Module05-09
C++ Boost：网络

C++ Boost - 网络

- 容器相关
- 字符串和文字处理
- 正则表达式
- 智能指针
- 函数对象相关
- 序列化
- 日期与时间
- 多线程
- ➔ 网络

网络 - 网络基本概念

- 网络：
  - 网络基本概念
  - Asio 核心概念
  - Asio 编程指南
  - 核心接口描述

网络 - 网络基本概念

- 网络基本概念

  网络相关的概念的介绍，请参考书籍：《 TCP/IP Network Administration, 3rd Edition 》前 2 个章节

  - Chapter 1. Overview of TCP/IP
    本章介绍了 ISO/OSI 七层模型、 TCP/IP 架构，以及各种协议的概况
  - Chapter 2. Delivering the Data
    本章介绍了数据传递的细节

  (该书作者： Craig Hunt)

网络 - 网络基本概念

- 网络：
  - 网络基本概念
  - Asio 核心概念
  - Asio 编程指南
  - 核心接口描述

网络 - Asio 核心概念

- Asio 的核心概念

  在使用 Boost.Asio 进行网络编程之前，我们需要了解下面几个概念：

  - Boost.Asio 剖析
  - 前摄器 (Proactor) 模式： 无线程并发
  - 线程和 Boost.Asio
  - Strands: 无明显加锁机制的线程调用
  - 缓冲区 (Buffers)
  - 流、 Short Reads 和 Short Writes

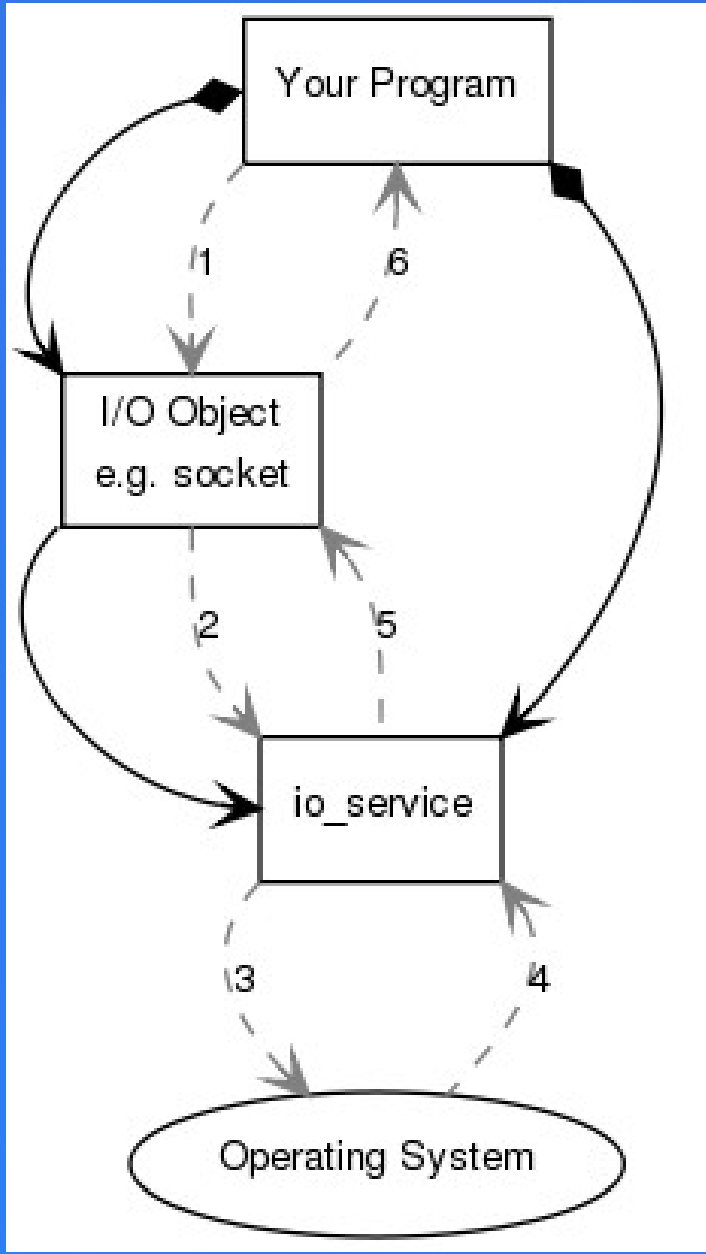
网络 - Asio 核心概念

- Asio 剖析
  - Boost.Asio 支持对 I/O 对象，如 Socket 执行同步 I/O 和异步 I/O

什么是同步 I/O，什么是异步 I/O ？

网络 - Asio 核心概念

- Asio 剖析 - 同步 I/O
  - 应用程序至少需要一个 io_service 对象，该对象表示程序连接到操作系统的 I/O 组件：
    boost::asio::io_service io_service;
  - 应用程序还需一个 I/O 对象如 TCP Socket 来执行 I/O 操作：
    boost::asio::ip::tcp::socket socket(io_service);



- Asio 剖析 - 同步 I/O（续 1 ）
  - 当执行一个同步连接后，将出现以下一系列的事件：
    - 1，程序通过调用 I/O 对象来发起连接：socket.connect(server_endpoint);
    - 2， I/O 对象将请求转发给 io_service
    - 3， io_service 通知操作系统执行连接操作
    - 4，操作系统将执行的结果返回给 io_service
    - 5， io_service 将操作过程中的错误码翻译成 boost::system::error_code，将结果转发给 I/O 对象如：TCP Socket

- Asio 剖析 - 同步 I/O（续 2）
  - 当执行一个同步连接后，将出现以下一系列的事件（续）：
    - 6，如果操作失败， io_service 将抛出一个异常

```
boost::system::system_error;
```

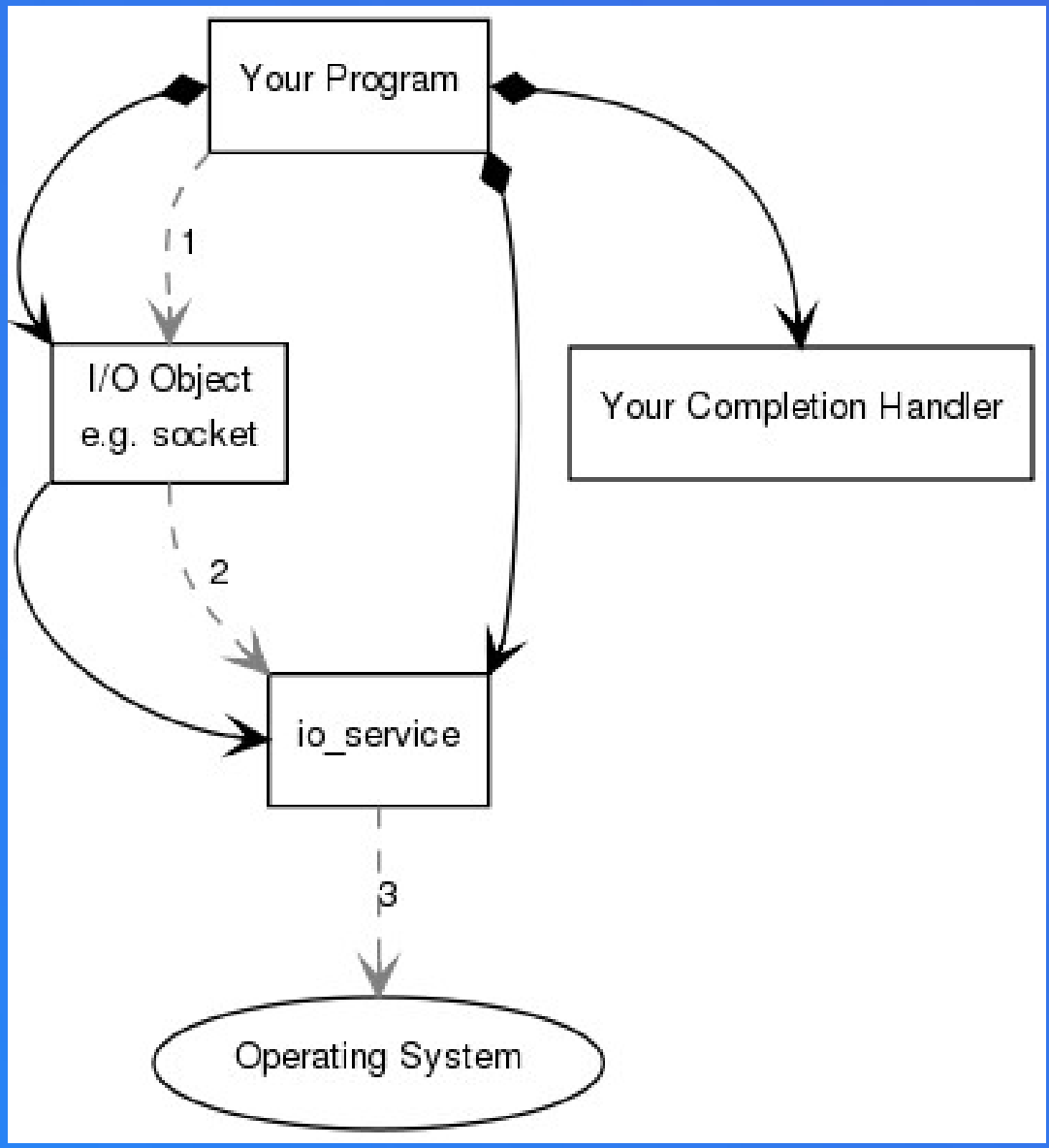
但是，如果连接的操作是按以下方式执行：

```
boost::system::error_code ec;
socket.connect(server_endpoint, ec);
```

结果被保存在 ec 中，这样就不会抛出异常。

网络 - Asio 核心概念

- Asio 剖析 - 异步 I/O
  - 图例（发起连接）



网络 - Asio 核心概念

- Asio 剖析 - 异步 I/O（续 1）
  - 1，应用程序通过调用 I/O 对象发起连接：

```
socket.async_connect(server_endpoint,
your_completion_handler);
```

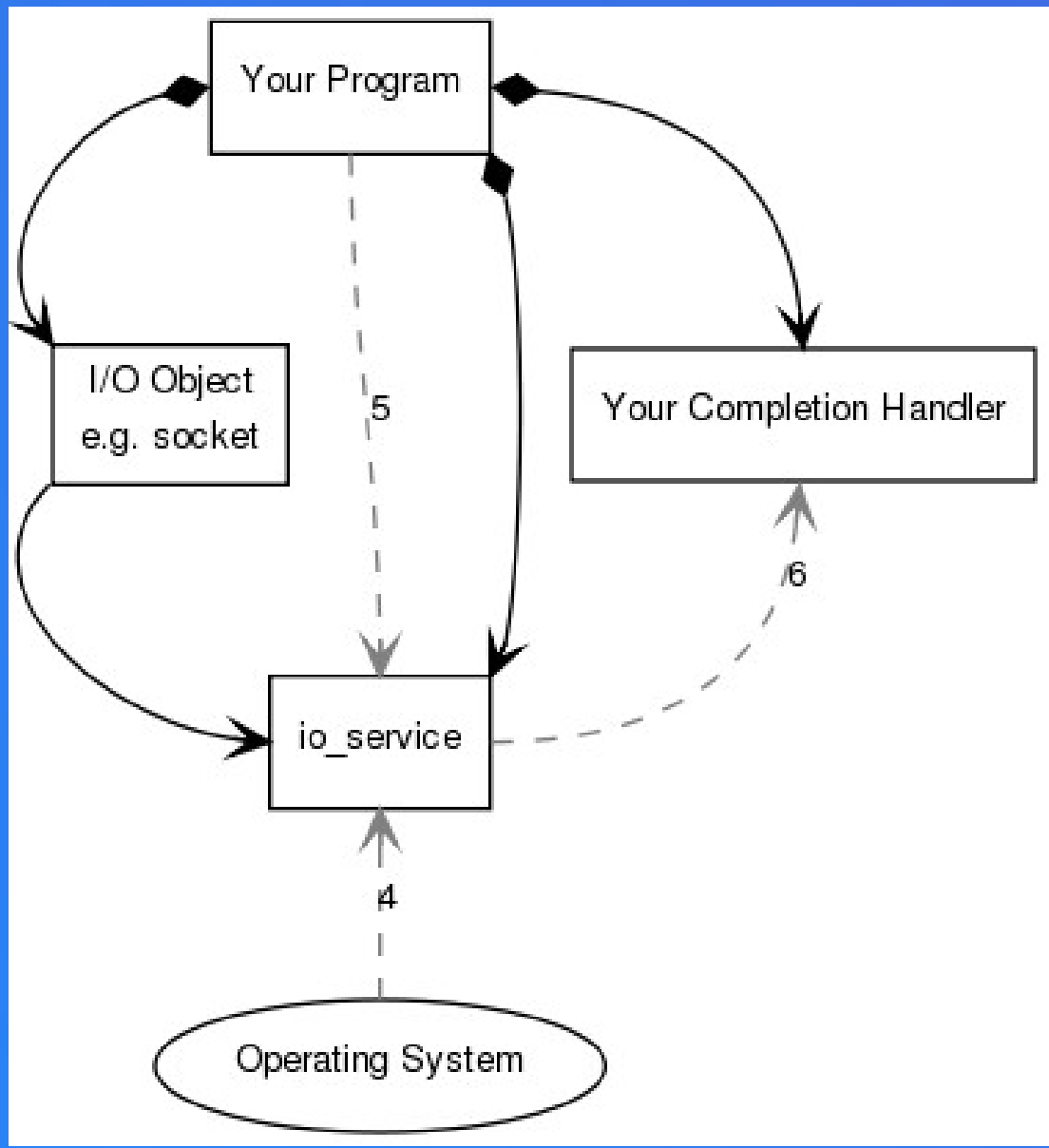
其中 your_completion_handler 是一个函数对象或函数，类似如下：

```
void your_completion_handler(const
boost::system::error_code& ec);
```

  - 2， I/O 对象将请求转发给 io_service
  - 3， io_service 告诉操作系统说它要发起一个异步连接

过了一会儿 ...（如下页图例所示）

- Asio 剖析 - 异步 I/O（续 2
  - 图例（完成处理器）

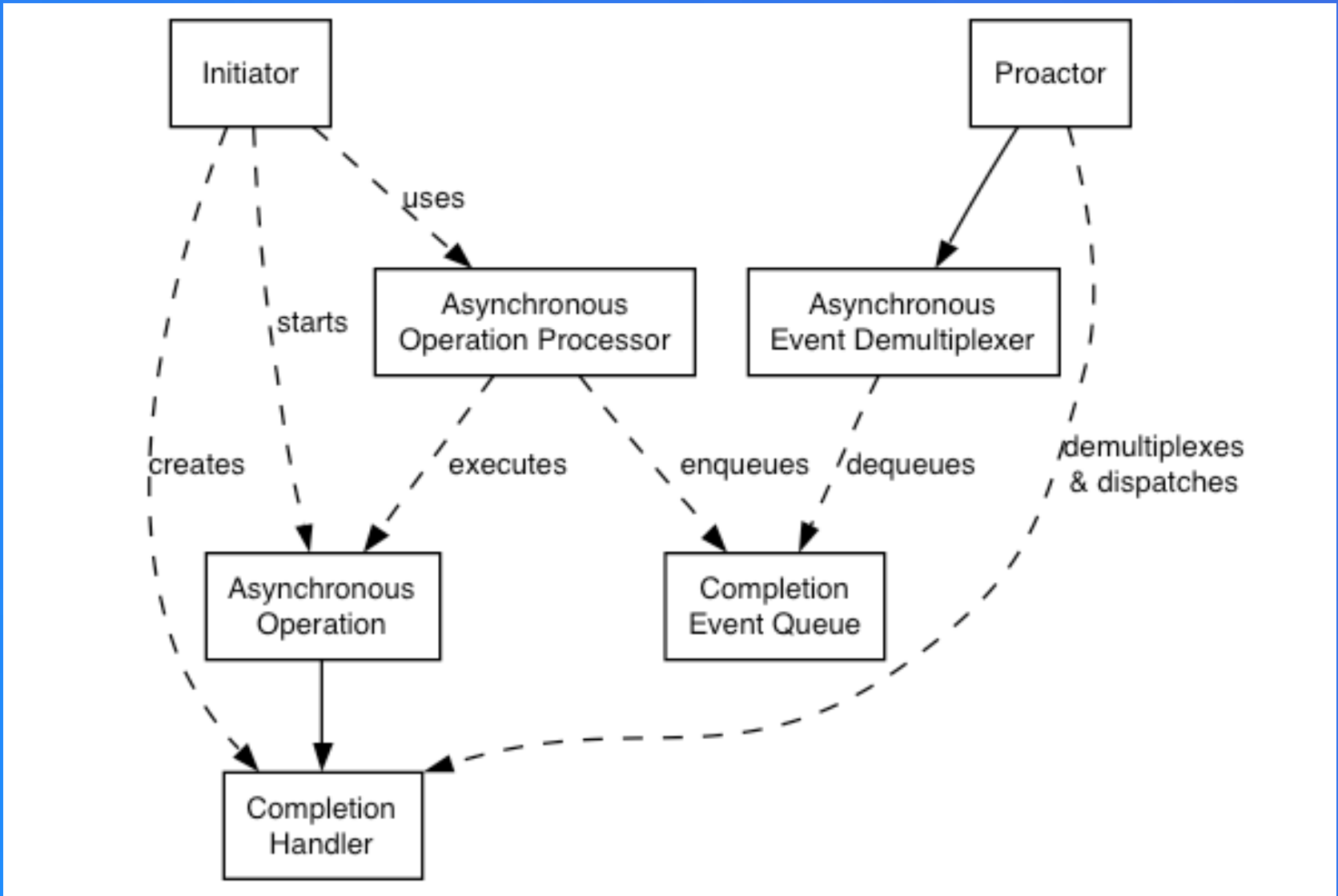


网络 - Asio 核心概念

- Asio 剖析 - 异步 I/O （续 1 ）
  - 4，操作系统完成了连接的操作，并且将结果放到一个 queue 中，供 io_service 取用
  - 5，为了能够获取上述操作的结果，应用程序必须调用 io_service::run() 。

    io_service::run() 操作将阻塞至所有异步操作结束，所以通常情况下，在我们发起第一个异步操作后就调用 io_service::run()
  - 6， io_service::run() 将操作结果从操作系统的 queue 中取出，并将其转换成 error_code ，再将其传给应用程序

- Proactor 模式简介



- Proactor 模式简介（续 1 ）

  Boost.Asio 的异步 I/O 基于 Proactor 模式

  - Asynchronous Operation （异步操作）
    - 如异步的读 / 写操作
  - Asynchronous Operation Processor （异步操作处理器）
    - 执行异步操作，且在操作完成时将事件放入完成事件队列 (Completion Event Queue)，在 asio 中， stream_socket_service 便是一个异步操作处理器
  - Completion Event Queue （完成事件队列）
    - 用于暂放完成事件，直到他们被异步事件多路分离器 (Asynchronous Event Demultiplexer) 取出
  - Completion Handler （完成处理器）
    - 处理异步操作的结果，他们通常是由 boost::bind() 创建的函数对象

网络 - Asio 核心概念

- Proactor 模式简介（续 2 ）
  - Asynchronous Event Demultiplexer （异步事件多路分离器）
    - 阻塞等待完成事件队列 (completion event queue) 中的事件，且返回完成事件给其调用者
  - Proactor （前摄器）
    - 调用异步事件分离器 (Asynchronous Event Demultiplexer) 来获取完成事件，并分派与该事件关联的完成处理器 (completion handler)(比如调用某个函数对象)。前摄器在 Asio 中是类 io_service
  - Initiator （发起者）
    - 发起异步操作的代码

网络 - Asio 核心概念

- 线程与 Asio
  - 线程安全：
    - 通常而言，对于不同对象的并发访问是安全的，但是对于单个对象的并发访问不是线程安全的。
    - 不过， io_service 则为安全的并发访问单个对象提供了强有力的保障
  - 线程池
    - 多个线程可以调用 io_service::run() 来建立一个线程池

网络 - Asio 核心概念

- 线程与 Asio（续 1）
  - 内部线程：
    - 在某些特定平台的实现中，可能需要一个或多个线程来模拟异步机制。尽可能的不要将这些线程暴露给库的调用者，特别是：
      - 不要在直接调用用户代码，并且：
      - 阻塞所有信号 (Signals)

网络 - Asio 核心概念

- Strands： 不需显式加锁的使用线程
  - 简单而言， strand 是将多线程串行化 (非并行) 的一种手段，使用 strand 可以在多线程代码中不必显式加锁 (如不显式使用 mutex 等)
  - strand 可以显式或隐式的使用：
    - 在单线程中调用 io_service::run() 表示所有事件处理器都在一个隐式的 strand 下执行，因为 io_service 保证这些事件处理器只在 io_service::run() 函数内部被调用
    - 当在一个连接中进行的一个异步操作链不可能出现事件处理器的并发执行的情形，这是一个隐式的 strand
    - 一个 io_service::strand 的实例就是一个显式的 strand ， 所有的事件处理器函数对象需要通过 io_service::strand::wrap() 进行包装，或者通过 io_service::strand 对象进行 post 或 dispatch

网络 - Asio 核心概念

- Buffer
  - 在 I/O 中 Buffer 是一种支持“散读聚写” (scatter-read & gather-write) 机制的基础设施、
  - 在 boost.Asio 中提供两种类别的 Buffer：
    - mutable_buffer
    - const_buffer

网络 - Asio 核心概念

- 流、 short read 与 short write
  - 在 boost.Asio 中，很多 I/O 对象是基于流的，这意味着：
    - 没有消息边界，数据在一个连续的字节流中传输
    - 读、写操作有可能会出现传输字节数比请求的少，这就是所谓的 short read 和 short write
  - 基于流的模式的类型：
    - SyncReadStream，通过调用成员函数 read_some() 执行同步读取操作
    - AsyncReadStream，通过调用成员函数 async_read_some() 执行异步读取操作
    - SyncWriteStream，通过调用成员函数 write_some() 执行同步写入操作
    - AsyncWriteStream，通过调用成员函数 async_write_some() 执行异步写入操作

网络 - Asio 核心概念

- 流、 short read 与 short write（续）
  - 典型的基于流操作的 I/O 对象：
    - ip::tcp::socket
    - ssl::stream<>
    - posix::stream_descriptor
    - windows::stream_handle

网络 - Asio 编程指南

- 网络：
  - 网络基本概念
  - Asio 核心概念
  - Asio 编程指南
  - 核心接口描述

网络 - Asio 编程指南

- Boost.Asio 编程指南
  - 通过代码来剖析 Asio 的编程套路：
    - 1， TCP 同步服务器
    - 2， TCP 同步客户端
    - 3， TCP 异步服务器
    - 4， TCP 异步客户端
    - 5， UDP 同步服务器
    - 6， UDP 同步客户端
    - 7， UDP 异步服务器
    - 8， UDP 异步客户端
    - 9，基于 asio::streambuf 的 I/O
    - 10， 定时器 (Timer)

网络 - TCP 同步服务器 S01A

- 服务器 S01A 功能介绍
  - 简介：服务器提供 3 种服务：告知当前时间、告知服务器端当前登录的用户、获取服务器内核、硬件平台等信息；客户端可以选择其中一种，服务器根据客户端的选择做出正确的回应。
  - 流程描述：
    - 客户端连接到服务器
    - 服务器端将服务列表发送给客户端
      - 列表大致如下：
        - 1. Get Time
        - 2. Who's Online
        - 3. System Info
    - 客户端选择指定的服务号发送给服务器端
    - 服务器端将相应的结果发送给客户端后，关闭这条连接
    - 客户端打印结果，退出程序

网络 - TCP 同步服务器 S01A

- 服务器 S01A 主要代码：

```
int main() {
    try {
        boost::asio::io_service io_service; // #1
        tcp::acceptor acceptor(io_service,
             tcp::endpoint(tcp::v4(), 8868));  // #2
        for (;;) {
            tcp::socket socket(io_service);  // #3
            acceptor.accept(socket); // #4
            // 1, send service list to client
            boost::system::error_code ignored_error;
            boost::asio::write(socket,
                 boost::asio::buffer(serviceList),
                boost::asio::transfer_all(), ignored_error);

            // 2, receive selection from client
            char selection[20];
            size_t n =
                socket.read_some(boost::asio::buffer(selection),
                                 ignored_error);
```

网络 - TCP 同步服务器 S01A

- 服务器 S01A 主要代码：

```
            // 3, send response
            string response = getServiceContent(atoi(selection));
            boost::asio::write(socket,
boost::asio::buffer(response),
                    boost::asio::transfer_all(), ignored_error);
        } // #6
    } catch (std::exception& e) {
        std::cerr << e.what() << std::endl;
    }
    return 0;
}
```

网络 - TCP 同步服务器 S01A

- 服务器 S01A 创建步骤：
  - #1，创建 io_service 实例
  - #2，创建 tcp::acceptor 实例，关联到 io_service
  - #3，创建 I/O 对象 tcp::socket，关联到 io_service
  - #4，调用 tcp::acceptor 实例的 accept() 方法，在 I/O 对象 tcp::socket 上同步接受来自客户端的连接
  - #5，根据业务逻辑的需要，进行一系列的同步 read() 和 write() 操作
  - #6，到该处，出了作用域，tcp::socket 对象销毁，代表服务器与客户端的 I/O 通道关闭

网络 - TCP 同步服务器 S01A

- 服务器 S01A 点评：
  - 优点：
    - 编程简单、明了，易于理解
  - 显著的缺点：
    - 属于迭代式事务处理，同一个时刻只能服务于一个客户端，如果当前客户端的事务未处理完毕，其它客户端将处于阻塞状态（无法连接上该服务器）
    - 所以，仅适合于并发访问不大，且事务处理简单、占用时间短的服务，如 date_time service 、 echo service 等等

网络 - TCP 同步服务器 S01A

- 服务器 S01A 第一次改进：
  - 目标：
    - 能够同时处理来自多个不同客户端的连接和消息来往
  - 分析：
    - 由于在第一个版本中，由于自始自终只有一个线程在执行所有操作： accept ， read ， write 等
    - 所以我们可以在循环中不断接受 (accept) 来自客户端的连接，但将后续的操作 (read,write 等 I/O 操作) 集中到一个线程中处理，一个客户端一个连接 (一个线程)

网络 - TCP 同步服务器 S01A

- 服务器 S01A 第一次改进的代码：

```
void handler(boost::shared_ptr<tcp::socket> socket) {
    // 1, send service list to client
    boost::system::error_code ignored_error;
    boost::asio::write(*socket, boost::asio::buffer(serviceList),
            boost::asio::transfer_all(), ignored_error);

    // 2, receive selection from client
    char selection[20] = "";
    socket->read_some(boost::asio::buffer(selection),
ignored_error);

    // 3, send response
    string response = getServiceContent(atoi(selection));
    boost::asio::write(*socket, boost::asio::buffer(response),
            boost::asio::transfer_all(), ignored_error);
}
```

网络 - TCP 同步服务器 S01A

- 服务器 S01A 第一次改进的代码（续）：

```
int main() {
    try {
        boost::asio::io_service io_service; // #1
        // 创建一个接受器，用于接受来自客户端的连接
        tcp::acceptor acceptor(io_service,
                               tcp::endpoint(tcp::v4(), 8868)); // #2
        for (;;) {
            // 创建一个socket对象，是服务器与客户端的通信通道
            boost::shared_ptr<tcp::socket> socket(new
tcp::socket(io_service)); // 注意：是socket指针
            // 如果接受到来自客户端发起的连接，将该连接与socket对象关联起来
            acceptor.accept(*socket); // #4
            // 为每个客户端创建一个处理线程
            boost::thread t(handler, socket); // #5
        } // #6
    } catch (std::exception& e) {
        std::cerr << e.what() << std::endl;
    }
}
```

网络 - TCP 同步服务器 S01A

- 服务器 S01A 第二次改进需求：
  - 当前的版本中，客户端连接到服务器后只能进行一个回合的交流，如果需要保持长连接，不断的交互，需要进行第二次改进，
  - 另外对 ifstream 对象的操作不是线程安全的
  - 以上两处需要改进，留作课下练习

网络 - TCP 同步客户端 C01A

- 客户端 C01A 功能介绍
  - 简介：该客户端负责发起到服务器 S01A 的连接，并向服务器请求特定的服务（具体 service 见 S01A 的描述）。

网络 - TCP 同步客户端 C01A

- 客户端 C01A 核心代码

```
int main(int argc, char* argv[]) {
    try {
        // 1, 创建io_service对象
        boost::asio::io_service io_service;

        // 2, 创建resolver对象关联到io_service对象
        tcp::resolver resolver(io_service);

        // 3, 创建一个查询对象
        tcp::resolver::query query("localhost", "8868");

        // 4, 用resolver对象和查询对象获取可用服务器地址
        tcp::resolver::iterator endpoint_iterator =
            resolver.resolve(query);
        tcp::resolver::iterator end;

        // 5, 创建tcp::socket对象, 关联到io_service
        tcp::socket socket(io_service);
```

网络 - TCP 同步客户端 C01A

- 客户端 C01A 核心代码（续 1 ）

```
// 6, socket对象发起到服务器端的同步连接操作
boost::system::error_code error =
     boost::asio::error::host_not_found;
while (error && endpoint_iterator != end) {
    socket.close();
    socket.connect(*endpoint_iterator++, error);
}
if (error) // 如果没有一个地址能连接成功，则抛出异常
    throw boost::system::system_error(error);

// 7, 一系列 同步read()和write()
char buf[512];
// receive service list from server
size_t len =
    socket.read_some(boost::asio::buffer(buf), error);
buf[len] = '\0';
cout << buf;

string selection;
cin >> selection;
```

网络 - TCP 同步客户端 C01A

- 客户端 C01A 核心代码（续 2 ）

```
        // send selection to server
        boost::asio::write(socket,
                           boost::asio::buffer(selection),
                           boost::asio::transfer_all(), error);

        // receive response from server
        len = socket.read_some(boost::asio::buffer(buf), error);
        buf[len] = '\0';
        cout << buf;
    } catch (std::exception& e) {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}
```

网络 - TCP 同步读取操作接口

- read() 和 receive()
  - tcp::socket 成员函数：

```
template<typename MutableBuffer>
std::size_t receive(const MutableBuffer& buffers);
template<typename MutableBuffer>
std::size_t receive(const MutableBuffer& buffers,
        socket_base::message_flags flags);
template<typename MutableBuffer>
std::size_t receive(const MutableBuffer& buffers,
        socket_base::message_flags flags,
boost::system::error_code& ec);

template<typename MutableBuffer>
size_t read_some(const MutableBuffer& buffers);
template<typename MutableBuffer>
size_t read_some(const MutableBuffer& buffers,
boost::system::error_code& ec);
```

注意： tcp::socket 没有名为 read() 的成员函数 .

网络 - TCP 同步读取操作接口

- read() 和 receive() （续 1 ）
  - 6 个 read() 自由函数：

```
template<typename SyncReadStream,
         typename MutableBufferSequence,
         typename CompletionCondition>
std::size_t read(SyncReadStream& s,
                 const MutableBufferSequence& buffers,
                 CompletionCondition completion_condition,
                 boost::system::error_code& ec);
template<typename SyncReadStream,
         typename MutableBufferSequence>
inline std::size_t read(SyncReadStream& s,
                 const MutableBufferSequence& buffers);
template<typename SyncReadStream,
         typename MutableBufferSequence,
         typename CompletionCondition>
inline std::size_t read(SyncReadStream& s,
                 const MutableBufferSequence& buffers,
                 CompletionCondition completion_condition);
```

网络 - TCP 同步读取操作接口

- read() 和 receive() （续 2 ）
  - 6 个 read() 自由函数（续）：

```
template<typename SyncReadStream, typename Allocator,
        typename CompletionCondition>
std::size_t read(SyncReadStream& s,
                 boost::asio::basic_streambuf<Allocator>& b,
                 CompletionCondition completion_condition,
                 boost::system::error_code& ec);

template<typename SyncReadStream, typename Allocator>
inline std::size_t read(SyncReadStream& s,
                 boost::asio::basic_streambuf<Allocator>& b);

template<typename SyncReadStream, typename Allocator,
        typename CompletionCondition>
inline std::size_t read(SyncReadStream& s,
                 boost::asio::basic_streambuf<Allocator>& b,
                 CompletionCondition completion_condition);
```

网络 - TCP 同步读取操作接口

- read() 和 receive()（续）
  - tcp::socket 成员函数和自由函数 read() 的区别：
    - 6 个自由函数阻塞至：
      - 读取指定的 bytes 数，或
      - 遇到错误，或
      - 符合指定的条件

      才返回。
    - tcp::socket 成员函数则是从 tcp 流中读取 1 到多个字节（即本次 I/O 可用的实际字节数），或遇到错误即返回

网络 - TCP 同步写入操作接口

- write() 和 send()
  - tcp::socket 成员函数：

```
template<typename ConstBuffer>
std::size_t send(const ConstBuffer& buffers);
template<typename ConstBuffer>
std::size_t send(const ConstBuffer& buffers,
        socket_base::message_flags flags);
template<typename ConstBuffer>
std::size_t send(const ConstBuffer& buffers,
        socket_base::message_flags flags,
        boost::system::error_code& ec);

template<typename ConstBuffer>
std::size_t write_some(const ConstBuffer& buffers);
template<typename ConstBuffer>
std::size_t write_some(const ConstBuffer& buffers,
        boost::system::error_code& ec);
```

注意：tcp::socket 没有名为 write() 的成员函数 .

网络 - TCP 同步写入操作接口

- write() 和 send()（续 1）
  - 6 个 write() 自由函数：

```
template <typename SyncWriteStream, typename ConstBuffer>
std::size_t write(SyncWriteStream& s,
        const ConstBuffer& buffers);

template <typename SyncWriteStream, typename ConstBuffer,
    typename CompletionCondition>
std::size_t write(SyncWriteStream& s,
        const ConstBuffer& buffers,
        CompletionCondition completion_condition);

template <typename SyncWriteStream, typename ConstBuffer,
    typename CompletionCondition>
std::size_t write(SyncWriteStream& s,
        const ConstBuffer& buffers,
        CompletionCondition completion_condition,
        boost::system::error_code& ec);
```

网络 - TCP 同步写入操作接口

- write() 和 send()（续 2）
  - 6 个 write() 自由函数：

```
template <typename SyncWriteStream, typename Allocator>
std::size_t write(SyncWriteStream& s,
        basic_streambuf<Allocator>& b);

template <typename SyncWriteStream, typename Allocator,
    typename CompletionCondition>
std::size_t write(SyncWriteStream& s,
        basic_streambuf<Allocator>& b,
        CompletionCondition completion_condition);

template <typename SyncWriteStream, typename Allocator,
    typename CompletionCondition>
std::size_t write(SyncWriteStream& s,
        basic_streambuf<Allocator>& b,
        CompletionCondition completion_condition,
        boost::system::error_code& ec);
```

网络 - TCP 同步写入操作接口

- 自由函数 write() 和 read() 的 CompletionCondition
  - 自由函数 write() 和 read() 的某些版本有一个参数是 CompletionCondition，该参数是一个函数对象或函数，符合下面形式：

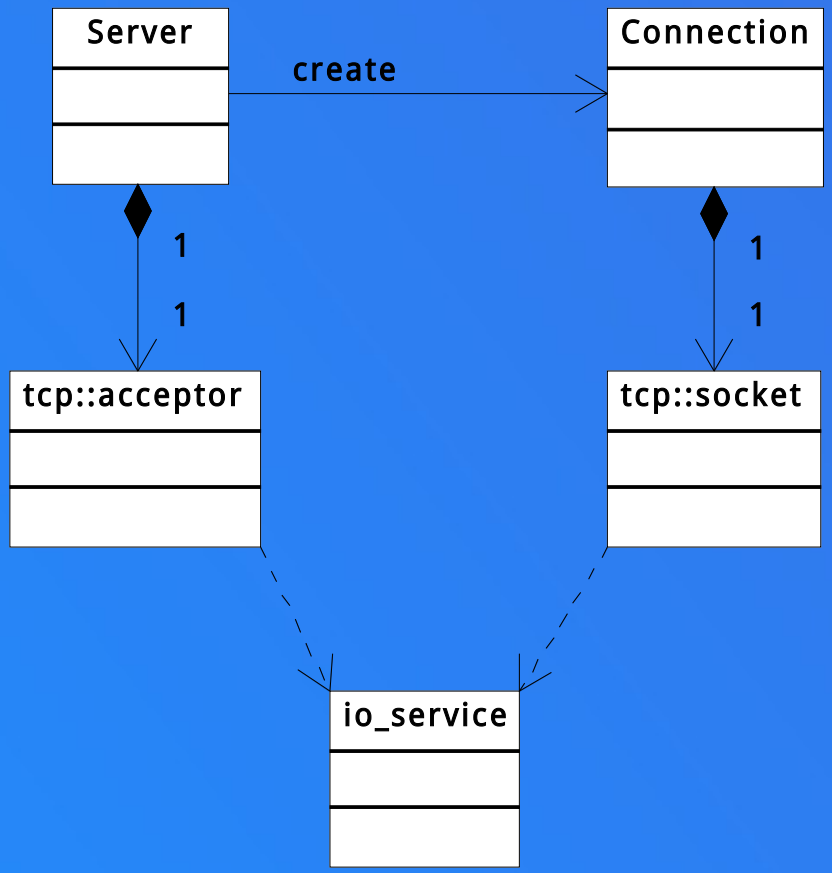
```
std::size_t completion_condition(
  // Result of latest write_some operation.
  const boost::system::error_code& error,

  // Number of bytes transferred so far.
  std::size_t bytes_transferred
);
```

  - 两个预定义的 completion_condition：
    - transfer_all_t，协助函数 transfer_all() 即返回该函数对象
    - transfer_at_least_t，协助函数 transfer_at_least() 即返回该函数对象

网络 - TCP 异步服务器 S02A

- 服务器 S02A 功能介绍
  - 简介：该服务器是一个基于异步 I/O 的 echo server，将客户端的消息原样返回给客户端。
  - 主要参与者：



网络 - TCP 异步服务器 S02A

- 该异步服务器实现的套路
  - Server 持有一个 tcp::acceptor 的实例，通过不断的调用 acceptor 的 async_accept()，异步接受来自客户端的连接，并创建 Connection 对象与该连接关联
  - Connection 对象拥有一个 I/O 对象 tcp::socket， Connection 对象创建后，通过相互循环调用 tcp::socket 的 async_read() 和 async_write() 一族读 / 写函数进行不断的异步读写操作
  - 所有的 I/O 事件的通知、完成事件处理器的调度都有 io_service 负责

网络 - TCP 异步服务器 S02A

- 服务器 S02A 代码：类 Connection

```
class Connection:
    public boost::enable_shared_from_this<Connection> {
public:
    Connection(boost::asio::io_service& service) :
        sock(service) {
    }

    void start() {
        sock.async_read_some(boost::asio::buffer(buf),
            boost::bind(&Connection::handleRead, //  #1
                        shared_from_this(),
                        boost::asio::placeholders::error));
    }

    tcp::socket& getSocket() {
        return sock;
    }
```

- 服务器 S02A 代码：类 Connection（续 1 ）

```
private:
    void handleRead(const boost::system::error_code& error) {
        if (!error) {
            boost::asio::async_write(sock,
                    boost::asio::buffer(buf),
                    boost::bind(&Connection::handleWrite,  // #2
                            shared_from_this(),
                            boost::asio::placeholders::error));
        }
    }
    void handleWrite(const boost::system::error_code& error) {
        if (!error) {
            memset(buf, 0, 512); // 注意: 重置buf
            sock.async_read_some(boost::asio::buffer(buf),
                boost::bind(&Connection::handleRead,  // #3
                            shared_from_this(),
                            boost::asio::placeholders::error));
        }
    }
```

网络 - TCP 异步服务器 S02A

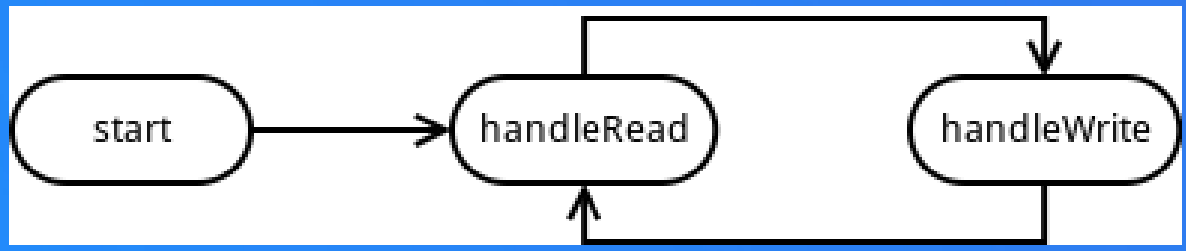
- 服务器 S02A 代码：类 Connection（续 2 ）

```
private:
    tcp::socket sock;
    char buf[512];
};

typedef boost::shared_ptr<Connection> ConnectionPtr;
```

网络 - TCP 异步服务器 S02A

- 服务器 S02A 代码：类 Connection 代码说明
  - #1，在 start() 函数中调用异步 I/O 函数 async_read_some()，将 Connection::handleRead() 成员函数作为完成处理器（即当 async_read_some() 操作完成后， io_service 将调度 ConnectionRead() ）
  - #2，在 Connection::handleRead() 成员函数中调用 async_write() 进行异步写操作，且将 Connection::handleWrite() 作为完成处理器
  - #3，在 Connection::handleWrite() 成员函数中调用 async_read_some() 进行异步读操作，且将 Connection::handleRead() 作为完成处理器



网络 - TCP 异步服务器 S02A

- 服务器 S02A 代码：类 Server

```
class Server {
public:
    Server(boost::asio::io_service& service) :
        acceptor(service, tcp::endpoint(tcp::v4(), 8868)) {
        start();
    }

private:
    void start() {
        ConnectionPtr conn(
                    new Connection(acceptor.io_service()));
        acceptor.async_accept(conn->getSocket(), boost::bind(
                &Server::handleAccept, this, conn,
                boost::asio::placeholders::error));
    }
```

网络 - TCP 异步服务器 S02A

- 服务器 S02A 代码：类 Server

```
    void handleAccept(ConnectionPtr con,
            const boost::system::error_code& error) {
        if (!error) {
            con->start();
            start();
        }
    }

private:
    tcp::acceptor acceptor;
};
```

网络 - TCP 异步服务器 S02A

- 服务器 S02A 代码： main() 函数

```
int main() {
    try {
        boost::asio::io_service service;
        Server server(service);
        service.run();  // 注意：与同步I/O不同，异步I/O需要调用run()
    } catch (exception& e) {
        cout << e.what() << endl;
    }
}
```

网络 - TCP 异步服务器 S02A

- 服务器 S02A 点评：
  - 与同步服务器相比，异步服务器的编程相对复杂，由于是异步 I/O ，每个异步 I/O 的方法都需要提供一个完成处理器
  - S02A 的一些局限：
    - 目前的逻辑严格按照消息一来一往的方式，即接收客户端的消息，且立即回送该消息，没办法打破这个顺序，所以不适合应对读写次数不匹配的场合

网络 - TCP 异步客户端 C02A

- 客户端 C02A 实现的功能
  - 不断发送消息给服务器端，并将服务器端的回应打印到屏幕

网络 - TCP 异步客户端 C02A

- 客户端 C02A 代码：类 Client

```
class Client {
public:
    Client(boost::asio::io_service& service,
           tcp::resolver::iterator endpointIterator) :
        sock(service) {
        tcp::endpoint endpoint = *endpointIterator;
        sock.async_connect(endpoint,
                boost::bind(&Client::handleConnect, this,
                 boost::asio::placeholders::error,
                ++endpointIterator));
    }
```

网络 - TCP 异步客户端 C02A

- 客户端 C02A 代码：类 Client（续 1）

```
private:
    void handleConnect(const boost::system::error_code& error,
            tcp::resolver::iterator endpointIterator) {
        if (!error) {
            char msg[BUF_SIZE] = { };
            cin.getline(msg, BUF_SIZE);
            cout << strlen(msg) << endl;
            boost::asio::async_write(sock,
                    boost::asio::buffer(msg, strlen(msg)),
                    boost::bind(&Client::handleWrite, this,
                            boost::asio::placeholders::error));
        else if (endpointIterator != tcp::resolver::iterator()) {
            sock.close();
            tcp::endpoint endpoint = *endpointIterator;
            sock.async_connect(endpoint,
                    boost::bind(&Client::handleConnect,
                    this, boost::asio::placeholders::error,
                    ++endpointIterator));
        }
    }
```

网络 - TCP 异步客户端 C02A

- 客户端 C02A 代码：类 Client（续 2）

```
void handleRead(const boost::system::error_code& error) {
    if (!error) {
        cout << buf << endl; // print received message
        char msg[BUF_SIZE] = { };
        cin.getline(msg, BUF_SIZE);
        cout << strlen(msg) << endl;
        boost::asio::async_write(sock,
                boost::asio::buffer(msg, strlen(msg)),
                 boost::bind(&Client::handleWrite, this,
                        boost::asio::placeholders::error));
    }
}
```

网络 - TCP 异步客户端 C02A

- 客户端 C02A 代码：类 Client（续 3 ）

```
    void handleWrite(const boost::system::error_code& error) {
        if (!error) {
            memset(buf, 0, 512); // 注意: 重置buf
        sock.async_read_some(boost::asio::buffer(buf),
                    boost::bind(&Client::handleRead, this,
                                boost::asio::placeholders::error));
        }
    }

private:
    tcp::socket sock;
    enum { BUF_SIZE = 512 };
    char buf[BUF_SIZE];
};
```

网络 - TCP 异步客户端 C02A

- 客户端 C02A 代码： main() 函数

```
int main() {
    try {
        boost::asio::io_service service;

        tcp::resolver resolver(service);
        tcp::resolver::query query("localhost", "8868");
        tcp::resolver::iterator iterator =
                resolver.resolve(query);

        Client client(service, iterator);

        service.run();
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << "\n";
    }

    return 0;
}
```

网络 - TCP 异步客户端 C02A

- 客户端 C02A 一些延伸
  - 该版本的客户端虽然使用了异步操作，但操作方式与同步客户端无异：
    - 客户端将消息发送给服务器 (写)，然后等待接收服务器的回应，完全是规律的写 / 读配对操作
  - 某些情况下：如即时消息 (IM) 客户端，用户可能连续发送多条消息，而没有收到消息，或者相反的情况。这种情形下，使用异步 I/O 是很合适的方式，但不能使用 async_read() 和 async_write 相互、循环调用的模式 (具体实现，留作课下练习)

网络 - TCP 异步读取操作

- async_read() 和 async_receive()
  - tcp::socket 的成员函数

```
template<typename MutableBuffer, typename ReadHandler>
void async_read_some(const MutableBuffer& buffers,
                     ReadHandler handler);

template<typename MutableBuffer, typename ReadHandler>
void async_receive(const MutableBuffer& buffers,
                   ReadHandler handler);

template<typename MutableBuffer, typename ReadHandler>
void async_receive(const MutableBuffer& buffers,
                   socket_base::message_flags flags,
                   ReadHandler handler);
```

网络 - TCP 异步读取操作

- async_read() 和 async_receive() （续 1 ）
  - async_read() 自由函数

```
template<typename AsyncReadStream, typename MutableBuffer,
        typename CompletionCondition, typename ReadHandler>
void async_read(AsyncReadStream& s,
                const MutableBuffer& buffers,
                CompletionCondition completion_condition,
                ReadHandler handler);

template<typename AsyncReadStream, typename MutableBuffer,
        typename ReadHandler>
void async_read(AsyncReadStream& s,
                const MutableBuffer& buffers,
                ReadHandler handler);
```

网络 - TCP 异步读取操作

- async_read() 和 async_receive() （续 2 ）
  - async_read() 自由函数（续）

```
template<typename AsyncReadStream, typename Allocator,
        typename CompletionCondition, typename ReadHandler>
void async_read(AsyncReadStream& s,
                boost::asio::basic_streambuf<Allocator>& b,
                CompletionCondition completion_condition,
                ReadHandler handler);

template<typename AsyncReadStream, typename Allocator,
typename ReadHandler>
void async_read(AsyncReadStream& s,
                boost::asio::basic_streambuf<Allocator>& b,
                ReadHandler handler);
```

网络 - TCP 异步读取操作

- async_read() 和 async_receive() （续 3 ）
  - async_read() 自由函数和 tcp::socket 成员函数的区别
    - 与同步的 read() 自由函数类似， async_read() 函数只有在下面 3 种情况下才完成：
      - 读满指定的 bytes ，或
      - 符合某个条件，或
      - 出现错误
    - 而 tcp::socket 的成员函数 async_read_some() 和 async_receive() 则从字节流中读取实际传输的 bytes

网络 - TCP 异步写入操作

- async_write() 和 async_send()
  - tcp::socket 的成员函数

```
template <typename ConstBuffer, typename WriteHandler>
void async_write_some(const ConstBuffer& buffers,
                      WriteHandler handler);

template <typename MutableBuffer, typename ReadHandler>
void async_send(const MutableBuffer& buffers,
                ReadHandler handler);

template <typename MutableBuffer, typename ReadHandler>
void async_send(const MutableBuffer& buffers,
                socket_base::message_flags flags,
                ReadHandler handler);
```

网络 - TCP 异步写入操作

- async_write() 和 async_send() （续 1 ）
  - async_write() 自由函数

```
template <typename AsyncWriteStream, typename ConstBuffer,
    typename WriteHandler>
void async_write(AsyncWriteStream& s,
                 const ConstBuffer& buffers,
                 WriteHandler handler);

template <typename AsyncWriteStream, typename ConstBuffer,
    typename CompletionCondition, typename WriteHandler>
void async_write(AsyncWriteStream& s,
                 const ConstBuffer& buffers,
                 CompletionCondition completion_condition,
                 WriteHandler handler);
```

网络 - TCP 异步写入操作

- async_write() 和 async_send() （续 2 ）
  - async_write() 自由函数

```
template <typename AsyncWriteStream, typename Allocator,
typename WriteHandler>
void async_write(AsyncWriteStream& s,
                 basic_streambuf<Allocator>& b,
                 WriteHandler handler);

template <typename AsyncWriteStream, typename Allocator,
    typename CompletionCondition, typename WriteHandler>
void async_write(AsyncWriteStream& s,
                 basic_streambuf<Allocator>& b,
                 CompletionCondition completion_condition,
                 WriteHandler handler);
```

网络 - UDP 同步服务器 S03A

- 服务器 S03A 实现的功能
  - 实现一个基于 UDP 的同步 Echo Server，将来自客户端的消息原样发送回给客户端

网络 - UDP 同步服务器 S03A

- 服务器 S03A 代码

```
int main() {
    try {
        boost::asio::io_service service; // #1
        udp::socket socket(service,
                           udp::endpoint(udp::v4(), 8868)); //#2

        char buf[512];
        for (;;) {
            memset(buf, 0, 512);
            udp::endpoint remoteEndpoint; // #3
            boost::system::error_code error;
            size_t len =
                socket.receive_from(boost::asio::buffer(buf),
                                    remoteEndpoint, 0, error);//#4
```

网络 - UDP 同步服务器 S03A

- 服务器 S03A 代码（续）

```
            if (error &&
                error != boost::asio::error::message_size)
                throw boost::system::system_error(error);

            boost::system::error_code ignoredError;
            socket.send_to(boost::asio::buffer(buf, len),
                           remoteEndpoint, 0,
                           ignoredError); // #5
        }
    } catch (std::exception& e) {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}
```

网络 - UDP 同步服务器 S03A

- 服务器 S03A 创建的步骤
  - #1，创建 io_service 对象
  - #2，创建一个 udp::socket 对象，与 io_service 对象关联，用于 I/O
  - #3，准备一个 udp::endpoint 对象，该对象将在 udp::socket::receive_from() 操作中初始化
  - #4， udp::socket::receive_from() 接收来自客户端的数据报，并记录对端的地址信息
  - #5， udp::socket::send_to() 将消息发送给客户端

网络 - UDP 同步客户端 C03A

- 客户端 C03A 实现的功能
  - 不断往服务器发送消息，并将返回的信息打印到屏幕

网络 - UDP 同步客户端 C03A

- 客户端 C03A 代码

```
int main(int argc, char* argv[]) {
    try {
        boost::asio::io_service service; // #1

        udp::resolver resolver(service); // #2
        udp::resolver::query query(udp::v4(),
                    "localhost", "8868"); // #3
        udp::endpoint receiverEndpoint =
                    *resolver.resolve(query); // #4

        udp::socket socket(service);
        socket.open(udp::v4()); // #5
        char buf[512];
        for (;;) {
            memset(buf, 0, 512);
            cin.getline(buf, 512);
            socket.send_to(boost::asio::buffer(buf, strlen(buf)),
                    receiverEndpoint); // #6
```

网络 - UDP 同步客户端 C03A

- 客户端 C03A 代码（续）

```
            memset(buf, 0, 512);
            udp::endpoint senderEndpoint; // #7
            size_t len =
                    socket.receive_from(boost::asio::buffer(buf),
                     senderEndpoint); // #8
            cout << buf << endl;
        }
    } catch (std::exception& e) {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}
```

网络 - UDP 同步客户端 C03A

- 服务器 S03A 创建的步骤
  - #1，创建 io_service 对象
  - #2，创建 udp::resolver 对象，与 io_service 对象关联，
  - #3，创建查询对象 udp::resolver::query 对象，
  - #4， resovler 对象使用 query 对象查询目标服务器的地址信息
  - #5，创建、打开 udp::socket 对象，与 io_service 对象关联，用于 I/O
  - #6， udp::socket::send_to() 将消息发送给服务器
  - #7，准备一个 udp::endpoint 对象，该对象将在 udp::socket::receive_from() 操作中初始化
  - #8， udp::socket::receive_from() 接收来自服务器的数据报

网络 - UDP 同步读取操作

- receive_from()

```
template<typename MutableBuffer>
std::size_t receive_from(const MutableBuffer& buffers,
        endpoint_type& sender_endpoint);

template<typename MutableBuffer>
std::size_t receive_from(const MutableBuffer& buffers,
        endpoint_type& sender_endpoint,
        socket_base::message_flags flags);

template<typename MutableBuffer>
std::size_t receive_from(const MutableBuffer& buffers,
        endpoint_type& sender_endpoint,
        socket_base::message_flags flags,
        boost::system::error_code& ec);
```

网络 - UDP 同步写入操作

- send_to()

```
template<typename ConstBuffer>
std::size_t send_to(const ConstBuffer& buffers,
        const endpoint_type& destination);

template<typename ConstBuffer>
std::size_t send_to(const ConstBuffer& buffers,
        const endpoint_type& destination,
        socket_base::message_flags flags);

template<typename ConstBuffer>
std::size_t send_to(const ConstBuffer& buffers,
        const endpoint_type& destination,
        socket_base::message_flags flags,
        boost::system::error_code& ec);
```

网络 - UDP 异步服务器 S04A

- 服务器 S04A 实现的功能
  - 实现一个基于 UDP 的异步 Echo Server，将来自客户端的消息原样发送回给客户端（功能完全等同于同步 UDP 服务器 S03A ）

网络 - UDP 异步服务器 S04A

- 服务器 S04A 代码：类 Server

```
class Server {
public:
    Server(boost::asio::io_service& service) :
        sock(service, udp::endpoint(udp::v4(), 8868)) {
        start();
    }

private:
    void start() {
        memset(buf, 0, BUF_SIZE);
        sock.async_receive_from(boost::asio::buffer(buf),
                remoteEndpoint,
                boost::bind(&Server::handleReceive, this,
                boost::asio::placeholders::error,
                boost::asio::placeholders::bytes_transferred));
    }
```

- 服务器 S04A 代码：类 Server（续）

```
    void handleReceive(const boost::system::error_code& error,
            std::size_t bytes_transferred) {
        if (!error || error == boost::asio::error::message_size) {
            sock.async_send_to(boost::asio::buffer(buf,
                bytes_transferred), remoteEndpoint,
                boost::bind(&Server::handleSend, this,
                            boost::asio::placeholders::error));
            start();
        }
    }
    void handleSend(const boost::system::error_code& /*error*/) {
    }
private:
    udp::socket sock;
    udp::endpoint remoteEndpoint;

    enum { BUF_SIZE = 512 };
    char buf[BUF_SIZE];
};
```

网络 - UDP 异步服务器 S04A

- 服务器 S04A 代码： main() 函数

```
int main() {
    try {
        boost::asio::io_service service;
        Server server(service);
        service.run(); // 注意: 一定要调用 run()函数
    } catch (std::exception& e) {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}
```

网络 - UDP 异步服务器 S04A

- 服务器 S04A 的创建步骤
  - 总体而言， S04A Server 与 S03A Server 的创建步骤大致相同，区别在于：
    - 使用异步 I/O 操作
    - 为异步 I/O 操作提供完成处理器
    - 以函数相互调用代替显式循环语句

网络 - UDP 异步客户端 C04A

- 客户端 C04A 实现的功能
  - 一个异步 Echo 客户端，功能同同步客户端 C03A

网络 - UDP 异步客户端 C04A

- 客户端 C04A 代码：类 Client

```
class Client {
public:
    Client(boost::asio::io_service& service,
           const udp::endpoint& remote) :
        remoteEndpoint(remote), sock(service, udp::v4()) {
        // sock.open(udp::v4());
        start();
    }

private:
    void start() {
        memset(buf, 0, BUF_SIZE);
        cin.getline(buf, BUF_SIZE);
        sock.async_send_to(boost::asio::buffer(buf, strlen(buf)),
                remoteEndpoint,
                boost::bind(&Client::handleSend, this,
                boost::asio::placeholders::error));
    }
```

网络 - UDP 异步客户端 C04A

- 客户端 C04A 代码：类 Client

```
    void handleSend(const boost::system::error_code& error) {
        if (!error) { memset(buf, 0, BUF_SIZE);
            udp::endpoint local;
      sock.async_receive_from(boost::asio::buffer(buf, BUF_SIZE),
      local, boost::bind(&Client::handleReceive, this,
                         boost::asio::placeholders::error));
        }
    }
    void handleReceive(const boost::system::error_code& error) {
        if (!error) {
            cout << buf << endl;
            start();
        }
    }
private:
    udp::endpoint remoteEndpoint;
    udp::socket sock;
    enum { BUF_SIZE = 512 };
    char buf[BUF_SIZE];
};
```

网络 - UDP 异步客户端 C04A

- 客户端 C04A 代码： main() 函数

```
int main() {
    try {
        boost::asio::io_service service;
        udp::resolver resolver(service);
        udp::resolver::query query(udp::v4(),
                                   "localhost", "8868");
        udp::endpoint receiverEndpoint =
                              *resolver.resolve(query);
        cout << receiverEndpoint.address().to_string() << endl;
        cout << receiverEndpoint.port() << endl;
        Client c(service, receiverEndpoint);

        service.run();
    } catch (exception& e) {
        cout << e.what() << endl;
    }
}
```

网络 - UDP 异步读取操作

- async_receive_from()

```
template<typename MutableBufferSequence, typename ReadHandler>
void async_receive_from(const MutableBufferSequence& buffers,
        endpoint_type& sender_endpoint, ReadHandler handler);

template<typename MutableBufferSequence, typename ReadHandler>
void async_receive_from(const MutableBufferSequence& buffers,
        endpoint_type& sender_endpoint,
        socket_base::message_flags flags,
        ReadHandler handler);
```

网络 - UDP 异步写入操作

- async_send_to()

```
template<typename ConstBufferSequence, typename WriteHandler>
void async_send_to(const ConstBufferSequence& buffers,
        const endpoint_type& destination, WriteHandler handler);

template<typename ConstBufferSequence, typename WriteHandler>
void async_send_to(const ConstBufferSequence& buffers,
        const endpoint_type& destination,
        socket_base::message_flags flags,
        WriteHandler handler);
```

网络 - 使用 asio::streambuf

- 两个简易 http 客户端
  - 通过两个简易的客户端熟悉基于 asio::streambuf 的 I/O
  - 熟悉基于行的 I/O 操作
    - read_until() 、 async_read_until() 等

网络 - 使用 asio::streambuf

- 两个简易 http 客户端代码
  - (见 Boost.Asio 的示例代码)

网络 - 使用 Timer

- 关于 Timer（定时器）
  - 在服务器端应用程序经常会使用：
    - 超时事件，如实现超时 accept 操作、超时 read/write 操作等
    - 周期性的操作，如服务器端定时向所有或部分客户端发送特定的消息 (如 keep_alive)
    - 或其它周期性的任务
  - Boost 中提供了一种 Timer 类型： deadline_timer ，与 I/O 事件类似， io_service 实例也负责调度 Timer 的超时事件

网络 - 使用 Timer

- 同步超时操作
  - 示例

```
#include <iostream>
#include <boost/asio.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>

int main() {
    boost::asio::io_service io;
    boost::asio::deadline_timer t(io,
        boost::posix_time::seconds(5));
    t.wait();

    std::cout << "Hello, world!\n";

    return 0;
}
```

网络 - 使用 Timer

- 异步超时操作
  - 示例

```
#include <iostream>
#include <boost/asio.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>

void print(const boost::system::error_code& /*e*/) {
    std::cout << "Hello, world!\n";
}

int main() {
    boost::asio::io_service io;
    boost::asio::deadline_timer t(io,
        boost::posix_time::seconds(5));
    t.async_wait(print);
    io.run();

    return 0;
}
```

网络 - 使用 io_service::strand

- 关于 io_service::strand
  - strand 是保证同一个 io_service 调度的线程不会并发执行，也即执行串行化，这样就避免了数据竞态
  - 也就是实现类使用多线程，但不需显式加锁

网络 - 使用 io_service::strand

- io_service::strand 示例

```
#include <iostream>
#include <boost/asio.hpp>
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>

class printer {
public:
    printer(boost::asio::io_service& io) :
        strand_(io), timer1_(io, boost::posix_time::seconds(1)),
        timer2_(io, boost::posix_time::seconds(1)), count_(0) {
        timer1_.async_wait(strand_.wrap(
            boost::bind(&printer::print1, this)));
        timer2_.async_wait(strand_.wrap(
            boost::bind(&printer::print2, this)));
    }
    ~printer() {
        std::cout << "Final count is " << count_ << "\n";
    }
```

网络 - 使用 io_service::strand

- io_service::strand 示例（续 1 ）

```
void print1() {
    if (count_ < 10) {
        std::cout << "Timer 1: " << count_ << "\n";
        ++count_;
        timer1_.expires_at(timer1_.expires_at()
                + boost::posix_time::seconds(1));
        timer1_.async_wait(strand_.wrap(
                boost::bind(&printer::print1, this)));
    }
}
void print2() {
    if (count_ < 10) {
        std::cout << "Timer 2: " << count_ << "\n";
        ++count_;
        timer2_.expires_at(timer2_.expires_at()
                + boost::posix_time::seconds(1));
        timer2_.async_wait(strand_.wrap(
                boost::bind(&printer::print2, this)));
    }
}
```

网络 - 使用 io_service::strand

- io_service::strand 示例（续 2 ）

```
private:
    boost::asio::strand strand_;
    boost::asio::deadline_timer timer1_;
    boost::asio::deadline_timer timer2_;
    int count_;
};

int main() {
    boost::asio::io_service io;
    printer p(io);
    boost::thread t(boost::bind(&boost::asio::io_service::run,
                                &io));
    io.run();
    t.join();

    return 0;
}
```

网络 - Asio 编程指南

- 网络：
  - 网络基本概念
  - Asio 核心概念
  - Asio 编程指南
  - 核心接口描述

网络 - 核心接口描述

- Asio 的核心接口
  - io_service
  - tcp::socket
  - udp::socket
  - tcp::acceptor
  - tcp::resolver
  - udp::resolver
  - tcp::endpoint
  - udp::endpoint
  - deadline_timer

网络 - 核心接口描述： io_service

- 关于 io_service
  - Proactor ，负责调用异步事件的多路分离器从事件队列中取出，且分派与这些事件关联的完成处理器
  - 在目前的版本中，异步事件多路分离器，在 Linux 平台可能使用的是：
    - epoll (kernel 2.6 以后)
    - select (kernel 2.4 或更早版本)

网络 - 核心接口描述： io_service

- io_service 主要接口
  - 内部类型：
    - class service： 所遇 service 的基类
    - class strand： 确保所有完成处理器串行化执行
    - class work： 当有任务需执行时通知 io_service
  - 主要成员函数：
    - run()： 执行事件处理循环
    - post()： 请求 io_service 调用指定的处理器，并立即返回
    - poll()： 执行事件处理循环来执行准备就绪的处理器
    - dispatch()： 请求 io_service 调用指定的处理器

    post() 和 dispatch() 所需的完成处理器的形式：

    ```
    void handler();
    ```

网络 - 核心接口描述： sockets

- tcp::socket 和 udp::socket
  - 主要的 I/O 对象
  - tcp::socket 和 udp::socket 的实际类型分别是：
    - typedef basic_stream_socket<tcp> socket;
    - typedef basic_datagram_socket<udp> socket;

网络 - 核心接口描述： tcp::acceptor

- tcp::acceptor (basic_socket_acceptor<tcp>)
  - 接受器，接受来自客户端的连接，用于 tcp 服务器端
  - 主要函数：
    - accept()： 同步接受来自客户端的连接
    - async_accept()： 异步接受来自客户端的连接
    - open()： 开启 acceptor 对象
    - bind()： 绑定到本地某个端口
    - listen()： 进入侦听状态
    - close()： 关闭该 acceptor
    - is_open()： 查看该 acceptor 是否已经开启
    - get_io_service()： 返回关联的 io_service 的引用
    - local_endpoint()： 返回本地地址、端口信息

网络 - 核心接口描述： resolvers

- tcp::resolver 和 udp::resolver
  - 主机名、服务解析，用于客户端解析指定的服务器所在的 ip 地址和指定的端口（指定的服务）
  - 各自的具体类型：
    - basic_resolver<tcp>
    - basic_resolver<udp>
  - 主要函数：
    - resolve()： 同步解析操作
    - async_resolve()： 异步解析操作
    - get_service_type()： 获取与之关联的 io_service 引用

网络 - 核心接口描述： endpoints

- tcp::endpoint 和 udp::endpoint
  - 用于表示网络通信的端点
  - 各自的具体类型：
    - basic_endpoint<tcp>
    - basic_endpoint<udp>
  - 主要函数：
    - address()： 获取 / 设置该端点的 ip 地址
    - port()： 获取 / 设置该端点的 port

网络 - 核心接口描述： deadline_timer

- deadline_timer
  - 定时器
  - 主要函数：
    - wait()： 同步等待
    - async_wait()： 异步等待
    - expires_at()： 获取 / 设置该 timer 到期的绝对时间
    - expires_from_now()： 获取 / 设置该 timer 从现在算起的相对到期时间
    - cancel()： 取消等待在该 timer 上的所有异步操作
    - get_io_service()： 获取与该 timer 关联的 io_service 对象的引用

网络 - More stuffs...

- 网络编程的涉及面非常广泛，并且很复杂，特别是在不同操作系统下进行网络程序的开发更是如此，不过 Boost.Asio 提供了一套完整、统一、高效并且易用的接口，大大降低了开发的难度
- 由于 Boost.Asio 所提供的接口十分的丰富，本次课程仅提取了其中核心的一部分，还有更多有用的接口尚需在课下熟悉
- 另外，要熟练进行网络应用的开发，以下是几点建议：
  - 熟悉网络基本概念，了解 IP/TCP 架构
  - 消化 Boost.Asio 库附带的示例代码，这里每个示例都体现了该库开发者的经验所在
  - 熟悉 Linux 下的原生 Socket API
  - 尽量熟悉 Linux 下的事件多路分离的 API 如： epoll 和 select 或其它系统下的对应机制（如 FreeBSD 的 kqueue 等）