

```
ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Updating device: %d\n"), deviceId));
ACE_OS::sleep(1);
```

3) 释放互斥体, 然后再 sleep 1 秒钟。(这里还要 sleep 1 秒钟的原因是: 我在调试的时候一直都是一个线程执行到底, 没有形成竞争的局面。为了演示 2 个线程的竞争, 这里再 sleep 一次, 让一直没有机会抢到 CPU 的线程也有机会。)

```
mutex_.release();
ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Release the mutex.\n")));
ACE_OS::sleep(1);
```

下面是 DeviceRepository.cpp 的完整代码

```
#include "DeviceRepository.h"

void DeviceRepository::updateDevice(int deviceId)
{
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Try to acquire the mutex...\n")));
    mutex_.acquire();
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Got the mutex.\n")));

    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Updating device: %d\n"), deviceId));
    ACE_OS::sleep(1);

    mutex_.release();
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|t) Release the mutex.\n")));
    ACE_OS::sleep(1);
}
```

- 4. 下面创建仓库的访问类：RepositoryVisitor。这个类是线程，我们不需要使用 ACE_Task 的队列能力，所以让 RepositoryVisitor 继承自 ACE_Task 的父类 ACE_Task_Base。

下面是 RepositoryVisitor.h 的完整代码

```
#ifndef REPOSITORYVISITOR_H
#define REPOSITORYVISITOR_H

#include "ace/Task.h"
#include "DeviceRepository.h"

class RepositoryVisitor: public ACE_Task_Base
{
public:
    RepositoryVisitor(DeviceRepository& rep):rep_(rep){}
    virtual int svc(void);

private:
    enum {NUM_USERS = 10};
    DeviceRepository& rep_;
};
#endif
```

- 5. 接下来的工作是实现 svc 方法。svc 的主要任务就是执行 10 次仓库的访问。

下面是 RepositoryVisitor.cpp 的完整代码

```
#include "RepositoryVisitor.h"

int RepositoryVisitor::svc()
{
    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Visitor Thread Running...\n")));
    for (int i = 0; i < NUM_USERS; i++)
    {
        this->rep_.updateDevice(i);
    }

    return 0;
}
```

- 6. 最后是 main 方法的代码。创建 ThreadMutex.cpp 文件。

main 方法里创建了一个仓库，以及两个线程。然后激活两个线程，等待它们执行完毕。

下面是 ThreadMutex.cpp 修改后的完整代码

```
#include "DeviceRepository.h"
#include "RepositoryVisitor.h"

int main(int argc, char* argv[])
{
    DeviceRepository rep;
    RepositoryVisitor visitor1(rep);
    RepositoryVisitor visitor2(rep);

    visitor1.activate();
    visitor2.activate();

    visitor1.wait();
    visitor2.wait();
}
```

- 7. 接下来编译运行。运行效果如下图：

```
[mzzheng@ipas ThreadMutex]$ ls
DeviceRepository.cpp  DeviceRepository.o  RepositoryVisitor.h  ThreadMutex
DeviceRepository.h  RepositoryVisitor.cpp  RepositoryVisitor.o  ThreadMutex.cpp
[mzzheng@ipas ThreadMutex]$ ./ThreadMutex
[ 10:56:28.616269](21770|3072433072) Visitor Thread Running...
[ 10:56:28.616800](21770|3072433072) Try to acquire the mutex...
[ 10:56:28.616948](21770|3072433072) Got the mutex.
[ 10:56:28.617075](21770|3072433072) Updating device: 0
[ 10:56:28.617264](21770|3061943216) Visitor Thread Running...
[ 10:56:28.617439](21770|3061943216) Try to acquire the mutex...
[ 10:56:29.623774](21770|3072433072) Release the mutex.
[ 10:56:29.624157](21770|3061943216) Got the mutex.
[ 10:56:29.624301](21770|3061943216) Updating device: 0
[ 10:56:30.633749](21770|3072433072) Try to acquire the mutex...
[ 10:56:30.634124](21770|3061943216) Release the mutex.
[ 10:56:30.634279](21770|3072433072) Got the mutex.
[ 10:56:30.634393](21770|3072433072) Updating device: 1
[ 10:56:31.643750](21770|3061943216) Try to acquire the mutex...
[ 10:56:31.644152](21770|3072433072) Release the mutex.
[ 10:56:31.644305](21770|3061943216) Got the mutex.
[ 10:56:31.644432](21770|3061943216) Updating device: 1
[ 10:56:32.653746](21770|3072433072) Try to acquire the mutex...
[ 10:56:32.654119](21770|3061943216) Release the mutex.
[ 10:56:32.654259](21770|3072433072) Got the mutex.
[ 10:56:32.654372](21770|3072433072) Updating device: 2
[ 10:56:33.663750](21770|3061943216) Try to acquire the mutex...
[ 10:56:33.664125](21770|3072433072) Release the mutex.
[ 10:56:33.664279](21770|3061943216) Got the mutex.
[ 10:56:33.664393](21770|3061943216) Updating device: 2
[ 10:56:34.673750](21770|3072433072) Try to acquire the mutex...
[ 10:56:34.674127](21770|3061943216) Release the mutex.
[ 10:56:34.674279](21770|3072433072) Got the mutex.
```

我们从运行结果可以看到：mutex 有效的完成了让线程串行访问资源的使命。

5.6.2 守卫(Guard)

前面的 mutex 工作的很美好，但是 mutex 的 acquire 和 release 是两条语句。这就有可能导致问题。如果忘了写 release 语句了怎么办？或者没有忘记 release，但是 mutex 保护的代码段之中 return 了，比如下面的代码：

```
❑ 0. void DeviceRepository::updateDevice(int deviceId)
    {
        mutex_.acquire();

        ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Updating device: %d\n"), deviceId));

        // Allocate a new object
        ACE_NEW_RETURN(object, Object, -1);
        // ...
        // Use the object

        mutex_.release();
    }
```

上面的代码看似没有问题，mutex 保护的代码段中使用 ACE_NEW_RETURN 创建一个 object。但是 ACE_NEW_RETURN 宏会在有错误发生时返回，从而造成 mutex 没有被释放，就导致了死锁。在真实的代码中，找出这样的返回语句可能更为困难，而在代码中四处加 release 也是很优雅的。

ACE 提供了一个方便的类来解决这个问题 – ACE_Guard：能帮助你简化代码，并防止死锁。

Guard 使用了一种常见的 C++ 手法：把构造器和析构器用于资源获取和释放。守卫类在构造时获取一个指定的锁，在销毁时释放这个锁。使用 Guard 总能保证锁的释放，而不管你的代码走的是什么非正常途径。

下面用 guard 改写前面的可能引起问题的代码：

```
❑ 0. void DeviceRepository::updateDevice(int deviceId)
    {
        // 创建 Guard 来守卫 mutex
        ACE_Guard<ACE_Thread_Mutex> guard(this->mutex_);
```

```

    ACE_DEBUG((LM_DEBUG, ACE_TEXT("[%T] (%P|%t) Updating device: %d\n"), deviceId));

    // 创建新对象。这里可能会出错返回。
    ACE_NEW_RETURN(object, Object, -1);
    // ...
    // Use the object

    // Guard 被释放,自动释放了锁。
}

```

通过前面的代码可以看出 Guard 的使用方法。ACE_Guard 模板类将锁的类型作为其模板参数，并且要求你传入锁对象。这样在进入和退出函数时自动获取和释放互斥体。

5.7 前行的路标

前面我们编写了一个线程版的 ReactorClient。这个程序为每个连接创建一个线程，每个 2 秒钟发送一条 “Iteration %d” 的消息过去。现在请写一个序列号产生器，每个线程要发送消息时，通过序列号产生器或者最新的序列号，使用这个序列号替换掉 消息里的 “%d”。

要求：

1. 序列号产生器产生的序列号是从 0 开始依次增大的，每个号码只使用一次。
2. 当号码达到 65535 时，再从 0 开始计数

这样虽然线程有多个，但是它们发的消息却不会重复了。它们发送的消息将是 Iteration 0, Iteration 1, , Iteration 65535.....（提示：序列号产生器里使用 mutex 及 ACE_Guard (可选)）

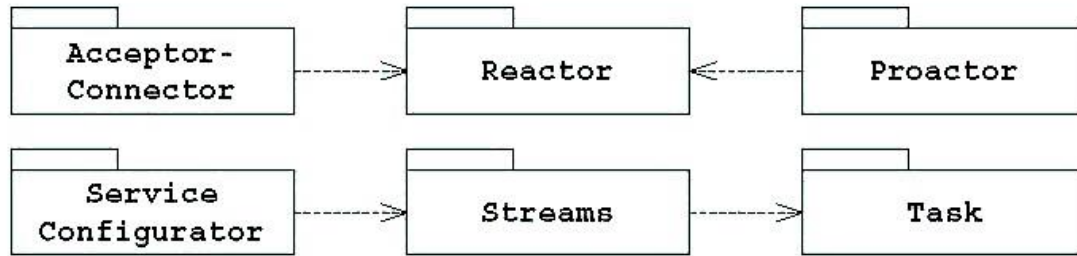
6. 总结

本指南介绍了 ACE 的 6 大框架中的 4 个，分别是 Reactor, Proactor, Acceptor-Connector, Task。

Reactor 和 Proactor 都是事件驱动的框架，不同的是一个是同步 I/O，一个是异步 I/O。两者各有利弊。

Acceptor-Connector 框架封装了连接的建立和管理，使我们可以更专注于业务逻辑。

Task 框架可以方便的创建线程，使用消息队列。



这些框架可以满足大部分的需要。ACE 还提供了 Streams 框架和 Service Configurator 等高级特性，我们将在《ACE 开发指南(高级)》中介绍它们。

7. 常见问题

1) **Q:** 使用消息队列时在 putq 时阻塞了怎么办?

A: 这是由于达到了消息队列的高水位标导致的，尝试加大消息队列的高水位标。

2) **Q:** Linux 上使用 ACE 库链接出错怎么办?

A: 请检查 ACE 库的编译环境是不是符合当前的 gcc 或者 g++ 版本。如果不对，请重新编译 ACE 库或者使用 ACE 库编译的 gcc 来编译链接你的程序。

3) **Q:** (请补充新问题)

A: