Document Number: N3388=12-0078
Date: 2012-04-23
Reply to: Christopher Kohlhoff <chris@kohlhoff.com>

与C++11一起使用 Asio

本文旨在介绍 Asio 库，并简要概述其实现以及与C++结合使用。

1. Asio 库的 C++11 变体

本文基于一个独立于 Boost 的 Asio 的变体，而没有使用 Boost 发行版的 Asio 库。此变体的目标包括：

- 在接口中仅使用C++11语言和库的特性。
- 示范 Asio 库可以仅使用 C++11 标准库和操作系统提供的功能来实现。

这个变体可以在 http://github.com/chriskohlhoff/asio/tree/cpp11-only 获取。

2. 使用 I/O 流处理简单用例

在许多应用程序中，网络不是核心功能，也不被看成是开发应用程序的程序员的核心能力。为了满足这些用例，Asio 为 TCP 套接字提供了一套高级接口，该接口是围绕熟悉的 C++ I/O 流框架设计的。

以这种方式使用库像是使用远程主机详细信息创建一个流对象一样简单：

```
tcp::iostream s("www.boost.org", "http");
```

接下来，确定当远程主机无响应时套接字是否放弃：

```
s.expires_from_now(std::chrono::seconds(60));
```

然后，根据需要发送和接收数据。在这个例子中，发送一个请求：

```
s << "GET / HTTP/1.0\r0.2cm\n";

s << "Host: www.boost.org\r\n";

s << "Accept: */*\r\n";

s << "Connection: close\r\n\r\n";
```

然后接收并处理响应：

```
std::string header;

while (std::getline(s, header) && header != "\r")
  std::cout << header << "\n";

std::cout << s.rdbuf();
```

如果在任何时候出现错误，可以使用 tcp::iostream 类的 error() 成员函数来确定失败的原因。

```
if (!s)
{
  std::cout << "Socket error: " << s.error().message() << "\n";
  return 1;
}
```

3. 理解同步操作

同步操作是在相应的操作系统操作完成之前不会将控制权返回给调用者的函数。在基于 Asio 的应用程序中，它们的用例通常分为两类：

- 不关心超时的简单程序，或者乐于依靠底层操作系统提供的超时行为。
- 需要细粒化控制系统调用的程序，并且知道同步操作会否阻塞的条件。

Asio 可用于在诸如套接字这样的 I/O 对象上执行同步和异步操作。然而，同步操作提供了一个机会，对 Asio 各个部分和你的程序以及它们如何一起工作做一个简单概念上的概述。作为一个介绍性的示例，让我们考虑一下在套接字上执行同步连接操作会发生什么。

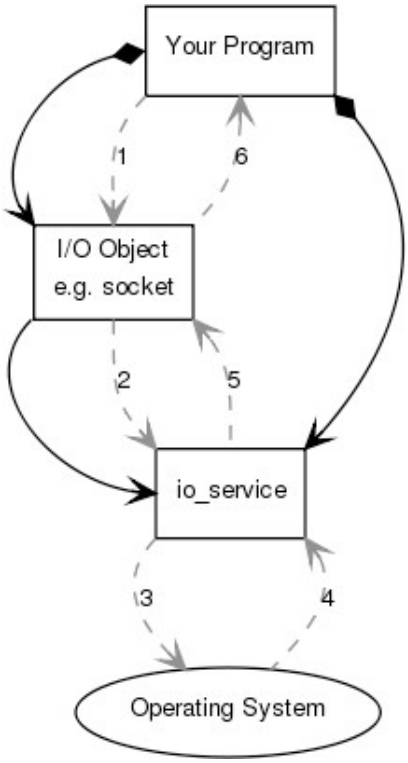
程序至少要有一个 io_service 对象。io_service 表示程序到操作系统 I/O 服务的链接。

```
asio::io_service io_service;
```

要执行 I/O 操作，程序需要一个 I/O 对象，例如一个 TCP 套接字：

```
tcp::socket socket(io_service);
```

当一个同步连接操作被执行，会依次发生下面的事件：



1. 程序通过调用 I/O 对象来开始连接操作：

   ```
   socket.connect(server_endpoint);
   ```

2. I/O 对象把请求转发给 io_service.

3. io_service 请求操作系统执行连接操作。

4. 操作系统把操作端结果返回给 io_service。

5. io_service 把来自操作的任何错误转换成 std::error_code 类型的对象。然后把结果转发回I/O 对象。

6. 如果操作失败， I/O 对象将抛出一个 std::system_error 类型的异常。如果开始连接操作的代码被改写为：

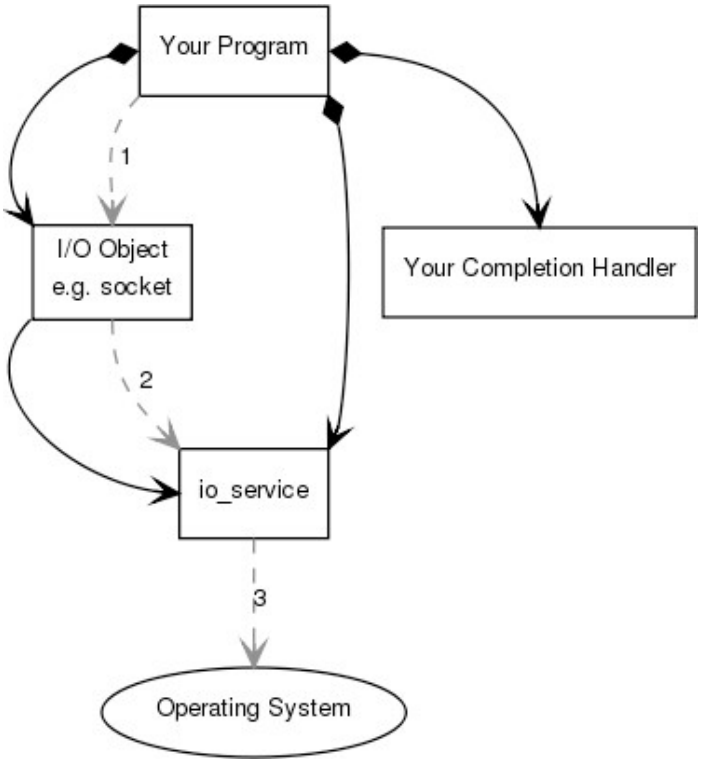
```
std::error_code ec;
socket.connect(server_endpoint, ec);
```

然后 error_code 类型变量 ec 会被设置为操作的结果，并且不会再引发异常。

4. 理解异步操作

异步操作不阻塞调用方，取而代之是当相应的操作系统操作完成时，向程序投递通知。大部分重要的基于 Asio 的程序会使用异步操作。

当使用异步操作时，会依次发生下面的事件：



1. 程序通过调用 I/O 对象开始异步连接操作：

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

async_connect() 函数是一个启动函数。 Asio 中的启动函数以 async_ 前缀命名。启动函数将函数对象，被成为完成处理函数，作为它的最后一个参数。

在这个特殊的场景中，your_completion_handler 是一个具有如下签名的函数或函数对象：

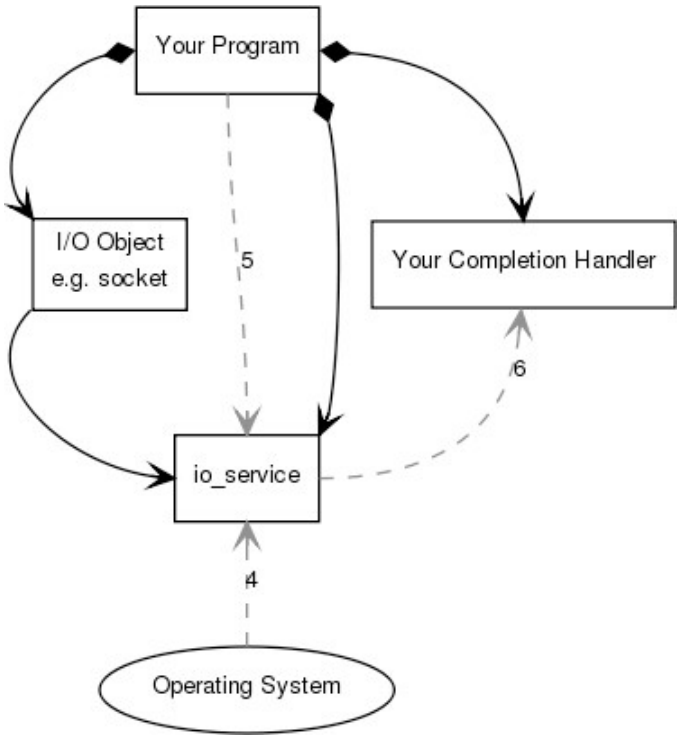
```
void your_completion_handler( const std::error_code& ec);
```

完成处理函数的确切签名依赖于正在被执行的异步操作。Asio 参考文档指出了每个操作相应的完成处理函数的合适形式。

2. I/O 转发请求给 io_service。

3. io_service 向操作系统发出信号，指示操作系统应该启动一个异步连接操作。

时间流逝。（在同步操作场景中，此等待会完全包括在连接操作期间）在此期间，操作系统名义上负责异步操作，这被称为未完成工作。



4. 操作系统通过将结果放入一个队列来通知连接操作完成，准备由 io_service 选取。

5. 程序必须调用 io_service::run() （或者与之相似的 io_service 成员函数）以便结果能够被取回。对 io_service::run() 的调用会在未完成工作期间阻塞[1]。通常应该在开始第一个异步操作后立即调用它。

6. 当进入对 io_service::run() 调用的内部时， io_service 从队列中取出操作的结果，把它转换成一个 error_code 类型对象，并且把它传递到完成处理函数。

5. 链式异步操作

异步操作被认为是未完成工作，直到它关联的完成处理函数被调用并且返回。完成处理函数可能反过来调用其它启动函数，进而创建更多的未完成工作。

---

[1] 未完成工作在逻辑上由 io_service::work 类型的对象表示。如果存在一个或多个 io_service::work 对象，io_service::run()函数将会阻塞，并且所有的异步操作的行为就像它们有一个关联的 io_service::work对象。

考虑一个连接后紧跟着在套接字上进行其它 I/O 操作的场景：

```
socket.async_connect( server_endpoint,
    [&](std::error_code ec)
    {
        if (!ec)
        {
            socket.async_read_some(asio::buffer(data),
                [&](std::error_code ec, std::size_t length)
                {
                    if (!ec)
                    {
                        async_write(socket, asio::buffer(data, length),
                            [&](std::error_code ec, std::size_t length)
                            {
                                // …
                            });
                    }
                });
        }

    });
```

异步连接操作的完成处理函数，这里以一个 C++11 的 lambda 表达式表示，启动一个异步的读操作。这个读操作有它自己关联的未完成工作，并且它的完成处理函数以异步写的形式也创建了更多工作。因此，在链中的所有操作完成前， io_service::run() 不会返回。

当然，在真实的程序中这些链通常更长，并且可能包含循环和分叉。在这些程序中，io_service::run() 可能会永远执行。

6. 处理错误

Asio 处理错误的方法基于这样一个观点，即异常并不总是处理错误的正确方法。在网络编程中，例如，经常会遇到如下错误：

- 无法连接远程 IP 地址。
- 网络连接断开。
- 尝试打开一个 IPv6 套接字，但却没有可用的 IPv6 网络接口。

这些可能是特殊条件，但同样可以作为正常控制流的一部分处理。如果你合理的预期它发生，它就不是例外。分别地：

- IP 地址是一个主机名称关联的地址列表中的一个，你想要尝试连接列表中的下一个地址。
- 网络不稳定。你想要重建连接，并在 n 次失败之后才放弃。
- 你的程序可以回退到 IPv4 套接字。

错误是否是异常，取决于程序中的上下文。此外，由于代码大小或性能限制，某些领域可能不愿意或无法使用异常。基于这个原因， 所有的异常都提供两种处理处理方式，抛出异常：

```
socket.connect(server_endpoint); // 出错时抛出 std::system_error。
```

和非抛异常的重载：

```
std::error_code ec;
socket.connect(server_endpoint, ec); // 设置 ec 来指明错误。
```

基于类似的原因，Asio 不会根据操作完成时是否出错来分别使用完成处理函数。那样做会在异步操作链中创建一个分叉，而这可能与程序对构成错误的条件的本意不匹配。

7. 管理对象生存期

当使用异步操作时，一个特别的挑战是对象生存期管理。Asio 采用的方法没有显示地支持管理对象生存期，相反地，生存期的要求是基于启动函数如何声明的规则在程序的控制之下：

- **按值（value），常量引用（const reference）和右值引用（rvalue reference）参数。**

  这些由库实现按需复制或移动。例如，库实现维护了一份完成处理函数对象的副本，直到这个处理函数被调用。

- **非常量引用（Non-const reference）参数， this 指针。**

  程序有责任保证对象在异步操作完成前保持有效。

很多基于 Asio 程序中被使用的一个方法是，绑定对象生存期到完成处理函数。这可以使用 std::shared_ptr<> 和 std::enable_shared_from_this<> 来完成：

```
class connection :
      std::enable_shared_from_this<connection>
{
   tcp::socket socket_;
   vector<unsigned char> data_;
   // ...
   void start_read()
   {
      socket_.async_read_some(socket_,
            asio::buffer(data_),
            std::bind(&connection::handle_read,
               shared_from_this(), _1, _2));
   }
   // ...
};
```

对于 C++11，这个方法可以在可用性和性能之间提供出色的权衡。Asio 能够利用可移动的完成处理函数最大限度降低与引用计数相关的成本。因为程序通常由异步操作链组成，指针的所有权可以沿着链之间传递，引用计数只在链的开始和结束才被更新。

但是，某些程序需要对对象的生存期、内存使用和执行效率进行精细控制。通过将对象生存期置于程序控制之下，这些用例也能获得支持。例如，另一种方法是在程序启动时创建全部对象，完成处理函数不需要在对象生存期中发挥作用（对象指针的所有权不需要借助完成处理函数在链之间传递），并且复制成本也微不足道。

8. 优化内存分配

很多异步操作需要分配对象以保存操作相关的状态。例如，窗口实现需要重叠派生（OVERLAPPED-derived）对象传递给 Win32 API 函数。

默认情况下，Asio 会使用 ::operator new() 操作符为这个 bookkeeping 信息分配空间，但是，异步操作链在这里提供了一个优化的机会。每个链可以有一个关联的内存块，并且这个内存块可以被链中的每个顺序操作重用，这意味着编写不执行持续内存分配的异步协议实现成为可能（例如，协议分三层，每一层包含头和负载，从上到下三层头分别是*16*，*12*，*8*字节，负载不超过*1450*字节，这样发送时，从最上层开始分配内存即可预留*1476*（*16+12+8+1450*）字节，第一层头偏移在*+28*（*16+12*）字节处，第二层头偏移在*+16*字节处，而第三层头偏移在*0*处，接收时按照相反的顺序，第三层从*0*字节处解析，然后将*+16*字节偏移后的数据向上传递给第二层，第二层解析自己的*12*字节头之后再将*+28*字节偏移后的数据向上传递到第一层，第一层解析自己的*8*字节头和头之后的负载数据，无论收发，数据在链中都无需再度分配，层之间传递数据只需要移动偏移指针）。

这个自定义内存分配的钩子是异步操作完成处理函数，处理函数确定正在被执行的操作中更大的上下文（The handler identifies the larger context in which the operation is being performed.）通过把这个完成处理函数传递给启动函数，Asio 可以在给操作系统发送信号令其开始异步操作之前分配必要的内存。

9. 处理并发

协议实现通常涉及多个异步操作链的协做。例如，一个操作链可能处理消息发送，另一个处理接收，第三个可能实现应用程序层超时。所有这些链需要访问普通变量，如套接字，计时器和缓冲区。而且，这些异步操作可能会永远继续下去。

异步操作提供了一种实现并发的方法，而无需线程的开销和复杂性。然而，Asio 的接口被设计成支持一系列线程方法，其中一些方法概述如下。

单线程设计

Asio 保证完成处理函数的调用只来自 io_service::run()[2] 中，因此，严格地只在一个线程中调用 io_service::run()，程序可以阻止并行执行处理函数。这个设计是大部分程序推荐的起点，因为不需要显示的同步机制。但是，必须保持处理函数简短并且不能阻塞。

将线程用于执行长时间任务

一种单线程设计的变种，这种设计依旧使用单个线程运行 io_service::run() 实现协议逻辑。长时间运行或阻塞任务被传递给后台线程，完成后，运行结果被投递回 io_service::run() 线程[3]。

通过使用无共享消息传递的方法，程序可以确保对象不会在 io_service::run() 线程和任何后台工作线程之间共享，因此也不需要显示的同步。

多个 io_service，每个 io_service 一个线程

在这种设计中，I/O 对象关联一个运行在单个线程的“主” io_service，不同的对象只通过消息传递进行通信。这种设计可以更高效使用多个CPU，并且限制竞争源。显示同步也不需要，同样也必须保证完成函数简单和非阻塞。

---

[2] 或者一个类似的来自 io_service 的成员函数 run_one()， poll() 或 poll_one()。

[3] Asio 支持使用 io_service 成员函数 post() 和 dispatch() 传递消息。

一个 io_service，多个线程

io_service::run() 可能被多个线程调用，为单个 io_service 设置一个线程池。这个实现以任意方式在线程之间分配可用的工作。

因为完成处理函数可以被任意线程调用，除非协议实现是微不足道的并且由单个操作链组成，否则可能需要某种形式的同步。Asio 为此提供了 io_service::strand 类。

strand 阻止关联到它的任何完成处理函数并发执行，在上面的例子中，我们有一个协议实现由三个操作链组成（用于发送，接收和超时），strand 确保相关联的链的处理函数是顺序执行的。其它协议实现执行在另外的 strand 上，它们仍然能够利用线程池中的其它任意线程。而且，不像互斥锁，如果 strand 在“正使用”时，它不会阻塞调用线程，而是会切换到其它 strand 上的已就绪处理函数，从而保持线程忙碌。

与使用自定义的内存分配一样，strand 同步使用一个与完成处理函数关联的钩子，也就是说，完成处理函数确定正在被执行的操作中较大的上下文（That is, the completion handler identifies the larger context in which the operation is being performed）。这种自定义调用钩子允许同步机制扩展到基于操作组合的抽象。就像我们下面会看到的那样。

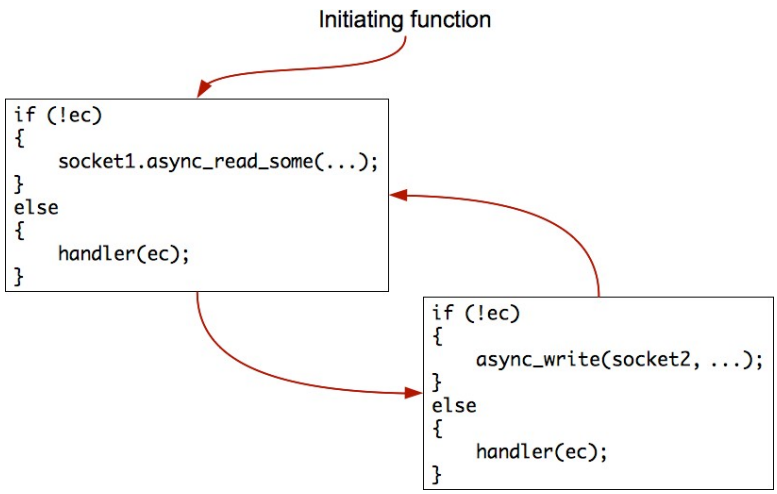
10. 推卸责任（Passing the buck）：开发高效的抽象

Asio 中一个关键的设计目标是支持高层抽象的创建，其实现的主要机制是异步操作的组合。在 Asio 术语中，这些被简称为“组合的操作”。

作为一个例子，考虑一个假定的用户定义的异步操作，这个操作实现把从一个套接字读取的全部数据完整透传给另一个套接字。启动函数可以定义如下：

```
template <typename Handler>void
async_transfer(
    tcp::socket& socket1, tcp::socket& socket2, std::array<unsigned
    char, 1024>& working_buffer,Handler handler);
```

这个函数会从两个底层异步操作方面实现：一个操作从 socket1 读，另一个操作写到 socket2。每个操作有一个中间完成处理函数，它们之间的关系展示在下图：



这些中间完成处理函数可以“推卸责任（pass the buck）”，即通过自定义分配和调用钩子来简单调用用户的完成处理函数钩子。通过这种方式，组合将推迟内存分配和同步的全部选择给抽象的用户，抽象的用户得以在易用性和效率之间选择适当的权衡，并且如果不需要显示同步，也不需要付出同步的代价。

Asio 提供了许多开箱即用的这种组合，比如非成员函数 async_connect()， async_read()， async_write() 和 async_read_until()。密切相关的组合操作也被分组在对象 Asio 中，如 buffered_stream<> 和 ssl::stream<> 模版。

11. 范围和可扩展性

鉴于 Asio 库的规模，提交给 WG21的提案仅限于一个最小的可行子集，该子集聚焦于 TCP 和 UDP ，缓冲区和计时器的网络。Asio 的接口被设计为允许用户和实现者通过许多机制扩展，其中一些机制描述如下。

额外的 I/O 服务

io_service 类实现了一个使用服务类型索引的可扩展，类型安全，多态的 I/O 服务。I/O 服务的存在是为了代表 I/O 对象管理操作系统的逻辑接口。特别是，存在在一类 I/O 对象之间共享的资源。例如，计时器可能以单计时器队列的形式实现，I/O 服务管理这些共享资源，并且被设计成不使用时不会产生成本。

Asio 实现额外的 I/O 服务以提供对某些操作系统特定功能的访问。 例如包括：

- 在句柄上执行重叠 I/O 的 Windows 特定服务。
- 等待内核对象如事件，进程和线程的 Windows 特定服务。
- 面向流的文件描述符的 POSIX 特定服务。
- 通过 signal() 或 POSIX sigaction() 实现信号处理安全集成的服务。

可以添加新的 I/O 服务而不会对库的现有使用者造成冲击。

套接字类型规定

尽管基于 Asio 的提案仅限于 TCP 和 UDP 套接字，但是接口是基于类型规定，例如 Protocol 和 Endpoint。 这些类型规定被设计成允许库使用其它类型的套接字。Asio 库本身使用这些类型规定添加了对 ICMP 和 UNIX 域套接字的支持。

流类型规定

Asio 库为同步和异步面向流的 I/O 定义了几个类型规定，这些类型规定在 TCP 套接字接口被实现，并且被抽象为诸如 ssl::stream<> 和 buffered_stream<>。通过实现这些类型规定，一个类可以与诸如 async_read()， async_read_until() 和 async_write() 之类的组合操作一起使用，类型规定也旨在用于诸如基于 HTTP 的异步包装器之类的更高级别抽象。