实战出精华

在具体的C++网络编程中提升你的逼格

John Torjo

Boost.Asio C++ 网络编程

关于作者

做为权威的 C++专家，除了偶尔用 C#和 Java 写程序，**John Torjo** 把他超过 15 年编程生涯中的大部分时间都贡献给了 C++。

他也很喜欢在 C++ Users Journal 和其他杂志上写一些编程相关的文章。

闲暇的时候，他喜欢玩扑克、开快车。他有很多个自由职业，其中一个就把他的两个爱好结合在一起，一个是玩扑克，另外一个是编程。如果你想联系他，可以发邮件到 john.code@torjo.com。

我要感谢我的朋友 Alexandru Chis, Aurelian Hale, Bela Tibor Bartha, Cristian Fatu, Horia Uifaleanu, Nicolae Ghimbovschi 以及 Ovidiu Deac。感谢他们对本书提出的反馈和意见。同时我也要感谢 Packt 公司各位对我频繁错过截稿日期行为的包容。然后最需要感谢的是 Chris Kohlhoff，Boost.Asio 的作者，是他写出了如此伟大的库。

把这本书献给我最好的朋友 Darius。

关于评审员

Béla Tibor Bartha

一个使用过多种技术和语言进行开发的专业软件工程师。尽管在过去的 4 年里，他做的是 iOS 和 OSX 应用开发，但是 C++陪伴他度过了早期个人游戏项目开发的激情岁月。

我要感谢 John，因为他我才能做这本书的评审

Nicolae Ghimbovschi

一个参加各类 C++项目超过 5 年的天才个人开发者。他主要参与一些企业通信工程的项目。作为一个狂热的 Linux 爱好者，他喜欢利用不同的操作系统、脚本工具和编程语言进行测试和实验。除了编程，他还喜欢骑自行车、瑜伽和冥想。

我要感谢 John 让我来评审这本书

关于译者

非主流程序猿 mmoaay，技术很烂，喜欢平面设计、鼠绘、交友、运动和翻译，但是确作为一只程序猿混迹在 IT 行业。热爱开源，技术烂就只好做做设计和翻译的工作。

微博：http://weibo.com/smmoaay

关于 avplayer

http://avplayer.org 中国第一技术社区。

目录

前言

第一章：Boost.Asio 入门

第二章：Boost.Asio 基本原理

第三章：回显服务端/客户端

TCP 回显服务端/客户端
    TCP 同步客户端
    TCP 同步服务端
    TCP 异步客户端
    TCP 同步服务端
    代码
UDP 回显服务端/客户端
    UDP 同步回显客户端
    UDP 同步回显服务端
总结

第四章：客户端和服务端

同步客户端/服务端
    同步客户端
    同步服务端
异步客户端/服务端
    异步客户端
    异步服务端
总结

第五章：同步 VS 异步

同步异步混合编程
客户端和服务端之间消息的互相传递
客户端软件中的同步 I/O
服务端软件中的同步 I/O
    同步服务端中的线程
客户端软件中的异步 I/O
服务端软件中的异步 I/O
    异步服务端中的线程
异步操作
代理实现
总结

第六章：Boost.Asio-其他特性

第七章：Boost.Asio-进阶

索引

前言

网络编程由来已久，并且是一个极富挑战性的任务。Boost.Asio 对网络编程做了一个极好的抽象，从而保证只需要少量的编程就可以实现一个优雅的客户端/服务端软件。在实现的过程中，它能让你体会到极大的乐趣。而且更为有益的是：Boost.Asio 包含了一些非网络的特性，用 Boost.Asio 写出来的代码紧凑、易读，而且如果你按照我在书中所讲的来做，你的代码会无懈可击。

这本书涵盖了什么？

*第一章：Boost.Asio 入门*将告诉你 Boost.Asio 是什么？怎么编译它？顺带着会有一些例子。你会发现 Boost.Asio 不仅仅是一个网络库。同时你也会接触到 Boost.Asio 中最核心的类 io_service。

*第二章：Boost.Asio 基本原理*包含了你必须了解的内容：什么时候用 Boost.Asio？我们将深入了解异步编程——一种比同步更需要技巧，且更有乐趣的编程方式。这一章也是在开发你自己的网络应用时可以作为参考的一章。

*第三章：回显服务端/客户端*将会告诉你如何实现一个小的客户端/服务端应用；也许这会是你写过的最简单的客户端/服务端应用。回显应用就是把客户端发过来的消息发送回去然后关闭客户端连接的服务。我们会先实现一个同步的版本，然后再实现一个异步的版本，这样就可以非常容易地看到它们之间的不同。

*第四章：客户端和服务端*会深入讨论如何用 Boost.Asio 创建一个简单的客户端/服务端应用。我们将讨论如何避免诸如内存泄漏和死锁的缺陷。所有的程序都只是实现一个简单的框架，从而使你能更方便地对它们进行扩展以满足你的需求。

*第五章：同步 VS 异步*会带你了解在同步和异步方式之间做选择时需要考虑的事情。首要的事情就是不要混淆它们。在这一章，我们会发现实现、测试和调试每个类型应用是非常容易的。

*第六章：Boost.Asio 的其他特性*将带你了解 Boost.Asio 一些不为人知的特性。你会发现，虽然 std streams 和 streambufs 有一点点难用，但是却表现出了它们得天独厚的优势。最后，是姗姗来迟的 Boost.Asio 协程，它可以让你用一种更易读的方式来写异步代码。（就好像写同步代码一样）

*第七章：Boost.Asio 进阶*包含了一些 Boost.Asio 进阶问题的处理。虽然在日常编程中不需要深入研究它们，但是了解它们对你有益无害（Boost.Asio 高级调试，SSL，Windows 特性，POSIX 特性等）。

读这本书你需要准备什么？

如果要编译 Boost.Asio 以及运行本书中的例子，你需要一个现代编译器。例如，Visual Studio 2008 及其以上版本或者 g++ 4.4 及其以上版本

这本书是为谁准备的？

这本书对于那些需要进行网络编程却不想深入研究复杂的原始网络 API 的开发者来说是一个福音。所有你需要的只是 Boost.Asio 提供的一套 API。作为著名 Boost C++库的一部分，你只需要额外添加几个#include 文件即可转换到 Boost.Asio。

在读这本书之前，你需要熟悉 Boost 核心库的一些知识，例如 Boost 智能指针、boost::noncopyable、Boost Functors、Boost Bind、shared_from_this/enabled_shared_from_this 和 Boost 线程（线程和互斥量）。同时还需要了解 Boost 的 Date/Time。读者还需要知道阻塞的概念以及“非阻塞”操作。

约定

本书使用不同样式的文字来区分不同种类的信息。这里给出这些样式的例子以及它们的解释。

文本中的代码会这样显示：“通常一个 *io_service* 的例子就足够了”。

一段代码是下面这个样子的：

```
read(stream, buffer [, extra options])

async_read(stream, buffer [, extra options], handler)

write(stream, buffer [, extra options])

async_write(stream, buffer [, extra options], handler)
```

专业词汇和重要的单词用黑体显示

[*！警告或者重要的注释在这样的一个框里面*]

[*？技巧在这样的一个框里面*]

读者反馈

我们欢迎来自读者的反馈。告诉我们你对这本书的看法——你喜欢哪部分，不喜欢哪部分。读者的反馈对我们非常重要，它能让我们写出对读者帮助更大的书。

你只需要发送一封邮件到 feedback@packtpub.com 即可进行反馈，注意在邮件的主题中注明书名。

如果你有一个擅长的专题，想撰写一本书或者为某本书做贡献。请阅读我们在 www.packtpub.com/authors 上的作者指引。

用户支持

现在你已经是 Packt 书籍的拥有者，我们将告诉你一些事项，让你购买本书得到的收益最大化。

下载示例代码

你可以在 http://www.packtpub.com 登录你的帐号，然后下载你所购买的书籍的全部示例代码。同时，你也可以通过访问 http://www.packtpub.com/support 进行注册，然后这些示例代码文件将直接发送到你的邮箱。

纠错

尽管我们已经尽最大的努力去保证书中内容的准确性，但是错误还是不可避免的。如果你在我们的书籍中发现了错误——也许是文字，也许是代码——如果你能将它们报告给我们，我们将不胜感激。这样的话，你不仅能帮助其他读者，同时也能帮助我们改进这本书的下一个版本。如果你发现任何需要纠正的地方，访问 http://www.packtpub.com/submit-errata，选择你的书籍，点击 **errata submission form** 链接，然后输入详细的纠错信息来将错误报告给我们。一经确定，你的提交就会通过，然后这个纠错就会被上传到我们的网站，或者添加到那本书的纠错信息区域的纠错列表中。所有已发现的纠错都可以访问 http://www.packtpub.com/support，然后通过选择书名的方式来查看。

答疑

如果你有关于本书任何方面的问题，你可以通过 questions@packtpub.com 联系我们。我们将尽我们最大的努力进行解答

第一章

Boost.Asio 入门

首先，让我们先来了解一下什么是 Boost.Asio？怎么编译它？了解的过程中我们会给出一些例子。然后在发现 Boost.Asio 不仅仅是一个网络库的同时你也会接触到 Boost.Asio 中最核心的类——*io_service*。

什么是 Boost.Asio

简单来说，Boost.Asio 是一个跨平台的、主要用于网络和其他一些底层输入/输出编程的 C++库。

计算机网络的设计方式有很多种，但是 Boost.Asio 的的方式远远优于其它的设计方式。它在 2005 年就被包含进 Boost，然后被大量 Boost 的用户测试并在很多项目中使用，比如 Remobo(http://www.remobo.com)，可以让你创建你自己的**即时私有网络(IPN)**的应用，libtorrent(http://www.rasterbar.com/products/libtorrent))一个实现了比特流客户端的库，PokerTH (http://www.pokerth.net)一个支持 LAN 和互联网对战的纸牌游戏。

Boost.Asio 在网络通信、COM 串行端口和文件上成功地抽象了输入输出的概念。你可以基于这些进行同步或者异步的输入输出编程。

```
read(stream, buffer [, extra options])
async_read(stream, buffer [, extra options], handler)
write(stream, buffer [, extra options])
```

```
async_write(stream, buffer [, extra options], handler)
```

从前面的代码片段可以看出，这些函数支持传入包含任意内容（不仅仅是一个 socket，我们可以对它进行读写）的流实例。

作为一个跨平台的库，Boost.Asio 可以在大多数操作系统上使用。能同时支持数千个并发的连接。其网络部分的灵感来源于**伯克利软件分发(BSD)socket**，它提供了一套可以支持**传输控制协议(TCP)**socket、**用户数据报协议(UDP)**socket 和 **Internet 控制消息协议(IMCP)**socket 的 API，而且如果有需要，你可以对其进行扩展以支持你自己的协议。

历史

Boost.Asio 在 2003 被开发出来，然后于 2005 年的 12 月引入到 Boost 1.35 版本中。原作者是 Christopher M. Kohlhoff，你可以通过 chris@kohlhoff.com 联系他。

这个库在以下的平台和编译器上测试通过：

- 32-bit 和 64-bit Windows，使用 Visual C++ 7.1 及以上
- Windows 下使用 MinGW
- Windows 下使用 Cygwin(确保已经定义 __USE_232_SOCKETS)
- 基于 2.4 和 2.6 内核的 Linux，使用 g++ 3.3 及以上
- Solaris 下使用 g++ 3.3 及以上
- MAC OS X 10.4 以上下使用 g++ 3.3 及以上

它也可能能在诸如 AIX 5.3，HP-UX 11i v3，QNX Neutrino 6.3，Solaris 下使用 Sun Studio 11 以上，True64 v5.1，Windows 下使用 Borland C++ 5.9.2 以上等平台上使用。（更多细节请咨询 www.boost.org）

依赖

Boost.Asio 依赖于如下的库：

- **Boost.System**：这个库为 Boost 库提供操作系统支持 (http://www.boost.org/doc/libs/1_51_0/doc/html/boost_system/index.html)
- **Boost.Regex**：使用这个库（可选的）以便你重载 *read_until()*或者 *async_read_until()* 时使用 *boost::regex* 参数。
- **Boost.DateTime**：使用这个库（可选的）以便你使用 Boost.Asio 中的计时器

- **OpenSSL**：使用这个库（可选的）以便你使用 Boost.Asio 提供的 SSL 支持。

编译 Boost.Asio

Boost.Asio 是一个只需要引入头文件就可以使用的库。然而，考虑到你的编译器和程序的大小，你可以选择用源文件的方式来编译 Boost.Asio。如果你想要这么做以减少编译时间，有如下几种方式：

在某个源文件中，添加*#include "boost/asio/impl/src.hpp"*（如果你在使用 SSL，添加*#include "boost/asio/ssl/impl/src.hpp"*） 在所有的源文件中，添加*#define BOOST_ASIO_SEPARATE_COMPILATION*

注意 Boost.Asio 依赖于 Boost.System，必要的时候还依赖于 Boost.Regex，所以你需要用如下的指令先编译 Boost：

bjam –with-system –with-regex stage

如果你还想同时编译 tests，你需要使用如下的指令：

bjam –with-system –with-thread –with-date_time –with-regex –with-serialization stage

这个库有大量的例子，你可以连同本书中的例子一块看看。

重要的宏

如果设置了 *BOOST_ASIO_DISABLE_THREADS*；不管你是否在编译 Boost 的过程中使用了线程支持，Boost.Asio 中的线程支持都会失效。

同步 VS 异步

首先，异步编程和同步编程是非常不同的。在同步编程中，所有的操作都是顺序执行的，比如从 socket 中读取（请求），然后写入（回应）到 socket 中。每一个操作都是阻塞的。因为操作是阻塞的，所以为了不影响主程序，当在 socket 上读写时，通常会创建一个或多个线程来处理 socket 的输入/输出。因此，同步的服务端/客户端通常是多线程的。

相反的，异步编程是事件驱动的。虽然启动了一个操作，但是你不知道它何时会结束；它只是提供一个回调给你，当操作结束时，它会调用这个 API，并返回操

作结果。对于有着丰富经验的 QT（诺基亚用来创建跨平台图形用户界面应用程序的库）程序员来说，这就是他们的第二天性。因此，在异步编程中，你只需要一个线程。

因为中途做改变会非常困难而且容易出错，所以你在项目初期（最好是一开始）就得决定用同步还是异步的方式实现网络通信。不仅 API 有极大的不同，你程序的语意也会完全改变（异步网络通信通常比同步网络通信更加难以测试和调试）。你需要考虑是采用阻塞调用和多线程的方式（同步，通常比较简单），或者是更少的线程和事件驱动（异步，通常更复杂）。

下面是一个基础的同步客户端例子：

```
using boost::asio;
io_service service;
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 2001);
ip::tcp::socket sock(service);
sock.connect(ep);
```

首先，你的程序至少需要一个 *io_service* 实例。Boost.Asio 使用 *io_service* 同操作系统的输入/输出服务进行交互。通常一个 *io_service* 的实例就足够了。然后，创建你想要连接的地址和端口，再建立 socket。把 socket 连接到你创建的地址和端口。

下面是一个简单的使用 Boost.Asio 的服务端：

```
typedef boost::shared_ptr<ip::tcp::socket> socket_ptr;
io_service service;
ip::tcp::endpoint ep( ip::tcp::v4(), 2001)); // listen on 2001
ip::tcp::acceptor acc(service, ep);
while ( true)
{
    socket_ptr sock(new ip::tcp::socket(service));
    acc.accept(*sock);
    boost::thread( boost::bind(client_session, sock));
}
void client_session(socket_ptr sock)
{
    while ( true)
    {
        char data[512];
        size_t len = sock->read_some(buffer(data));
        if ( len > 0)
            write(*sock, buffer("ok", 2));
```

```
    }
}
```
首先，同样是至少需要一个 *io_service* 实例。然后你指定你想要监听的端口，再创建一个接收器——一个用来接收客户端连接的对象。 在接下来的循环中，你创建一个虚拟的 socket 来等待客户端的连接。然后当一个连接被建立时，你创建一个线程来处理这个连接。

在 client_session 线程中来读取一个客户端的请求，进行解析，然后返回结果。

而创建一个异步的客户端，你需要做如下的事情：

```
using boost::asio;
io_service service;
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 2001);
ip::tcp::socket sock(service);
sock.async_connect(ep, connect_handler);
service.run();
void connect_handler(const boost::system::error_code & ec)
{
    // 如果 ec 返回成功我们就可以知道连接成功了
}
```
在程序中你需要创建至少一个 *io_service* 实例。你需要指定连接的地址以及创建 socket。

当连接完成时（其完成处理程序）你就异步地连接到了指定的地址和端口，也就是说，*connect_handler* 被调用了。

当 *connect_handler* 被调用时，检查错误代码（*ec*），如果成功，你就可以向服务端进行异步的写入。

注意：只要还有待处理的异步操作，*servece.run()*循环就会一直运行。在上述例子中，只执行了一个这样的操作，就是 socket 的 *async_connect*。在这之后，*service.run()*就退出了。

每一个异步操作都有一个完成处理程序——一个操作完成之后被调用的函数。下面的代码是一个基本的异步服务端

```
using boost::asio;
typedef boost::shared_ptr<ip::tcp::socket> socket_ptr;
io_service service;
ip::tcp::endpoint ep( ip::tcp::v4(), 2001)); // 监听端口 2001
ip::tcp::acceptor acc(service, ep);
```

```
socket_ptr sock(new ip::tcp::socket(service));
start_accept(sock);
service.run();
void start_accept(socket_ptr sock)
{
    acc.async_accept(*sock, boost::bind( handle_accept, sock, _1) );
}
void handle_accept(socket_ptr sock, const boost::system::error_code &
    err)
{
    if ( err) return;
    // 从这里开始，你可以从 socket 读取或者写入
    socket_ptr sock(new ip::tcp::socket(service));
    start_accept(sock);
}
```

在上述代码片段中，首先，你创建一个 *io_service* 实例，指定监听的端口。然后，你创建接收器 acc——一个接受客户端连接，创建虚拟的 socket，异步等待客户端连接的对象。

最后，运行异步 *service.run()*循环。当接收到客户端连接时，*handle_accept* 被调用（调用 *async_accept* 的完成处理程序）。如果没有错误，这个 socket 就可以用来做读写操作。

在使用这个 socket 之后，你创建了一个新的 socket，然后再次调用 *start_accept()*，用来创建另外一个“等待客户端连接”的异步操作，从而使 *service.run()*循环一直保持忙碌状态。

异常处理 VS 错误代码

Boost.Asio 允许同时使用异常处理或者错误代码，所有的异步函数都有抛出错误和返回错误码两种方式的重载。当函数抛出错误时，它通常抛出 *boost::system::system_error* 的错误。

```
using boost::asio;
ip::tcp::endpoint ep;
ip::tcp::socket sock(service);
sock.connect(ep); // 第一行
boost::system::error_code err;
sock.connect(ep, err); // 第二行
```

在前面的代码中，*sock.connect(ep)*会抛出错误，*sock.connect(ep, err)*则会返回一个错误码。

看一下下面的代码片段：

```
try
{
    sock.connect(ep);
} catch(boost::system::system_error e)
{
    std::cout << e.code() << std::endl;
}
```

下面的代码片段和前面的是一样的：

```
boost::system::error_code err;
sock.connect(ep, err);
if ( err)
    std::cout << err << std::endl;
```

当使用异步函数时，你可以在你的回调函数里面检查其返回的错误码。异步函数从来不抛出异常，因为这样做毫无意义。那谁会捕获到它呢？

在你的异步函数中，你可以使用异常处理或者错误码（随心所欲），但要保持一致性。同时使用这两种方式会导致问题，大部分时候是崩溃（当你不小心出错，忘记去处理一个抛出来的异常时）。如果你的代码很复杂（调用很多 socket 读写函数），你最好选择异常处理的方式，把你的读写包含在一个函数 *try {} catch* 块里面。

```
void client_session(socket_ptr sock)
{
    try
    {
        ...
    } catch ( boost::system::system_error e)
    {
        // 处理错误
    }
}
```

如果使用错误码，你可以使用下面的代码片段很好地检测连接是何时关闭的：

```
char data[512];
boost::system::error_code error;
size_t length = sock.read_some(buffer(data), error);
if (error == error::eof)
    return; // 连接关闭
```

Boost.Asio 的所有错误码都包含在ˆ的命名空间中（以便你创造一个大型的 switch 来检查错误的原因）。如果想要了解更多的细节，请参照 *boost/asio/error.hpp* 头文件

Boost.Asio 中的线程

当说到 Boost.Asio 的线程时，我们经常在讨论：

- *io_service*:*io_service* 是线程安全的。几个线程可以同时调用 *io_service::run()*。大多数情况下你可能在一个单线程函数中调用 *io_service::run()*，这个函数必须等待所有异步操作完成之后才能继续执行。然而，事实上你可以在多个线程中调用 *io_service::run()*。这会阻塞所有调用 *io_service::run()*的线程。只要当中任何一个线程调用了 *io_service::run()*，所有的回调都会同时被调用；这也就意味着，当你在一个线程中调用 *io_service::run()*时，所有的回调都被调用了。
- *socket*:*socket* 类不是线程安全的。所以，你要避免在某个线程里读一个 socket 时，同时在另外一个线程里面对其进行写入操作。（通常来说这种操作都是不推荐的，更别说 Boost.Asio）。
- *utility*:就 *utility* 来说，因为它不是线程安全的，所以通常也不提倡在多个线程里面同时使用。里面的方法经常只是在很短的时间里面使用一下，然后就释放了。

除了你自己创建的线程，Boost.Asio 本身也包含几个线程。但是可以保证的是那些线程不会调用你的代码。这也意味着，只有调用了 *io_service::run()*方法的线程才会调用回调函数。

不仅仅是网络通信

除了网络通信，Boost.Asio 还包含了其他的 I/O 功能。

Boost.Asio 支持信号量，比如 *SIGTERM*（软件终止）、*SIGINT*（中断信号）、*SIGSEGV*（段错误）等等。 你可以创建一个 *signal_set* 实例，指定异步等待的信号量，然后当这些信号量产生时，就会调用你的异步处理程序：

```
void signal_handler(const boost::system::error_code & err, int signal)
{
    // 纪录日志，然后退出应用
}
boost::asio::signal_set sig(service, SIGINT, SIGTERM);
sig.async_wait(signal_handler);
```

如果 *SIGINT* 产生，你就能在你的 *signal_handler* 回调中捕获到它。

你可以使用 Boost.Asio 轻松地连接到一个串行端口。在 Windows 上端口名称是 *COM7*，在 POSIX 平台上是*/dev/ttyS0*。

```
io_service service;
serial_port sp(service, "COM7");
```

打开端口后，你就可以使用下面的代码设置一些端口选项，比如端口的波特率、奇偶校验和停止位。

```
serial_port::baud_rate rate(9600);
sp.set_option(rate);
```

打开端口后，你可以把这个串行端口看做一个流，然后基于它使用自由函数对串行端口进行读/写操作。比如 *async_read(), write, async_write(),* 就像下面的代码片段：

```
char data[512];
read(sp, buffer(data, 512));
```

Boost.Asio 也可以连接到 Windows 的文件，然后同样使用自由函数，比如 *read(), asyn_read()*等等，就像下面的代码片段：

```
HANDLE h = ::OpenFile(...);
windows::stream_handle sh(service, h);
char data[512];
read(h, buffer(data, 512));
```

对于 POXIS 文件描述符，比如管道，标准 I/O 和各种设备（但不包括普通文件）你也可以这样做，就像下面的代码所做的一样：

```
posix::stream_descriptor sd_in(service, ::dup(STDIN_FILENO));
char data[512];
read(sd_in, buffer(data, 512));
```

计时器

一些 I/O 操作需要一个超时时间。这只能应用在异步操作上（同步意味着阻塞，因此没有超时时间）。例如，下一条信息必须在 100 毫秒内从你的同伴那传递给你。

```
bool read = false;
void deadline_handler(const boost::system::error_code &)
{
    std::cout << (read ? "read successfully" : "read failed") << std::endl;
}
void read_handler(const boost::system::error_code &)
```

```
{
    read = true;
}
ip::tcp::socket sock(service);
…
read = false;
char data[512];
sock.async_read_some(buffer(data, 512));
deadline_timer t(service, boost::posix_time::milliseconds(100));
t.async_wait(&deadline_handler);
service.run();
```

在上述代码片段中，如果你在超时之前读完了数据，*read* 则被设置成 true，这样我们的伙伴就及时地通知了我们。否则，当 *deadline_handler* 被调用时，*read* 还是 false，也就意味着我们的操作超时了。

Boost.Asio 也支持同步计时器，但是它们通常和一个简单的 sleep 操作是一样的。*boost::this_thread::sleep(500);*这段代码和下面的代码片段完成了同一件事情：

```
deadline_timer t(service, boost::posix_time::milliseconds(500));
t.wait();
```

io_service 类

你应该已经发现大部分使用 Boost.Asio 编写的代码都会使用几个 *io_service* 的实例。*io_service* 是这个库里面最重要的类；它负责和操作系统打交道，等待所有异步操作的结束，然后为每一个异步操作调用其完成处理程序。

如果你选择用同步的方式来创建你的应用，你则不需要考虑我将在这一节向你展示的东西。 你有多种不同的方式来使用 *io_service*。在下面的例子中，我们有 3 个异步操作，2 个 socket 连接操作和一个计时器等待操作：

- 有一个 *io_service* 实例和一个处理线程的单线程例子：

```
io_service service; // 所有 socket 操作都由 service 来处理
ip::tcp::socket sock1(service);
ip::tcp::socket sock2(service);
sock1.async_connect( ep, connect_handler);
sock2.async_connect( ep, connect_handler);
deadline_timer t(service, boost::posixtime::seconds(5));
t.async_wait(timeout_handler);
service.run();
```

- 有一个 io_service 实例和多个处理线程的多线程例子：

```
io_service service;
ip::tcp::socket sock1(service);
ip::tcp::socket sock2(service);
sock1.asyncconnect( ep, connect_handler);
sock2.async_connect( ep, connect_handler);
deadline_timer t(service, boost::posixtime::seconds(5));
t.async_wait(timeout_handler);
for ( int i = 0; i < 5; ++i )
    boost::thread( run_service);
void run_service()
{
    service.run();
}
```

- 有多个 *io_service* 实例和多个处理线程的多线程例子：

```
io_service service[2];
ip::tcp::socket sock1(service[0]);
ip::tcp::socket sock2(service[1]);
sock1.async_connect( ep, connect_handler);
sock2.async_connect( ep, connect_handler);
deadline_timer t(service[0], boost::posixtime::seconds(5));
t.async_wait(timeout_handler);
for ( int i = 0; i < 2; ++i)
    boost::thread( boost::bind(run_service, i));
void run_service(int idx)
{
    service[idx].run();
}
```

首先，要注意你不能拥有多个 *io_service* 实例却只有一个线程。下面的代码片段没有任何意义：

```
for ( int i = 0; i < 2; ++i)
    service[i].run();
```

上面的代码片段没有意义是因为 *service[1].run()*需要 *service[0].run()*先结束。因此，所有由 *service[1]*处理的异步操作都需要等待，这显然不是一个好主意。

在前面的 3 个方案中，我们在等待 3 个异步操作结束。为了解释它们之间的不同点，我们假设：过一会操作 1 完成，然后接着操作 2 完成。同时我们假设每一个完成处理程序需要 1 秒钟来完成执行。

在第一个例子中，我们在一个线程中等待三个操作全部完成，第 1 个操作一完成，我们就调用它的完成处理程序。尽管操作 2 紧接着完成了，但是操作 2 的完成处理程序需要在 1 秒钟后，也就是操作 1 的完成处理程序完成时才会被调用。

第二个例子，我们在两个线程中等待 3 个异步操作结束。当操作 1 完成时，我们在第 1 个线程中调用它的完成处理程序。当操作 2 完成时，紧接着，我们就在第 2 个线程中调用它的完成处理程序（当线程 1 在忙着响应操作 1 的处理程序时，线程 2 空闲着并且可以回应任何新进来的操作）。

在第三个例子中，因为操作 1 是 *sock1* 的 *connect*，操作 2 是 *sock2* 的 *connect*，所以应用程序会表现得像第二个例子一样。线程 1 会处理 *sock1 connect* 操作的完成处理程序，线程 2 会处理 *sock2* 的 *connect* 操作的完成处理程序。然而，如果 *sock1* 的 *connect* 操作是操作 1，*deadline_timer t* 的超时操作是操作 2，线程 1 会结束正在处理的 *sock1connect* 操作的完成处理程序。因而，*deadline_timer t* 的超时操作必须等 *sock1 connect* 操作的完成处理程序结束（等待 1 秒钟），因为线程 1 要处理 *sock1* 的连接处理程序和 *t* 的超时处理程序。

下面是你需要从前面的例子中学到的：

- 第一种情况是非常基础的应用程序。因为是串行的方式，所以当几个处理程序需要被同时调用时，你通常会遇到瓶颈。如果一个处理程序需要花费很长的时间来执行，所有随后的处理程序都不得不等待。
- 第二种情况是比较适用的应用程序。他是非常强壮的——如果几个处理程序被同时调用了（这是有可能的），它们会在各自的线程里面被调用。唯一的瓶颈就是所有的处理线程都很忙的同时又有新的处理程序被调用。然而，这是有快速的解决方式的，增加处理线程的数目即可。
- 第三种情况是最复杂和最难理解的。你只有在第二种情况不能满足需求时才使用它。这种情况一般就是当你有成千上万实时（socket）连接时。你可以认为每一个处理线程（运行 *io_service::run()*的线程）有它自己的 *select/epoll* 循环；它等待任意一个 socket 连接，然后等待一个读写操作，当它发现这种操作时，就执行。大部分情况下，你不需要担心什么，唯一你需要担心的就是当你监控的 socket 数目以指数级的方式增长时（超过 1000 个的 socket）。在那种情况下，有多个 select/epoll 循环会增加应用的响应时间。

如果你觉得你的应用程序可能需要转换到第三种模式，请确保监听操作的这段代码（调用 *io_service::run()*的代码）和应用程序其他部分是隔离的，这样你就可以很轻松地对其进行更改。

最后，需要一直记住的是如果没有其他需要监控的操作，*.run()*就会结束，就像下面的代码片段：

```
io_service service;
tcp::socket sock(service);
sock.async_connect( ep, connect_handler);
service.run();
```

在上面的例子中，只要 sock 建立了一个连接，*connect_handler* 就会被调用，然后接着 *service.run()*就会完成执行。

如果你想要 *service.run()*接着执行，你需要分配更多的工作给它。这里有两个方式来完成这个目标。一种方式是在 *connect_handler* 中启动另外一个异步操作来分配更多的工作。 另一种方式会模拟一些工作给它，用下面的代码片段：

```
typedef boost::shared_ptr work_ptr;
work_ptr dummy_work(new io_service::work(service));
```

上面的代码可以保证 *service.run()*一直运行直到你调用 *useservice.stop()*或者 *dummy_work.reset(0);//* 销毁 dummy_work.

总结

做为一个复杂的库，Boost.Asio 让网络编程变得异常简单。构建起来也简单。而且在避免使用宏这一点上也做得很好；它虽然定义了少部分的宏来做选项开关，但是你需要关心的很少。

Boost.Asio 支持同步和异步编程。他们有很大不同；你需要在项目早期就选择其中的一种来实现，因为它们之间的转换是非常复杂而且易错的。

如果你选择同步，你可以选择异常处理或者错误码，从异常处理转到错误码；只需要在 call 函数中增加一个参数即可（错误码）。

Boost.Asio 不仅仅可以用来做网络编程。它还有其他更多的特性，这让它显得更有价值，比如信号量，计时器等等。

下一章我们将深入研究大量 Boost.Asio 中用来做网络编程的函数和类。同时我们也会学一些异步编程的诀窍。

第二章

Boost.Asio 基本原理

这一章涵盖了使用 Boost.Asio 时必须知道的一些事情。我们也将深入研究比同步编程更复杂、更有乐趣的异步编程。

网络 API

这一部分包含了当使用 Boost.Asio 编写网络应用程序时必须知道的事情。

Boost.Asio 命名空间

Boost.Asio 的所有内容都包含在 boost::asio 命名空间或者其子命名空间内。

- *boost::asio*：这是核心类和函数所在的地方。重要的类有 io_service 和 streambuf。类似 *read, read_at, read_until* 方法，它们的异步方法，它们的写方法和异步写方法等自由函数也在这里。
- *boost::asio::ip*：这是网络通信部分所在的地方。重要的类有 *address, endpoint, tcp, udp 和 icmp*，重要的自由函数有 *connect* 和 *async_connect*。要注意的是在 *boost::asio::ip::tcp::socket* 中间，*socket* 只是 *boost::asio::ip::tcp* 类中间的一个 *typedef* 关键字。
- *boost::asio::error*：这个命名空间包含了调用 I/O 例程时返回的错误码
- *boost::asio::ssl*：包含了 SSL 处理类的命名空间
- *boost::asio::local*：这个命名空间包含了 POSIX 特性的类
- *boost::asio::windows*：这个命名空间包含了 Windows 特性的类

IP 地址

对于 IP 地址的处理，Boost.Asio 提供了 *ip::address* , *ip::address_v4* 和 *ip::address_v6* 类。 它们提供了相当多的函数。下面列出了最重要的几个：

- *ip::address(v4_or_v6_address)*:这个函数把一个 v4 或者 v6 的地址转换成 *ip::address*
- *ip::address:from_string(str)*：这个函数根据一个 IPv4 地址（用.隔开的）或者一个 IPv6 地址（十六进制表示）创建一个地址。

- *ip::address::to_string()* ：这个函数返回这个地址的字符串。
- *ip::address_v4::broadcast([addr, mask])*:这个函数创建了一个广播地址 *ip::address_v4::any()*：这个函数返回一个能表示任意地址的地址。
- *ip::address_v4::loopback(), ip_address_v6::loopback()*：这个函数返回环路地址（为 v4/v6 协议）
- *ip::host_name()*：这个函数用 string 数据类型返回当前的主机名。

大多数情况你会选择用 *ip::address::from_string*：

```
ip::address addr = ip::address::from_string("127.0.0.1");
```

如果你想通过一个主机名进行连接，下面的代码片段是无用的：

```
// 抛出异常
ip::address addr = ip::address::from_string("www.yahoo.com");
```

端点

端点是使用某个端口连接到的一个地址。不同类型的 socket 有它自己的 *endpoint* 类，比如 *ip::tcp::endpoint*、*ip::udp::endpoint* 和 *ip::icmp::endpoint*

如果想连接到本机的 80 端口，你可以这样做：

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 80);
```

有三种方式来让你建立一个端点：

- *endpoint()*：这是默认构造函数，某些时候可以用来创建 UDP/ICMP socket。
- *endpoint(protocol, port)*：这个方法通常用来创建可以接受新连接的服务器端 socket。
- *endpoint(addr, port)*:这个方法创建了一个连接到某个地址和端口的端点。

例子如下：

```
ip::tcp::endpoint ep1;
ip::tcp::endpoint ep2(ip::tcp::v4(), 80);
ip::tcp::endpoint ep3( ip::address::from_string("127.0.0.1), 80);
```

如果你想连接到一个主机（不是 IP 地址），你需要这样做：

```
// 输出 "87.248.122.122"
io_service service;
ip::tcp::resolver resolver(service);
ip::tcp::resolver::query query("www.yahoo.com", "80");
ip::tcp::resolver::iterator iter = resolver.resolve( query);
ip::tcp::endpoint ep = *iter;
```

```
std::cout << ep.address().to_string() << std::endl;
```
你可以用你需要的 socket 类型来替换 tcp。首先，为你想要查询的名字创建一个查询器，然后用 resolve()函数解析它。如果成功，它至少会返回一个入口。你可以利用返回的迭代器，使用第一个入口或者遍历整个列表来拿到全部的入口。

给定一个端点，可以获得他的地址，端口和 IP 协议（v4 或者 v6）：

```
std::cout << ep.address().to_string() << ":" << ep.port()
<< "/" << ep.protocol() << std::endl;
```

套接字

Boost.Asio 有三种类型的套接字类：*ip::tcp, ip::udp* 和 *ip::icmp*。当然它也是可扩展的，你可以创建自己的 socket 类，尽管这相当复杂。如果你选择这样做，参照一下 *boost/asio/ip/tcp.hpp, boost/asio/ip/udp.hpp* 和 *boost/asio/ip/icmp.hpp*。它们都是含有内部 typedef 关键字的超小类。

你可以把 *ip::tcp, ip::udp, ip::icmp* 类当作占位符；它们可以让你便捷地访问其他类/函数，如下所示：

- *ip::tcp::socket, ip::tcp::acceptor, ip::tcp::endpoint,ip::tcp::resolver, ip::tcp::iostream*
- *ip::udp::socket, ip::udp::endpoint, ip::udp::resolver*
- *ip::icmp::socket, ip::icmp::endpoint, ip::icmp::resolver*

socket 类创建一个相应的 *socket*。而且总是在构造的时候传入 io_service 实例：

```
io_service service;
ip::udp::socket sock(service)
sock.set_option(ip::udp::socket::reuse_address(true));
```
每一个 socket 的名字都是一个 typedef 关键字

- *ip::tcp::socket = basic_stream_socket*
- *ip::udp::socket = basic_datagram_socket*
- *ip::icmp::socket = basic_raw_socket*

同步错误码

所有的同步函数都有抛出异常或者返回错误码的重载，比如下面的代码片段：

```
sync_func( arg1, arg2 ... argN); // 抛出异常
```

```
boost::system::error_code ec;
sync_func( arg1 arg2, ..., argN, ec); // 返回错误码
```

在这一章剩下的部分，你会见到大量的同步函数。简单起见，我省略了有返回错误码的重载，但是不可否认它们确实是存在的。

socket 成员方法

这些方法被分成了几组。并不是所有的方法都可以在各个类型的套接字里使用。这个部分的结尾将有一个列表来展示各个方法分别属于哪个 socket 类。

注意所有的异步方法都立刻返回，而它们相对的同步实现需要操作完成之后才能返回。

连接相关的函数

这些方法是用来连接或绑定 socket、断开 socket 字连接以及查询连接是活动还是非活动的：

- *assign(protocol,socket)*：这个函数分配了一个原生的 socket 给这个 socket 实例。当处理老（旧）程序时会使用它（也就是说，原生 socket 已经被建立了）
- *open(protocol)*：这个函数用给定的 IP 协议（v4 或者 v6）打开一个 socket。你主要在 UDP/ICMP socket，或者服务端 socket 上使用。
- *bind(endpoint)*：这个函数绑定到一个地址
- *connect(endpoint)*：这个函数用同步的方式连接到一个地址
- *async_connect(endpoint)*：这个函数用异步的方式连接到一个地址
- *is_open()*：如果套接字已经打开，这个函数返回 true
- *close()*：这个函数用来关闭套接字。调用时这个套接字上任何的异步操作都会被立即关闭，同时返回 *error::operation_aborted* 错误码。
- *shutdown(type_of_shutdown)*：这个函数立即使 send 或者 receive 操作失效，或者两者都失效。
- *cancel()*：这个函数取消套接字上所有的异步操作。这个套接字上任何的异步操作都会立即结束，然后返回 *error::operation_aborted* 错误码。

例子如下：

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 80);
ip::tcp::socket sock(service);
sock.open(ip::tcp::v4()); n
sock.connect(ep);
```

```
sock.write_some(buffer("GET /index.html\r\n"));
char buff[1024];
sock.read_some(buffer(buff,1024));
sock.shutdown(ip::tcp::socket::shutdown_receive);
sock.close();
```

读写函数

这些是在套接字上执行 I/O 操作的函数。

对于异步函数来说，处理程序的格式 *void handler(const boost::system::error_code& e, size_t bytes)*都是一样的

- *async_receive(buffer, [flags,] handler)*：这个函数启动从套接字异步接收数据的操作。
- *async_read_some(buffer,handler)*：这个函数和 *async_receive(buffer, handler)*功能一样。
- *async_receive_from(buffer, endpoint[, flags], handler)*：这个函数启动从一个指定端点异步接收数据的操作。
- *async_send(buffer [, flags], handler)*：这个函数启动了一个异步发送缓冲区数据的操作。
- *async_write_some(buffer, handler)*：这个函数和 a*sync_send(buffer, handler)*功能一致。
- *async_send_to(buffer, endpoint, handler)*：这个函数启动了一个异步 send 缓冲区数据到指定端点的操作。
- *receive(buffer [, flags])*：这个函数异步地从所给的缓冲区读取数据。在读完所有数据或者错误出现之前，这个函数都是阻塞的。
- *read_some(buffer)*：这个函数的功能和 *receive(buffer)*是一致的。
    -
    - receive_from(buffer, endpoint [, flags])*：这个函数异步地从一个指定的端点获取数据并写入到给定的缓冲区。在读完所有数据或者错误出现之前，这个函数都是阻塞的。
- *send(buffer [, flags])*：这个函数同步地发送缓冲区的数据。在所有数据发送成功或者出现错误之前，这个函数都是阻塞的。
- *write_some(buffer)*：这个函数和 *send(buffer)*的功能一致。
- *send_to(buffer, endpoint [, flags])*：这个函数同步地把缓冲区数据发送到一个指定的端点。在所有数据发送成功或者出现错误之前，这个函数都是阻塞的。
- *available()*：这个函数返回有多少字节的数据可以无阻塞地进行同步读取。

稍后我们将讨论缓冲区。让我们先来了解一下标记。标记的默认值是 0，但是也可以是以下几种：

- *ip::socket_type::socket::message_peek*：这个标记只监测并返回某个消息，但是下一次读消息的调用会重新读取这个消息。
- *ip::socket_type::socket::message_out_of_band*：这个标记处理带外（OOB）数据，OOB 数据是被标记为比正常数据更重要的数据。关于 OOB 的讨论在这本书的内容之外。
- *ip::socket_type::socket::message_do_not_route*：这个标记指定数据不使用路由表来发送。
- *ip::socket_type::socket::message_end_of_record*：这个标记指定的数据标识了记录的结束。在 Windows 下不支持。

你最常用的可能是 *message_peek*，使用方法请参照下面的代码片段：

```
char buff[1024];
sock.receive(buffer(buff), ip::tcp::socket::message_peek );
memset(buff,0, 1024);
// 重新读取之前已经读取过的内容
sock.receive(buffer(buff) );
```

下面的是一些教你如何同步或异步地从不同类型的套接字上读取数据的例子：

- 例 1 是在一个 TCP 套接字上进行同步读写：

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 80);
ip::tcp::socket sock(service);
sock.connect(ep);
sock.write_some(buffer("GET /index.html\r\n"));
std::cout << "bytes available " << sock.available() << std::endl;
char buff[512];
size_t read = sock.read_some(buffer(buff));
```

- 例 2 是在一个 UDP 套接字上进行同步读写：

```
ip::udp::socket sock(service);
sock.open(ip::udp::v4());
ip::udp::endpoint receiver_ep("87.248.112.181", 80);
sock.send_to(buffer("testing\n"), receiver_ep);
char buff[512];
ip::udp::endpoint sender_ep;
sock.receive_from(buffer(buff), sender_ep);
```

[？注意：就像上述代码片段所展示的那样，使用 receive_from 从一个 UDP 套接字读取数据时，你需要构造一个默认的端点]

- 例 3 是从一个 UDP 服务套接字中异步读取数据：

```
using namespace boost::asio;
io_service service;
ip::udp::socket sock(service);
boost::asio::ip::udp::endpoint sender_ep;
char buff[512];

void on_read(const boost::system::error_code & err, std::size_t read_bytes)
{
    std::cout << "read " << read_bytes << std::endl;
    sock.async_receive_from(buffer(buff), sender_ep, on_read);
}

int main(int argc, char* argv[])
{
    ip::udp::endpoint ep(ip::address::from_string("127.0.0.1"),8001);
    sock.open(ep.protocol());
    sock.set_option(boost::asio::ip::udp::socket::reuse_address(true));
    sock.bind(ep);
    sock.async_receive_from(buffer(buff,512), sender_ep, on_read);
    service.run();
}
```

套接字控制：

这些函数用来处理套接字的高级选项：

- *get_io_service()*：这个函数返回构造函数中传入的 io_service 实例
- *get_option(option)*：这个函数返回一个套接字的属性
- *set_option(option)*：这个函数设置一个套接字的属性
- *io_control(cmd)*：这个函数在套接字上执行一个 I/O 指令

这些是你可以获取/设置的套接字选项：

名字	描述	类型
broadcast	如果为 true，允许广播消息	bool
debug	如果为 true，启用套接字级别的调试	bool
do_not_route	如果为 true，则阻止路由选择只使用本地接口	bool
enable_connection_aborted	如果为 true，记录在 accept()时中断的连接	bool
keep_alive	如果为 true，会发送心跳	bool
linger	如果为 true，套接字会在有未发送数据的情况下挂起 close()	bool
receive_buffer_size	套接字接收缓冲区大小	int
receive_low_watemark	规定套接字输入处理的最小字节数	int
reuse_address	如果为 true，套接字能绑定到一个已用的地址	bool
send_buffer_size	套接字发送缓冲区大小	int
send_low_watermark	规定套接字数据发送的最小字节数	int
ip::v6_only	如果为 true，则只允许 IPv6 的连接	bool

每个名字代表了一个内部套接字 typedef 或者类。下面是对它们的使用：

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 80);
ip::tcp::socket sock(service);
sock.connect(ep);
// TCP 套接字可以重用地址
ip::tcp::socket::reuse_address ra(true);
sock.set_option(ra);
// 获取套接字读取的数据
ip::tcp::socket::receive_buffer_size rbs;
```

```
sock.get_option(rbs);
std::cout << rbs.value() << std::endl;
// 把套接字的缓冲区大小设置为 8192
ip::tcp::socket::send_buffer_size sbs(8192);
sock.set_option(sbs);
```

[?在上述特性工作之前，套接字要被打开。否则，会抛出异常]

TCP VS UDP VS ICMP

就像我之前所说，不是所有的成员方法在所有的套接字类中都可用。我做了一个包含成员函数不同点的列表。如果一个成员函数没有出现在这，说明它在所有的套接字类都是可用的。

名字	TCP	UDP	ICMP
async_read_some	是	-	-
async_receive_from	-	是	是
async_write_some	是	-	-
async_send_to	-	是	是
read_some	是	-	-
receive_from	-	是	是
write_some	是	-	-
send_to	-	是	是

其他方法

其他与连接和 I/O 无关的函数如下：

- *local_endpoint()*：这个方法返回套接字本地连接的地址。
- *remote_endpoint()*：这个方法返回套接字连接到的远程地址。
- *native_handle()*：这个方法返回原始套接字的处理程序。你只有在调用一个 Boost.Asio 不支持的原始方法时才需要用到它。
- *non_blocking()*：如果套接字是非阻塞的，这个方法返回 true，否则 false。
- *native_non_blocking()*：如果套接字是非阻塞的，这个方法返回 true，否则返回 false。但是，它是基于原生的套接字来调用本地的 api。所以通常来说，你不需要调用这个方法（non_blocking()已经缓存了这个结果）；你只有在直接调用 native_handle()这个方法的时候才需要用到这个方法。
- *at_mark()*：如果套接字要读的是一段 OOB 数据，这个方法返回 true。这个方法你很少会用到。

其他需要考虑的事情

最后要注意的一点，套接字实例不能被拷贝，因为拷贝构造方法和＝操作符是不可访问的。

```
ip::tcp::socket s1(service), s2(service);
s1 = s2; // 编译时报错
ip::tcp::socket s3(s1); // 编译时报错
```

这是非常有意义的，因为每一个实例都拥有并管理着一个资源（原生套接字本身）。如果我们允许拷贝构造，结果是我们会有两个实例拥有同样的原生套接字；这样我们就需要去处理所有者的问题（让一个实例拥有所有权？或者使用引用计数？还是其他的方法）Boost.Asio 选择不允许拷贝（如果你想要创建一个备份，请使用共享指针）

```
typedef boost::shared_ptr<ip::tcp::socket> socket_ptr;
socket_ptr sock1(new ip::tcp::socket(service));
socket_ptr sock2(sock1); // ok
socket_ptr sock3;
sock3 = sock1; // ok
```

套接字缓冲区

当从一个套接字读写内容时，你需要一个缓冲区，用来保存读取和写入的数据。缓冲区内存的有效时间必须比 I/O 操作的时间要长；你需要保证它们在 I/O 操作结束之前不被释放。 对于同步操作来说，这很容易；当然，这个缓冲区在 receive 和 send 时都存在。

```
char buff[512];
...
sock.receive(buffer(buff));
strcpy(buff, "ok\n");
sock.send(buffer(buff));
```

但是在异步操作时就没这么简单了，看下面的代码片段：

```
// 非常差劲的代码 ...
void on_read(const boost::system::error_code & err, std::size_t read_bytes)
{ ... }
void func() {
    char buff[512];
    sock.async_receive(buffer(buff), on_read);
}
```

在我们调用 *async_receive()*之后，buff 就已经超出有效范围，它的内存当然会被释放。当我们开始从套接字接收一些数据时，我们会把它们拷贝到一片已经不属于我们的内存中；它可能会被释放，或者被其他代码重新开辟来存入其他的数据，结果就是：内存冲突。

对于上面的问题有几个解决方案：

- 使用全局缓冲区
- 创建一个缓冲区，然后在操作结束时释放它
- 使用一个集合对象管理这些套接字和其他的数据，比如缓冲区数组

第一个方法显然不是很好，因为我们都知道全局变量非常不好。此外，如果两个实例使用同一个缓冲区怎么办？

下面是第二种方式的实现：

```
void on_read(char * ptr, const boost::system::error_code & err, std::size_t
read_bytes)
{
    delete[] ptr;
}
....
char * buff = new char[512];
sock.async_receive(buffer(buff, 512), boost::bind(on_read,buff,_1,_2))
```

或者，如果你想要缓冲区在操作结束后自动超出范围，使用共享指针

```
struct shared_buffer
{
    boost::shared_array<char> buff;
```

```
    int size;
    shared_buffer(size_t size) : buff(new char[size]), size(size) { }
    mutable_buffers_1 asio_buff() const
    {
        return buffer(buff.get(), size);
    }
};

// 当 on_read 超出范围时，boost::bind 对象被释放了，
// 同时也会释放共享指针
void on_read(shared_buffer, const boost::system::error_code & err, std::size_t
read_bytes) {}
sock.async_receive(buff.asio_buff(), boost::bind(on_read,buff,_1,_2));
```

shared_buffer 类拥有实质的 *shared_array<>*，*shared_array<>*存在的目的是用来保存 *shared_buffer* 实例的拷贝－当最后一个 *share_array<>*元素超出范围时，*shared_array<>*就被自动销毁了，而这就是我们想要的结果。

因为 Boost.Asio 会给完成处理句柄保留一个拷贝，当操作完成时就会调用这个完成处理句柄，所以你的目的达到了。那个拷贝是一个 boost::bind 的仿函数，它拥有着实际的 *shared_buffer* 实例。这是非常优雅的！

第三个选择是使用一个连接对象来管理套接字和其他数据，比如缓冲区，通常来说这是正确的解决方案但是非常复杂。在这一章的末尾我们会对这种方法进行讨论。

缓冲区封装函数

纵观所有代码，你会发现：无论什么时候，当我们需要对一个 buffer 进行读写操作时，代码会把实际的缓冲区对象封装在一个 buffer()方法中，然后再把它传递给方法调用：

```
char buff[512];
sock.async_receive(buffer(buff), on_read);
```

基本上我们都会把缓冲区包含在一个类中以便 Boost.Asio 的方法能遍历这个缓冲区，比方说，使用下面的代码：

```
sock.async_receive(some_buffer, on_read);
```

实例 *some_buffer* 需要满足一些需求，叫做 *ConstBufferSequence* 或者 *MutableBufferSequence*（你可以在 Boost.Asio 的文档中查看它们）。创建你自己的类去处理这些需求的细节是非常复杂的，但是 Boost.Asio 已经提供了一些

类用来处理这些需求。所以你不用直接访问这些缓冲区，而可以使用 *buffer()*方法。

自信地讲，你可以把下面列出来的类型都包装到一个 buffer()方法中：

- 一个 char[] const 数组
- 一个字节大小的 void *指针
- 一个 std::string 类型的字符串
- 一个 POD const 数组（POD 代表纯数据，这意味着构造器和释放器不做任何操作）
- 一个 pod 数据的 std::vector
- 一个包含 pod 数据的 boost::array
- 一个包含 pod 数据的 std::array

下面的代码都是有效的：

```
struct pod_sample { int i; long l; char c; };
...
char b1[512];
void * b2 = new char[512];
std::string b3; b3.resize(128);
pod_sample b4[16];
std::vector<pod_sample> b5; b5.resize(16);
boost::array<pod_sample,16> b6;
std::array<pod_sample,16> b7;
sock.async_send(buffer(b1), on_read);
sock.async_send(buffer(b2,512), on_read);
sock.async_send(buffer(b3), on_read);
sock.async_send(buffer(b4), on_read);
sock.async_send(buffer(b5), on_read);
sock.async_send(buffer(b6), on_read);
sock.async_send(buffer(b7), on_read);
```

总的来说就是：与其创建你自己的类来处理 *ConstBufferSequence* 或者 *MutableBufferSequence* 的需求，不如创建一个能在你需要的时候保留缓冲区，然后返回一个 mutable_buffers_1 实例的类，而我们早在 shared_buffer 类中就这样做了。

read/write/connect 自由函数

Boost.Asio 提供了处理 I/O 的自由函数，我们分四组来分析它们。

connect 方法

这些方法把套接字连接到一个端点。

- *connect(socket, begin [, end] [, condition])*：这个方法遍历队列中从 start 到 end 的端点来尝试同步连接。begin 迭代器是调用 *socket_type::resolver::query* 的返回结果（你可能需要回顾一下端点这个章节）。特别提示 end 迭代器是可选的；你可以忽略它。你还可以提供一个 condition 的方法给每次连接尝试之后调用。用法是 *Iterator connect_condition(const boost::system::error_code & err,Iterator next);*。你可以选择返回一个不是 *next* 的迭代器，这样你就可以跳过一些端点。
- *async_connect(socket, begin [, end] [, condition], handler)*：这个方法异步地调用连接方法，在结束时，它会调用完成处理方法。用法是 *void handler(constboost::system::error_code & err, Iterator iterator);*。传递给处理方法的第二个参数是连接成功端点的迭代器（或者 end 迭代器）。

它的例子如下：

```
using namespace boost::asio::ip;
tcp::resolver resolver(service);
tcp::resolver::iterator iter =
resolver.resolve(tcp::resolver::query("www.yahoo.com","80"));
tcp::socket sock(service);
connect(sock, iter);
```

一个主机名可以被解析成多个地址，而 *connect* 和 *async_connect* 能很好地把你从尝试每个地址然后找到一个可用地址的繁重工作中解放出来，因为它们已经帮你做了这些。

read/write 方法

这些方法对一个流进行读写操作（可以是套接字，或者其他表现得像流的类）：

- *async_read(stream, buffer [, completion] ,handler)*：这个方法异步地从一个流读取。结束时其处理方法被调用。处理方法的格式是：*void handler(const boost::system::error_ code & err, size_t bytes)*;。你可以选择指定一个完成处理方法。完成处理方法会在每个 *read* 操作调用成功之后调用，然后告诉 Boost.Asio*async_read* 操作是否完成（如果没有完成，它会继续读取）。它的格式是：*size_t completion(const boost::system::error_code& err, size_t bytes_transfered)* 。当这个完成处理方法返回 0 时，我们认为*read* 操作完成；如果

它返回一个非 0 值，它表示了下一个 *async_read_some* 操作需要从流中读取的字节数。接下来会有一个例子来详细展示这些。

- *async_write(stream, buffer [, completion], handler)*：这个方法异步地向一个流写入数据。参数的意义和 *async_read* 是一样的。
- *read(stream, buffer [, completion])*：这个方法同步地从一个流中读取数据。参数的意义和 *async_read* 是一样的。
- *write(stream, buffer [, completion])*: 这个方法同步地向一个流写入数据。参数的意义和 *async_read* 是一样的。

```
async_read(stream, stream_buffer [, completion], handler)
async_write(strean, stream_buffer [, completion], handler)
write(stream, stream_buffer [, completion])
read(stream, stream_buffer [, completion])
```

首先，要注意第一个参数变成了流，而不单是 socket。这个参数包含了 socket 但不仅仅是 socket。比如，你可以用一个 Windows 的文件句柄来替代 socket。当下面情况出现时，所有 read 和 write 操作都会结束：

- 可用的缓冲区满了（当读取时）或者所有的缓冲区已经被写入（当写入时）
- 完成处理方法返回 0（如果你提供了这么一个方法）
- 错误发生时

下面的代码会异步地从一个 socket 中间读取数据直到读取到'\n'：

```
io_service service;
ip::tcp::socket sock(service);
char buff[512];
int offset = 0;
size_t up_to_enter(const boost::system::error_code &, size_t bytes)
{
    for ( size_t i = 0; i < bytes; ++i)
        if ( buff[i + offset] == '\n')
            return 0;
    return 1;
 }
void on_read(const boost::system::error_code &, size_t) {}
...
async_read(sock, buffer(buff), up_to_enter, on_read);
```

Boost.Asio 也提供了一些简单的完成处理仿函数：

- transfer_at_least(n)
- transfer_exactly(n)

- transfer_all()

例子如下：

```
char buff[512];
void on_read(const boost::system::error_code &, size_t) {}
// 读取 32 个字节
async_read(sock, buffer(buff), transfer_exactly(32), on_read);
```

上述的 4 个方法，不使用普通的缓冲区，而使用由 Boost.Asio 的 *std::streambuf* 类继承来的 *stream_buffer* 方法。stl 流和流缓冲区非常复杂；下面是例子：

```
io_service service;
void on_read(streambuf& buf, const boost::system::error_code &, size_t)
{
    std::istream in(&buf);
    std::string line;
    std::getline(in, line);
    std::cout << "first line: " << line << std::endl;
}
int main(int argc, char* argv[])
{
    HANDLE file = ::CreateFile("readme.txt", GENERIC_READ, 0, 0, OPEN_ALWAYS,
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, 0);
    windows::stream_handle h(service, file);
    streambuf buf;
    async_read(h, buf, transfer_exactly(256),
    boost::bind(on_read,boost::ref(buf),_1,_2));
    service.run();
}
```

在这里，我向你们展示了如何在一个 Windows 文件句柄上调用 *async_read*。读取前 256 个字符，然后把它们保存到缓冲区中，当操作结束时。*on_read* 被调用，再创建 *std::istream* 用来传递缓冲区，读取第一行（*std::getline*），最后把它输出到命令行中。

read_until/async_read_until 方法

这些方法在条件满足之前会一直读取：

- *async_read_until(stream, stream_buffer, delim, handler)*:这个方法启动一个异步 *read* 操作。*read* 操作会在读取到某个分隔符时结束。分隔符可以是字符,*std::string*

或者 *boost::regex*。处理方法的格式为：*void handler(const boost::system::error_code & err, size_t bytes);*。

- *async_read_until(strem, stream_buffer, completion, handler)*：这个方法和之前的方法是一样的，但是没有分隔符，而是一个完成处理方法。完成处理方法的格式为：*pair< iterator,bool > completion(iterator begin, iterator end);*，其中迭代器的类型为 *buffers_iterator< streambuf::const_buffers_type >*。你需要记住的是这个迭代器是支持随机访问的。你扫描整个区间（begin，end），然后决定 read 操作是否应该结束。返回的结果是一个结果对，第一个成员是一个迭代器，它指向最后被这个方法访问的字符；第二个成员指定 read 操作是否需要结束，需要时返回 true，否则返回 false。
- *read_until(stream, stream_buffer, delim)*：这个方法执行一个同步的 *read* 操作，参数的意义和 *async_read_until* 一样。
- *read_until(stream, stream_buffer, completion)*：这个方法执行一个同步的 read 操作，参数的意义和 *async_read_until* 一样。

下面这个例子在读到一个指定的标点符号之前会一直读取：

```
typedef buffers_iterator<streambuf::const_buffers_type> iterator;
std::pair<iterator, bool> match_punct(iterator begin, iterator end)
{
    while ( begin != end)
        if ( std::ispunct(*begin))
            return std::make_pair(begin,true);
    return std::make_pair(end,false);
}
void on_read(const boost::system::error_code &, size_t) {}
...
streambuf buf;
async_read_until(sock, buf, match_punct, on_read);
```

如果我们想读到一个空格时就结束，我们需要把最后一行修改为：

```
async_read_until(sock, buff, ' ', on_read);
```

*_at 方法

这些方法用来在一个流上面做随机存取操作。由你来指定 *read* 和 *write* 操作从什么地方开始（*offset*）：

- *async_read_at(stream, offset, buffer [, completion], handler)*：这个方法在指定的流的 offset 处开始执行一个异步的 read 操作，当操作结束时，它会调用 handler。

handler 的格式为：*void handler(const boost::system::error_code& err, size_t bytes);*。*buffer* 可以是普通的 *wrapper()*封装或者 *streambuf* 方法。如果你指定一个 completion 方法，它会在每次 read 操作成功之后调用，然后告诉 Boost.Asio *async_read_at* 操作已经完成（如果没有，则继续读取）。它的格式为：*size_t completion(const boost::system::error_code& err, size_t bytes);*。当 completion 方法返回 0 时，我们认为 *read* 操作完成了；如果返回一个非零值，它代表了下一次调用流的 *async_read_some_at* 方法的最大读取字节数。

- *async_write_at(stream, offset, buffer [, completion], handler)*：这个方法执行一个异步的 write 操作。参数的意义和 *async_read_at* 是一样的
- *read_at(stream, offset, buffer [, completion])*：这个方法在一个执行的流上，指定的 *offset* 处开始 read。参数的意义和 *async_read_at* 是一样的
- *write_at(stream, offset, buffer [, completion])*：这个方法在一个执行的流上，指定的 *offset* 处开始 write。参数的意义和 *async_read_at* 是一样的

这些方法不支持套接字。它们用来处理流的随机访问；也就是说，流是可以随机访问的。套接字显然不是这样（套接字是不可回溯的）。

下面这个例子告诉你怎么从一个文件偏移为 256 的位置读取 128 个字节：

```
io_service service;
int main(int argc, char* argv[])
{
    HANDLE file = ::CreateFile("readme.txt", GENERIC_READ, 0, 0, OPEN_ALWAYS,
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, 0);
    windows::random_access_handle h(service, file);
    streambuf buf;
    read_at(h, 256, buf, transfer_exactly(128));
    std::istream in(&buf);
    std::string line;
    std::getline(in, line);
    std::cout << "first line: " << line << std::endl;
}
```

异步编程

这部分对异步编程时可能碰到的一些问题进行了深入的探究。我建议你先读一遍，然后在接下来读这本书的过程中，再经常回过头来看看，从而增强你对这部分的理解。

异步的需求

就像我之前所说的，同步编程比异步编程简单很多。这是因为，线性的思考是很简单的（调用 A，调用 A 结束，调用 B，调用 B 结束，然后继续，这是以事件处理的方式来思考）。后面你会碰到这种情况，比如：五件事情，你不知道它们执行的顺序，也不知道他们是否会执行！

尽管异步编程更难，但是你会更倾向于选择使用它，比如：写一个需要处理很多并发访问的服务端。并发访问越多，异步编程就比同步编程越简单。

假设：你有一个需要处理 1000 个并发访问的应用，从客户端发给服务端的每个信息都会再返回给客户端，以‘\n’结尾。

同步方式的代码，1 个线程：

```
using namespace boost::asio;
struct client
{
    ip::tcp::socket sock;
    char buff[1024]; // 每个信息最多这么大
    int already_read; // 你已经读了多少
};
std::vector<client> clients;
void handle_clients()
{
    while ( true)
        for ( int i = 0; i < clients.size(); ++i)
            if ( clients[i].sock.available() ) on_read(clients[i]);
}

void on_read(client & c)
{
    int to_read = std::min( 1024 - c.already_read, c.sock.available());
    c.sock.read_some( buffer(c.buff + c.already_read, to_read));
    c.already_read += to_read;
    if ( std::find(c.buff, c.buff + c.already_read, '\n') < c.buff +
        c.already_read)
    {
        int pos = std::find(c.buff, c.buff + c.already_read, '\n') - c.buff;
        std::string msg(c.buff, c.buff + pos);
        std::copy(c.buff + pos, c.buff + 1024, c.buff);
        c.already_read -= pos;
```

```
        on_read_msg(c, msg);
    }
}
void on_read_msg(client & c, const std::string & msg)
{
    // 分析消息，然后返回
    if ( msg == "request_login")
        c.sock.write( "request_ok\n");
    else if ...
}
```

有一种情况是在任何服务端（和任何基于网络的应用）都需要避免的，就是代码无响应的情况。在我们的例子里，我们需要 *handle_clients()*方法尽可能少的阻塞。如果方法在某个点上阻塞，任何进来的信息都需要等待方法解除阻塞才能被处理。

为了保持响应，只在一个套接字有数据的时候我们才读，也就是说，*if (clients[i].sock.available()) on_read(clients[i])*。在 *on_read* 时，我们只读当前可用的；调用 *read_until(c.sock, buffer(...), '\n')*会是一个非常糟糕的选择，因为直到我们从一个指定的客户端读取了完整的消息之前，它都是阻塞的（我们永远不知道它什么时候会读取到完整的消息）

这里的瓶颈就是 *on_read_msg()*方法；当它执行时，所有进来的消息都在等待。一个良好的 *on_read_msg()*方法实现会保证这种情况基本不会发生，但是它还是会发生（有时候向一个套接字写入数据，缓冲区满了时，它会被阻塞） 同步方式的代码，10 个线程

```
using namespace boost::asio;
struct client
{
    // ... 和之前一样
    bool set_reading()
    {
        boost::mutex::scoped_lock lk(cs_);
        if ( is_reading_) return false; // 已经在读取
        else { is_reading_ = true; return true; }
    }
    void unset_reading()
    {
        boost::mutex::scoped_lock lk(cs_);
        is_reading_ = false;
    }
private:
```

```
    boost::mutex cs_;
    bool is_reading_;
};
std::vector<client> clients;
void handle_clients()
{
    for ( int i = 0; i < 10; ++i)
        boost::thread( handle_clients_thread);
}
void handle_clients_thread()
{
    while ( true)
        for ( int i = 0; i < clients.size(); ++i)
            if ( clients[i].sock.available() )
                if ( clients[i].set_reading())
                {
                    on_read(clients[i]);
                    clients[i].unset_reading();
                }
}
void on_read(client & c)
{
    // 和之前一样
}
void on_read_msg(client & c, const std::string & msg)
{
    // 和之前一样
}
```

为了使用多线程，我们需要对线程进行同步，这就是 *set_reading()*和 *set_unreading()*所做的。*set_reading()*方法非常重要，比如你想要一步实现“判断是否在读取然后标记为读取中”。但这是有两步的（“判断是否在读取”和“标记为读取中”），你可能会有两个线程同时为一个客户端判断是否在读取，然后你会有两个线程同时为一个客户端调用 *on_read*，结果就是数据冲突甚至导致应用崩溃。

你会发现代码变得极其复杂。

同步编程有第三个选择，就是为每个连接开辟一个线程。但是当并发的线程增加时，这就成了一种灾难性的情况。

然后，让我们来看异步编程。我们不断地异步读取。当一个客户端请求某些东西时，*on_read* 被调用，然后回应，然后等待下一个请求（然后开始另外一个异步的 read 操作）。

异步方式的代码，10 个线程

```
using namespace boost::asio;
io_service service;
struct client
{
    ip::tcp::socket sock;
    streambuf buff; // 从客户端取回结果
}
std::vector<client> clients;
void handle_clients()
{
    for ( int i = 0; i < clients.size(); ++i)
        async_read_until(clients[i].sock, clients[i].buff, '\n',
    boost::bind(on_read, clients[i], _1, _2));
    for ( int i = 0; i < 10; ++i)
        boost::thread(handle_clients_thread);
}
void handle_clients_thread()
{
    service.run();
}
void on_read(client & c, const error_code & err, size_t read_bytes)
{
    std::istream in(&c.buff);
    std::string msg;
    std::getline(in, msg);
    if ( msg == "request_login")
        c.sock.async_write( "request_ok\n", on_write);
    else if ...
    ...
    // 等待同一个客户端下一个读取操作
    async_read_until(c.sock, c.buff, '\n', boost::bind(on_read, c, _1, _2));
}
```

发现代码变得有多简单了吧？client 结构里面只有两个成员，*handle_clients()*仅仅调用了 *async_read_until*，然后它创建了 10 个线程，每个线程都调用 *service.run()*。这些线程会处理所有来自客户端的异步 read 操作，然后分发所有向客户端的异步 write 操作。另外需要注意的一件事情是：*on_read()*一直在为下一次异步 read 操作做准备（看最后一行代码）。

异步 run(), run*one(), poll(), poll* one()

为了实现监听循环，*io_service* 类提供了 4 个方法，比如：*run(), run_one(), poll()* 和 *poll_one()*。虽然大多数时候使用 *service.run()*就可以，但是你还是需要在这里学习其他方法实现的功能。

持续运行

再一次说明，如果有等待执行的操作，*run()*会一直执行，直到你手动调用 *io_service::stop()*。为了保证 *io_service* 一直执行，通常你添加一个或者多个异步操作，然后在它们被执行时，你继续一直不停地添加异步操作，比如下面代码：

```
using namespace boost::asio;
io_service service;
ip::tcp::socket sock(service);
char buff_read[1024], buff_write[1024] = "ok";
void on_read(const boost::system::error_code &err, std::size_t bytes);
void on_write(const boost::system::error_code &err, std::size_t bytes)
{
    sock.async_read_some(buffer(buff_read), on_read);
}
void on_read(const boost::system::error_code &err, std::size_t bytes)
{
    // ... 处理读取操作 ...
    sock.async_write_some(buffer(buff_write,3), on_write);
}
void on_connect(const boost::system::error_code &err)
{
    sock.async_read_some(buffer(buff_read), on_read);
}
int main(int argc, char* argv[])
{
    ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 2001);
    sock.async_connect(ep, on_connect);
    service.run();
}
```

1. 当 *service.run()*被调用时，有一个异步操作在等待。
2. 当 socket 连接到服务端时，*on_connect* 被调用了，它会添加一个异步操作。
3. 当 *on_connect* 结束时，我们会留下一个等待的操作（*read*）。
4. 当 *on_read* 被调用时，我们写入一个回应，这又添加了另外一个等待的操作。

5. 当 *on_read* 结束时，我们会留下一个等待的操作（*write*）。
6. 当 *on_write* 操作被调用时，我们从服务端读取另外一个消息，这也添加了另外一个等待的操作。
7. 当 *on_write* 结束时，我们有一个等待的操作（read）。
8. 然后一直继续循环下去，直到我们关闭这个应用。

run_one(), poll(), poll_one() 方法

我在之前说过异步方法的 handler 是在调用了 *io_service::run* 的线程里被调用的。因为在至少 90%～95%的时候，这是你唯一要用到的方法，所以我就把它说得简单了。对于调用了 *run_one(), poll()*或者 *poll_one()*的线程这一点也是适用的。

*run_one()*方法最多执行和分发一个异步操作：

- 如果没有等待的操作，方法立即返回 0
- 如果有等待操作，方法在第一个操作执行之前处于阻塞状态，然后返回 1

你可以认为下面两段代码是等效的：

```
io_service service;
service.run(); // 或者
while ( !service.stopped()) service.run_once();
```

你可以使用 *run_once()*启动一个异步操作，然后等待它执行完成。

```
io_service service;
bool write_complete = false;
void on_write(const boost::system::error_code & err, size_t bytes)
{ write_complete = true; }
 …
std::string data = "login ok”;
write_complete = false;
async_write(sock, buffer(data), on_write);
do service.run_once() while (!write_complete);
```

还有一些使用 *run_one()*方法的例子，包含在 Boost.Asio 诸如 *blocking_tcp_client.cpp* 和 *blocking_udp_client.cpp* 的文件中。

poll_one 方法以非阻塞的方式最多运行一个准备好的等待操作：

- 如果至少有一个等待的操作，而且准备好以非阻塞的方式运行，poll_one 方法会运行它并且返回 1
- 否则，方法立即返回 0

操作正在等待并准备以非阻塞方式运行，通常意味着如下的情况：

- 一个计时器过期了，然后它的 *async_wait* 处理方法需要被调用
- 一个 I/O 操作完成了（比如 *async_read*），然后它的 hanlder 需要被调用
- 之前被加入 *io_services* 实例队列中的自定义 handler（这会在之后的章节中详解）

你可以使用 *poll_one* 去保证所有 I/O 操作的 handler 完成运行，同时做一些其他的工作

```
io_service service;
while ( true)
{
    // 运行所有完成了 IO 操作的 handler
    while ( service.poll_one()) ;
    // ... 在这里做其他的事情 …
}
```

*poll()*方法会以非阻塞的方式运行所有等待的操作。下面两段代码是等效的：

```
io_service service;
service.poll(); // 或者
while ( service.poll_one()) ;
```

所有上述方法都会在失败的时候抛出 *boost::system::system_error* 异常。这是我们所不希望发生的事情；这里抛出的异常通常都是致命的，也许是资源耗尽，或者是你 handler 的其中一个抛出了异常。另外，每个方法都有一个不抛出异常，而是返回一个 *boost::system::error_code* 的重载：

```
io_service service;
boost::system::error_code err = 0;
service.run(err);
if ( err) std::cout << "Error " << err << std::endl;
```

异步工作

异步工作不仅仅指用异步地方式接受客户端到服务端的连接、异步地从一个 socket 读取或者写入到 socket。它包含了所有可以异步执行的操作。

默认情况下，你是不知道每个异步 handler 的调用顺序的。除了通常的异步调用（来自异步 socket 的读取/写入/接收）。你可以使用 *service.post()*来使你的自定义方法被异步地调用。例如：

```
#include <boost/thread.hpp>
```

```
#include <boost/bind.hpp>
#include <boost/asio.hpp>
#include <iostream>
using namespace boost::asio;
io_service service;
void func(int i)
{
    std::cout << "func called, i= " << i << std::endl;
}

void worker_thread()
{
    service.run();
}

int main(int argc, char* argv[])
{
    for ( int i = 0; i < 10; ++i)
        service.post(boost::bind(func, i));
    boost::thread_group threads;
    for ( int i = 0; i < 3; ++i)
        threads.create_thread(worker_thread);
    // 等待所有线程被创建完
    boost::this_thread::sleep( boost::posix_time::millisec(500));
    threads.join_all();
}
```

在上面的例子中，*service.post(some_function)*添加了一个异步方法调用。

这个方法在某一个调用了 *service.run()*的线程中请求 *io_service* 实例，然后调用给定的 *some_funtion* 之后立即返回。在我们的例子中，这个线程是我们之前创建的三个线程中的一个。你不能确定异步方法调用的顺序。你不要期待它们会以我们调用 *post()*方法的顺序来调用。下面是运行之前代码可能得到的结果：

```
func called, i= 0
func called, i= 2
func called, i= 1
func called, i= 4
func called, i= 3
func called, i= 6
func called, i= 7
func called, i= 8
func called, i= 5
func called, i= 9
```

有时候你会想让一些异步处理方法顺序执行。比如，你去一个餐馆（*go_to_restaurant*），下单（*order*），然后吃（*eat*）。你需要先去餐馆，然后下单，最后吃。这样的话，你需要用到 *io_service::strand*，这个方法会让你的异步方法被顺序调用。看下面的例子：

```
using namespace boost::asio;
io_service service;
void func(int i)
{
    std::cout << "func called, i= " << i << "/" << boost::this_thread::get_id()
<< std::endl;
}
void worker_thread()
{
    service.run();
}
int main(int argc, char* argv[])
{
    io_service::strand strand_one(service), strand_two(service);
    for ( int i = 0; i < 5; ++i)
        service.post( strand_one.wrap( boost::bind(func, i)));
    for ( int i = 5; i < 10; ++i)
        service.post( strand_two.wrap( boost::bind(func, i)));
    boost::thread_group threads;
    for ( int i = 0; i < 3; ++i)
        threads.create_thread(worker_thread);
    // 等待所有线程被创建完
    boost::this_thread::sleep( boost::posix_time::millisec(500));
    threads.join_all();
}
```

在上述代码中，我们保证前面的 5 个线程和后面的 5 个线程是顺序执行的。*func called, i = 0* 在 *func called, i = 1* 之前被调用，然后调用 *func called, i = 2*……同样 *func called, i = 5* 在 *func called, i = 6* 之前，*func called, i = 6* 在 *func called, i = 7* 被调用……你需要注意的是尽管方法是顺序调用的，但是不意味着它们都在同一个线程执行。运行这个程序可能得到的一个结果如下：

```
func called, i= 0/002A60C8
func called, i= 5/002A6138
func called, i= 6/002A6530
func called, i= 1/002A6138
func called, i= 7/002A6530
func called, i= 2/002A6138
func called, i= 8/002A6530
```

```
func called, i= 3/002A6138
func called, i= 9/002A6530
func called, i= 4/002A6138
```

异步 post() VS dispatch() VS wrap()

Boost.Asio 提供了三种让你把处理方法添加为异步调用的方式：

- *service.post(handler)*：这个方法能确保其在请求 *io_service* 实例，然后调用指定的处理方法之后立即返回。handler 稍后会在某个调用了 *service.run()*的线程中被调用。
- *service.dispatch(handler)*：这个方法请求 *io_service* 实例去调用给定的处理方法，但是另外一点，如果当前的线程调用了 *service.run()*，它可以在方法中直接调用 handler。
- *service.wrap(handler)*：这个方法创建了一个封装方法，当被调用时它会调用 *service.dispatch(handler)*，这个会让人有点困惑，接下来我会简单地解释它是什么意思。

在之前的章节中你会看到关于 *service.post()*的一个例子，以及运行这个例子可能得到的一种结果。我们对它做一些修改，然后看看 *service.dispatch()*是怎么影响输出的结果的：

```
using namespace boost::asio;
io_service service;
void func(int i)
{
    std::cout << "func called, i= " << i << std::endl;
}
void run_dispatch_and_post()
{
    for ( int i = 0; i < 10; i += 2) {
    service.dispatch(boost::bind(func, i));
    service.post(boost::bind(func, i + 1));
    }
}
int main(int argc, char* argv[])
{
    service.post(run_dispatch_and_post);
    service.run();
}
```

在解释发生了什么之前，我们先运行程序，观察结果：

```
func called, i= 0
func called, i= 2
func called, i= 4
func called, i= 6
func called, i= 8
func called, i= 1
func called, i= 3
func called, i= 5
func called, i= 7
func called, i= 9
```

偶数先输出，然后是奇数。这是因为我用 *dispatch()*输出偶数，然后用 *post()*输出奇数。*dispatch()*会在返回之前调用 hanlder，因为当前的线程调用了 *service.run()*，而 *post()*每次都立即返回了。 现在，让我们讲讲 *service.wrap(handler)*。*wrap()*返回了一个仿函数，它可以用来做另外一个方法的参数：

```
using namespace boost::asio;
io_service service;
void dispatched_func_1()
{
    std::cout << "dispatched 1" << std::endl;
}
void dispatched_func_2()
{
    std::cout << "dispatched 2" << std::endl;
}
void test(boost::function<void()> func)
{
    std::cout << "test" << std::endl;
    service.dispatch(dispatched_func_1);
    func();
}
void service_run()
{
    service.run();
}
int main(int argc, char* argv[])
{
    test( service.wrap(dispatched_func_2));
    boost::thread th(service_run);
    boost::this_thread::sleep( boost::posix_time::millisec(500));
    th.join();
}
```

*test(service.wrap(dispatched_func_2));*会把 *dispatched_ func_2* 包装起来创建一个仿函数，然后传递给 *test* 当作一个参数。当 *test()*被调用时，它会分发调用方法 1，然后调用 *func()*。这时，你会发现调用 *func()*和 *service.dispatch(dispatched_func_2)*是等价的，因为它们是连续调用的。程序的输出证明了这一点：

```
test
dispatched 1
dispatched 2
```

io_service::strand 类（用来序列化异步调用）也包含了 *poll(), dispatch()*和 *wrap()* 等成员函数。它们的作用和 *io_service* 的 *poll(), dispatch()*和 *wrap()*是一样的。然而，大多数情况下你只需要把 *io_service::strand::wrap()*方法做为 *io_service::poll()*或者 *io_service::dispatch()*方法的参数即可。

保持活动

假设你需要做下面的操作：

```
io_service service;
ip::tcp::socket sock(service);
char buff[512];
...
read(sock, buffer(buff));
```

在这个例子中，*sock* 和 *buff* 的存在时间都必须比 *read()*调用的时间要长。也就是说，在调用 *read()*返回之前，它们都必须有效。这就是你所期望的；你传给一个方法的所有参数在方法内部都必须有效。当我们采用异步方式时，事情会变得比较复杂。

```
io_service service;
ip::tcp::socket sock(service);
char buff[512];
void on_read(const boost::system::error_code &, size_t) {}
...
async_read(sock, buffer(buff), on_read);
```

在这个例子中，*sock* 和 *buff* 的存在时间都必须比 *read()*操作本身时间要长，但是 read 操作持续的时间我们是不知道的，因为它是异步的。

当使用 socket 缓冲区的时候，你会有一个 *buffer* 实例在异步调用时一直存在（使用 *boost::shared_array<>*）。在这里，我们可以使用同样的方式，通过创建一

个类并在其内部管理 socket 和它的读写缓冲区。然后，对于所有的异步操作，传递一个包含智能指针的 *boost::bind* 仿函数给它：

```
using namespace boost::asio;
io_service service;
struct connection : boost::enable_shared_from_this<connection>
{
    typedef boost::system::error_code error_code;
    typedef boost::shared_ptr<connection> ptr;
    connection() : sock_(service), started_(true) {}
    void start(ip::tcp::endpoint ep)
    {
        sock_.async_connect(ep, boost::bind(&connection::on_connect,
            shared_from_this(), _1));
    }
    void stop()
    {
        if ( !started_) return;
        started_ = false;
        sock_.close();
    }
    bool started() { return started_; }
private:
    void on_connect(const error_code & err)
    {
        // 这里你决定用这个连接做什么：读取或者写入
        if ( !err) do_read();
        else stop();
    }
    void on_read(const error_code & err, size_t bytes)
    {
        if ( !started() ) return;
        std::string msg(read_buffer_, bytes);
        if ( msg == "can_login") do_write("access_data");
        else if ( msg.find("data ") == 0) process_data(msg);
        else if ( msg == "login_fail") stop();
    }
    void on_write(const error_code & err, size_t bytes)
    {
        do_read();
    }
    void do_read()
    {
```

```
        sock_.async_read_some(buffer(read_buffer_),
        boost::bind(&connection::on_read, shared_from_this(),   _1, _2));
    }
    void do_write(const std::string & msg)
    {
        if ( !started() ) return;
        // 注意: 因为在做另外一个 async_read 操作之前你想要发送多个消息,
        // 所以你需要多个写入 buffer
        std::copy(msg.begin(), msg.end(), write_buffer_);
        sock_.async_write_some(buffer(write_buffer_, msg.size()),
        boost::bind(&connection::on_write, shared_from_this(), _1, _2));
    }

    void process_data(const std::string & msg)
    {
        // 处理服务端来的内容，然后启动另外一个写入操作
    }
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
    bool started_;
};

int main(int argc, char* argv[])
{
    ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
    connection::ptr(new connection)->start(ep);
}
```

在所有异步调用中，我们传递一个 *boost::bind* 仿函数当作参数。这个仿函数内部包含了一个智能指针，指向 *connection* 实例。只要有一个异步操作等待时，Boost.Asio 就会保存 *boost::bind* 仿函数的拷贝，这个拷贝保存了指向连接实例的一个智能指针，从而保证 *connection* 实例保持活动。问题解决！

当然，*connection* 类仅仅是一个框架类；你需要根据你的需求对它进行调整（它看起来会和当前服务端例子的情况相当不同）。

你需要注意的是创建一个新的连接是相当简单的：*connection::ptr(new connection)- >start(ep)*。这个方法启动了到服务端的（异步）连接。当你需要关闭这个连接时，调用 *stop()*。

当实例被启动时（*start()*），它会等待客户端的连接。当连接发生时。*on_connect()* 被调用。如果没有错误发生，它启动一个 read 操作（*do_read()*）。当 read 操作结束时，你就可以解析这个消息；当然你应用的 *on_read()*看起来会各种各样。而当你写回一个消息时，你需要把它拷贝到缓冲区，然后像我在 *do_write()*方法中所做的一样将其发送出去，因为这个缓冲区同样需要在这个异步写操作中一直存活。最后需要注意的一点——当写回时，你需要指定写入的数量，否则，整个缓冲区都会被发送出去。

总结

网络 api 实际上要繁杂得多，这个章节只是做为一个参考，当你在实现自己的网络应用时可以回过头来看看。

Boost.Asio 实现了端点的概念，你可以认为是 IP 和端口。如果你不知道准确的 IP，你可以使用 *resolver* 对象将主机名，例如 *www.yahoo.com* 转换为一个或多个 IP 地址。

我们也可以看到 API 的核心——socket 类。Boost.Asio 提供了 *TCP*、*UDP* 和 *ICMP* 的实现。而且你还可以用你自己的协议来对它进行扩展；当然，这个工作不适合缺乏勇气的人。

异步编程是刚需。你应该已经明白为什么有时候需要用到它，尤其在写服务端的时候。调用 *service.run()*来实现异步循环就已经可以让你很满足，但是有时候你需要更进一步，尝试使用 *run_one()*、*poll()*或者 *poll_one()*。

当实现异步时，你可以异步执行你自己的方法；使用 *service.post()*或者 *service.dispatch()*。

最后，为了使 socket 和缓冲区（read 或者 write）在整个异步操作的生命周期中一直活动，我们需要采取特殊的防护措施。你的连接类需要继承自 *enabled_shared_from_this*，然后在内部保存它需要的缓冲区，而且每次异步调用都要传递一个智能指针给 *this* 操作。

下一章会进行实战操作；在实现回显客户端/服务端应用时会有大量的编程实践。

第三章

回显服务端/客户端

在这一章，我们将会实现一个小的客户端/服务端应用，这可能会是你写过的最简单的客户端/服务端应用。回显应用就是一个把客户端发过来的任何内容回显给其本身，然后关闭连接的的服务端。这个服务端可以处理任何数量的客户端。每个客户端连接之后发送一个消息，服务端接收到完成消息后把它发送回去。在那之后，服务端关闭连接。

因此，每个回显客户端连接到服务端，发送一个消息，然后读取服务端返回的结果，确保这是它发送给服务端的消息就结束和服务端的会话。

我们首先实现一个同步应用，然后实现一个异步应用，以便你可以很容易对比他们：

为了节省空间，下面的代码有一些被裁剪掉了。你可以在附加在这本书的代码中看到全部的代码。

TCP 回显服务端/客户端

对于 TCP 而言，我们需要一个额外的保证；每一个消息以换行符结束('\n')。编写一个同步回显服务端/客户端非常简单。

我们会展示编码内容，比如同步客户端，同步服务端，异步客户端和异步服务端。

TCP 同步客户端

在大多数有价值的例子中，客户端通常比服务端编码要简单（因为服务端需要处理多个客户端请求）。 下面的代码展示了不符合这条规则的一个例外：

```
size_t read_complete(char * buf, const error_code & err, size_t bytes)
{
    if ( err) return 0;
    bool found = std::find(buf, buf + bytes, '\n') < buf + bytes;
    // 我们一个一个读取直到读到回车，不缓存
    return found ? 0 : 1;
```

```
}
void sync_echo(std::string msg)
{
    msg += "\n”;
    ip::tcp::socket sock(service);
    sock.connect(ep);
    sock.write_some(buffer(msg));
    char buf[1024];
    int bytes = read(sock, buffer(buf), boost::bind(read_complete,buf,_1,_2));
    std::string copy(buf, bytes - 1);
    msg = msg.substr(0, msg.size() - 1);
    std::cout << "server echoed our " << msg << ": "<< (copy == msg ? "OK" :
        "FAIL") << std::endl;
    sock.close();
}
int main(int argc, char* argv[])
{
    char* messages[] = { "John says hi", "so does James", "Lucy just got home",
        "Boost.Asio is Fun!", 0 };
    boost::thread_group threads;
    for ( char ** message = messages; *message; ++message)
    {
        threads.create_thread( boost::bind(sync_echo, *message));
        boost::this_thread::sleep( boost::posix_time::millisec(100));
    }
    threads.join_all();
}
```

核心功能 *sync_echo*。它包含了连接到服务端，发送信息然后等待回显的所有逻辑。

你会发现，在读取时，我使用了自由函数 *read()*，因为我想要读'\n'之前的所有内容。*sock.read_some()*方法满足不了这个要求，因为它只会读可用的，而不是全部的消息。

*read()*方法的第三个参数是完成处理句柄。当读取到完整消息时，它返回 0。否则，它会返回我下一步（直到读取结束）能都到的最大的缓冲区大小。在我们的例子中，返回结果始终是 1，因为我永远不想读的消息比我们需要的更多。

在 *main()*中，我们创建了几个线程；每个线程负责把消息发送到客户端，然后等待操作结束。如果你运行这个程序，你会看到下面的输出：

```
server echoed our John says hi: OK
server echoed our so does James: OK
```

```
server echoed our Lucy just got home: OK
server echoed our Boost.Asio is Fun!: OK
```

注意：因为我们是同步的，所以不需要调用 *service.run()*。

TCP 同步服务端

回显同步服务端的编写非常容易，参考如下的代码片段：

```
io_service service;
size_t read_complete(char * buff, const error_code & err, size_t bytes)
{
    if ( err) return 0;
    bool found = std::find(buff, buff + bytes, '\n') < buff + bytes;
    // 我们一个一个读取直到读到回车，不缓存
    return found ? 0 : 1;
}
void handle_connections()
{
    ip::tcp::acceptor acceptor(service,
        ip::tcp::endpoint(ip::tcp::v4(),8001));
    char buff[1024];
    while ( true)
    {
        ip::tcp::socket sock(service);
        acceptor.accept(sock);
        int bytes = read(sock, buffer(buff),
        boost::bind(read_complete,buff,_1,_2));
        std::string msg(buff, bytes);
        sock.write_some(buffer(msg));
        sock.close();
    }
}
int main(int argc, char* argv[])
{
    handle_connections();
}
```

服务端的逻辑主要在 *handle_connections()*。因为是单线程，我们接受一个客户端请求，读取它发送给我们的消息，然后回显，然后等待下一个连接。可以确定，当两个客户端同时连接时，第二个客户端需要等待服务端处理完第一个客户端的请求。

还是要注意因为我们是同步的，所以不需要调用 *service.run()*。

TCP 异步客户端

当我们开始异步时，编码会变得稍微有点复杂。我们会构建在**第二章 保持活动**中展示的 *connection* 类。

观察这个章节中接下来的代码，你会发现每个异步操作启动了新的异步操作，以保持 *service.run()*一直工作。 首先，核心功能如下：

```
#define MEM_FN(x)       boost::bind(&self_type::x, shared_from_this())
#define MEM_FN1(x,y)    boost::bind(&self_type::x, shared_from_this(),y)
#define MEM_FN2(x,y,z)  boost::bind(&self_type::x, shared_from_this(),y,z)
class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr> ,
    boost::noncopyable
{
    typedef talk_to_svr self_type;
    talk_to_svr(const std::string & message) : sock_(service), started_(true),
        message_(message) {}
    void start(ip::tcp::endpoint ep)
    {
        sock_.async_connect(ep, MEM_FN1(on_connect,_1));
    }
public:
    typedef boost::system::error_code error_code;
    typedef boost::shared_ptr<talk_to_svr> ptr;
    static ptr start(ip::tcp::endpoint ep, const std::string &message)
    {
        ptr new_(new talk_to_svr(message));
        new_->start(ep);
        return new_;
    }
    void stop()
    {
        if ( !started_) return;
        started_ = false;
        sock_.close();
    }
    bool started() { return started_; }
    ...
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
```

```
    bool started_;
    std::string message_;
};
```

我们需要一直使用指向 *talk_to_svr* 的智能指针，这样的话当在 *tack_to_svr* 的实例上有异步操作时，那个实例是一直活动的。为了避免错误，比如在栈上构建一个 *talk_to_svr* 对象的实例时，我把构造方法设置成了私有而且不允许拷贝构造（继承自 *boost::noncopyable*）。

我们有了核心方法，比如 *start(),stop()*和 *started()*，它们所做的事情也正如它们名字表达的一样。如果需要建立连接，调用 *talk_to_svr::start(endpoint, message)* 即可。我们同时还有一个 read 缓冲区和一个 write 缓冲区。（*readbuufer* 和 *writebuffer*）。

MEM_FN 是一个方便使用的宏，它们通过 shared_ptr_from_this() *方法强制使用一个指向 this* 的智能指针。

下面的几行代码和之前的解释非常不同：

```
//等同于 "sock_.async_connect(ep, MEM_FN1(on_connect,_1));"
sock_.async_connect(ep,boost::bind(&talk_to_svr::on_connect,shared_ptr_from
    _this(),_1));
sock_.async_connect(ep, boost::bind(&talk_to_svr::on_connect,this,_1));
```

在上述例子中，我们正确的创建了 *async_connect* 的完成处理句柄；在调用完成处理句柄之前它会保留一个指向 *talk_to_server* 实例的智能指针，从而保证当其发生时 *talk_to_server* 实例还是保持活动的。

在接下来的例子中，我们错误地创建了完成处理句柄，当它被调用时，*talk_to_server* 实例很可能已经被释放了。 从 socket 读取或写入时，你使用如下的代码片段：

```
void do_read()
{
    async_read(sock_, buffer(read_buffer_), MEM_FN2(read_complete,_1,_2),
        MEM_FN2(on_read,_1,_2));
}
void do_write(const std::string & msg)
{
    if ( !started() ) return;
    std::copy(msg.begin(), msg.end(), write_buffer_);
    sock_.async_write_some( buffer(write_buffer_, msg.size()),
        MEM_FN2(on_write,_1,_2));
}
```

```
size_t read_complete(const boost::system::error_code & err, size_t bytes)
{
    // 和 TCP 客户端中的类似
}
```

*do_read()*方法会保证当 *on_read()*被调用的时候，我们从服务端读取一行。*do_write()*方法会先把信息拷贝到缓冲区（考虑到当 *async_write* 发生时 msg 可能已经超出范围被释放），然后保证实际的写入操作发生时 *on_write()*被调用。

然后是最重要的方法，这个方法包含了类的主要逻辑：

```
void on_connect(const error_code & err)
{
    if ( !err)      do_write(message_ + "\n");
    else           stop();
}
void on_read(const error_code & err, size_t bytes)
{
    if ( !err) {
        std::string copy(read_buffer_, bytes - 1);
        std::cout << "server echoed our " << message_ << ": " << (copy == message_ ?
            "OK" : "FAIL") << std::endl;
    }
    stop();
}
void on_write(const error_code & err, size_t bytes)
{
    do_read();
}
```

当连接成功之后，我们发送消息到服务端,*do_write()*。当 write 操作结束时，*on_write()*被调用，它初始化了一个 *do_read()*方法，当 *do_read()*完成时。*on_read()*被调用；这里，我们简单的检查一下返回的信息是否是服务端的回显，然后退出服务。 我们会发送三个消息到服务端让它变得更有趣一点：

```
int main(int argc, char* argv[])
{
    ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
    char* messages[] = { "John says hi", "so does James", "Lucy got home", 0 };
    for ( char ** message = messages; *message; ++message)
    {
        talk_to_svr::start( ep, *message);
        boost::this_thread::sleep( boost::posix_time::millisec(100));
    }
    service.run();
```

```
}
```

上述的代码会生成如下的输出：

```
server echoed our John says hi: OK
server echoed our so does James: OK
server echoed our Lucy just got home: OK
```

TCP 异步服务端

核心功能和同步服务端的功能类似，如下：

```
class talk_to_client : public boost::enable_shared_from_this<talk_to_
   client>, boost::noncopyable
{
    typedef talk_to_client self_type;
    talk_to_client() : sock_(service), started_(false) {}
public:
    typedef boost::system::error_code error_code;
    typedef boost::shared_ptr<talk_to_client> ptr;
    void start()
    {
        started_ = true;
        do_read();
    }

    static ptr new_()
    {
        ptr new_(new talk_to_client);
        return new_;
    }
    void stop()
    {
        if ( !started_) return;
        started_ = false;
        sock_.close();
    }
    ip::tcp::socket & sock() { return sock_;}
    ...
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
    bool started_;
```

```
};
```

因为我们是非常简单的回显服务，这里不需要 *is_started()*方法。对每个客户端，仅仅读取它的消息，回显，然后关闭它。

do_read()，*do_write()*和 *read_complete()*方法和 TCP 同步服务端的完全一致。主要的逻辑同样是在 *on_read()*和 *on_write()*方法中：

```
void on_read(const error_code & err, size_t bytes)
{
    if ( !err) {
        std::string msg(read_buffer_, bytes);
        do_write(msg + "\n");
    }
    stop();
}
void on_write(const error_code & err, size_t bytes)
{
    do_read();
}
```

对客户端的处理如下：

```
ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::v4(),8001));
void handle_accept(talk_to_client::ptr client, const error_code & err)
{
    client->start();
    talk_to_client::ptr new_client = talk_to_client::new_();
    acceptor.async_accept(new_client->sock(),
    boost::bind(handle_accept,new_client,_1));
}
int main(int argc, char* argv[])
{
    talk_to_client::ptr client = talk_to_client::new_();
    acceptor.async_accept(client->sock(),
    boost::bind(handle_accept,client,_1));
    service.run();
}
```

每一次客户端连接到服务时，*handle_accept* 被调用，它会异步地从客户端读取，然后同样异步地等待一个新的客户端。

代码

你会在这本书相应的代码中得到所有 4 个应用（TCP 回显同步客户端，TCP 回显同步服务端，TCP 回显异步客户端，TCP 回显异步服务端）。当测试时，你可以使用任意客户端/服务端组合（比如，一个异步客户端和一个同步服务端）。

UDP 回显服务端/客户端

因为 UDP 不能保证所有信息都抵达接收者，我们不能保证“信息以回车结尾”。没收到消息，我们只是回显，但是没有 socket 去关闭（在服务端），因为我们是 UDP。

UDP 同步回显客户端

UDP 回显客户端比 TCP 回显客户端要简单：

```
ip::udp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
void sync_echo(std::string msg)
{
    ip::udp::socket sock(service, ip::udp::endpoint(ip::udp::v4(), 0));
    sock.send_to(buffer(msg), ep);
    char buff[1024];
    ip::udp::endpoint sender_ep;
    int bytes = sock.receive_from(buffer(buff), sender_ep);
    std::string copy(buff, bytes);
    std::cout << "server echoed our " << msg << ": " << (copy == msg ? "OK" :
        "FAIL") << std::endl;
    sock.close();
}
int main(int argc, char* argv[])
{
    char* messages[] = { "John says hi", "so does James", "Lucy got home", 0 };
    boost::thread_group threads;
    for ( char ** message = messages; *message; ++message)
    {
        threads.create_thread( boost::bind(sync_echo, *message));
        boost::this_thread::sleep( boost::posix_time::millisec(100));
    }
    threads.join_all();
}
```

所有的逻辑都在 *synch_echo()*中；连接到服务端，发送消息，接收服务端的回显，然后关闭连接。

UDP 同步回显服务端

UDP 回显服务端会是你写过的最简单的服务端：

```
io_service service;
void handle_connections()
{
    char buff[1024];
    ip::udp::socket sock(service, ip::udp::endpoint(ip::udp::v4(), 8001));
    while ( true)
    {
        ip::udp::endpoint sender_ep;
        int bytes = sock.receive_from(buffer(buff), sender_ep);
        std::string msg(buff, bytes);
        sock.send_to(buffer(msg), sender_ep);
    }
}
int main(int argc, char* argv[])
{
    handle_connections();
}
```

它非常简单，而且能很好的自释。 我把异步 UDP 客户端和服务端留给读者当作一个练习。

总结

我们已经写了完整的应用，最终让 Boost.Asio 得以工作。回显应用是开始学习一个库时非常好的工具。你可以经常学习和运行这个章节所展示的代码，这样你就可以非常容易地记住这个库的基础。 在下一章，我们会建立更复杂的客户端/服务端应用，我们要确保避免低级错误，比如内存泄漏，死锁等等。

第四章

客户端和服务端

在这一章节，我们会深入学习怎样使用 Boost.Asio 建立非凡的客户端和服务端应用。你可以运行并测试它们，而且在理解之后，你可以把它们做为框架来构造自己的应用。

在接下来的例子中：

- 客户端使用一个用户名（无密码）登录到服务端
- 所有的连接由客户端建立，当客户端请求时服务端回应
- 所有的请求和回复都以换行符结尾（'\n'）
- 对于 5 秒钟没有 ping 操作的客户端，服务端会自动断开其连接

客户端可以发送如下请求：

- 获得所有已连接客户端的列表
- 客户端可以 ping，当它 ping 时，服务端返回 *ping ok* 或者 *ping client_list_chaned*（在接下来的例子中，客户端重新请求已连接的客户端列表）

为了更有趣一点，我们增加了一些难度：

- 每个客户端登录 6 个用户连接，比如 Johon,James,Lucy,Tracy,Frank 和 Abby
- 每个客户端连接随机地 ping 服务端（随机 7 秒；这样的话，服务端会时不时关闭一个连接）

同步客户端/服务端

首先，我们会实现同步应用。你会发现它的代码很直接而且易读的。而且因为所有的网络调用都是阻塞的，所以它不需要独立的线程。

同步客户端

同步客户端会以你所期望的串行方式运行；连接到服务端，登录服务器，然后执行连接循环，比如休眠一下，发起一个请求，读取服务端返回，然后再休眠一会，然后一直循环下去……

因为我们是同步的，所以我们让事情变得简单一点。首先，连接到服务器，然后再循环，如下：

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
void run_client(const std::string & client_name)
{
    talk_to_svr client(client_name);
    try
    {
        client.connect(ep);
        client.loop();
    } catch(boost::system::system_error & err)
    {
        std::cout << "client terminated " << std::endl;
    }
}
```

下面的代码片段展示了 talk_to_svr 类：

```
struct talk_to_svr
{
    talk_to_svr(const std::string & username) : sock_(service), started_(true),
        username_(username) {}
    void connect(ip::tcp::endpoint ep)
    {
        sock_.connect(ep);
    }
    void loop()
    {
        write("login " + username_ + "\n");
        read_answer();
        while ( started_)
        {
            write_request();
            read_answer();
            boost::this_thread::sleep(millisec(rand() % 7000));
        }
    }
```

```
    std::string username() const { return username_; }
    ...
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    int already_read_;
    char buff_[max_msg];
    bool started_;
    std::string username_;
};
```

在这个循环中，我们仅仅填充 1 个比特，做一个 ping 操作之后就进入睡眠状态，之后再读取服务端的返回。我们的睡眠是随机的（有时候超过 5 秒），这样服务端就有可能在某个时间点断开我们的连接：

```
void write_request()
{
    write("ping\n");
}
void read_answer()
{
    already_read_ = 0;
    read(sock_, buffer(buff_), boost::bind(&talk_to_svr::read_complete, this,
        _1, _2));
    process_msg();
}
void process_msg()
{
    std::string msg(buff_, already_read_);
    if ( msg.find("login ") == 0) on_login();
    else if ( msg.find("ping") == 0) on_ping(msg);
    else if ( msg.find("clients ") == 0) on_clients(msg);
    else std::cerr << "invalid msg " << msg << std::endl;
}
```

对于读取结果，我们使用在之前章节就有说到的 *read_complete* 来保证我们能读到换行符（'\n'）。这段逻辑在 *process_msg()*中，在这里我们读取服务端的返回，然后分发到正确的方法去处理：

```
void on_login() { do_ask_clients(); }
void on_ping(const std::string & msg)
{
    std::istringstream in(msg);
    std::string answer;
    in >> answer >> answer;
```

```
    if ( answer == "client_list_changed")
        do_ask_clients();
}
void on_clients(const std::string & msg)
{
    std::string clients = msg.substr(8);
    std::cout << username_ << ", new client list:" << clients;
}
void do_ask_clients()
{
    write("ask_clients\n");
    read_answer();
}
void write(const std::string & msg) { sock_.write_some(buffer(msg)); }
size_t read_complete(const boost::system::error_code & err, size_t bytes)
{
    // ... 和之前一样
}
```

在读取服务端对我们 ping 操作的返回时，如果得到的消息是 *client_list_changed*，我们就需要重新请求客户端列表。

同步服务端

同步服务端也是相当简单的。它只需要两个线程，一个负责接收新的客户端连接，另外一个负责处理已经存在的客户端请求。它不能使用单线程，因为等待新的客户端连接是一个阻塞操作，所以我们需要另外一个线程来处理已经存在的客户端请求。

正常来说服务端都比客户端要难实现。一方面，它要管理所有已经连接的客户端。因为我们是同步的，所以我们需要至少两个线程，一个负责接受新的客户端连接（因为 accept()是阻塞的）而另一个负责回复已经存在的客户端。

```
void accept_thread()
{
    ip::tcp::acceptor acceptor(service,ip::tcp::endpoint(ip::tcp::v4(),
        8001));
    while ( true)
    {
        client_ptr new_( new talk_to_client);
        acceptor.accept(new_->sock());
```

```
        boost::recursive_mutex::scoped_lock lk(cs);
        clients.push_back(new_);
    }
}

void handle_clients_thread()
{
    while ( true)
    {
        boost::this_thread::sleep( millisec(1));
        boost::recursive_mutex::scoped_lock lk(cs);
        for(array::iterator b = clients.begin(), e = clients.end(); b!= e; ++b)
            (*b)->answer_to_client();
        // 删除已经超时的客户端
        clients.erase(std::remove_if(clients.begin(), clients.end(),
        boost::bind(&talk_to_client::timed_out,_1)), clients.end());
    }
}
int main(int argc, char* argv[])
{
    boost::thread_group threads;
    threads.create_thread(accept_thread);
    threads.create_thread(handle_clients_thread);
    threads.join_all();
}
```

为了分辨客户端发送过来的请求我们需要保存一个客户端的列表。 每个 *talk_to_client* 实例都拥有一个 socket，socket 类是不支持拷贝构造的，所以如果你想要把它们保存在一个 *std::vector* 对象中，你需要一个指向它的智能指针。这里有两种实现的方式：在 *talk_to_client* 内部保存一个指向 socket 的智能指针然后创建一个 *talk_to_client* 实例的数组，或者让 *talk_to_client* 实例用变量的方式保存 socket，然后创建一个指向 *talk_to_client* 智能指针的数组。我选择后者，但是你也可以选前面的方式：

```
typedef boost::shared_ptr<talk_to_client> client_ptr;
typedef std::vector<client_ptr> array;
array clients;
boost::recursive_mutex cs; // 用线程安全的方式访问客户端数组
```

talk_to_client 的主要代码如下：

```
struct talk_to_client : boost::enable_shared_from_this<talk_to_client>
{
    talk_to_client() { ... }
    std::string username() const { return username_; }
```

```
    void answer_to_client()
    {
        try
        {
            read_request();
            process_request();
        } catch ( boost::system::system_error&)
        { stop(); }

        if ( timed_out())
            stop();
    }
    void set_clients_changed() { clients_changed_ = true; }
    ip::tcp::socket & sock() { return sock_; }
    bool timed_out() const
    {
        ptime now = microsec_clock::local_time();
        long long ms = (now - last_ping).total_milliseconds();
        return ms > 5000 ;
    }
    void stop()
    {
        boost::system::error_code err; sock_.close(err);
    }
    void read_request()
    {
        if ( sock_.available())
            already_read_ += sock_.read_some(buffer(buff_ + already_read_,
        max_msg - already_read_));
    }
...
private:
    // ...  和同步客户端中的一样
    bool clients_changed_;
    ptime last_ping;
};
```

上述代码拥有非常好的自释能力。其中最重要的方法是 *read_request()*。它只在存在有效数据的情况才读取，这样的话，服务端永远都不会阻塞：

```
void process_request()
{
    bool found_enter = std::find(buff_, buff_ + already_read_, '\n') < buff_
        + already_read_;
    if ( !found_enter)
```

```
        return; // 消息不完整
        // 处理消息
    last_ping = microsec_clock::local_time();
    size_t pos = std::find(buff_, buff_ + already_read_, '\n') - buff_;
    std::string msg(buff_, pos);
    std::copy(buff_ + already_read_, buff_ + max_msg, buff_);
    already_read_ -= pos + 1;
    if ( msg.find("login ") == 0) on_login(msg);
    else if ( msg.find("ping") == 0) on_ping();
    else if ( msg.find("ask_clients") == 0) on_clients();
    else std::cerr << "invalid msg " << msg << std::endl;
}
void on_login(const std::string & msg)
{
    std::istringstream in(msg);
    in >> username_ >> username_;
    write("login ok\n");
    update_clients_changed();
}
void on_ping()
{
    write(clients_changed_ ? "ping client_list_changed\n" : "ping ok\n");
    clients_changed_ = false;
}
void on_clients()
{
    std::string msg;
    { boost::recursive_mutex::scoped_lock lk(cs);
        for( array::const_iterator b = clients.begin(), e = clients.end() ; b !=
            e; ++b)
            msg += (*b)->username() + " ";
    }
    write("clients " + msg + "\n");
}
void write(const std::string & msg){sock_.write_some(buffer(msg)); }
```

观察 *process_request()*。当我们读取到足够多有效的数据时，我们需要知道我们是否已经读取到整个消息(如果 *found_enter* 为真)。这样做的话，我们可以使我们避免一次读多个消息的可能（'\n'之后的消息也被保存到缓冲区中），然后我们解析读取到的整个消息。剩下的代码都是很容易读懂的。

异步客户端/服务端

现在，是比较有趣（也比较难）的异步实现！ 当查看示意图时，你需要知道 Boost.Asio 代表由 Boost.Asio 执行的一个异步调用。例如 *do_read()*，Boost.Asio 和 *on_read()*代表了从 *do_read()*到 *on_read()*的逻辑流程，但是你永远不知道什么时候轮到 *on_read()*被调用，你只是知道你最终会调用它。

异步客户端

到这里事情会变得有点复杂，但是仍然是可控的。当然你也会拥有一个不会阻塞的应用。

下面的代码你应该已经很熟悉：

```
#define MEM_FN(x)       boost::bind(&self_type::x, shared_from_this())
#define MEM_FN1(x,y)    boost::bind(&self_type::x, shared_from_
    this(),y)
#define MEM_FN2(x,y,z)  boost::bind(&self_type::x, shared_from_
    this(),y,z)
class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr>,
boost::noncopyable
{
    typedef talk_to_svr self_type;
    talk_to_svr(const std::string & username) : sock_(service), started_(true),
username_(username), timer_
(service) {}
    void start(ip::tcp::endpoint ep)
    {
        sock_.async_connect(ep, MEM_FN1(on_connect,_1));
    }
public:
    typedef boost::system::error_code error_code;
    typedef boost::shared_ptr<talk_to_svr> ptr;
    static ptr start(ip::tcp::endpoint ep, const std::string & username)
    {
        ptr new_(new talk_to_svr(username));
        new_->start(ep);
        return new_;
    }
    void stop()
    {
        if ( !started_) return;
        started_ = false;
```

```
        sock_.close();
    }
    bool started() { return started_; }
    ...
private:
    size_t read_complete(const boost::system::error_code &err, size_t bytes)
    {
        if ( err) return 0;
        bool found = std::find(read_buffer_, read_buffer_ + bytes, '\n') <
            read_buffer_ + bytes;
        return found ? 0 : 1;
    }
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
    bool started_;
    std::string username_;
    deadline_timer timer_;
};
```

你会看到额外还有一个叫 *deadlinetimer timer* 的方法用来 ping 服务端；而且 ping 操作同样是随机的。 下面是类的逻辑：

```
void on_connect(const error_code & err)
{
       if ( !err)      do_write("login " + username_ + "\n");
       else           stop();
}
void on_read(const error_code & err, size_t bytes)
{
    if ( err) stop();
    if ( !started() ) return;
    // 处理消息
    std::string msg(read_buffer_, bytes);
    if ( msg.find("login ") == 0) on_login();
    else if ( msg.find("ping") == 0) on_ping(msg);
    else if ( msg.find("clients ") == 0) on_clients(msg);
}
void on_login()
{
    do_ask_clients();
}
void on_ping(const std::string & msg)
```

```
{
    std::istringstream in(msg);
    std::string answer;
    in >> answer >> answer;
    if ( answer == "client_list_changed") do_ask_clients();
    else postpone_ping();
}
void on_clients(const std::string & msg)
{
    std::string clients = msg.substr(8);
    std::cout << username_ << ", new client list:" << clients ;
    postpone_ping();
}
```

在 *on_read()*中，首先的两行代码是亮点。在第一行，如果出现错误，我们就停止。而第二行，如果我们已经停止了（之前就停止了或者刚好停止），我们就返回。反之如果所有都是 OK，我们就对收到的消息进行处理。

最后是 *do_*方法，实现如下：

```
void do_ping() { do_write("ping\n"); }
void postpone_ping()
{
    timer_.expires_from_now(boost::posix_time::millisec(rand() % 7000));
    timer_.async_wait( MEM_FN(do_ping));
}
void do_ask_clients() { do_write("ask_clients\n"); }
void on_write(const error_code & err, size_t bytes) { do_read(); }

void do_read()
{
    async_read(sock_, buffer(read_buffer_), MEM_FN2(read_complete,_1,_2),
        MEM_FN2(on_read,_1,_2));
}
void do_write(const std::string & msg)
{
    if ( !started() ) return;
    std::copy(msg.begin(), msg.end(), write_buffer_);
    sock_.async_write_some( buffer(write_buffer_, msg.size()),
        MEM_FN2(on_write,_1,_2));
}
```

注意每一个 *read* 操作都会触发一个 ping 操作

- 当 *read* 操作结束时，*on_read()*被调用

- *on_read()*调用 *on_login()*，*on_ping()*或者 *on_clients()*
- 每一个方法要么发出一个 ping，要么请求客户端列表
- 如果我们请求客户端列表，当 *read* 操作接收到它们时，它会发出一个 ping 操作。

异步服务端

这个示意图是相当复杂的；从 Boost.Asio 出来你可以看到 4 个箭头指向 *on_accept*，*on_read*，*on_write* 和 *on_check_ping*。这也就意味着你永远不知道哪个异步调用是下一个完成的调用，但是你可以确定的是它是这 4 个操作中的一个。

现在，我们是异步的了；我们可以继续保持单线程。接受客户端连接是最简单的部分，如下所示：

```
ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::v4(), 8001));
void handle_accept(talk_to_client::ptr client, const error_code & err)
{
    client->start();
    talk_to_client::ptr new_client = talk_to_client::new_();
    acceptor.async_accept(new_client->sock(),
boost::bind(handle_accept,new_client,_1));
}
int main(int argc, char* argv[])
{
    talk_to_client::ptr client = talk_to_client::new_();
    acceptor.async_accept(client->sock(),boost::bind(handle_accept,client,
        _1));
    service.run();
}
```

上述代码会一直异步地等待一个新的客户端连接（每个新的客户端连接会触发另外一个异步等待操作）。 我们需要监控 *client list changed* 事件（一个新客户端连接或者一个客户端断开连接），然后当事件发生时通知所有的客户端。因此，我们需要保存一个客户端连接的数组，否则除非你不需要在某一时刻知道所有连接的客户端，你才不需要这样一个数组。

```
class talk_to_client;
typedef boost::shared_ptr<talk_to_client>client_ptr;
typedef std::vector<client_ptr> array;
```

```
array clients;
```

connection 类的框架如下：

```
class talk_to_client : public boost::enable_shared_from_this<talk_to_client> ,
boost::noncopyable
{
    talk_to_client() { ... }
public:
    typedef boost::system::error_code error_code;
    typedef boost::shared_ptr<talk_to_client> ptr;
    void start()
    {
        started_ = true;
        clients.push_back( shared_from_this());
        last_ping = boost::posix_time::microsec_clock::local_time();
        do_read(); //首先，我们等待客户端连接
    }
    static ptr new_() { ptr new_(new talk_to_client); return new_; }
    void stop()
    {
        if ( !started_) return;
        started_ = false;
        sock_.close();
        ptr self = shared_from_this();
        array::iterator it = std::find(clients.begin(), clients.end(), self);
        clients.erase(it);
        update_clients_changed();
    }
    bool started() const { return started_; }
    ip::tcp::socket & sock() { return sock_;}
    std::string username() const { return username_; }
    void set_clients_changed() { clients_changed_ = true; }
    …
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
    bool started_;
    std::string username_;
    deadline_timer timer_;
    boost::posix_time::ptime last_ping;
    bool clients_changed_;
};
```

我会用 *talk_to_client* 或者 *talk_to_server* 来调用 *connection* 类，从而让你更明白我所说的内容。

现在你需要用到之前的代码了；它和我们在客户端应用中所用到的是一样的。我们还有另外一个 *stop()*方法，这个方法用来从客户端数组中移除一个客户端连接。

服务端持续不断地等待异步的 *read* 操作：

```
void on_read(const error_code & err, size_t bytes)
{
    if ( err) stop();
    if ( !started() ) return;
    std::string msg(read_buffer_, bytes);
    if ( msg.find("login ") == 0) on_login(msg);
    else if ( msg.find("ping") == 0) on_ping();
    else if ( msg.find("ask_clients") == 0) on_clients();
}
void on_login(const std::string & msg)
{
    std::istringstream in(msg);
    in >> username_ >> username_;
    do_write("login ok\n");
    update_clients_changed();
}
void on_ping()
{
    do_write(clients_changed_ ? "ping client_list_changed\n" : "ping ok\n");
    clients_changed_ = false;
}
void on_clients()
{
    std::string msg;
    for(array::const_iterator b =clients.begin(),e =clients.end(); b != e; ++b)
        msg += (*b)->username() + " ";
    do_write("clients " + msg + "\n");
}
```

这段代码是简单易懂的；需要注意的一点是：当一个新客户端登录，我们调用 *update_clients_changed()*，这个方法为所有客户端将 *clientschanged* 标志为 *true*。

服务端每收到一个请求就用相应的方式进行回复，如下所示：

```
void do_ping() { do_write("ping\n"); }
```

```
void do_ask_clients() { do_write("ask_clients\n"); }
void on_write(const error_code & err, size_t bytes) { do_read(); }
void do_read()
{
    async_read(sock_, buffer(read_buffer_), MEM_FN2(read_complete,_1,_2),
        MEM_FN2(on_read,_1,_2));
    post_check_ping();
}
void do_write(const std::string & msg)
{
    if ( !started() ) return;
    std::copy(msg.begin(), msg.end(), write_buffer_);
    sock_.async_write_some( buffer(write_buffer_, msg.size()),
        MEM_FN2(on_write,_1,_2));
}
size_t read_complete(const boost::system::error_code & err, size_t bytes)
{
    // ... 就像之前
}
```

在每个 *write* 操作的末尾，*on_write()*方法被调用，这个方法会触发另外一个异步读操作，这样的话“等待请求－回复请求”这个循环就会一直执行，直到客户端断开连接或者超时。

在每次读操作开始之前，我们异步等待 5 秒钟来观察客户端是否超时。如果超时，我们关闭它的连接：

```
void on_check_ping()
{
    ptime now = microsec_clock::local_time();
    if ( (now - last_ping).total_milliseconds() > 5000)
        stop();
    last_ping = boost::posix_time::microsec_clock::local_time();
}
void post_check_ping()
{
    timer_.expires_from_now(boost::posix_time::millisec(5000));
    timer_.async_wait( MEM_FN(on_check_ping));
}
```

这就是整个服务端的实现。你可以运行并让它工作起来！

在代码中，我向你们展示了这一章我们学到的东西，为了更容易理解，我把代码稍微精简了下；比如，大部分的控制台输出我都没有展示，尽管在这本书附赠的

代码中它们是存在的。我建议你自己运行这些例子，因为从头到尾读一次代码能加强你对本章展示应用的理解。

总结

我们已经学到了怎么写一些基础的客户端/服务端应用。我们已经避免了一些诸如内存泄漏和死锁的低级错误。所有的编码都是框架式的，这样你就可以根据你自己的需求对它们进行扩展。

在接下来的章节中，我们会更加深入地了解使用 Boost.Asio 进行同步编程和异步编程的不同点，同时你也会学会如何嵌入你自己的异步操作。

第五章

同步 VS 异步

Boost.Asio 的作者做了一个很惊艳的工作：它可以让你在同步和异步中自由选择，从而更好地适应你的应用。

在之前的章节中，我们已经学习了各种类型应用的框架，比如同步客户端，同步服务端，异步客户端，异步服务端。它们中的每一个都可以作为你应用的基础。如果要更加深入地学习各种类型应用的细节，请继续。

混合同步异步编程

Boost.Asio 库允许你进行同步和异步的混合编程。我个人认为这是一个坏主意，但是 Boost.Asio（就像 C++一样）在你需要的时候允许你深入底层。

通常来说，当你写一个异步应用时，你会很容易掉入这个陷阱。比如在响应一个异步 *write* 操作时，你做了一个同步 *read* 操作：

```
io_service service;
ip::tcp::socket sock(service);
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
void on_write(boost::system::error_code err, size_t bytes)
{
    char read_buff[512];
    read(sock, buffer(read_buff));
}
async_write(sock, buffer("echo"), on_write);
```

毫无疑问，同步 *read* 操作会阻塞当前的线程，从而导致其他任何正在等待的异步操作变成挂起状态（对这个线程）。这是一段糟糕的代码，因为它会导致整个应用变得无响应或者整个被阻塞掉（所有异步运行的端点都必须避免阻塞，而执行一个同步的操作违反了这个原则）。

当你写一个同步应用时，你不大可能执行异步的 *read* 或者 *write* 操作，因为同步地思考已经意味着用一种线性的方式思考（执行 A，然后执行 B，再执行 C，等等）。

我唯一能想到的同步和异步同时工作的场景就是同步操作和异步操作是完全隔离的，比如，同步和异步从一个数据库进行读写。

从客户端传递信息到服务端 VS 从服务端传递信息到客户端

成功的客户端/服务端应用一个很重要的部分就是来回传递消息（服务端到客户端和客户端到服务端）。你需要指定用什么来标记一个消息。换句话说，当读取一个输入的消息时，你怎么判断它被完整读取了？

标记消息结尾的方式完全取决于你（标记消息的开始很简单，因为它就是前一个消息之后传递过来的第一个字节），但是要保证消息是简单且连续的。

你可以：

- 消息大小固定（这不是一个很好的主意，如果我们需要发送更多的数据怎么办？）
- 通过一个特殊的字符标记消息的结尾，比如'\n'或者'\0'
- 在消息的头部指定消息的大小

我在整本书中间采用的方式都是“使用'\n'标记消息的结尾”。所以，每次读取一条消息都会如下：

```
char buff_[512];
// 同步读取
read(sock_, buffer(buff_), boost::bind(&read_complete, this, _1, _2));
// 异步读取
async_read(sock_, buffer(buff_),MEM_FN2(read_complete,_1,_2),
    MEM_FN2(on_read,_1,_2));
size_t read_complete(const boost::system::error_code & err, size_t bytes)
{
    if ( err) return 0;
    already_read_ = bytes;
    bool found = std::find(buff_, buff_ + bytes, '\n') < buff_ + bytes;
    // 一个一个读，直到读到回车，无缓存
    return found ? 0 : 1;
}
```

我把在消息头部指定消息长度这种方式作为一个练习留给读者；这非常简单。

客户端应用中的同步 I/O

同步客户端一般都能归类到如下两种情况中的一种：

- 它向服务端请求一些东西，读取结果，然后处理它们。然后请求一些其他的东西，然后一直持续下去。事实上，这很像之前章节里说到的同步客户端。
- 从服务端读取消息，处理它，然后写回结果。然后读取另外一条消息，然后一直持续下去。

两种情况都使用“发送请求－读取结果”的策略。换句话说，一个部分发送一个请求到另外一个部分然后另外一个部分返回结果。这是实现客户端/服务端应用非常简单的一种方式，同时这也是我非常推荐的一种方式。

你可以创建一个 *Mambo Jambo* 类型的客户端服务端应用，你可以随心所欲地写它们中间的任何一个部分，但是这会导致一场灾难。（你怎么知道当客户端或者服务端阻塞的时候会发生什么？）。

上面的情况看上去会比较相似，但是它们非常不同：

- 前者，服务端响应请求（服务端等待来自客户端的请求然后回应）。这是一个请求式连接，客户端从服务端拉取它需要的东西。
- 后者，服务端发送事件到客户端然后由客户端响应。这是一个推式连接，服务端推送通知/事件到客户端。

你大部分时间都在做请求式客户端/服务端应用，这也是比较简单，同时也是比较常见的。

你可以把拉取请求（客户端到服务端）和推送请求（服务端到客户端）结合起来，但是，这是非常复杂的，所以你最好避免这种情况 。把这两种方式结合的问题在于：如果你使用“发送请求－读取结果”策略。就会发生下面一系列事情：

- 客户端写入（发送请求）
- 服务端写入（发送通知到客户端）
- 客户端读取服务端写入的内容，然后将其作为请求的结果进行解析
- 服务端阻塞以等待客户端的返回的结果，这会在客户端发送新请求的时候发生
- 服务端把发送过来的请求当作它等待的结果进行解析
- 客户端会阻塞（服务端不会返回任何结果，因为它把客户端的请求当作它通知返回的结果）

在一个请求式客户端/服务端应用中，避免上面的情况是非常简单的。你可以通过实现一个 ping 操作的方式来模拟一个推送式请求，我们假设每 5 秒钟客户端 ping 一次服务端。如果没有事情需要通知，服务端返回一个类似 *ping ok* 的结果，如果有事情需要通知，服务端返回一个 *ping [event_name]*。然后客户端就可以初始化一个新的请求去处理这个事件。

复习一下，第一种情况就是之前章节中的同步客户端应用，它的主循环如下：

```
void loop()
{
    // 对于我们登录操作的结果
    write("login " + username_ + "\n");
    read_answer();
    while ( started_)
    {
        write_request();
        read_answer();
        ...
    }
}
```

我们对其进行修改以适应第二种情况：

```
void loop()
{
    while ( started_)
    {
        read_notification();
        write_answer();
    }
}
void read_notification()
{
    already_read_ = 0;
    read(sock_, buffer(buff_), boost::bind(&talk_to_svr::read_complete, this,
        _1, _2));
    process_notification();
}
void process_notification()
{
    // ... 看通知是什么，然后准备回复
}
```

服务端应用中的同步 I/O

类似客户端，服务端也被分为两种情况用来匹配之前章节中的情况 1 和情况 2。同样，两种情况都采用“发送请求－读取结果”的策略。

第一种情况是我们在之前章节实现过的同步服务端。当你是同步时读取一个完整的请求不是很简单，因为你需要避免阻塞（通常来说是能读多少就读多少）：

```
void read_request()
{
    if ( sock_.available())
}
already_read_ += sock_.read_some(buffer(buff_ + already_read_, max_msg -
    already_read_));
```

只要一个消息被完整读到，就对它进行处理然后回复给客户端：

```
void process_request()
{
    bool found_enter = std::find(buff_, buff_ + already_read_, '\n') < buff_
        + already_read_;
    if ( !found_enter)
        return; // 消息不完整
    size_t pos = std::find(buff_, buff_ + already_read_, '\n') - buff_;
    std::string msg(buff_, pos);
    ...
    if ( msg.find("login ") == 0) on_login(msg);
    else if ( msg.find("ping") == 0) on_ping();
    else ...
}
```

如果我们想让服务端变成一个推送服务端，我们通过如下的方式修改：

```
typedef std::vector<client_ptr> array;
array clients;
array notify;
std::string notify_msg;
void on_new_client()
{
    // 新客户端连接时，我们通知所有客户端这个事件
    notify = clients;
    std::ostringstream msg;
    msg << "client count " << clients.size();
    notify_msg = msg.str();
    notify_clients();
}
```

```
void notify_clients()
{
    for ( array::const_iterator b = notify.begin(), e = notify.end(); b != e;
        ++b)
    {
        (*b)->sock_.write_some(notify_msg);
    }
}
```

*on_new_client()*方法是事件之一，这个事件我们需要通知所有的客户端。*notify_clients* 是通知所有对一个事件感兴趣客户端的方法。它发送消息但是不等待每个客户端返回的结果，因为那样的话就会导致阻塞。当客户端返回一个结果时，客户端会告诉我们它为什么回复（然后我们就可以正确地处理它）。

同步服务端中的线程

这是一个非常重要的关注点：我们开辟多少线程去处理服务端请求？ 对于一个同步服务端，我们至少需要一个处理新连接的线程：

```
void accept_thread()
{
    ip::tcp::acceptor acceptor(service,
ip::tcp::endpoint(ip::tcp::v4(),8001));
    while ( true)
    {
        client_ptr new_( new talk_to_client);
        acceptor.accept(new_->sock());
        boost::recursive_mutex::scoped_lock lk(cs);
        clients.push_back(new_);
    }
}
```

对于已经存在的客户端：

- 我们可以是单线程。这是最简单的，同时也是我在**第四章 同步服务端**中采用的实现方式。它可以很轻松地处理 100-200 并发的客户端而且有时候会更多，对于大多数情况来说这已经足够用了。
- 我们可以对每个客户端开一个线程。这不是一个很好的选择；他会浪费很多线程而且有时候会导致调试困难，而且当它需要处理 200 以上并发的客户端的时候，它可能马上会到达它的瓶颈。
- 我们可以用一些固定数量的线程去处理已经存在的客户端

第三种选择是同步服务端中最难实现的；整个 *talk_to_client* 类需要是线程安全的。然后，你需要一个机制来确定哪个线程处理哪个客户端。对于这个问题，你有两个选择：

- 将特定的客户端分配给特定的线程；比如，线程 1 处理前面 20 个客户端，线程 2 处理 21 到 40 个线程，等等。当一个线程在使用时（我们在等待被客户端阻塞的一些东西），我们从已存在客户端列表中将其取出来。等我们处理完之后，再把它放回到列表中。每个线程都会循环遍历已经存在的客户端列表，然后把拥有完整请求的第一个客户端提出来（我们已经从客户端读取了一条完整的消息），然后回复它。
- 服务端可能会变得无响应
    - 第一种情况，被同一个线程处理的几个客户端同时发送请求，因为一个线程在同一时刻只能处理一个请求。所以这种情况我们什么也不能做。
    - 第二种情况，如果我们发现并发请求大于当前线程个数的时候。我们可以简单地创建新线程来处理当前的压力。

下面的代码片段有点类似之前的 *answer_to_client* 方法，它向我们展示了第二种方法的实现方式：

```
struct talk_to_client : boost::enable_shared_from_this<talk_to_client>
{
    ...
    void answer_to_client()
    {
        try
        {
            read_request();
            process_request();
        } catch ( boost::system::system_error&)
        { stop(); }
    }
};
```

我们需要对它进行修改使它变成下面代码片段的样子：

```
struct talk_to_client : boost::enable_shared_from_this<talk_to_client>
{
    boost::recursive_mutex cs;
    boost::recursive_mutex cs_ask;
    bool in_process;
    void answer_to_client()
```

```
    {
        { boost::recursive_mutex::scoped_lock lk(cs_ask);
            if ( in_process)
                return;
            in_process = true;
        }
        { boost::recursive_mutex::scoped_lock lk(cs);
            try {
                read_request();
                process_request();
            }catch ( boost::system::system_error&)
            {
                stop();
            }
        }
        { boost::recursive_mutex::scoped_lock lk(cs_ask);
            in_process = false;
        }
    }
};
```

当我们在处理一个客户端请求的时候，它的 *in_process* 变量被设置成 *true*，其他的线程就会忽略这个客户端。额外的福利就是 *handle_clients_thread()*方法不需要做任何修改；你可以随心所欲地创建你想要数量的 *handle_clients_thread()* 方法。

客户端应用中的异步 I/O

主流程和同步客户端应用有点类似，不同的是 Boost.Asio 每次都位于 async_read 和 async_write 请求中间。

第一种情况是我在**第四章 客户端和服务端** 中实现过的。你应该还记得在每个异步操作结束的时候，我都启动另外一个异步操作，这样 *service.run()*方法才不会结束。

为了适应第二种情况，你需要使用下面的代码片段：

```
void on_connect()
{
    do_read();
}
```

```
void do_read()
{
    async_read(sock_, buffer(read_buffer_), MEM_FN2(read_complete,_1,_2),
       MEM_FN2(on_read,_1,_2));
}
void on_read(const error_code & err, size_t bytes)
{
    if ( err) stop();
    if ( !started() ) return;
    std::string msg(read_buffer_, bytes);
    if ( msg.find("clients") == 0) on_clients(msg);
    else ...
}
void on_clients(const std::string & msg)
{
    std::string clients = msg.substr(8);
    std::cout << username_ << ", new client list:" << clients ;
    do_write("clients ok\n");
}
```

注意只要我们成功连接上，我们就开始从服务端读取。每个 *on_[event]*方法都会通过写一个回复给服务端的方式来结束我们。

使用异步的美好在于你可以使用 Boost.Asio 进行管理，从而把 I/O 网络操作和其他异步操作结合起来。尽管它的流程不像同步的流程那么清晰，你仍然可以用同步的方式来想象它。

假设，你从一个 web 服务器读取文件然后把它们保存到一个数据库中（异步地）。你可以把这个过程想象成下面的流程图：

服务端应用的异步 I/O

现在要展示的是两个普遍的情况，情况 1（拉取）和情况 2（推送）

第一种情况同样是我在**第 4 章 客户端和服务端** 中实现的异步服务端。在每一个异步操作最后，我都会启动另外一个异步操作，这样的话 service.run()就不会结束。现在要展示的是被剪裁过的框架代码。下面是 talk_to_client 类所有的成员：

```
void start()
```

```
{
    ...
    do_read(); // first, we wait for client to login
}
void on_read(const error_code & err, size_t bytes)
{
    std::string msg(read_buffer_, bytes);
    if ( msg.find("login ") == 0) on_login(msg);
    else if ( msg.find("ping") == 0) on_ping();
    else
    ...
}
void on_login(const std::string & msg)
{
    std::istringstream in(msg);
    in >> username_ >> username_;
    do_write("login ok\n");
}
void do_write(const std::string & msg)
{
    std::copy(msg.begin(), msg.end(), write_buffer_);
    sock_.async_write_some( buffer(write_buffer_, msg.size()),
        MEM_FN2(on_write,_1,_2));
}
void on_write(const error_code & err, size_t bytes) { do_read(); }
```

简单来说，我们始终等待一个 *read* 操作，而且只要一发生，我们就处理然后将结果返回给客户端。

我们把上述代码进行修改就可以完成一个推送服务端

```
void start()
{
    ...
    on_new_client_event();
}
void on_new_client_event()
{
    std::ostringstream msg;
    msg << "client count " << clients.size();
    for ( array::const_iterator b = clients.begin(), e = clients.end();
        (*b)->do_write(msg.str());
}
void on_read(const error_code & err, size_t bytes)
{
```

```
    std::string msg(read_buffer_, bytes);
    // 在这里我们基本上只知道我们的客户端接收到我们的通知
}
void do_write(const std::string & msg)
{
    std::copy(msg.begin(), msg.end(), write_buffer_);
    sock_.async_write_some( buffer(write_buffer_, msg.size()),
        MEM_FN2(on_write,_1,_2));
}
void on_write(const error_code & err, size_t bytes) { do_read(); }
```

只要有一个事件发生，我们假设*是 on_new_client_event*，所有需要被通知到的客户端就都收到一条信息。当它们回复时，我们简单认为他们已经确认收到事件。注意我们永远不会把正在等待的异步操作用尽（所以，*service.run()*不会结束），因为我们一直在等待一个新的客户端：

```
ip::tcp::acceptor acc(service, ip::tcp::endpoint(ip::tcp::v4(), 8001));
void handle_accept(talk_to_client::ptr client, const error_code & err)
{
    client->start();
    talk_to_client::ptr new_client = talk_to_client::new_();
    acc.async_accept(new_client->sock(), bind(handle_accept,new_client,_1));
}
```

异步服务端中的多线程

我在**第 4 章 客户端和服务端** 展示的异步服务端是单线程的，所有的事情都发生在 main()中：

```
int main()
{
    talk_to_client::ptr client = talk_to_client::new_();
    acc.async_accept(client->sock(), boost::bind(handle_accept,client,_1));
    service.run();
}
```

异步的美妙之处就在于可以非常简单地把单线程变为多线程。你可以一直保持单线程直到你的并发客户端超过 200。然后，你可以使用如下的代码片段把单线程变成 100 个线程：

```
boost::thread_group threads;
void listen_thread()
{
    service.run();
```

```
}
void start_listen(int thread_count)
{
    for ( int i = 0; i < thread_count; ++i)
        threads.create_thread( listen_thread);
}
int main(int argc, char* argv[])
{
    talk_to_client::ptr client = talk_to_client::new_();
    acc.async_accept(client->sock(), boost::bind(handle_accept,client,_1));
    start_listen(100);
    threads.join_all();
}
```

当然，一旦你选择了多线程，你需要考虑线程安全。尽管你在线程 A 中调用了 *async_**，但是它的完成处理流程可以在线程 B 中被调用（因为线程 B 也调用了* service.run()*）。对于它本身而言这不是问题。只要你遵循逻辑流程，也就是从* async_read()*到* on_read()*，从* on_read()*到* process_request*，从* process_request *到* async_write()*，从* async_write()*到* on_write()*，从* on_write() *到* async_read()*，然后在你的* talk_to_client*类中也没有被调用的公有方法，这样的话尽管不同的方法可以在不同的线程中被调用，它们还是会被有序地调用。从而不需要互斥量。

这也意味着对于一个客户端，只会有一个异步操作在等待。假如在某些情况，一个客户端有两个异步方法在等待，你就需要互斥量了。这是因为两个等待的操作可能正好在同一个时间完成，然后我们就会在两个不同的线程中间同时调用他们的完成处理函数。所以，这里需要线程安全，也就是需要使用互斥量。 在我们的异步服务端中，我们确实同时有两个等待的操作：

```
void do_read()
{
    async_read(sock_, buffer(read_buffer_),MEM_FN2(read_complete,_1,_2),
MEM_FN2(on_read,_1,_2));
    post_check_ping();
}
void post_check_ping()
{
    timer_.expires_from_now(boost::posix_time::millisec(5000));
    timer_.async_wait( MEM_FN(on_check_ping));
}
```

当在做一个 *read* 操作时，我们会异步等待 *read* 操作完成和超时。所以，这里需要线程安全。

我的建议是，如果你准备使用多线程，从开始就保证你的类是线程安全的。通常这不会影响它的性能（当然你也可以在配置中设置开关）。同时，如果你准备使用多线程，从一个开始就使用。这样的话你能尽早地发现可能存在的问题。一旦你发现一个问题，你首先需要检查的事情就是：单线程运行的时候是否会发生？如果是，它很简单；只要调试它就可以了。否则，你可能忘了对一些方法加锁（互斥量）。

因为我们的例子需要是线程安全的，我已经把 *talk_to_client* 修改成使用互斥量的了。同时，我们也有一个客户端连接的列表，它也需要自己的互斥量，因为我们有时需要访问它。

避免死锁和内存冲突不是那么容易。下面是我需要对 *update_client_changed()* 方法进行修改的地方：

```
void update_clients_changed()
{
    array copy;
    { boost::recursive_mutex::scoped_lock lk(clients_cs); copy = clients; }
    for( array::iterator b = copy.begin(), e = copy.end(); b != e; ++b)
        (*b)->set_clients_changed();
}
```

你需要避免的是同时有两个互斥量被锁定（这会导致死锁）。在我们的例子中，我们不想 *clients_cs* 和一个客户端的 *cs_*互斥量同时被锁住

异步操作

Boost.Asio 同样允许你异步地运行你任何一个方法。仅仅需要使用下面的代码片段：

```
void my_func()
{
    ...
}
service.post(my_func);
```

这样就可以保证 *my_func* 在调用了 *service.run()*方法的某个线程中间被调用。你同样可以异步地调用一个有完成处理 handler 的方法，方法的 handler 会在方法结束的时候通知你。伪代码如下：

```
void on_complete()
{
    ...
```

```
}
void my_func()
{
    ...
    service.post(on_complete);
}
async_call(my_func);
```

没有现成的 *async_call* 方法，因此，你需要自己创建。幸运的是，它不是很复杂，参考下面的代码片段：

```
struct async_op : boost::enable_shared_from_this<async_op>, ...
{
    typedef boost::function<void(boost::system::error_code)>completion_func;
    typedef boost::function<boost::system::error_code ()> op_func;
    struct operation { ... };
    void start() {
        { boost::recursive_mutex::scoped_lock lk(cs_);
            if ( started_) return; started_ = true; }
        boost::thread t(boost::bind(&async_op::run,this));
    }
    void add(op_func op, completion_func completion, io_service &service)
    {
        self_ = shared_from_this();
        boost::recursive_mutex::scoped_lock lk(cs_);
        ops_.push_back( operation(service, op, completion));
        if ( !started_) start();
    }
    void stop()
    {
        boost::recursive_mutex::scoped_lock lk(cs_);
        started_ = false; ops_.clear();
    }
private:
    boost::recursive_mutex cs_;
    std::vector<operation> ops_;
    bool started_;
    ptr self_;
};
```

async_op 方法创建了一个后台线程，这个线程会运行（*run()*）你添加（*add()*）到它里面的所有的异步操作。为了让事情简单一些，每个操作都包含下面的内容：

- 一个异步调用的方法
- 当第一个方法结束时被调用的一个完成处理 handler

- 会运行完成处理 handler 的 io_service 实例。这也是完成时通知你的地方。参考下面的代码：

```
struct async_op : boost::enable_shared_from_this<async_op>, private
boost::noncopyable
{
    struct operation
    {
        operation(io_service & service, op_func op, completion_func completion) :
            service(&service), op(op), completion(completion) , work(new
            io_service::work(service)) {}
        operation() : service(0) {}
        io_service * service;
        op_func op;
        completion_func completion;
        typedef boost::shared_ptr<io_service::work> work_ptr;
        work_ptr work;
    };
    ...
};
```

它们被 *operation* 结构体包含在内部。注意当有一个操作在等待时，我们在操作的构造方法中构造一个 *io_service::work* 实例，从而保证直到我们完成异步调用之前 *service.run()*都不会结束（当 *io_service::work* 实例保持活动时，*service.run()* 就会认为它有工作需要做）。参考下面的代码片段：

```
struct async_op : ...
{
    typedef boost::shared_ptr<async_op> ptr;
    static ptr new_() { return ptr(new async_op); }
    ...
    void run()
    {
        while ( true)
        {
            {
                boost::recursive_mutex::scoped_lock lk(cs_);
                if ( !started_) break;
            }
            boost::this_thread::sleep(boost::posix_time::millisec(10));
            operation cur;
            {
                boost::recursive_mutex::scoped_lock lk(cs_);
                if ( !ops_.empty())
```

```
            {
                cur = ops_[0];
                ops_.erase(ops_.begin());
            }
        }
        if ( cur.service)
            cur.service->post(boost::bind(cur.completion, cur.op()));
    }
    self_.reset();
  }
};
```

*run()*方法就是后台线程；它仅仅观察是否有工作需要做，如果有，就一个一个地运行这些异步方法。在每个调用结束的时候，它会调用相关的完成处理方法。

为了测试，我们创建一个会被异步执行的 *compute_file-checksum* 方法

```
size_t checksum = 0;
boost::system::error_code compute_file_checksum(std::string file_name)
{
    HANDLE file = ::CreateFile(file_name.c_str(),GENERIC_READ, 0,
        0,OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, 0);
    windows::random_access_handle h(service, file);
    long buff[1024];
    checksum = 0;
    size_t bytes = 0, at = 0;
    boost::system::error_code ec;
    while ( (bytes = read_at(h, at, buffer(buff), ec)) > 0)
    {
        at += bytes; bytes /= sizeof(long);
        for ( size_t i = 0; i < bytes; ++i)
            checksum += buff[i];
    }
    return boost::system::error_code(0,boost::system::generic_category());
}
void on_checksum(std::string file_name, boost::system::error_code)
{
    std::cout << "checksum for " << file_name << "=" << checksum << std::endl;
}
int main(int argc, char* argv[])
{
    std::string fn = "readme.txt";
    async_op::new_()->add( service,
    boost::bind(compute_file_checksum,fn),boost::bind(on_checksum,fn,_1));
    service.run();
```

```
}
```

注意我展示给你的只是实现异步调用一个方法的一种可能。除了像我这样实现一个后台线程，你可以使用一个内部 *io_service* 实例，然后推送（*post()*）异步方法给这个实例调用。这个作为一个练习留给读者。

你也可以扩展这个类让其可以展示一个异步操作的进度（比如，使用百分比）。这样做你就可以在主线程通过一个进度条来显示进度。

代理实现

代理一般位于客户端和服务端之间。它接受客户端的请求，可能会对请求进行修改，然后接着把请求发送到服务端。然后从服务端取回结果，可能也会对结果进行修改，然后接着把结果发送到客户端。

代理有什么特别的？我们讲述它的目的在于：对每个连接，你都需要两个 sokect，一个给客户端，另外一个给服务端。这些都给实现一个代理增加了不小的难度。

实现一个同步的代理应用比异步的方式更加复杂；数据可能同时从两个端过来（客户端和服务端），也可能同时发往两个端。这也就意味着如果我们选择同步，我们就可能在一端向另一端 *read()*或者 *write()*，同时另一端向这一端 *read()*或者 *write()*时阻塞，这也就意味着最终我们会变得无响应。

根据下面几条实现一个异步代理的简单例子：

- 在我们的方案中，我们在构造函数中能拿到两个连接。但不是所有的情况都这样，比如对于一个 web 代理来说，客户端只告诉我们服务端的地址。
- 因为比较简单，所以不是线程安全的。参考如下的代码：

```
class proxy  : public boost::enable_shared_from_this<proxy>
{
    proxy(ip::tcp::endpoint ep_client, ip::tcp::endpoint ep_server) : ... {}
public:
    static ptr start(ip::tcp::endpoint ep_client,
        ip::tcp::endpoint ep_svr)
    {
        ptr new_(new proxy(ep_client, ep_svr));
        // … 连接到两个端
        return new_;
    }
```

```
    void stop()
    {
        // ... 关闭两个连接
    }
    bool started() { return started_ == 2; }
private:
    void on_connect(const error_code & err)
    {
        if ( !err)
        {
            if ( ++started_ == 2) on_start();
        } else stop();
    }
    void on_start()
    {
        do_read(client_, buff_client_);
        do_read(server_, buff_server_);
    }
...
private:
    ip::tcp::socket client_, server_;
    enum { max_msg = 1024 };
    char buff_client_[max_msg], buff_server_[max_msg];
    int started_;
};
```

这是个非常简单的代理。当我们两个端都连接时，它开始从两个端读取（*on_start()* 方法）：

```
class proxy  : public boost::enable_shared_from_this<proxy>
{
    ...
    void on_read(ip::tcp::socket & sock, const error_code& err, size_t bytes)
    {
        char * buff = &sock == &client_ ? buff_client_ : buff_server_;
        do_write(&sock == &client_ ? server_ : client_, buff, bytes);
    }
    void on_write(ip::tcp::socket & sock, const error_code &err, size_t bytes)
    {
        if ( &sock == &client_) do_read(server_, buff_server_);
        else do_read(client_, buff_client_);
    }
    void do_read(ip::tcp::socket & sock, char* buff)
    {
```

```
        async_read(sock, buffer(buff, max_msg),
            MEM_FN3(read_complete,ref(sock),_1,_2),
            MEM_FN3(on_read,ref(sock),_1,_2));
    }
    void do_write(ip::tcp::socket & sock, char * buff, size_t size)
    {
        sock.async_write_some(buffer(buff,size),
            MEM_FN3(on_write,ref(sock),_1,_2));
    }
    size_t read_complete(ip::tcp::socket & sock, const error_code & err, size_t
        bytes)
    {
        if ( sock.available() > 0)
            return sock.available();
        return bytes > 0 ? 0 : 1;
    }
};
```

对每一个成功的读取操作（*on_read*），它都会发送消息到另外一个部分。只要消息一发送成功（*on_write*），我们就从来源那部分再次读取。

使用下面的代码片段让这个流程运转起来：

```
int main(int argc, char* argv[])
{
    ip::tcp::endpoint ep_c(ip::address::from_string("127.0.0.1"),8001);
    ip::tcp::endpoint ep_s(ip::address::from_string("127.0.0.1"),8002);
    proxy::start(ep_c, ep_s);
    service.run();
}
```

你会注意到我在读和写中重用了 buffer。这个重用是 ok 的，因为从客户端读取到的消息在新消息被读取之前就已经写入到服务端，反之亦然。这也意味着这种特别的实现方式会碰到响应性的问题。当我们正在处理到 B 部分的写入时，我们不会从 A 读取（我们会在写入到 B 部分完成时重新从 A 部分读取）。你可以通过下面的方式重写实现来克服这个问题：

- 使用多个读取 buffer
- 对每个成功的 *read* 操作，除了异步写回到另外一个部分，还需要做额外的一个 *read*（读取到一个新的 buffer）
- 对每个成功的 *write* 操作，销毁（或者重用）这个 buffer

我会把这个当作练习留给你们。

小结

在选择同步或者异步时需要考虑很多事情。最先需要考虑的就是避免混淆它们。

在这一章中，我们已经看到：

- 实现，测试，调试各个类型的应用是多么简单
- 线程是如何影响你的应用的
- 应用的行为是怎么影响它的实现的（拉取或者推送类型）
- 选择异步时怎样去嵌入自己的异步操作

接下来，我们会了解一些 Boost.Asio 不那么为人知晓的特性，中间就有我最喜欢的 Boost.Asio 特性－协程，它可以让你轻松地取异步之精华，去异步之糟粕。

第六章

Boost.Asio－其他特性

这章我们讲了解一些 Boost.Asio 不那么为人所知的特性。标准的 stream 和 streambuf 对象有时候会更难用一些，但正如你所见，它们也有它们的益处。最后，你会看到姗姗来迟的 Boost.Asio 协程的入口，它可以让你的异步代码变得非常易读。这是非常惊人的一个特性。

标准 stream 和标准 I/O buffer

读这一章节之前你需要对 STL stream 和 STL streambuf 对象有所了解。

Boost.Asio 在处理 I/O 操作时支持两种类型的 buffer：

- *boost::asio::buffer()*：这种 buffer 关联着一个 Boost.Asio 的操作（我们使用的 buffer 被传递给一个 Boost.Asio 的操作）
- *boost::asio::streambuf*：这个 buffer 继承自 *std::streambuf*，在网络编程中可以和 STL stream 一起使用

纵观全书，之前的例子中最常见的例子如下：

```
size_t read_complete(boost::system::error_code, size_t bytes){ ... }
char buff[1024];
read(sock, buffer(buff), read_complete);
write(sock, buffer("echo\n"));
```

通常来说使用这个就能满足你的需要，如果你想要更复杂，你可以使用 *streambuf* 来实现。

这个就是你可以用 *streambuf* 对象做的最简单也是最坏的事情：

```
streambuf buf;
read(sock, buf);
```

这个会一直读到 *streambuf* 对象满了，然后因为 *streambuf* 对象可以通过自己重新开辟空间从而获取更多的空间，它基本会读到连接被关闭。

你可以使用 *read_until* 一直读到一个特定的字符串：

```
streambuf buf;
read_until(sock, buf, "\n");
```

这个例子会一直读到一个“\n”为止，把它添加到 *buffer* 的末尾，然后退出 *read* 方法。

向一个 *streambuf* 对象写一些东西，你需要做一些类似下面的事情：

```
streambuf buf;
std::ostream out(&buf);
out << "echo" << std::endl;
write(sock, buf);
```

这是非常直观的；你在构造函数中传递你的 *streambuf* 对象来构建一个 STL stream，将其写入到你想要发送的消息中，然后使用 *write* 来发送 buffer 的内容。

Boost.Asio 和 STL stream

Boost.Asio 在集成 STL stream 和网络方面做了很棒的工作。也就是说，如果你已经在使用 STL 扩展，你肯定就已经拥有了大量重载了操作符<<和>>的类。从 socket 读或者写入它们就好像在公园漫步一样简单。

假设你有下面的代码片段：

```
struct person
{
    std::string first_name, last_name;
    int age;
};
std::ostream& operator<<(std::ostream & out, const person & p)
{
    return out << p.first_name << " " << p.last_name << " " << p.age;
}
std::istream& operator>>(std::istream & in, person & p)
{
    return in >> p.first_name >> p.last_name >> p.age;
}
```

通过网络发送这个 *person* 就像下面的代码片段这么简单：

```
streambuf buf;
std::ostream out(&buf);
person p;
// … 初始化 p
out << p << std::endl;
write(sock, buf);
```

另外一个部分也可以非常简单的读取：

```
read_until(sock, buf, "\n");
std::istream in(&buf);
person p;
in >> p;
```

使用 *streambuf* 对象（当然，也包括它用来写入的 *std::ostream* 和用来读取的 *std::istream*）时最棒的部分就是你最终的编码会很自然：

- 当通过网络写入一些要发送的东西时，很有可能你会有多个片段的数据。所以，你需要把数据添加到一个 buffer 里面。如果那个数据不是一个字符串，你需要先把它转换成一个字符串。当使用<<操作符时这些操作默认都已经做了。
- 同样，在另外一个部分，当读取一个消息时，你需要解析它，也就是说，读取到一个片段的数据时，如果这个数据不是字符串，你需要将它转换为字符串。当你使用>>操作符读取一些东西时这些也是默认就做了的。

最后要给出的是一个非常著名，非常酷的诀窍，使用下面的代码片段把 *streambuf* 的内容输出到 console 中

```
streambuf buf;
...
std::cout << &buf << std::endl; //把所有内容输出到 console 中
```

同样的，使用下面的代码片段来把它的内容转换为一个 *string*

```
std::string to_string(streambuf &buf)
{
    std::ostringstream out;
    out << &buf;
    return out.str();
}
```

streambuf 类

我之前说过，*streambuf* 继承自 *std::streambuf*。就像 *std::streambuf* 本身，它不能拷贝构造。

另外，它有一些额外的方法，如下：

- *streambuf([max_size,][allocator])*：这个方法构造了一个 *streambuf* 对象。你可以选择指定一个最大的 buffer 大小和一个分配器，分配器用来在需要的时候分配/释放内存。

- *prepare(n)*：这个方法返回一个子 buffer，用来容纳连续的 n 个字符。它可以用来读取或者写入。方法返回的结果可以在任何 Boost.Asio 处理 *read/write* 的自由函数中使用，而不仅仅是那些用来处理 *streambuf* 对象的方法。
- *data()*：这个方法以连续的字符串形式返回整个 buffer 然后用来写入。方法返回的结果可以在任何 Boost.Asio 处理写入的自由函数中使用，而不仅仅是那些用来处理 streambuf 对象的方法。
- *comsume(n)*：在这个方法中，数据从输入队列中被移除（从 read 操作）
- *commit(n)*：在这个方法中，数据从输出队列中被移除(从 write 操作)然后加入到输入队列中（为 read 操作准备）。
- *size()*：这个方法以字节为单位返回整个 streambuf 对象的大小。
- *max_size()*：这个方法返回最多能保存的字节数。

除了最后的两个方法，其他的方法不是那么容易理解。首先，大部分时间你会把 *streambuf* 以参数的方式传递给 *read/write* 自由函数，就像下面的代码片段展示的一样：

```
read_until(sock, buf, "\n"); // 读取到 buf 中
write(sock, buf); // 从 buf 写入
```

如果你想之前的代码片段展示的一样把整个 buffer 都传递到一个自由函数中，方法会保证把 buffer 的输入输出指针指向的位置进行增加。也就是说，如果有数据需要读，你就能读到它。比如：

```
read_until(sock, buf, '\n');
std::cout << &buf << std::endl;
```

上述代码会把你刚从 socket 写入的东西输出。而下面的代码不会输出任何东西：

```
read(sock, buf.prepare(16), transfer_exactly(16) );
std::cout << &buf << std::endl;
```

字节被读取了，但是输入指针没有移动，你需要自己移动它，就像下面的代码片段所展示的：

```
read(sock, buf.prepare(16), transfer_exactly(16) );
buf.commit(16);
std::cout << &buf << std::endl;
```

同样的，假设你需要从 *streambuf* 对象中写入，如果你使用了 *write* 自由函数，则需要像下面一样：

```
streambuf buf;
std::ostream out(&buf);
```

```
out << "hi there" << std::endl;
write(sock, buf);
```

下面的代码会把 hi there 发送三次：

```
streambuf buf;
std::ostream out(&buf);
out << "hi there" << std::endl;
for ( int i = 0; i < 3; ++i)
    write(sock, buf.data());
```

发生的原因是因为 buffer 从来没有被消耗过，因为数据还在。如果你想消耗它，使用下面的代码片段：

```
streambuf buf;
std::ostream out(&buf);
out << "hi there" << std::endl;
write(sock, buf.data());
buf.consume(9);
```

总的来说，你最好选择一次处理整个 *streambuf* 实例。如果需要调整则使用上述的方法。

尽管你可以在读和写操作时使用同一个 *streambuf*，你仍然建议你分开使用两个，一个读另外一个写，它会让事情变的简单，清晰，同时你也会减少很多导致 bug 的可能。

处理 streambuf 对象的自由函数

下面列出了 Boost.Asio 中处理 streambuf 对象的自由函数：

- *read(sock, buf[, completion_function])*：这个方法把内容从 socket 读取到 *streambuf* 对象中。*completion* 方法是可选的。如果有，它会在每次 *read* 操作成功之后被调用，然后告诉 Boost.Asio 这个操作是否完成（如果没有，它继续读取）。它的格式是：*size_t completion(const boost::system::error_code & err, size_t bytes_transfered);*，如果 *completion* 方法返回 0，我们认为 *read* 操作完成了，如果非 0，它表示下一次调用 stream 的 *read_some* 方法需要读取的最大的字节数。
- *read_at(random_stream, offset, buf [, completion_function])*：这个方法从一个支持随机读取的 stream 中读取。注意它没有被应用到 socket 中（因为他们没有随机读取的模型，它们是单向的，一直向前）。
- *read_until(sock, buf, char | string | regex | match_condition)*：这个方法一直读到满足一个特性的条件为止。或者是一个 char 类型的数据被读到，

或者是一个字符串被读到，或者是一个目前读到的字符串能匹配的正则表达式，或者 *match_condition* 方法告诉我们需要结束这个方法。*match_condition* 方法的格式是：*pair match(iterator begin, iterator end);* ，*iterator* 代表 *buffers_ iterator*。如果匹配到，你需要返回一个 *pair*（*passed_end_of_match* 被设置成 true）。如果没有匹配到，你需要返回 *pair*（begin 被设置为 false）。

- *write(sock, buf [, completion_function])*: 这个方法写入 *streambuf* 对象所有的内容。*completion* 方法是可选的，它的表现和 *read()*的 *completion* 方法类似：当 write 操作完成时返回 0，或者返回一个非 0 数代表下一次调用 stream 的 *write_some* 方法需要写入的最大的字节数。
- *write_at(random_stream,offset, buf [, completion_function])*: 这个方法用来向一个支持随机存储的 stream 写入。同样，它没有被应用到 socket 中。
- *async_read(sock, buf [, competion_function], handler)*: 这个方法是 *read()*的异步实现，handler 的格式为：*void handler(const boost::system::error_code, size_t bytes)*。
- *async_read_at(radom_stream, offset, buf [, completion_function] , handler)*: 这个方法是 *read_at()*的异步实现。
- *asyncread_until (sock, buf, char | string | regex | match condition, handler)*: 这个方法是 *read_until()*的异步实现。
- *async_write(sock, buf [, completion_function] , handler)*: 这个方法是 *write()*的异步实现。
- *async_write_at(random_stream,offset, buf [, completion_function] , handler)*: 这个方法是 *write_at()*的异步实现。

我们假设你需要一直读取直到读到一个元音字母：

```
streambuf buf;
bool is_vowel(char c)
{
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}
size_t read_complete(boost::system::error_code, size_t bytes)
{
    const char * begin = buffer_cast<const char*>( buf.data());
    if ( bytes == 0) return 1;
    while ( bytes > 0)
        if ( is_vowel(*begin++)) return 0;
        else --bytes;
```

```
    return 1;
}
...
read(sock, buf, read_complete);
```

这里需要注意的事情是对 *read_complete()*中 buffer 的访问，也就是 *buffer_cast<>* 和 *buf.data*。

如果你使用正则，上面的例子会更简单：

```
read_until(sock, buf, boost::regex("^[aeiou]+") );
```

或者我们修改例子来让 *match_condition* 方法工作起来：

```
streambuf buf;
bool is_vowel(char c)
{
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}
typedef buffers_iterator<streambuf::const_buffers_type> iterator;
std::pair<iterator,bool> match_vowel(iterator b, iterator e)
{
    while ( b != e)
        if ( is_vowel(*b++)) return std::make_pair(b, true);
    return std::make_pair(e, false);
}
...
size_t bytes = read_until(sock, buf, match_vowel);
```

当使用 *read_until* 时会有个难点：你需要记住你已经读取的字节数，因为下层的 buffer 可能多读取了一些字节（不像使用 *read()*时）。比如：

```
std::cout << &buf << std::endl;
```

上述代码输出的字节可能比 *read_until* 读取到的多。

协程

Boost.Asio 的作者在 2009-2010 年间实现了非常酷的一个部分，协程，它能让你更简单地设计你的异步应用。

它们可以让你同时享受同步和异步两个世界中最好的部分，也就是：异步编程但是很简单就能遵循流程控制，就好像应用是按流程实现的。

正常的流程已经在情形 1 种展示了，如果使用协程，你会尽可能的接近情形 2。

简单来说，就是协程允许在方法中的指定位置开辟一个入口来暂停和恢复运行。

如果要使用协程，你需要在 *boost/libs/asio/example/http/server4* 目录下的两个头文件：*yield.hpp* 和 *coroutine.hpp*。在这里，Boost.Asio 定义了两个虚拟的关键词（宏）和一个类：

- *coroutine*：这个类在实现协程时被你的连接类继承或者使用。
- *reenter(entry)*：这个是协程的主体。参数 *entry* 是一个指向 *coroutine* 实例的指针，它被当作一个代码块在整个方法中使用。
- *yield code*：它把一个声明当作协程的一部分来运行。当下一次进入方法时，操作会在这段代码之后执行。

为了更好的理解，我们来看一个例子。我们会重新实现 **第四章 异步客户端** 中的应用，这是一个可以登录，ping，然后能告诉你其他已登录客户端的简单客户端应用。 核心代码和下面的代码片段类似：

```
class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr>, public
coroutine, boost::noncopyable
{
    ...
    void step(const error_code & err = error_code(), size_t bytes = 0)
    {
        reenter(this)
        {
            for (;;)
            {
                yield async_write(sock_, write_buffer_, MEM_FN2(step,_1,_2) );
                yield async_read_until( sock_, read_buffer_,"\n",
                    MEM_FN2(step,_1,_2));
                yield service.post( MEM_FN(on_answer_from_server));
            }
        }
    }
};
```

首先改变的事就是：我们只有一个叫做 *step()*的方法，而没有大量类似 *connect()，on_connect()，on_read()，do_read()，on_write()，do_write()*等等的成员方法。

方法的主体在 *reenter(this) { for (;;) { }}* 内。你可以把 *reenter(this)*当作我们上次运行的代码，所以这次我们执行的是下一次的代码。

在*reenter*代码块中，你会发现几个*yield*声明。你第一次进入方法时，*async_write*方法被执行，第二次 *async_read_until* 方法被执行，第三次 *service.post* 方法被执行，然后第四次 *async_write* 方法被执行，然后一直循环下去。

你需要一直记住 *for(;;){}*实例。参考下面的代码片段：

```
void step(const error_code & err = error_code(), size_t bytes = 0)
{
    reenter(this)
    {
        yield async_write(sock_, write_buffer_, MEM_FN2(step,_1,_2) );
        yield async_read_until( sock_, read_buffer_,
            "\n",MEM_FN2(step,_1,_2));
        yield service.post(MEM_FN(on_answer_from_server));
    }
}
```

如果我们第三次使用上述的代码片段，我们会进入方法然后执行 *service.post*。当我们第四次进入方法时，我们跳过 *service.post*，不执行任何东西。当执行第五次时仍然不执行任何东西，然后一直这样下去：

```
class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr>, public
coroutine, boost::noncopyable
{
    talk_to_svr(const std::string & username) : ... {}
    void start(ip::tcp::endpoint ep)
    {
        sock_.async_connect(ep, MEM_FN2(step,_1,0) );
    }
    static ptr start(ip::tcp::endpoint ep, const std::string &username)
    {
        ptr new_(new talk_to_svr(username));
        new_->start(ep);
        return new_;
    }
    void step(const error_code & err = error_code(), size_t bytes = 0)
    {
        reenter(this)
        {
            for (;;)
            {
                if ( !started_)
                {
                    started_ = true;
```

```
                std::ostream out(&write_buf_);
                out << "login " << username_ << "\n";
            }
            yield async_write(sock_, write_buf_,MEM_FN2(step,_1,_2));
            yield async_read_until ( sock_,read_buf_,"\n",
                MEM_FN2(step,_1,_2));
            yield service.post(MEM_FN(on_answer_from_server));
        }
    }
    void on_answer_from_server()
    {
        std::istream in(&read_buf_);
        std::string word;
        in >> word;
        if ( word == "login") on_login();
        else if ( word == "ping") on_ping();
        else if ( word == "clients") on_clients();
        read_buf_.consume( read_buf_.size());
        if (write_buf_.size() > 0) service.post(MEM_FN2(step,error_code(),0));
    }
    ...
private:
    ip::tcp::socket sock_;
    streambuf read_buf_, write_buf_;
    bool started_;
    std::string username_;
    deadline_timer timer_;
};
```

当我们启动连接时，*start()*被调用，然后它会异步地连接到服务端。当连接完成时，我们第一次进入 *step()*。也就是我们发送我们登录信息的时候。

在那之后，我们调用 *async_write*，然后调用 *async_read_until*，再处理消息（*on_answer_from_server*）。

我们在 *on_answer_from_server* 处理接收到的消息；我们读取第一个字符，然后把它分发到相应的方法。剩下的消息（如果还有一些消息没读完）我们都忽略掉：

```
class talk_to_svr : ...
{
    ...
    void on_login() { do_ask_clients(); }
```

```
    void on_ping()
    {
        std::istream in(&read_buf_);
        std::string answer; in >> answer;
        if ( answer == "client_list_changed")
            do_ask_clients();
        else postpone_ping();
    }
    void on_clients()
    {
        std::ostringstream clients; clients << &read_buf_;
        std::cout << username_ << ", new client list:" << clients.str();
        postpone_ping();
    }
    void do_ping()
    {
        std::ostream out(&write_buf_); out << "ping\n";
        service.post( MEM_FN2(step,error_code(),0));
    }
    void postpone_ping()
    {
        timer_.expires_from_now(boost::posix_time::millisec(rand() % 7000));
        timer_.async_wait( MEM_FN(do_ping));
    }
    void do_ask_clients()
    {
        std::ostream out(&write_buf_);
        out << "ask_clients\n";
    }
};
```

完整的例子还会更复杂一点，因为我们需要随机地 ping 服务端。实现这个功能我们需要在第一次请求客户端列表完成之后做一个 ping 操作。然后，在每个从服务端返回的 ping 操作的结果中，我们做另外一个 ping 操作。

使用下面的代码片段来执行整个过程：

```
int main(int argc, char* argv[])
{
    ip::tcp::endpoint ep(ip::address::from_string("127.0.0.1"),8001);
    talk_to_svr::start(ep, "John");
    service.run();
}
```

使用协程，我们节约了 15 行代码，而且代码也变的更加易读。

在这里我们仅仅接触了协程的一点皮毛。如果你想要了解更多，请登录作者的个人主页：http://blog.think-async.com/2010_03_01_archive.html。

总结

我们已经了解了如何使用 Boost.Asio 玩转 STL stream 和 streambuf 对象。我们也了解了如何使用协程来让我们的代码更加简洁和易读。

下面就是重头戏了，比如 Asio VS Boost.Asio，高级调试，SSL 和平台相关特性。

第七章

Boost.Asio－进阶话题

这一章对 Boost.Asio 的一些进阶话题进行了阐述。在日常编程中深入研究这些问题是不太可能的，但是知道这些肯定是有好处的：

- 如果调试失败，你需要看 Boost.Asio 能帮到你什么
- 如果你需要处理 SSL，看 Boost.Asio 能帮你多少
- 如果你指定一个操作系统，看 Boost.Asio 为你准备了哪些额外的特性

Asio VS Boost.Asio

Boost.Asio 的作者也保持了 Asio。你可以用 Asio 的方式来思考，因为它在两种情况中都有：Asio（非 Boost 的）和 Boost.Asio。作者声明过更新都会先在非 Boost 中出现，然后过段时间后，再加入到 Boost 的发布中。

不同点被归纳到下面几条：

- Asio 被定义在 *asio::*的命名空间中，而 Boost.Asio 被定义在 *boost::asio::* 中
- Asio 的主头文件是 *asio.hpp*，而 Boost.Asio 的头文件是 *boost/asio.hpp*
- Asio 也有一个启动线程的类（和 *boost::thread* 一样）
- Asio 提供它自己的错误码类(*asio::error_code* 代替 *boost::system::error_code*，然后 *asio:system_error* 代替 *boost::systrem::system_error*)

你可以在这里查阅更多 Asio 的信息：http://think_async.com

你需要自己决定你选择的版本，我选择 Boost.Asio。下面是一些当你做选择时需要考虑的问题：

- Asio 的新版本比 Boost.Asio 的新版本发布要早（因为 Boost 的版本更新比较少）
- Asio 只有头文件（而 Boost.Asio 的部分依赖于其他 Boost 库，这些库可能需要编译）

- Asio 和 Boost.Asio 都是非常成熟的，所以除非你非常需要一些 Asio 新发布的特性，Boost.Asio 是非常保险的选择，而且你也可以同时拥有其他 Boost 库的资源

尽管我不推荐这样，你可以在一个应用中同时使用 Asio 和 Boost.Asio。在允许的情况下这是很自然的，比如你使用 Asio，然后一些第三方库是 Boost.Asio，反之亦然。

调试

调试同步应用往往比调试异步应用要简单。对于同步应用，如果阻塞了，你会跳转进入调试，然后你会知道你在哪（同步意味着有序的）。然而如果是异步，事件不是有序发生的，所以在调试中是非常难知道到底发生了什么的。

为了避免这种情况，首先，你需要深入了解协程。如果实现正确，基本上你一点也不会碰到异步调试的问题。

以防万一，在做异步编码的时候，Boost.Asio 还是对你伸出了援手；Boost.Asio 允许“句柄追踪”，当 *BOOST_ASIO_ENABLE_HANDLER_TRACKING* 被定义时，Boost.Asio 会写很多辅助的输出到标准错误流，纪录时间，异步操作，以及操作和完成处理 handler 的关系。

句柄追踪信息

虽然输出信息不是那么容易理解，但是有总比没有好。Boost.Asio 的输出是 *@asio|||* 。 第一个标签永远都是*@asio*，因为其他代码也会输出到标准错误流（和 *std::error* 相当），所以你可以非常简单的用这个标签过滤从 Boost.Asio 打印出来的信息。*timestamp* 实例从 1970 年 1 月 1 号到现在的秒数和毫秒数。*action* 实例可以是下面任何一种：

- *>n*：这个在我们进入 handler *n* 的时候使用。*description* 实例包含了我们发送给 handler 的参数。
- *<n*：这个在我们退出 handler *n* 的时候使用。
- *!n*：这个当我们因为异常退出 handler *n* 的时候使用。
- *-n*：这个当 handler *n* 在没有调用的情况就退出的时候使用；可能是因为 io_service 实例被删除地太快了（在 *n* 有机会被调用之前）

- *n*m：*这个当 handler* n *创建了一个新的有完成处理 hanlder m* 的异步操作时被调用。*description* 实例展示的就是异步操作开始的地方。当你看到*>m*（开始）和*<m*（结束）时 *completion* 句柄被调用了。
- *n*：就像在 *description* 中展示的一样，这个当 handler *n* 做了一个操作的时候使用（可能是 *close* 或者 *cancel* 操作）。你一般可以忽略这些信息。

当 *n* 是 0 时，操作是在所有（异步）handler 之外被执行的；你经常会在第一个操作时看到这个，或者当你使用的信号量其中一个被触发时。

你需要特别注意类型为*!n* 和*-n* 的信息，这些信息大部分都意味着你的代码有错误。在第一种情形中，异步方法没有抛出异常，所以，异常一定是你自己造成的；你不能让异常跑出你的 *completion* 句柄。第二种情形中，你可能太早就销毁了 *io_service* 实例，在所有完成处理句被调用之前。

一个例子

为了向你展示一个带辅助输出信息的例子，我们修改了在**第六章 Boost.Asio 其他特性** 中使用的例子。你所需要做的仅仅是在包含 *boost/asio.hpp* 之前添加一个*#define*

```
#define BOOST_ASIO_ENABLE_HANDLER_TRACKING
#include <boost/asio.hpp>
...
```

同时，我们也在用户登录和接收到第一个客户端列表时将信息输出到控制台中。输出会如下所示：

```
@asio|1355603116.602867|0*1|socket@008D4EF8.async_connect
@asio|1355603116.604867|>1|ec=system:0
@asio|1355603116.604867|1*2|socket@008D4EF8.async_send
@asio|1355603116.604867|<1|
@asio|1355603116.604867|>2|ec=system:0,bytes_transferred=11
@asio|1355603116.604867|2*3|socket@008D4EF8.async_receive
@asio|1355603116.604867|<2|
@asio|1355603116.605867|>3|ec=system:0,bytes_transferred=9
@asio|1355603116.605867|3*4|io_service@008D4BC8.post
@asio|1355603116.605867|<3|
@asio|1355603116.605867|>4|
John logged in
@asio|1355603116.606867|4*5|io_service@008D4BC8.post
@asio|1355603116.606867|<4|
@asio|1355603116.606867|>5|
```

```
@asio|1355603116.606867|5*6|socket@008D4EF8.async_send
@asio|1355603116.606867|<5|
@asio|1355603116.606867|>6|ec=system:0,bytes_transferred=12
@asio|1355603116.606867|6*7|socket@008D4EF8.async_receive
@asio|1355603116.606867|<6|
@asio|1355603116.606867|>7|ec=system:0,bytes_transferred=14
@asio|1355603116.606867|7*8|io_service@008D4BC8.post
@asio|1355603116.607867|<7|
@asio|1355603116.607867|>8|
John, new client list: John
```

让我们一行一行分析：

- 我们进入 *async_connect*，它创建了句柄 1（在这个例子中，所有的句柄都是 *talk_to_svr::step*）
- 句柄 1 被调用（当成功连接到服务端时）
- 句柄 1 调用 *async_send*，这创建了句柄 2（这里，我们发送登录信息到服务端）
- 句柄 1 退出
- 句柄 2 被调用，11 个字节被发送出去（login John）
- 句柄 2 调用 *async_receive*，这创建了句柄 3（我们等待服务端返回登录的结果）
- 句柄 2 退出
- 句柄 3 被调用，我们收到了 9 个字节（login ok）
- 句柄 3 调用 *on_answer_from_server*（这创建了句柄 4）
- 句柄 3 退出
- 句柄 4 被调用，这会输出 John logged in
- 句柄 4 调用了另外一个 step（句柄 5），这会写入 *ask_clients*
- 句柄 4 退出
- 句柄 5 进入
- 句柄 5，*async_send_ask_clients*，创建句柄 6
- 句柄 5 退出
- 句柄 6 调用 *async_receive*，这创建了句柄 7（我们等待服务端发送给我们已存在的客户端列表）
- 句柄 6 退出
- 句柄 7 被调用，我们接受到了客户端列表
- 句柄 7 调用 *on_answer_from_server*（这创建了句柄 8）
- 句柄 7 退出
- 句柄 8 进去，然后输出客户端列表（*on_clients*）

这需要时间去理解，但是一旦你理解了，你就可以分辨出有问题的输出，从而找出需要被修复的那段代码。

句柄追踪信息输出到文件

默认情况下，句柄的追踪信息被输出到标准错误流（相当于 *std::cerr*）。而把输出重定向到其他地方的可能性是非常高的。对于控制台应用，输出和错误输出都被默认输出到相同的地方，也就是控制台。但是对于一个 windows（非命令行）应用来说，默认的错误流是 null。

你可以通过命令行把错误输出重定向，比如：

```
some_application 2>err.txt
```

或者，如果你不是很懒，你可以代码实现，就像下面的代码片段

```
//  对于 Windows
HANDLE h = CreateFile("err.txt", GENERIC_WRITE, 0, 0, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL , 0);
SetStdHandle(STD_ERROR_HANDLE, h);
// 对于 Unix
int err_file = open("err.txt", O_WRONLY);
dup2(err_file, STDERR_FILENO);
```

SSL

Boost.Asio 提供了一些支持基本 SSL 的类。它在幕后使用的其实是 OpenSSL，所以，如果你想使用 SSL，首先从 www.openssl.org 下载 OpenSSL 然后构建它。你需要注意，构建 OpenSSL 通常来说不是一个简单的任务，尤其是你没有一个常用的编译器，比如 Visual Studio。

假如你成功构建了 OpenSSL，Boost.Asio 就会有一些围绕它的封装类：

- *ssl::stream*：它代替 *ip:::socket* 来告诉你用什么
- *ssl::context*：这是给第一次握手用的上下文
- *ssl::rfc2818_verification*：使用这个类可以根据 RFC 2818 协议非常简单地通过证书认证一个主机名

首先，你创建和初始化 SSL 上下文，然后使用这个上下文打开一个连接到指定远程主机的 socket，然后做 SSL 握手。握手一结束，你就可以使用 Boost.Asio 的 *read*/write**等自由函数。

下面是一个连接到 Yahoo！的 HTTPS 客户端例子：

```
#include <boost/asio.hpp>
#include <boost/asio/ssl.hpp>
using namespace boost::asio;
io_service service;
int main(int argc, char* argv[])
{
    typedef ssl::stream<ip::tcp::socket> ssl_socket;
    ssl::context ctx(ssl::context::sslv23);
    ctx.set_default_verify_paths();
    // 打开一个到指定主机的 SSL socket
    io_service service;
    ssl_socket sock(service, ctx);
    ip::tcp::resolver resolver(service);
    std::string host = "www.yahoo.com";
    ip::tcp::resolver::query query(host, "https");
    connect(sock.lowest_layer(), resolver.resolve(query));
    // SSL 握手
    sock.set_verify_mode(ssl::verify_none);
    sock.set_verify_callback(ssl::rfc2818_verification(host));
    sock.handshake(ssl_socket::client);
    std::string req = "GET /index.html HTTP/1.0\r\nHost: " + host + "\r\nAccept:
        */*\r\nConnection: close\r\n\r\n";
    write(sock, buffer(req.c_str(), req.length()));
    char buff[512];
    boost::system::error_code ec;
    while ( !ec)
    {
        int bytes = read(sock, buffer(buff), ec);
        std::cout << std::string(buff, bytes);
    }
}
```

第一行能很好的自释。当你连接到远程主机，你使用 *sock.lowest_layer()*，也就是说，你使用底层的 socket（因为 *ssl::stream* 仅仅是一个封装）。接下来三行进行了握手。握手一结束，你使用 Booat.Asio 的 *write()*方法做了一个 HTTP 请求，然后读取（*read()*）所有接收到的字节。

当实现 SSL 服务端的时候，事情会变的有点复杂。Boost.Asio 有一个 SSL 服务端的例子，你可以在 *boost/libs/asio/example/ssl/server.cpp* 中找到。

Boost.Asio 的 Windows 特性

接下来的特性只赋予 Windows 操作系统

流处理

Boost.Asio 允许你在一个 Windows 句柄上创建封装，这样你就可以使用大部分的自由函数，比如 *read()，read_until()，write()，async_read()，async_read_until()* 和 *async_write()*。下面告诉你如何从一个文件读取一行：

```
HANDLE file = ::CreateFile("readme.txt", GENERIC_READ, 0, 0, OPEN_ALWAYS,
    FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, 0);
windows::stream_handle h(service, file);
streambuf buf;
int bytes = read_until(h, buf, '\n');
std::istream in(&buf);
std::string line;
std::getline(in, line);
std::cout << line << std::endl;
```

stream_handle 类只有在 I/O 完成处理端口正在被使用的情况下才有效（这是默认情况）。如果情况满足，*BOOST_ASIO_HAS_WINDOWS_STREAM_HANDLE* 就被定义

随机访问句柄

Boost.Asio 允许对一个指向普通文件的句柄进行随机读取和写入。同样，你为这个句柄创建一个封装，然后使用自由函数，比如 *read_at()，write_at()，async_read_at()，async_write_at()*。要从 1000 的地方读取 50 个字节，你需要使用下面的代码片段：

```
HANDLE file = ::CreateFile("readme.txt", GENERIC_READ, 0, 0, OPEN_ALWAYS,
    FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, 0);
windows::random_access_handle h(service, file);
char buf[50];
int bytes = read_at(h, 1000, buffer( buf));
std::string msg(buf, bytes);
std::cout << msg << std::endl;
```

对于 Boost.Asio，随机访问句柄只提供随机访问，你不能把它们当作流句柄使用。也就是说，自由函数，比如：*read()，read_until()，write()*以及他们的相对的异步方法都不能在一个随机访问的句柄上使用。

random_access_handle 类只有在 I/O 完成处理端口在使用中才有效（这是默认情况）。如果情况满足，
BOOST_ASIO_HAS_WINDOWS_RANDOM_ACCESS_HANDLE 就被定义

对象句柄

你可以通过 Windows 句柄等待内核对象，比如修改通知，控制台输入，事件，内存资源通知，进程，信号量，线程或者可等待的计时器。或者简单来说，所有可以调用 *WaitForSingleObject* 的东西。你可以在它们上面创建一个 *object_handle* 封装，然后在上面使用 *wait()*或者 *async_wait()*：

```
void on_wait_complete(boost::system::error_code err) {}
...
HANDLE evt = ::CreateEvent(0, true, true, 0);
windows::object_handle h(service, evt);
// 同步等待
h.wait();
// 异步等待
h.async_wait(on_wait_complete);
```

Boost.Asio POSIX 特性

这些特性只在 Unix 操作系统上可用

本地 socket

Boost.Asio 提供了对本地 socket 的基本支持（也就是著名的 Unix 域 socket）。

本地 socket 是一种只能被运行在主机上的应用访问的 socket。你可以使用本地 socket 来实现简单的进程间通讯，连接两端的方式是把一个当作客户端而另一个当作服务端。对于本地 socket，端点是一个文件，比如*/tmp/whatever*。很酷的一件事情是你可以给指定的文件赋予权限，从而禁止机器上指定的用户在文件上创建 socket。

你可以用客户端 socket 的方式连接，如下面的代码片段：

```
local::stream_protocol::endpoint ep("/tmp/my_cool_app");
local::stream_protocol::socket sock(service);
sock.connect(ep);
```

你可以创建一个服务端 socket，如下面的代码片段：

```
::unlink("/tmp/my_cool_app");
local::stream_protocol::endpoint ep("/tmp/my_cool_app");
local::stream_protocol::acceptor acceptor(service, ep);
local::stream_protocol::socket sock(service);
acceptor.accept(sock);
```

只要 socket 被成功创建，你就可以像用普通 socket 一样使用它；它和其他 socket 类有相同的成员方法，而且你也可以在使用了 socket 的自由函数中使用。

注意本地 socket 只有在目标操作系统支持它们的时候才可用，也就是 *BOOST_ASIO_HAS_LOCAL_SOCKETS*（如果被定义）

连接本地 socket

最终，你可以连接两个 socket，或者是无连接的（数据报），或者是基于连接的（流）：

```
// 基于连接
local::stream_protocol::socket s1(service);
local::stream_protocol::socket s2(service);
local::connect_pair(s1, s2);
// 数据报
local::datagram_protocol::socket s1(service);
local::datagram_protocol::socket s2(service);
local::connect_pair(s1, s2);
```

在内部，*connect_pair* 使用的是不那么著名的 *POSIX socketpair()*方法。基本上它所作的事情就是在没有复杂 socket 创建过程的情况下连接两个 socket；而且只需要一行代码就可以完成。这在过去是实现线程通信的一种简单方式。而在现代编程中，你可以避免它，然后你会发现在处理使用了 socket 的遗留代码时它非常有用。

POSIX 文件描述符

Boost.Asio 允许在一些 POSIX 文件描述符，比如管道，标准 I/O 和其他设备（但是不是在普通文件上）上做一些同步和异步的操作。 一旦你为这样一个 POSIX 文件描述符创建了一个 *stream_descriptor* 实例，你就可以使用一些 Boost.Asio 提供的自由函数。比如 *read()*，*read_until()*，*write()*，*async_read()*，*async_read_until()*和 *async_write()*。

下面告诉你如何从 stdin 读取一行然后输出到 stdout：

```
size_t read_up_to_enter(error_code err, size_t bytes) { ... }
posix::stream_descriptor in(service, ::dup(STDIN_FILENO));
posix::stream_descriptor out(service, ::dup(STDOUT_FILENO));
char buff[512];
int bytes = read(in, buffer(buff), read_up_to_enter);
write(out, buffer(buff, bytes));
```

stream_descriptor 类只在目标操作系统支持的情况下有效，也就是 *BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR*（如果定义了）

Fork

Boost.Asio 支持在程序中使用 *fork()*系统调用。你需要告诉 *io_service* 实例 *fork()* 方法什么时候会发生以及什么时候发生了。参考下面的代码片段：

```
service.notify_fork(io_service::fork_prepare);
if (fork() == 0)
{
    // 子进程
    service.notify_fork(io_service::fork_child);
    ...
} else
{
    // 父进程
    service.notify_fork(io_service::fork_parent);
    ...
}
```

这意味着会在不同的线程使用即将被调用的 *service*。尽管 Boost.Asio 允许这样，我还是强烈推荐你使用多线程，因为使用 *boost::thread* 简直就是小菜一碟。

总结

为简单明了的代码而奋斗。学习和使用协程会最小化你需要做的调试工作，但仅仅是在代码中有潜在 bug 的情况下，Boost.Asio 才会伸出援手，这一点在关于调试的章节中就已经讲过。

如果你需要使用 SSL，Boost.Asio 是支持基本的 SSL 编码的

最终，如果已经知道应用是针对专门的操作系统的，你可以享用 Boost.Asio 为那个特定的操作系统准备的特性。

第八章

结束

就目前来说，网络编程是非常重要的。作为 21 世纪所有 C++程序员的必学内容，我们对 Boost.Asio 的理论进行了深入理解并付诸实践。因为本书的内容都可以很简单的进行阅读、测试、理解和扩展，所以你可以把它当作一个参考以及便携的 Boost.Asio 样例库。

最后，希望你能以读本书为乐，以编程为乐。

https://mmoaay.gitbooks.io/boost-asio-cpp-network-programming-chinese/content/index.html