本文档是 Boost.Asio 1.86.0 中文文档。

Boost.Asio 是一个跨平台的 C++ 库，致力于提供网络和底层 IO 编程，为开发者提供一个使用 C++ 异步模型的途径。

第一章　概述

对 Boost.Asio 特性和基本原理的概述。

1.1　基本原理

大多数程序以某种方式和外界交流，不管是通过一个文件、网络、串口，还是控制终端。有的时候，比如网络通讯，单独的 I/O 操作可能需要很长时间来完成。这对软件开发带来了很大的挑战。

Boost.Asio 提供了一些工具来管理这些长时间的操作，而不需要程序依赖基于线程和显式上锁的并发模型。

Boost.Asio 库意在使用 C++ 进行系统级别编程。特别的，Boost.Asio 强调下列目标：

- **可移植性：**该库应该支持一系列的常用的操作系统，并在这些操作系统上提供一致的行为。
- **可扩展性：**该库应该能够支持开发可以扩展到处理成千上万的并发连接的网络应用程序。库在每个操作系统上的实现应该使用最佳的机制，以实现这种可扩展性。
- **效率：**该库应支持诸如散布-聚集 I/O（scatter-gather I/O）等技术，并允许程序尽量减少数据复制。
- **借鉴已有的 API 模型（例如 BSD 套接字（sockets）API）：**BSD 套接字 API 被广泛实现和理解，并且有大量的文献覆盖这一主题。其他编程语言也常常使用类似的网络 API 接口。因此，Boost.Asio 在设计时应尽可能利用现有的实践和接口标准。
- **易用：**这个库应通过采取工具包（toolkit）而非框架（framework）的方法，来降低新用户的学习门槛。也就是说，它应尽量减少新用户在学习时

的初始投入时间，让他们只需了解一些基本规则和指导方针。之后，用户只需理解他们所使用的具体函数即可。

- **进一步抽象的基础：**这个库应允许开发其他提供更高层次抽象的库。例如，可以基于这个库实现一些常用协议的库，如 HTTP 协议的实现。

尽管 Boost.Asio 起初是为了网络出现的，它的异步 IO 的概念被扩展延伸到了操作系统其他的资源，比如串口、文件描述符等。

1.2 基础 Boost.Asio 剖析

Boost.Asio 可以有同步或者异步的操作，例如 Sockets 。在使用 Boost.Asio 之前，有一个各种 Boost.Asio 不同部分的概念图是有所帮助的。

作为一个入门示例，让我们来考虑一下当你在一个 Socket 上执行连接操作时会发生什么。我们将从研究同步操作开始。

你的程序至少含有一个 IO execution context，就是 boost::asio::io_context 、boost::asio::thread_pool 或者 boost::asio::system_context 对象。这个 IO execution context 代表着你的程序如何连接操作系统的 IO 服务。

```
boost::asio::io_context io_context;
```

为了执行 IO 操作，程序需要一个 IO object ，比如一个 TCP Socket 。

```
boost::asio::ip::tcp::socket socket(io_context);
```

当一个同步连接操作执行后，下面一系列的事件就会发生：

- 程序在 socket 对象（IO 对象）上调用方法来初始化连接操作

```
socket.connect(server_port);
```

- IO 对象进而执行 IO 执行上下文
- IO 执行上下文通过系统调用进行连接操作
- 操作系统将返回值返回给 IO 上下文的调用处
- IO 上下文将所有错误统一转化为一个 boost::system::error_code 的对象来表示。一个 error_code 可以被比较，测试是不是任何 bool 值。之后，结果就被返回给 IO 对象

- 如果操作失败,IO 对象抛出一个异常,类型是 boost::system::system_error。如果初始化代码写成如下形式:

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

那么，ec 就被设置为这个操作的结果，而不会抛出异常

以上是同步连接的操作，当使用异步操作时，会有不同的事件序列。

- 程序在 socket 对象（IO 对象）上调用方法来初始化连接操作

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

这里的 your_completion_handler 是一个函数或者一个有这样签名的函数对象:

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

具体的签名还要看具体的异步操作

- IO 对象进而执行 IO 执行上下文
- IO 执行上下文向操作系统发送请求，需要开启一个异步连接

  程序等待时间（在同步操作中，程序需要等待操作系统连接操作结束），在异步操作中，不需要等待，调用直接返回。
- 操作系统通过在一个 queue 中放置一个结果来提醒连接操作完成，准备被 IO 执行上下文获取
- 当使用 io_context 作为上下文时，程序必须调用 io_context::run()，进而保证操作的结果被获取到。调用这个函数时，如果有未完成的异步操作，程序会阻塞，所以最好在第一个异步操作之后就调用它
- 在这个 run 函数中，IO 执行上下文会将结果从 queue 出队，转换为 error_code ，再传递给 your_completion_handler

下面给出一个实例，作为执行逻辑的参考

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

using boost::asio::ip::tcp;

void on_connect(const boost::system::error_code& error)
    {
        if (!error) {
            std::cout << "Connected to the server!"
                << std::endl;
        } else {
            std::cerr << "Failed to connect: " <<
                error.message() << std::endl;
        }
}

void on_write(const boost::system::error_code& error,
    std::size_t bytes_transferred) {
        if (!error) {
            std::cout << "Message sent! Bytes
                transferred: " << bytes_transferred
                << std::endl;
        } else {
            std::cerr << "Failed to send message: "
                << error.message() << std::endl;
        }
}

int main() {
        try {
            boost::asio::io_context io_context;

            // 创建一个TCP socket
```

```
        tcp::socket socket(io_context);

        // 解析服务器地址
        tcp::resolver resolver(io_context);
        auto endpoints = resolver.resolve
            ("127.0.0.1", "8080");

        // 异步连接到服务器
        boost::asio::async_connect(socket,
            endpoints, on_connect);

        // 运行I/O上下文以处理异步操作
        io_context.run();

        // 构建要发送的消息
        std::string message = "Hello, Server!";

        // 异步发送消息到服务器
        boost::asio::async_write(socket, boost::
            asio::buffer(message), on_write);

        // 再次运行I/O上下文以处理异步写操作
        io_context.run();
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what()
            << std::endl;
    }

    return 0;
}
```

io_context.run() 会阻塞异步操作直到完成，这就为总体上的异步操作执行顺序提供了基准。

1.3 异步模型

本节会从高层视角展示 Boost.Asio 库的核心异步模型。该模型将异步操作作为异步组合的基本构建块，但将其与组合机制解耦。Boost.Asio 的异步操作支持回调、futures、纤程、协程和尚待想象的方法。允许应用编程人员基于合适的 trade-offs 来选择一种方式。

1.3.1 Asynchronous Operations

Asynchronous Operation 是 Boost.Asio 异步模型组成的基本单元。Asynchronous Operations 代表一种工作，它们在后台启动和运行，同时，用户初始化这些工作的代码可以继续做其他事情（不会忙等）。

理论上，异步操作的生命周期可以被描述为下列事件和阶段：Initiating function 是一个可以被用户调用来开启一个异步操作的函数。

Completion handler 是由用户提供的，move-only 的函数对象，它最多会被调用一次，并产生异步操作的结果。Completion handler 的调用告诉用户这样的事情：操作完成，并且这个操作的副作用被完成。

Initiating function 和 Completion handler 被合并到用户的代码中，如下所示：

同步操作体现为单个函数，因此具有几个固有的语义属性。异步操作采用了同步操作中的一些语义属性，以促进灵活高效的组合。

同步操作特性：

- 当同步操作是泛型时，返回类型是从函数及其参数中确定性地导出的
- 如果同步操作需要临时资源（如内存、文件描述符或线程），则在从函数返回之前释放此资源。

异步操作等价特性:

- 当异步操作是泛型操作时，完成处理程序的参数类型和顺序是从发起函数及其参数中确定性地导出的
- 如果异步操作需要临时资源（如内存、文件描述符或线程），则在调用完成处理程序之前释放此资源。

后者是异步操作的一个重要属性，因为它允许完成处理程序在不重叠资源使用的情况下启动进一步的异步操作。考虑在链中反复重复相同操作的平凡（相对常见）情况：

通过确保在完成处理程序运行之前释放资源，我们避免了操作链的峰值资源使用量加倍。

1.3.2 Asynchronous Agents

Asynchronous Agents 是一系列的异步操作的组合。每个异步操作都作为 Asynchronous Agents 的一部分运行，即使 Asynchronous Agents 只有单个异步操作。Asynchronous Agents 是一个实体，可以和其他 agents 同时工作。Asynchronous Agents 对于异步操作来说就像是线程对于同步操作的地位。

然而，异步代理是一种纯粹的概念性构造，它允许我们推理程序中异步操作的上下文和组合。“异步代理”这个名称不会出现在库中，使用哪种具体机制（Such as chains of lambdas, coroutines, fibers, state machines, etc）来组合代理中的异步操作也不重要。

可以将异步代理看作如下：

异步代理交替等待异步操作完成，然后为该操作运行完成处理程序。在代理的上下文中，这些完成处理程序表示不可分割的可调度工作单元。

1.3.3 Associated Characteristics and Associators

Asynchronous agent 具有 associated characteristics ，指定了异步操作在作为该代理的一部分组合时的行为方式，例如：

- 分配器，决定代理的异步操作如何获取内存资源。
- 取消槽，决定代理的异步操作如何支持取消。
- 执行器，决定代理的完成处理程序将如何排队和运行。

当在异步代理中运行异步操作时，其实现可能会查询这些相关特征，并使用它们来满足它们所代表的要求或偏好。异步操作通过将关联符特征应用于完成处理程序来执行这些查询。每个特征都有相应的关联特征。

关联特征可能专门用于具体的完成处理程序类型，以：

- 接受异步操作提供的默认特性，按原样返回此默认值
- 返回该特性的无关实现，或
- 调整提供的默认值，以引入完成处理程序所需的其他行为。

Associator 规范

给定一个名为 associated_R 的关联器特征 (The associator traits are named associated_allocator, associated_executor, and associated_cancellation_slot.)，具有：

- s 类型的源值 s（在这种情况下为完成处理程序及其类型），
- 定义相关特征的句法和语义要求的一组类型要求（或概念）R，以及
- 满足类型要求 R 的 c 类候选值 c，表示异步操作提供的相关特性的默认实现

异步操作使用关联器特性来计算：

- 类型 associated_R<S，C>：type，以及
- 关联的值 associated_R<S，C>：get（S，C）

满足 R 中定义的要求。为了方便起见，这些也可以分别通过类型别名 associated_R_t<S，C> 和函数 get_associate_R（S，C）访问。

该特征的主要模板被指定为：

- 如果 S:：R_type 格式正确，则将嵌套类型别名类型定义为 S:：R_ type，并定义一个返回 S.get_R () 的静态成员函数
- 否则，如果 associator<associated_R，S，C>：：type 格式良好并且表示一个类型，则继承自 associator<associated_R，S，C>
- 否则，将嵌套类型别名类型定义为 C，并定义一个返回 C 的静态成员函数 get。

1.3.4 Child Agents

代理内的异步操作本身可以通过子代理来实现 (In Boost.Asio these asynchronous operations are referred to as composed operations)。就父代理而言，它正在等待单个异步操作的完成。构成子代理的异步操作按顺序运行，当最终完成处理程序运行时，父代理将恢复。

与单个异步操作一样，构建在子代理上的异步操作必须在调用完成处理程序之前释放其临时资源。我们也可以将这些子代理视为在调用完成处理程序之前结束其生命周期的资源。

当异步操作创建子代理时，它可能会将父代理的相关特征传播 (Typically, by specialising the associator trait and forwarding to the outer completion handler) 到子代理。然后，这些相关特征可以通过异步操作和子代理的其他层递归传播。这种异步操作的堆叠复制了同步操作的另一个属性。

同步操作的特性：

- 同步操作的组合可以被重构，以使用在同一线程上运行的子函数（即简单调用），而不会改变功能

异步操作的等价特性：

- 异步代理可以被重构为使用异步操作和共享父代理相关特征的子代理，而不会改变功能。

最后，一些异步操作可以通过并发运行的多个子代理来实现。在这种情况下，异步操作可以选择性地传播父代理的相关特征。

1.3.5 Executors

每个异步代理都有一个关联的执行器。代理的执行器决定代理的完成处理程序如何排队并最终运行。

执行器的示例用法包括：

- 协调一组在共享数据结构上运行的异步代理，确保代理的完成处理程序永远不会并发运行（In Boost.Asio, this kind of executor is called a strand.）
- 确保代理在靠近数据或事件源（如 NIC）的指定执行资源（如 CPU）上运行。
- 表示一组相关的代理，从而使动态线程池能够做出更明智的调度决策（例如将代理作为一个单元在执行资源之间移动）。
- 将所有完成处理程序排队在 GUI 应用程序线程上运行，以便它们可以安全地更新用户界面元素。
- 按原样返回异步操作的默认执行器，以尽可能靠近触发操作完成的事件运行完成处理程序。
- 调整异步操作的默认执行器，使其在每个完成处理程序之前和之后运行代码，如日志记录、用户授权或异常处理。

- 为异步代理及其完成处理程序指定优先级。

异步代理中的异步操作使用代理的关联执行器来：

- 在操作未完成时跟踪异步操作所代表的工作的存在。
- 取消完成处理程序的队列，以便在操作完成时执行。
- 如果这样做可能会导致无意的递归和堆栈溢出，请确保完成处理程序不会重新运行。

因此，异步代理的关联执行器代表了一种策略，即代理应如何、在何处以及何时运行，该策略被指定为组成代理的代码的跨领域关注点。

1.3.6 Allocators

每个异步代理都有一个关联的分配器。代理的分配器是代理的异步操作使用的接口，用于获取每次操作的稳定内存资源（POSMs）。这个名称反映了这样一个事实，即内存是针对每个操作的，因为内存只在该操作的生命周期内保留，并且是稳定的，因为在整个操作过程中，内存保证在该位置可用。

第二章　使用 Asio

2.1　支持平台

下列平台和编译器的组合能够工作：

- Linux using g++ 4.6 or later
- Linux using clang 3.4 or later
- FreeBSD using g++ 9 or later
- macOS using Xcode 10 or later
- Win32 using Visual C++ 11.0 (Visual Studio 2012) or later
- Win64 using Visual C++ 11.0 (Visual Studio 2012) or later

下列平台可能正常工作：

- AIX
- Android
- HP-UX
- iOS
- NetBSD
- OpenBSD
- QNX Neutrino
- Solaris
- Tru64

- Win32 using MinGW.
- Win32 using Cygwin.

2.2 依赖

为了链接使用 Boost.Asio 的程序，下列库必须可用：

- 使用 boost::system::error_code 和 boost::system::system_error classes ，需要 Boost.System
- 使用 spawn() 启动协程，可能需要 Boost.Coroutine
- 使用任何 read_until() or async_read_until() 的重载，接收 boost::regex 参数，可能需要 Boost.Regex
- 使用 Boost.Asio's SSL 支持，可能需要 OpenSSL

第三章　指导教程

第四章　示例