Vasilui 2 сентября 2013 в 18:25

«Boost.Asio C++ Network Programming». Глава 1: Приступая к работе с Boost.Asio

C++, API

Из песочницы Tutorial

Привет Хабралюди!
Это мой первый пост, поэтому не судите строго. Я хочу начать **вольный** перевод книги John Torjo «Boost.Asio C++ Network Programming» вот ссылка на нее.

Содержание:

Во-первых разберем что есть Boost.Asio, как его собрать, а так же несколько примеров. Вы узнаете, что Boost.Asio больше, чем сетевая библиотека. Так же вы узнаете о самом важном классе, который находится в самом сердце Boost.Asio — `io_service` .

Что такое Boost.Asio?

Если коротко, Boost.Asio это, большей частью, кросс-платформенная C++ библиотека для программирования сетей и некоторых других низкоуровневых программ ввода/вывода.
Есть много реализаций для решения сетевых задач, но Boost.Asio превзошел их все; он был принят в Boost в 2005 и с тех пор был протестирован большим количеством пользователей Boost, используется во многих проектах, таких как:

- Remobo , позволяет вам создавать собственную IPN
- libtorrent , является библиотекой, которая реализует Bittorrent клиент
- PokerTH , представляет собой игру в покер с поддержкой LAN и Internet

Boost.Asio успешно абстрагирует понятия input и output, которые работают не только для работы в сети, но и для последовательных COM-портов, файлов и так далее. Кроме этого вы можете делать input или output программирование синхронным или асинхронным:

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

Как вы успели заметить в предыдущем фрагменте кода функции принимают экземпляр потока, которым может быть что угодно (не только сокет, мы можем читать и писать в него).
Библиотека является переносимой, работает на большинстве операционных систем и хорошо масштабируется при более чем тысяче одновременных подключений. Сетевая часть была последователем BSD(Berkeley Software Distribution) сокетов. Предоставляется API для работы с TCP (Transmission Control Protocol) сокетами, UDP (User Datagram Protocol) сокетами, IMCP(Internet Control Message Protocol) сокетами, так же библиотека является расширяемой, так, если вы хотите, то можете адаптировать ее на свой собственный протокол.

История

Boost.Asio была принята в Boost 1.35 в декабре 2005г, после начала разработки в 2003г. Первоначально автором является Кристофер Кохлхофф (Christopher M. Kohlhoff), с ним можно связаться по адресу chris@kohlhoff.com.
Библиотека была протестирована на следующих платформах и компиляторах:

- 32-bit и 64-bit Windows, используя Visual C++ 7.1 и выше
- Windows, с использованием MinGW
- Windows, используя Cygwin (убедитесь, что определен `__USE_232_SOCKETS`)
- Linux на основе ядер 2.4 и 2.6, используя g++ 3.3 и выше
- Solaris, используя g++ 3.3 и выше
- MAC OS X 10.4+, используя g++ 3.3 и выше

Это также может работать на платформах, таких как AIX 5.3, HP-UX 11i v3, QNX Neutrino 6.3, Solaris, с использованием Sun Studio 11+, True64 v5.1, Windows, с использованием Borland C++ 5.9.2+ (проконсультируйтесь на www.boost.org для уточнения деталей).

Зависимости

Boost.Asio зависит от следующих библиотек:

- Boost.System: эта библиотека предоставляет поддержку операционной системе для библиотеки Boost (http://www.boost.org /doc/libs/1_51_0/doc/html/boost_system/index.html)
- Boost.Regex: эта библиотека (опция) используется в случае, если вы используете `read_until()` или `async_read_until()` , которые принимают `boost::regex` параметр
- Boost.DateTime: эта библиотека (опция) используется, если вы используете таймеры Boost.Asio
- OpenSSL: эта библиотека (опция) используется, если вы решите использовать SSL поддержку, предоставляемую Boost.Asio

Сборка Boost.Asio

Boost.Asio это чисто заголовочная библиотека. Однако, в зависимости от компилятора и размера вашей программы, вы можете выбрать создание Boost.Asio как исходного файла. Вы можете сделать это для уменьшения времени компиляции. Это может быть сделано следующими способами:

- Только в одном из ваших файлов, используя `#include <boost/asio/impl/src.hpp>` (если вы используете SSL, то `#include <boost/asio/ssl/impl/src.hpp>`)
- Используя `#define BOOST_ASIO_SEPARATE_COMPILATION` во всех ваших исходных файлах

Заметьте, что Boost.Asio зависит от Boost.System и необязательно от Boost.Regex, так что вам нужно, по крайней мере, построить библиотеку boost, используя следующий код:

```
bjam –with-system –with-regex stage
```

Если вы так же хотите построить тесты, то вы должны использовать следующий код:

```
bjam –with-system –with-thread –with-date_time –with-regex –withserialization stage
```

Библиотека поставляется с большим количеством примеров, которые вы можете проверить, наряду с примерами, которые приводятся в этой книге.

Важные макросы

Используйте `BOOST_ASIO_DISABLE_THREADS` , если установлено; он отключает поддержку потоков в Boost.Asio, независимо от того был ли Boost скомпилирован с поддержкой потоков.

Синхронный против асинхронного

Во-первых, асинхронное программирование чрезвычайно отличается от синхронного программирования. В синхронном программировании все операции вы делаете в последовательном порядке, такие как чтение (запрос) из сокета S, а затем написать (ответ) в сокет. Каждая из операций является блокирующей. Так как операции блокирующие, то чтобы не прерывать основную программу, в то время как вы читаете или записываете в сокет, вы обычно создаете один или несколько потоков, которые имеют дело с сокетами ввода/вывода. Таким образом, синхронные сервер/клиенты, как правило, многопоточны.
С другой стороны, асинхронные программы управляются событиями. Вы начинаете операцию, но вы не знаете, когда она закончится; вы предоставляете `callback` функцию, которая будет вызываться API с результатом операции, когда операция завершится. Для программистов, которые имеют большой опыт работы с QT – кросс-платформенной библиотекой от Nokia для создания графических приложений пользовательского интерфейса, это вторая натура. Таким образом, в асинхронном программировании вам не обязательно иметь больше, чем один поток.
Вы должны решить на ранней стадии вашего проекта (желательно в начале) какой подход вы будете использовать: синхронный или асинхронный, так как переключение на полпути будет трудным и подвержено ошибкам; существенно отличается не только API, семантика вашей программы будет сильно изменена (асинхронные сети, как правило, труднее для тестирования и отладки, чем синхронные). Подумайте перед тем как захотите использовать либо блокирующие вызовы и много потоков (синхронная, как правило, проще) либо мало потоков и события (асинхронный, как правило, более сложный).
Вот простой пример синхронного клиента:

```
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);
```

Во-первых, ваша программа должна иметь экземпляр `io_service` . Boost.Asio использует `io_service` для общения с сервисом ввода/вывода операционной системы. Обычно одного экземпляра `io_service` бывает достаточно. Далее создайте

адрес и порт к которому вы хотите подключиться. Создайте сокет. Подключите сокет к вашему адресу и порту:

```
//Here is a simple synchronous server:using boost::asio;
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));
    }
}
```

Опять же, ваша первая программа должна иметь, по крайней мере, хотя бы один экземпляр `io_service` . Затем вы указываете порт для прослушивания и создаете акцептор (приемник) — один объект, который принимает клиентские подключения.
В следующем цикле вы создаете фиктивный сокет и ждете подключение клиента. После того как соединение установлено, вы создаете поток, который будет заниматься этой связью.
В потоке, в функции `client_session` вы слушаете запросы клиента, интерпретируете их и отвечаете.
Для создания простого асинхронного клиента вы будете делать что-то похожее на следующее:

```
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)
 {
    // here we know we connected successfully
    // if ec indicates success
}
```

Ваша программа должна иметь, по крайней мере один экземпляр `io_service` . Вы указываете где подключается и создается сокет. Затем, как только установлено соединение, вы асинхронно подключаетесь к адресу и порту (это завершение обработчика), то есть вызывается `connect_handler` .
После вызова `connect_handler` проверьте код на ошибки(`ec`), и в случае успеха вы можете асинхронно написать серверу.
Обратите внимание, что цикл `service.run()` будет выполняться до тех пор пока имеются незаконченные асинхронные операции. В предыдущем примере есть только одна такая операция, это сокет `async_connect` . После этого

`service.run()` завершается.
Каждая асинхронная операция имеет завершающий обработчик, функцию, которая будет вызвана, когда операция завершится. Следующий код это простой асинхронный сервер:

```
using boost::asio;
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);
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;
    // at this point, you can read/write to the socket
    socket_ptr sock(new ip::tcp::socket(service));
    start_accept(sock);
}
```

В предыдущем фрагменте кода, во-первых, вы создаете экземпляр `io_service` . Затем вы указываете порт, который будете прослушивать. Потом создаете акцептор – объект для приема клиентских подключений, а так же создаете фиктивный сокет и асинхронно ждете подключения клиента.
Наконец, запускаете асинхронный цикл `service.run()` . При подключении клиента вызывается `handle_accept` (завершающий обработчик для вызова `async_accept`). Если нет ошибок, то вы можете использовать этот сокет для операций чтения/записи.
После использования сокета, вы создаете новый сокет и снова вызываете `start_accept()` , которая добавляет подобную асинхронную операцию «ждем подключения клиента», оставляя цикл `service.run()` занятым.

Исключения против кодов ошибок

Boost.Asio позволяет использовать как исключения так и коды ошибок. Все синхронные функции имеют перегрузки выбрасывающие исключения в результате ошибки или возвращает код ошибки. Если функция падает, то она выбрасывает `boost::system::system_error` ошибку.

```
using boost::asio;
ip::tcp::endpoint ep;
ip::tcp::socket sock(service);
sock.connect(ep); // Line 1
boost::system::error_code err;
sock.connect(ep, err); // Line 2
```

В предыдущем коде `sock.connect(ep)` выбрасывает исключение в случае ошибки и `sock.connect(ep, err)` вернет код ошибки.
Взгляните на следующий фрагмент кода:

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

Следующий участок кода похож на предыдущий:

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

В случае, если вы используете асинхронные функции, все они возвращают код ошибки, который вы можете проверить в вашей функции обратного вызова. Асинхронные функции никогда не выкидывают исключений и не имеет никакого смысла делать это. А кто его поймает?
В ваших синхронных функциях вы можете использовать как исключения так и коды ошибок (что больше захотите), но пользуйтесь чем то одним. Их смешивание может привести к проблемам или даже падению (когда вы по ошибке забываете обработать исключение). Если ваш код является комплексным (вызываются функции чтения/записи в сокет), то вам, вероятно, предпочтительней пользоваться исключениями и выполнять функции чтения и записи в `try {} catch` блоке.

```
void client_session(socket_ptr sock)
{
    try
     {
        ...
    }
    catch ( boost::system::system_error e)
    {
        // handle the error
    }
}
```

Если вы используете коды ошибок, то вы можете хорошо увидеть, когда соединение будет закрыто, как показано в следующем фрагменте кода:

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

Все коды ошибок Boost.Asio находятся в пространстве имен `boost::asio::error` (в случае, если вы хотите сделать полный перебор для нахождения неисправности). Так же вы можете посмотреть `boost/asio/error.hpp` для получения более

подробной информации.

Потоки в Boost.Asio

Когда дело доходит до потоков в Boost.Asio, нам надо поговорить о следующем:

- `io_service` : класс `io_service` является потоко-безопасным. Несколько потоков могут вызвать `io_service::run()` . Чаще всего вы, вероятно, вызываете `io_service::run()` из одного потока, так что функция ждет пока все блокирующие асинхронные функции будут выполнены. Тем не менее вы можете вызывать `io_service::run()` из нескольких потоков. Это блокирует все потоки, которые будут вызывать `io_service::run()` . Все функции обратного вызов будут вызваны в контекстах вех потоков, которые вызвали `io_service::run()` ; это так же означает, что если вы вызвали `io_service::run()` , только в одном потоке, то все функции обратного вызова будут вызваны в контексте данного потока.

- `socket` : классы сокетов не являются потоко-безопасными. Таким образом вам следует избегать таких ситуаций как читать из сокета в одном потоке, а писать в него в другом (это рекомендуется вообще, не говоря уже о Boost.Asio).

- `utility` : Классы `utility` обычно не имеет смысла использовать в нескольких потоках, они не потоко-безопасны. Большинство из них используются короткое время, а затем выходят из области видимости.

Библиотека Boost.Asio сама по себе может использовать несколько потоков кроме вашего, но гарантируется, что из этих потоков не будет вызываться ваш код. Это в свою очередь означает, что функции обратного вызова будут вызваны только в тех потоках, откуда вызвана `io_service::run()` .

Не только сети

Boost.Asio в дополнение к сетям предоставляет и другие объекты ввода/вывода.
Boost.Asio позволяет использовать такие сигналы как `SIGTERM` (завершить программу), `SIGINT` (прерывание сигнала), SIGSEGV (нарушение сегмента) и другие.
Вы создаете экземпляр `signal_set` и указываете какие сигналы ждать асинхронно и когда один из них случится, то вызовется ваш асинхронный обработчик:

```
void signal_handler(const boost::system::error_code & err, int signal)
{
    // log this, and terminate application
}
boost::asio::signal_set sig(service, SIGINT, SIGTERM);
sig.async_wait(signal_handler);
```

Если сгенерируется `SIGINT` , то вы попадете в обработчик `signal_handler` .
Используя Boost.Asio, вы можете легко подключиться к последовательному порту. Имя порта COM7 на Windows или /dev/ttyS0 на POSIX платформах:

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

После открытия вы можете установить некоторые параметры, такие как скорость передачи данных порта, четность, стоп-биты, как указано в следующем фрагменте кода:

```
serial_port::baud_rate rate(9600);
```

```
sp.set_option(rate);
```

Если порт открыт, то вы можете обрабатывать его в потоке, к тому же рекомендуется использовать свободные функции для чтения и/или записи в последовательный порт, например, `read()`, `async_read()`, `write()`, `async_write()`, как показано в следующем примере:

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

Boost.Asio позволяет также подключаться к Windows-файлам и опять же использовать свободные функции, такие как `read()`, `asyn_read()` и другие, как показано ниже:

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

Тоже самое вы можете сделать и с дескрипторами POSIX файлов, таких как пайпы, стандартные I/O, различными устройствами (но не с обычными файлами), как это сделано в следующем фрагменте:

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

Таймеры

Некоторые операции ввода/вывода могут иметь временные ограничения для завершения. Вы можете применить это только к асинхронным операциям (так как синхронные средства блокирования не имеют временных ограничений). Например, следующее сообщение от вашего партнера должно прийти к вам через 100 миллисекунд:

```
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();
```

В предыдущем фрагменте кода если мы читаем наши данные до окончания времени, `read` установлено в `true`, то наш партнер достучался до нас вовремя. В противном случае, когда вызывается `deadline_handler`, `read` по-прежнему установлен в `false`, что означает, что мы не связались до конца отведенного времени.
Boost.Asio позволяет использовать синхронные таймеры, но, обычно, они эквивалентны простой операции `sleep`. Строчка `boost::this_thread::sleep(500);` и следующий фрагмент сделают тоже самое:

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

Класс io_service

Вы уже видели, что большая часть кода, которая использует Boost.Asio будет использовать некоторый экземпляр `io_service`. `io_service` – самый важный класс в библиотеке, он имеет дело с операционной системой, ждет конца всех асинхронных операций, а потом при завершении вызывает обработчик для каждой такой операции.
Если вы решили создать ваше приложение синхронным, то вам не нужно беспокоиться о том, что я собираюсь показать в этом разделе.
Вы можете использовать экземпляр `io_service` несколькими способами. В следующих примерах мы имеем три асинхронных операции, два соединенных сокета и таймер ожидания:

- Один поток с одним экземпляром `io_service` и одним обработчиком:

```
io_service service_;
// all the socket operations are handled by service_
ip::tcp::socket sock1(service_);
// all the socket operations are handled by service_
ip::tcp::socket sock2(service_);
sock1.async_connect( ep, connect_handler);
sock2.async_connect( ep, connect_handler);
deadline_timer t(service_, boost::posix_time::seconds(5));
t.async_wait(timeout_handler);
service_.run();
```

- Много потоков с одним экземпляром `io_service` и несколькими обработчиками:

```
io_service service_;
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::posix_time::seconds(5));
t.async_wait(timeout_handler);
for ( int i = 0; i < 5; ++i)
    boost::thread( run_service);
```

```
void run_service()
{
    service_.run();
}
```

- Много потоков с несколькими экземплярами `io_service` и несколькими обработчиками:

```
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::posix_time::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();
}
```

Прежде всего обратите внимание на то, что вы не можете иметь несколько экземпляров `io_service` в одном потоке. Не имеет смысла писать следующий код:

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

Предыдущий участок кода не имеет никакого смысла, потому что `service_[1].run()` потребует `service_[0].run()` при закрытии первого. Так что все асинхронные операции `service_[1]` должны будут ждать обработки, а это не очень хорошая идея.
Во всех трех предыдущих примерах мы ждали три асинхронные операции для завершения. Чтобы объяснить различия, мы будем считать, что, спустя некоторое время, завершится операция 1 и сразу после этого завершится операция 2. Так же мы предположим, что каждому обработчику потребуется секунда, чтобы завершиться.
В первом случае мы ждем завершения всех трех операций в одном потоке. После завершения первой операции мы вызываем ее обработчик. Даже если операция 2 завершается сразу же после первой, мы будем вынуждены ждать секунду для вызова ее обработчика после завершения первой операции.
Во втором случае мы ждем завершения трех операций в двух потоках. После завершения первой операции мы вызываем ее обработчик в первом потоке. Как только завершиться операция 2, мы сразу же вызовем ее обработчик во втором потоке (в то время как первый поток занят, ожидая завершения обработчика первой операции, второй поток может свободно ответить на завершение любой другой операции).
В последнем случае, если операция 1 будет `connect к sock1` и операция 2 `connect к sock2`, то приложение будет вести себя как и во втором случае. Первый поток будет обрабатывать обработчик `connect для sock1`, а второй поток – обработчик `connect для sock2`. Однако, если из `sock1` является операцией 1 и тайм-аут `deadline_timer t` это операция 2, то первый поток будет обрабатывать обработчик для `connect из sock1`. Таким образом, обработчику тайм-аута `deadline_timer t` придется ждать конца работы обработчика `connect из sock1` (он будет ждать одну секунду), в

первом потоке обрабатывается как подключение к обработчику `sock1`, так и обработчик тайм-аута `t`.
Вот что вы должны извлечь из предыдущих примеров:

- Ситуация 1 для базовых приложений. Вы всегда будете сталкиваться с проблемами, если несколько обработчиков должны быть вызваны одновременно или же если они должны будут вызываться последовательно. Если один обработчик требует много времени для окончания, то остальным обработчикам придется подождать.

- Ситуация 2 для большинства случаев. Это очень хорошо, если несколько обработчиков должны быть вызваны одновременно и каждый из них вызывается в отдельном потоке. Единственное узкое место может возникнуть, если все обрабатывающие потоки заняты и в то же время должны быть вызваны новые обработчики. Тем не менее, в качестве простого решения можно просто увеличить количество потоков-обработчиков.

- Ситуация 3 является наиболее сложной и более гибкой. Вы должны ее использовать только тогда, когда ситуации 2 недостаточно. Это вероятно будет возможно, когда у вас имеется более тысячи одновременных подключений (сокетов). Вы можете считать, что каждый поток-обработчик (поток, запустивший `io_service::run()`) имеет свой собственный цикл `select/epoll`; он ждет все сокеты, контролирует операции чтения/записи и найдя хотя бы одну такую операцию начинает обрабатывать ее. В большинстве случаев вам не о чем беспокоиться, беспокоиться можно только в том случае, когда количество сокетов растет экспоненциально (более 1000 сокетов). В таком случае наличие нескольких циклов `select/epoll` может увеличить время отклика.

Если вы думаете, что ваше приложение когда-нибудь перейдет к ситуации 3, то убедитесь, что участок кода (код, который вызывает `io_service::run()`) изолирован от остального кода, чтобы его можно было легко изменять.
И наконец всегда помните, что `.run()` всегда будет закончен, если больше нет операций для контроля, как показано в примере ниже:

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

В этом случае как только сокет установил соединение будет вызван `connect_handler` и `service.run()` завершится.
Если вы хотите, чтобы `service.run()` продолжил работать, вы должны предоставить ему больше работы. Есть два способа решения данной проблемы. Одним из способов является увеличение нагрузки на `connect_handler`, запустив еще одну асинхронную операцию.
Второй способ заключается в имитации некоторой его работы, используя следующий код:

```
typedef boost::shared_ptr<io_service::work> work_ptr;
work_ptr dummy_work(new io_service::work(service_));
```

Приведенный выше код обеспечит постоянную работу `service_.run()` до тех пор пока вы не вызовете `useservice_.stop()` или `dummy_work.reset(0); // destroy dummy_work`.

Резюме.

Boost.Asio является сложной библиотекой, которая делает программирование сетей довольно простым. Собрать ее просто. Она работает достаточно хорошо, избегая использования макросов; предоставляется несколько макросов для включения опций вкл/выкл, но есть несколько вещей, о которых нельзя забывать.
Boost.Asio поддерживает как синхронное так и асинхронное программирование. Эти два подхода очень разные и вы должны выбрать какой-то один и как можно раньше, так как переключение является довольно сложным и подвержено ошибкам.
Если вы выбрали синхронный подход, то вы можете выбирать между исключениями и кодами ошибок, переход от исключений к кодам ошибок довольно прост, надо добавить еще один аргумент в вызов функции (код ошибки).

Boost.Asio служит не только для программирования сетей. У этой библиотеки есть несколько особенностей, которые делают ее более ценной, такие как сигналы, таймеры и так далее.
В следующей главе мы будем вникать в множество функций и классов Boost.Asio, предусматривающие сети. Кроме того мы узнаем несколько трюков об асинхронном программировании.

Ну вот и все на сегодня, если Вам понравилось, то я с удовольствием продолжу перевод. Обо всех замечаниях пишите в комментариях.

Всем удачи!

68,5 Карма 0,0 Рейтинг

Подписаться

Кузьминых Василий @Vasilui
Пользователь

Комментарии 10

AndrewShmig 2 сентября 2013 в 21:35 +2

> Я хочу начать **вольный** перевод книги

Где-то я это уже видел :)

Спасибо за перевод! Надеюсь у вас хватит сил дойти до конца в этом трудном деле!

Ответить

Cupper 2 сентября 2013 в 23:23 +4

Хороший перевод. Правда как то сумбурно описана ситуация 3. Даже зная что там происходит, только со второго раза понял написанное.

Мое скромное имхо по поводу Boost: все хорошо кроме случаев когда нужны операции с таймаутами. Через ихние дедлайн таймеры это просто пипец. Вот нет бы просто в async_read/write/что_там_еще передать timeout и при его истечении в обработчике получить соответствующую ошибку (как например при потери соединения).

Ответить

RainM 3 сентября 2013 в 10:17 **+3**

Спасибо большое за работу! Надеюсь увидеть остальные главы!

Ответить

ProstoTyoma 3 сентября 2013 в 18:27 **0**

С COM портами и Asio у меня были проблемы. Он непредсказуемо разбивал входящие данные, пришедшие непрерывно, на куски. Это мешало работать с Modbus, у которого начало пакета детектится по времени между пакетами. После перехода на другую библиотеку, фактически Win API, заработало нормально.

Ответить

Cupper 4 сентября 2013 в 00:21 **+1**

Это не asio их разбивал, это их компорт так отдавал. Это нормальное поведение для COM порта на сколько я знаю. Я так же с ним работал. В начале я побайтно читал находя начало сообщения (синхронизовывал) а потом X байт за раз. Мне повезло у меня длина сообщения однозначно определялась в начале пакета данных.

Ответить

Vasilui 4 сентября 2013 в 08:42 **0**

Мне также пришлось поработать с COM портом и использовал я именно Asio. Читал побайтно до конца сообщения, так как точно знал символ конца сообщения. Все прекрасно работает и по сей день.

Ответить

ThisNameWasFree 14 октября 2013 в 00:42 **+1**

Автор, вот энто:

```
Try
{
…
}
catch ( boost::system::system_error e)
{
// handle the error
}
```

Заменить следующим:

```
try
{
…
}
catch ( boost::system::system_error &e)
{
// handle the error
}
```

Ответить

maxood 26 января 2019 в 01:22 0

тогда уж этим -

```
try {
…
} catch (const boost::system::system_error &e) {
// handle the error
}
```

Ответить

kovalexius 6 ноября 2015 в 12:05 0

Здравствуйте, господа — знатоки boost::asio!

Подскажите, умеет ли boost::asio слушать через определенный сетевой интерфейс? И умеет ли boost::asio коннектиться к удаленному интерфейсу через заданный локальный сетевой интерфейс? Если конечно этих интерфейсов несколько.

Если да, то краткий пример кода-намёк, если не трудно.

Спасибо!

Ответить

 **afiskon** 20 декабря 2015 в 14:26 0

Благодарю за перевод. Не могу однако не выразить обеспокоенность легальностью таких переводов.

Ответить

Написать комментарий

Предпросмотр Отправить Markdown (?)

САМОЕ ЧИТАЕМОЕ

Сутки Неделя Месяц

Собеседование в Яндекс: театр абсурда :/

+564 182k 449 1102 +1075

Дата-центр возле Амстердама называют «выгребной ямой интернета», но он продолжает работу

+42 22,9k 45 80 +54

Люди подозревают, что технологии — отстой, потому что они на самом деле отстой

+57 25k 40 256 +256

Листая старые подшивки. Взгляд изнутри на компьютерную прессу 90-х

+101 14,5k 55 92 +92

E-com-корзины как полуфабрикат. Кейсы вендоров

Мегапост