



Vasilui 9 сентября 2013 в 14:56

«Boost.Asio C++ Network Programming». Глава 2: Основы Boost.Asio. Часть 1

C++, API

Tutorial

Всем привет!

Продолжаю перевод книги John Torjo «Boost.Asio C++ Network Programming». Вторая глава получилась большая, поэтому разобью ее на две части. В этой части мы поговорим именно про основы Boost.Asio, а во второй части речь пойдет про асинхронное программирование.

Содержание:

- [Глава 1: Приступая к работе с Boost.Asio](#)
- Глава 2: Основы Boost.Asio
 - **Часть 1: Основы Boost.Asio**
 - [Часть 2: Асинхронное программирование](#)
- [Глава 3: Echo Сервер/Клиент](#)
- [Глава 4: Клиент и Сервер](#)
- [Глава 5: Синхронное против асинхронного](#)
- [Глава 6: Boost.Asio – другие особенности](#)
- [Глава 7: Boost.Asio – дополнительные темы](#)

В этой главе мы рассмотрим то, что вам обязательно знать, используя Boost.Asio. Мы углубимся в асинхронное программирование, которое намного сложнее, чем синхронное и гораздо более интересное.

Сетевое API

В этом разделе показано, что вам необходимо знать, чтобы написать сетевое приложение с использованием Boost.Asio.

Пространства имен Boost.Asio

Все в Boost.Asio находится в пространстве имен `boost::asio` или его подпространстве, рассмотрим их:

- `boost::asio` : Это где находятся все основные классы и функции. Главные классы это `io_service` и `streambuf` . Здесь находятся такие функции как `read`, `read_at`, `read_until` , их асинхронные копии, а так же функции записи и их асинхронные копии.
- `boost::asio::ip` : Это место, где находится сетевая часть библиотеки. Основные классы это `address`, `endpoint`, `tcp`, `udp`, `icmp` , а основные функции это `connect` и `async_connect` . Обратите внимание, что `socket` в `boost::asio::ip::tcp::socket` это просто `typedef` внутри класса `boost::asio::ip::tcp` .
- `boost::asio::error` : Это пространство имен содержит коды ошибок, которые вы можете получить при вызове

подпрограммы ввода/вывода

- `boost::asio::ssl` : Это пространство имен содержит классы, имеющие дело с SSL.
- `boost::asio::local` : Это пространство имен содержит POSIX-специфичные классы
- `boost::asio::windows` : Это пространство имен содержит Windows-специфичные классы

IP адреса

Для работы с IP адресами Boost.Asio предоставляет классы `ip::address`, `ip::address_v4` и `ip::address_v6`. Они предоставляют множество функций. Вот наиболее важные из них:

- `ip::address(v4_or_v6_address)` : Эта функция конвертирует v4 или v6 адрес в `ip::address`
- `ip::address::from_string(str)` : Эта функция создает адрес из IPv4 адреса (разделенных точками) или из IPv6 (шестнадцатиричный формат)
- `ip::address::to_string()` : Эта функция возвращает представление адреса в благоприятном строчном виде
- `ip::address_v4::broadcast([addr, mask])` : Эта функция создает broadcast адрес
- `ip::address_v4::any()` : Эта функция возвращает адрес, который олицетворяет любой адрес
- `ip::address_v4::loopback()`, `ip::address_v6::loopback()` : Эта функция возвращает шлейф адресов (из v4/v6 протокола)
- `ip::host_name()` : Эта функция возвращает имя текущего хоста в виде строки

Скорее всего чаще всего вы будете использовать функцию `ip::address::from_string` :

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

Если вам необходимо подключиться к имени хоста, читайте дальше. Следующий код не будет работать:

```
// throws an exception
ip::address addr = ip::address::from_string("www.yahoo.com");
```

Конечные точки (Endpoints)

Конечная точка это адрес подключения вместе с портом. Каждый тип сокетов имеет свой endpoint класс, например, `ip::tcp::endpoint`, `ip::udp::endpoint`, и `ip::icmp::endpoint`.

Если вы хотите подключиться к `localhost` по 80 порту, то вам нужно написать следующее:

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

Вы можете создать конечную точку тремя способами:

- `endpoint()` : конструктор по умолчанию и он может быть иногда использован для UDP/ICMP сокетов

- `endpoint(protocol, port)` : обычно используется на серверных сокетах для приема новых подключений
- `endpoint(addr, port)` : создание конечной точки по адресу и порту

Вот несколько примеров:

```
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);
```

Если же вы хотите подключиться к хосту (не IP адресу), то вам нужно сделать следующее:

```
// outputs "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;
```

Можете заменить tcp на нужный вам тип сокета. Во-первых, создайте запрос с именем, к которому хотите подключиться, это можно реализовать с помощью функции `resolve()` . В случае успеха вернется хотя бы одна запись.

Получив конечную точку, вы можете получить из нее адрес, порт и IP протокол (v4 или v6):

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

Сокеты

Boost.Asio включает в себя три типа классов сокетов: `ip::tcp`, `ip::udp`, и `ip::icmp`, ну и, конечно же, расширяется. Вы можете создать свой собственный класс сокета, хотя это довольно сложно. В случае если вы все же решите сделать это посмотрите `boost/asio/ip/tcp.hpp`, `boost/asio/ip/udp.hpp`, и `boost/asio/ip/icmp.hpp`. Все они это довольно маленькие классы с внутренними `typedef` ключевыми словами.

Вы можете думать о классах `ip::tcp`, `ip::udp`, `ip::icmp` как о заполнителях; они дают возможность легко добраться до других классов/функций, которые определяются следующим образом:

- `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`

Класс `socket` создает соответствующий сокет. Вы всегда передаете экземпляр `io_service` в конструктор:

```
io_service service;
ip::udp::socket sock(service)
sock.set_option(ip::udp::socket::reuse_address(true));
```

Каждое имя сокета имеет typedef :

- `ip::tcp::socket= basic_stream_socket`
`ip::udp::socket= basic_datagram_socket<ud p>`
`ip::icmp::socket= basic_raw_socket`

Коды ошибок синхронных функций

Все синхронные функции имеют перегрузки, которые выбрасывают исключения или возвращают код ошибки, как показано ниже:

```
sync_func( arg1, arg2 ... argN); // throws
boost::system::error_code ec;
sync_func( arg1 arg2, ..., argN, ec); // returns error code
```

В оставшейся части главы, Вы увидите много синхронных функций. Чтобы не усложнять, я опустил показ перегрузок, которые возвращают код ошибки, но они существуют.

Функции сокетов

Все функции разделены на несколько групп. Не все функции доступны для каждого типа сокета. Список в конце этого раздела покажет вам, какие функции к какому классу сокетов относятся.

Отметим, что все асинхронные функции отвечают мгновенно, в то время как их коллеги синхронные отвечают только после того, как операция была завершена.

Соединительно-связывающие функции

Это функции, которые подключаются или соединяются с сокетом, отключают его и делают запрос о подключении, активно оно или нет:

- `assign(protocol, socket)` : эта функция присваивает сырой(естественный) сокет к экземпляру сокета. Используйте ее при работе с наследуемым кодом (то есть когда сырые сокеты уже созданы).
- `open(protocol)` : эта функция открывает сокет с заданным IP-протоколом (v4 или v6). Вы будете использовать ее в основном для UDP/ICMP сокетов или серверных сокетов
- `bind(endpoint)` : эта функция связывается с данным адресом.
- `connect(endpoint)` : эта функция синхронно подключается по данному адресу.
- `async_connect(endpoint)` : эта функция асинхронно подключается по данному адресу.
- `is_open()` : эта функция возвращает true если сокет открыт.
- `close()` : эта функция закрывает сокет. Любые асинхронные операции на этом сокете немедленно прекращаются и возвращают `error::operation_aborted` код ошибки.
- `shutdown(type_of_shutdown)` : эта функция отключает операцию `send` , `receive` или обе сразу же после вызова.
- `cancel()` : эта функция отменяет все асинхронные операции на этом сокете. Все асинхронные операции на этом сокете будут немедленно завершены и вернут `error::operation_aborted` код ошибки.

Приведем небольшой пример:

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

Функции чтения/записи

Это функции, которые выполняют ввод/вывод на сокете.

Для асинхронных функций обработчик имеем следующую сигнатуру `void handler(const boost::system::error_code& e, size_t bytes);`. А вот сами функции:

- `async_receive(buffer, [flags], handler)` : эта функция запускает асинхронную операцию получения данных от сокета.
- `async_read_some(buffer, handler)` : эта функция эквивалента `async_receive(buffer, handler)`.
- `async_receive_from(buffer, endpoint[, flags], handler)` : эта функция запускает асинхронное получение данных от определенного адреса.
- `async_send(buffer[, flags], handler)` : эта функция запускает операцию асинхронной передачи данных из буфера
- `async_write_some(buffer, handler)` : эта функция эквивалентна `async_send(buffer, handler)`.
- `async_send_to(buffer, endpoint, handler)` : эта функция запускает операцию асинхронной передачи данных из буфера по определенному адресу.
- `receive(buffer[, flags])` : эта функция синхронно принимает данные в буфер. Функция заблокирована пока не начнут приходить данные или если произошла ошибка.
- `read_some(buffer)` : эта функция эквивалентна `receive(buffer)`.
- `receive_from(buffer, endpoint[, flags])` : эта функция синхронно принимает данные от определенного адреса в данный буфер. Функция заблокирована пока не начали приходить данные или если произошла ошибка.
- `send(buffer[, flags])` : эта функция синхронно отправляет данные из буфера. Функция заблокирована пока идет отправка данных или если произошла ошибка.
- `write_some(buffer)` : эта функция эквивалентна `send(buffer)`.
- `send_to(buffer, endpoint[, flags])` : эта функция синхронно передает данные из буфера по данному адресу. Функция заблокирована пока идет отправка данных или если произошла ошибка.
- `available()` : эта функция возвращает количество байт, которое можно считать синхронно, без блокировки.

Мы будем говорить о буферах в ближайшее время. Давайте рассмотрим флаги. Значение по умолчанию для флагов равно 0, но возможны комбинации:

- `ip::socket_type::socket::message_peek` : этот флаг только заглядывает в сообщение. Он вернет сообщение, но при следующем вызове, чтобы прочитать сообщение нужно будет его перечитать.
- `ip::socket_type::socket::message_out_of_band` : этот флаг обрабатывает вне-полосные данные(out-of-band). ООВ данные это данные, которые помечены как более важные, по сравнению с обычными данными. Обсуждение ООВ данных выходит за рамки данной книги.
- `ip::socket_type::socket::message_do_not_route` : этот флаг указывает, что сообщение должно быть отправлено без использования таблиц маршрутизации.
- `ip::socket_type::socket::message_end_of_record` : этот флаг указывает на то, что данные помечены маркером об окончании записи. Это не поддерживается в Windows.

Скорее всего вы использовали `message_peek` , если когда-нибудь писали следующий код:

```
char buff[1024];
sock.receive(buffer(buff), ip::tcp::socket::message_peek );
memset(buff, 1024, 0);
// re-reads what was previously read
sock.receive(buffer(buff) );
```

Ниже приведены примеры, которые дают указания читать как синхронно так и асинхронно различным типам сокетов:

- Пример 1: синхронное чтение и запись в TCP сокет:

```
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));
```

- Пример 2: синхронное чтение и запись в UDP сокет:

```
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);
```

Обратите внимание, что читая из UDP сокета с использованием `receive_from` , вам необходимо использовать конструктор по умолчанию конечной точки, как показано в предыдущем примере.

- Пример 3: асинхронное чтение из UDP серверного сокета:

```
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();
}

```

Управление сокетом

Эти функции работают с дополнительными параметрами сокета:

- `get_io_service()` : эта функция возвращает экземпляр `io_service` , который был принят в конструкторе.
- `get_option(option)` : эта функция возвращает параметр сокета
- `set_option(option)` : эта функция устанавливает параметр сокета
- `io_control(cmd)` : эта функция приводит в исполнение команды ввода/вывода на сокете.

Следующие параметры вы можете получать/задавать сокету:

Имя	Определение	Тип
<code>broadcast</code>	Если <code>true</code> , то позволяет широковещательные сообщения	<code>bool</code>
<code>debug</code>	Если <code>true</code> , то позволяет отладку на уровне сокетов	<code>bool</code>
<code>do_not_route</code>	Если <code>true</code> , то предотвращает маршрутизацию и использует только локальные интерфейсы	<code>bool</code>
<code>enable_connection_aborted</code>	Если <code>true</code> , то переподключает оборванное соединение	<code>bool</code>
<code>keep_alive</code>	Если <code>true</code> , то посылает <code>keep-alives</code>	<code>bool</code>
<code>linger</code>	Если <code>true</code> , то сокет задерживается на <code>close()</code> , если есть не сохраненные данные	<code>bool</code>
<code>receive_buffer_size</code>	Размер буфера приема	<code>int</code>

receive_low_watermark	Предоставляет минимальное число байт при обработке входного сокета	int
reuse_address	Если true , то сокет может быть связан с адресом, который уже используется	bool
send_buffer_size	Размер буфера отправки	int
send_low_watermark	Предоставляет минимальное число байт для отправки в выходном сокете	int
ip::v6_only	Если true , то позволяет использовать только IPv6 связи	bool

Каждое имя представляет собой внутренний typedef сокета или класса. Вот как их можно использовать:

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 80);
ip::tcp::socket sock(service);
sock.connect(ep);
// TCP socket can reuse address
ip::tcp::socket::reuse_address ra(true);
sock.set_option(ra);
// get sock receive buffer size
ip::tcp::socket::receive_buffer_size rbs;
sock.get_option(rbs);
std::cout << rbs.value() << std::endl;
// set sock's buffer size to 8192
ip::tcp::socket::send_buffer_size sbs(8192);
sock.set_option(sbs);
```

Сокет должен быть открыт для работы предыдущей функции, иначе вылетит исключение.

TCP против UDP и ICMP

Как я уже сказал, не все функции-члены доступны для всех классов сокетов. Я составил список, где функции-члены отличаются. Если функции-члена здесь нет, то это означает, что она присутствует во всех классах сокетов:

Name	TCP	UDP	ICMP
async_read_some	Yes	-	-
async_write_some	Yes	-	-
async_send_to	-	Yes	Yes
read_some	Yes	-	-
receive_from	-	Yes	Yes
write_some	Yes	-	-
send_to	-	Yes	Yes

Прочие функции

Остальные функции, связанные с соединением или вводом/выводом:

- `local_endpoint()` : эта функция возвращает адрес, если сокет подключен локально.
- `remote_endpoint()` : эта функция возвращает удаленные адреса, куда сокет был подключен.
- `native_handle()` : эта функция возвращает чистый сокет. Ее нужно использовать только тогда, когда вы хотите использовать функции для работы с чистыми сокетами, которые не поддерживаются Boost.Asio.
- `non_blocking()` : эта функция возвращает `true` если сокет неблокирующий, иначе `false`.
- `native_non_blocking()` : эта функция возвращает `true` если сокет неблокирующий, иначе `false`. Тем не менее он будет вызывать чистый API для естественного сокета. Как правило вам это не нужно (`non_blocking()` всегда кэширует этот результат); вы должны использовать ее только тогда, когда вы непосредственно имеете дело с `native_handle()`.
- `at_mark()` : эта функция возвращает `true`, если вы собираетесь читать в сокете ООВ данные. Она нужна очень редко.

Другие соображения

И напоследок, экземпляр сокета не может быть скопирован, так как конструктор копирования и `operator=` недоступны.

```
ip::tcp::socket s1(service), s2(service);
s1 = s2; // compile time error
ip::tcp::socket s3(s1); // compile time error
```

В этом много смысла, так как каждый экземпляр хранит и управляет ресурсами (сам естественный сокет). Если бы мы использовали копирующий конструктор, то в конечном итоге мы имели два экземпляра одного и того же сокета; они должны были бы как то управлять правом собственности (либо один экземпляр имеет право собственности, или используется подсчет ссылок, или какой-то другой метод). В Boost.Asio было принято запретить копирование (если вы хотите создавать копии, просто используйте общий (`shared`) указатель).

```
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
```

Буферы сокетов

При чтении или записи в сокет, вам понадобится буфер, который будет содержать входящие или исходящие данные. Память буфера должна пережить операции ввода/вывода; вы должны убедиться, что до тех пор пока она не освобождена, она не выйдет из области видимости пока длятся операции ввода/вывода.

Это очень просто для синхронных операций; конечно, `buff` должен пережить обе операции `receive` и `send`:

```
char buff[512];
```

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

И это не так просто для асинхронных операций, как показано в следующем фрагменте:

```
// very bad code ...
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);
}
```

После вызова `async_receive()`, `buff` выйдет из области видимости, таким образом его память будет освобождена. Когда мы собираемся на самом деле получить некоторые данные на сокете, то надо скопировать их в память больше нам не принадлежащую; она может быть либо освобождена, либо перераспределена по коду для других данных, при этом всем имеет повреждение памяти.

Есть несколько решений поставленной задачи:

- Использовать глобальный буфер
- Создать буфер и уничтожить его, когда операция завершится
- Иметь объект связи для поддержки сокета и дополнительные данные, такие как буфер (ы).

Первое решение не очень удобно, так как мы все знаем, что глобальные переменные это плохо. Кроме того, что произойдет, если два обработчика будут использовать один и тот же буфер?

Вот как вы можете реализовать второе решение:

```
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));
```

Если вы хотите, чтобы буфер автоматически выходил из области видимости, когда завершается операция, то используйте общий указатель (`shared pointer`):

```
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);
}
};
// when on_read goes out of scope, the boost::bind object is released,
// and that will release the shared_buffer as well
void on_read(shared_buffer, const boost::system::error_code & err,
std::size_t read_bytes) {}
...
shared_buffer buff(512);
sock.async_receive(buff.asio_buff(), boost::bind(on_read, buff, _1, _2));

```

Класс `shared_buffer` содержит внутри себя `shared_array<>`, который является копией экземпляра `shared_buffer`, так что `shared_array<>` будет оставаться в живых; когда последний выйдет из области видимости, `shared_array<>` автоматически разрушится, как раз то, что мы хотели.

Это работает как и следовало ожидать, так как Boost.Asio будет держать копию завершающего обработчика, который вызывается при завершении операции. Эта копия является функтором `boost::bind`, который внутри хранит копию нашего экземпляра `shared_buffer`. Это очень аккуратно!

Третий вариант состоит в использовании объекта связи, который поддерживает сокет и содержит дополнительные данные, такие как буферы, обычно это правильное решение, но довольно сложное. Оно будет рассмотрено в конце этой главы.

Врапер функции буфера

В коде, который мы видели раньше, мы всегда нуждались в буфере для операций чтения/записи, код оборачивался в объект реального буфера, в вызов `buffer()` и передачу его функции:

```

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

```

В основном оборачивается любой буфер, который у нас есть в классе, что позволяет функциям из Boost.Asio итерироваться по буферу. Скажите, вы используете следующий код:

```

sock.async_receive(some_buffer, on_read);

```

Экземпляр `some_buffer` должен удовлетворять некоторым требованиям, а именно `ConstBufferSequence` или `MutableBufferSequence` (вы можете посмотреть о них более подробно в документации по Boost.Asio). Подробности создания своего собственного класса для удовлетворения этих требований являются довольно сложными, но Boost.Asio уже содержит некоторые классы, моделирующие эти требования. Вы не обращаетесь к ним напрямую, вы используете функцию `buffer()`.

Достаточно сказать, что вы можете обернуть все ниже следующее в функцию `buffer()`:

- константный массив символов

- `void*` и размер в символах
- строку `std::string`
- константный массив `POD[]` (POD подходит для старых данных, то есть конструктор и деструктор ничего не делают)
- массив `std::vector` из любых POD
- массив `boost::array` из любых POD
- массив `std::array` из любых POD

Следующий код рабочий:

```
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);
```

В общем, вместо того, чтобы создавать свой собственный класс для удовлетворения требований `ConstBufferSequence` или `MutableBufferSequence`, вы можете создать класс, который будет содержать буфер до тех пор, пока это необходимо и возвращать экземпляр `mutable_buffers_1`, это то же самое, что мы делали в классе `shared_buffer` ранее.

Независимые функции чтения/записи/подключения

Boost.Asio дает вам независимые функции для работы с вводом/выводом. Я разделил их на четыре группы.

Функции подключения

Эти функции подключают сокет или конечный адрес:

- `connect(socket, begin [, end] [, condition])`: эта функция пытается синхронно подключить каждый конечный адрес в последовательности, начиная с `begin` и заканчивая `end`. Итератор `begin` является результатом вызова `socket_type::resolver::query` (если хотите, можете посмотреть раздел «Конечные точки» еще раз). Указание конечного итератора не является обязательным, вы можете забыть об этом. Вы можете указать `condition` функции, которое вызывается перед каждой попыткой подключения. Это есть сигнатура `Iterator connect_condition(const boost::system::error_code & err, Iterator next)`. Вы можете выбрать другой итератор для возврата, чем следующий, это позволяет пропускать некоторые конечные адреса.

- `async_connect(socket, begin [, end] [, condition], handler)`: эта функция выполняет асинхронное подключение и в конце вызывает обработчик завершения. Сигнатура обработчика следующая `void handler(const boost::system::error_code & err, Iterator iterator)`; . Вторым параметром, передаваемым в обработчик является успешно подключенный конечный адрес (или итератор `end` в противном случае).

Приведем следующий пример:

```
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);
```

Имя хоста может вмещать более одного адреса, таким образом `connect` и `async_connect` освобождают вас от бремени проверять каждый адрес, чтобы понять какой из них доступен; они делают это за вас.

Функции чтения/записи

Это функции чтения или записи в поток (который может быть как сокетом так и любым другим классом, который ведет себя как поток):

- `async_read(stream, buffer [, completion] , handler)` : эта функция асинхронно читает из потока. По завершении вызывается обработчик. Он имеет следующую сигнатуру `void handler(const boost::system::error_code & err, size_t bytes)`; . При необходимости вы сами можете задать `completion` функцию. `Completion` функция вызывается после каждой успешной операции `read` , и сообщает Boost.Asio если функция `async_read` завершилась (если нет, то функция продолжит чтение). Следующий параметр это `size_t completion (const boost::system::error_code& err, size_t bytes_transferred)` . Когда эта завершающая функция возвращает 0, мы считаем, что операция чтения завершилась; если же вернулось ненулевое значение, то это означает, что максимальное количество байт будет прочитано при следующем вызове операции `async_read_some` в потоке. Далее будет рассмотрен пример для более лучшего понимания.
- `async_write(stream, buffer [, completion], handler)` : это функция асинхронной записи в поток. Список аргументов схож с `async_read` .
- `read(stream, buffer [, completion])` : это функция синхронного чтения из потока. Список аргументов схож с `async_read`
- `write(stream, buffer [, completion])` : это функция синхронной записи в поток. Список аргументов схож с `async_read` .
- `async_read(stream, stream_buffer [, completion], handler)`
- `async_write(stream, stream_buffer [, completion], handler)`
- `write(stream, stream_buffer [, completion])`
- `read(stream, stream_buffer [, completion])`

Во-первых заметим, что вместо сокета первым аргументом передается поток. Сюда может передаваться сокет, но этим не ограничивается. Например, вместо сокета вы можете использовать файл Windows.

Каждая операция чтения или записи закончится при выполнении одного из следующих условий:

- Предоставленный буфер заполнится (для чтения) или все данные в буфере будут записаны (для записи).
- `Completion` функция вернет 0 (если вы предоставили одну из таких функций).
- Если произошла ошибка.

Следующий код асинхронно читает, пока не встретит `'\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);
```

Кроме того, Boost.Asio предоставляет для помощи несколько `completion` функций:

- `transfer_at_least(n)`
- `transfer_exactly(n)`
- `transfer_all()`

Это иллюстрируется в следующем примере:

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

Последние четыре функции вместо обычного буфера используют функцию `stream_buffer` из Boost.Asio, это производная от `std::streambuf`. Сами потоки и их буферы из STL являются очень гибкими, вот пример:

```
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();
}

```

Здесь я показал вам, что вы можете вызвать `async_read` (или что-то подобное) для файлов Windows. Мы считываем первые 256 символов и сохраняем их в буфер. Когда операция чтения завершится, будет вызвана `on_read`, я создаю `std::istream` буфер, считываю первую строку (`std::getline`) и вывожу это в консоль.

Функции `read_until/async_read_until`

Эти функции производят чтение пока не будет выполнено некоторое условие:

- `async_read_until(stream, stream_buffer, delim, handler)`: эта функция начинает операцию асинхронного чтения. Операция чтения остановится как только будет прочитан разделитель (`delim`). Разделителем может быть любой символ, `std::string` или `boost::regex`. Сигнатура обработчика следующая `void handler(const boost::system::error_code & err, size_t bytes);`.
- `async_read_until(stream, stream_buffer, completion, handler)`: эта функция такая же как и предыдущая, но вместо разделителя мы имеем завершающую функцию. Она имеет следующую сигнатуру `pair<iterator, bool> completion(iterator begin, iterator end);`, когда итератор есть `buffers_iterator<streambuf::const_buffers_type>`. Вам необходимо помнить, что итератор имеет тип итератора произвольного доступа. Вы смотрите в диапазоне (`begin`, `end`) и сами решаете должна ли операция чтения завершиться или нет. У вас возвращается пара; первый член которой это итератор указывающий на последний символ прочитанный функцией; второй член это `true`, в противном случае, если операция чтения должна прекратиться, то `false`.
- `read_until(stream, stream_buffer, delim)`: эта функция выполняет операцию синхронного чтения. Значения параметров являются такими же как и у `async_read_until`.

В следующем примере мы будем читать до знаков препинания:

```

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);
```

Если бы мы хотели читать до пробела, то изменили бы последнюю строчку на:

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

Функции *_at

Это функции случайных операций чтения/записи в поток. Вы указываете, где операции чтения/записи должны начаться (указываете смещение):

- `async_read_at(stream, offset, buffer [, completion], handler)` : эта функция начинает операцию асинхронного чтения в данном потоке, начиная со смещения `offset` . При завершении операции будет вызван обработчик `handler (const boost::system::error_code& err, size_t bytes)` ; . Буфер может быть как обычной обвязкой `buffer()` так и функцией `streambuf` . Если задать функцию завершения, то она будет вызываться после каждого успешного чтения и сообщает Boost.Asio если операция `async_read_at operation` завершилась (если нет, то чтение продолжится). Сигнатура завершающей функции следующая `size_t completion(const boost::system::error_code& err, size_t bytes)` ; . Когда эта функция возвращает 0, то мы считаем, что операция чтения завершилась, если же вернулось ненулевое значение, то это означает, что максимальное число байт будет прочитано при следующем вызове `async_read_some_at` в этом потоке.
- `async_write_at(stream, offset, buffer [, completion], handler)` : эта функция запускает операцию асинхронной записи. Параметры схожи с `async_read_at` .
- `read_at(stream, offset, buffer [, completion])` : эта функция читает со смещением в данном потоке. Параметры схожи с `async_read_at` .
- `read_at(stream, offset, buffer [, completion])` : эта функция читает со смещением в данном потоке. Параметры схожи с `async_read_at` .

Эти функции не имеют дело с сокетами. Они имеют дело с потоками случайного доступа; другими словами, с потоками, которые могут быть доступны в случайном порядке. Сокеты явно не этот случай (сокеты являются `forward-only`).

Вот как вы можете прочитать 128 байт из файла, начиная со смещением в 256:

```
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;
}
+38 190 77,2k 6 Поделиться
```


**68,5**

Карма

0,0

Рейтинг



Подписаться

Кузьминых Василий @Vasilui

Пользователь

ПОХОЖИЕ ПУБЛИКАЦИИ

17 сентября 2015 в 03:51

Инкапсуляция интерфейсов. Делаем API в C++ удобным и понятным11 1 158 22 **+22**

11 июля 2014 в 04:16

О бедном C++ API замолвите слово!20 35,9k 174 71 **+71**

30 апреля 2013 в 10:53

Добавляем Web API для программы на C++ с помощью библиотеки POCO24 22,2k 117 6 **+6**

КУРСЫ

**C# Developer. Professional**

30 апреля 2021 • 6 месяцев • 65 000 Р • OTUS

**Программирование на языке C (Си)**

9 апреля 2021 • 4 недели • 23 350 Р • Специалист.ру

**Язык программирования C# 7.3**

12 апреля 2021 • 4 недели • 26 990 Р • Специалист.ру

**Linux (Ubuntu). Уровень 2. Программирование в Linux на C**

19 апреля 2021 • 3 недели • 23 990 Р • Специалист.ру

**Python для веб-разработки**

9 апреля 2021 • 9 месяцев • 59 400 Р • SkillFactory

[Больше курсов на Хабр Карьере](#)**Комментарии 6**Отслеживать новые в ☐ почте ☐ трекере**alex_shabalin** 9 сентября 2013 в 19:05**+1**

Интересно и понятно. Спасибо.

[Ответить](#)**ArtemE** 9 сентября 2013 в 22:54**+1**[@Vasilui](#), вы переводите с целью лучше изучить материал? Хороший перевод, технический и довольно грамотный.

[Ответить](#)



Vasilui 10 сентября 2013 в 08:50

+2

И с этой целью тоже, а вообще хочется, чтобы в сети было по-больше русскоязычного материала по таким хорошим библиотекам, как Boost.Asio.

[Ответить](#)

Cupper 10 сентября 2013 в 12:29

0

Кто бы мог по простому объяснить смысл вот этих опций сокетов

debug, enable_connection_aborted, linger

а то просто перевод документации на них не проясняет зачем они и что они делают.

[Ответить](#)



Dubovik_a 26 февраля 2017 в 01:19

0

Слишком похоже на автоматический перевод. Для русскоговорящего человека совсем не понимать.

[Ответить](#)

RoyalDutch 21 марта 2021 в 13:31

0

Вот это повторяется дважды:

| read_at(stream, offset, buffer [, completion]): эта функция читает со смещением в данном потоке. Параметры схожи с async_read_at.

Думаю вместо второго раза имелось ввиду это:

| write_at(stream, offset, buffer [, completion]): эта функция пишет со смещением в данном потоке. Параметры схожи с async_read_at.

Это единственный материал на русском, который мне получилось найти — спасибо большое за проделанную работу. Теперь хоть что то становится понятно

[Ответить](#)

Написать комментарий

B / **I** **U** **5** **”** **↗** **≡** **</>** **🖼** **+** **👤** *****

Предпросмотр

Отправить

☐ Markdown (?)

САМОЕ ЧИТАЕМОЕ

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
------	-------	----	----	-----

Две отвёртки, кожа с эффектом софт-тач и 10-нм процессор: обзор ноутбука на базе Intel Evo

Мегапост

Ваш аккаунт	Разделы	Информация	Услуги
Профиль	Публикации	Устройство сайта	Реклама
Трекер	Новости	Для авторов	Тарифы
Диалоги	Хабы	Для компаний	Контент
Настройки	Компании	Документы	Семинары
ППА	Пользователи	Соглашение	Мегaproекты
	Песочница	Конфиденциальность	Мерч

Если нашли опечатку в посте, выделите ее и нажмите Ctrl+Enter, чтобы сообщить автору.