



Vasilui 9 октября 2013 в 10:09

«Boost.Asio C++ Network Programming». Глава 6: – другие особенности

Программирование, C++, API

Tutorial

Всем привет!

Продолжаю перевод книги John Torjo «Boost.Asio C++ Network Programming».

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 некоторые из не очень известных особенностей Boost.Asio. Объекты `std streams` и `streambuf` иногда немного сложнее в использовании, но, как вы сами убедитесь, у них есть свои преимущества. Наконец, вы увидите довольно позднее добавление в Boost.Asio — `co-routines`, которое позволит вам иметь асинхронный код, но легко читаемый (как буд-то бы он синхронный). Это довольно удивительная особенность.

std потоки и std буферы ввода/вывода

Вы должны быть знакомы с такими объектами как `STL streams` и `STL streambuf` для того, чтобы понимать вещи, написанные в этом разделе.

В Boost.Asio есть два типа буферов для работы с вводом/выводом:

- `boost::asio::buffer()`
- `boost::asio::streambuf`

На протяжении всей книги вы в основном видели примерно следующее:

```
size_t read_complete(boost::system::error_code, size_t bytes){ ... }
```

```
char buff[1024];
read(sock, buffer(buff), read_complete);
write(sock, buffer("echo\n"));
```

Обычно вам будет этого достаточно. Но, если вы хотите большей гибкости, то можете использовать `streambuf`. Вот самое простое и худшее, что вы можете сделать с объектом `streambuf`:

```
streambuf buf;
read(sock, buf);
```

Это чтение будет идти до тех пор, пока не заполнится объект `streambuf`, а так как объект `streambuf` может перераспределить себя, чтобы вместить в себя больше места, то, в основном, чтение будет идти до тех пор, пока соединение не будет закрыто. Вы можете использовать функцию `read_until`, чтобы прочитать до последнего знака:

```
streambuf buf;
read_until(sock, buf, "\n");
```

Здесь чтение будет идти до символа `'\n'`, затем в буфер добавится то, что прочтено и выйдет из функции чтения. Чтобы написать что-то в объект `streambuf` вы будете делать что-то похожее на следующее:

```
streambuf buf;
std::ostream out(&buf);
out << "echo" << std::endl;
write(sock, buf);
```

Это довольно просто, вам надо создать поток STL, поместить туда объект `streambuf` при конструировании, записать в него сообщение, которое вы хотите отправить, а затем использовать функцию `write` для отправки содержимого буфера.

Boost.Asio и потоки STL

С Boost.Asio проделали большую работу по интеграции STL потоков и сетей. А именно, если вы уже широко используете STL, то у вас уже должно быть много классов с перегруженными операторами `>>` и `<<`. Чтение и запись в сокеты понравятся вам больше, чем прогулка по парку.

Скажем, у вас есть следующий фрагмент кода:

```
struct person
{
    std::string first_name, last_name;
    int age;
};
std::ostream& operator<<(std::ostream & out, const person & p)
{
    return out << p.first_name << " " << p.last_name << " " << p.age;
}
std::istream& operator>>(std::istream & in, person & p)
```

```
{
    return in >> p.first_name >> p.last_name >> p.age;
}
```

Отправить данные человека по сети так же просто, как показано ниже:

```
streambuf buf;
std::ostream out(&buf);
person p;
// ... initialize p
out << p << std::endl;
write(sock, buf);
```

Другая сторона может так же просто это прочитать:

```
read_until(sock, buf, "\n");
std::istream in(&buf);
person p;
in >> p;
```

Действительно хорошая сторона использования объектов `streambuf` и, конечно, соответствующих `std::ostream` для записи или `std::istream` для чтения, заключается в том, что в конечном итоге вы напишите код, который будет считаться нормальным:

- При написании чего-то, что будет передаваться по сети, очень вероятно, что вы будете иметь больше одной части данных. Таким образом, в конечном итоге, вы добавите данные в буфер. Если эти данные не являются строкой, то вы должны в первую очередь преобразовать их в строку. Все это происходит по умолчанию при использовании оператора `<<`.
- То же самое происходит и на другой стороне, при чтении сообщения; вам нужно разобрать его, то есть, прочитать одну часть данных за один раз и, если данные не являются строкой, необходимо их преобразовать. Все это происходит по умолчанию, если вы при чтении используете оператор `>>`.

Наконец, известен довольно крутой трюк, чтобы сбросить содержимое объекта `streambuf` в консоли, используйте следующий код:

```
streambuf buf;
...
std::cout << &buf << std::endl; // dumps all content to the console
```

Аналогичным образом, для преобразования его содержимого в строку, используйте следующий фрагмент кода:

```
std::string to_string(streambuf &buf)
{
    std::ostringstream out;
    out << &buf;
    return out.str();
}
```

Класс `streambuf`

Как я уже говорил, `streambuf` произведен от `std::streambuf`. Как и `std::streambuf` у него нет конструктора копирования. Кроме того, у него есть несколько дополнительных функций, таких как:

- `streambuf ([max_size,] [allocator])` : эта функция создает объект **`streambuf`**. При необходимости, вы можете опционально задать максимальный размер буфера и аллокатор, который будет использоваться для выделения/освобождения памяти.
- `prepare(n)` : эта функция возвращает под-буфер, используемый для размещения непрерывной последовательности из `n` символов. Он может быть использован для чтения или записи. Результат работы этой функции может быть использован с любой независимой функцией из Boost.Asio производящей чтение/запись, а не только с теми, которые работают с объектами `streambuf`.
- `data()` : эта функция возвращает весь буфер в виде непрерывной последовательности символов и используется для записи. Результат работы этой функции может быть использован с любой независимой функцией из Boost.Asio, производящей запись, а не только с теми, которые работают с объектами `streambuf`.
- `consume(n)` : в этой функции данные удаляются из входной последовательности (из операции чтения).
- `commit(n)` : в этой функции данные удаляются из выходной последовательности (из операции записи) и добавляются к входной последовательности (в операции чтения).
- `size()` : эта функция возвращает размер в символах всего объекта `streambuf`.
- `max_size()` : эта функция возвращает максимальное количество символов, которое может содержаться в объекте `streambuf`.

За исключением двух последних функций, остальные не так легко понять. Прежде всего, в большинстве случаев, вы будете посылать экземпляр `streambuf` в качестве аргумента для чтения/записи независимой функции, как показано ниже:

```
read_until(sock, buf, "\n"); // reads into buf
write(sock, buf); // writes from buf
```

Если вы посылаете весь буфер независимой функции, как показано в предыдущем фрагменте, то функция сначала убедится надо ли ей будет увеличивать размер буфера, искать входные и выходные указатели. Другими словами, если есть данные для чтения, то вы сможете их прочитать.

Например:

```
read_until(sock, buf, '\n');
std::cout << &buf << std::endl;
```

В предыдущем фрагменте сбросится то, что вы только что прочитали из сокета. Следующий пример не будет дампом чего-то:

```
read(sock, buf.prepare(16), transfer_exactly(16) );
std::cout << &buf << std::endl;
```

Байты считываются, но указатель не перемещается. Вы должны двигать его самостоятельно, как показано ниже:

```
read(sock, buf.prepare(16), transfer_exactly(16) );
buf.commit(16);
std::cout << &buf << std::endl;
```

Аналогично, если вы хотите записать в объект `streambuf` и если вы используете независимую функцию записи, то используйте следующий фрагмент кода:

```
streambuf buf;
std::ostream out(&buf);
out << "hi there" << std::endl;
write(sock, buf);
```

Следующий код пошлет `hi there` три раза:

```
streambuf buf;
std::ostream out(&buf);
out << "hi there" << std::endl;
for ( int i = 0; i < 3; ++i)
    write(sock, buf.data());
```

Это происходит, потому что буфер никогда не уничтожается и данные остаются там. Если вы хотите, чтобы данные уничтожались, то посмотрите как это реализуется:

```
streambuf buf;
std::ostream out(&buf);
out << "hi there" << std::endl;
write(sock, buf.data());
buf.consume(9);
```

В заключении, вы должны предпочесть иметь дело с целым экземпляром `streambuf`. Используйте предыдущие функции, если хотите тонкой настройки.

Даже если вы можете использовать один и тот же экземпляр `streambuf` для чтения и записи, то я все равно рекомендую вам два отдельных экземпляра, один для чтения, другой для записи. Это воспринимается проще и яснее, и вы избежите многих возможных ошибок.

Независимые функции, работающие с объектами `streambuf`

В следующем списке показаны независимые функции из Boost.Asio, которые работают с объектами `streambuf`:

- `read (sock, buf [, completion_function])`: эта функция читает из сокета в объект `streambuf`. Завершающая функция является не обязательной. Если же она есть, то она вызывается после каждой успешной операции чтения и сообщает Boost.Asio, если операция завершена (если нет, то продолжает читать). Ее сигнатура выглядит следующим образом: `size_t completion(const boost::system::error_code & err, size_t bytes_transferred)`. При завершении функция возвращает 0, имеется в виду, если операция чтения завершилась полностью; если она возвращает ненулевое значение, то это означает, что вернулось максимальное количество байт для следующего вызова потоковой функции `read_some`.
- `read_at(radom_stream, offset, buf [, completion_function])`: эта функция читает из случайного потока.

Обратите внимание, что это не относится к сокетам (так как они не моделируют концепцию случайного потока).

- `read_until(sock, buf, char | string | regex | match_condition)` : эта функция читает пока выполняется данное условие. Либо должен быть прочитан определенный символ, либо какая-либо строка или регулярное выражение совпадет с одной из прочитанных строк, либо функция `match_condition` скажет нам, что надо выйти из функции. Сигнатура функции `match_condition` следующая: `match_condition is pair<iterator, bool> match(iterator begin, iterator end)` ; где главный итератор это `buffers_iterator <streambuf::const_buffers_type>` . Если совпадение найдено, то вернется пара (`passed-end-of-match` установится в `true`), если же совпадений не выявлено, то вернется другая пара (`begin` установится в `false`).
- `write(sock, buf [, completion_function])` : эта функция записывает все содержимое в объект `streambuf` . Завершающая функция является необязательной и ее поведение похоже на завершающую функцию `read()` : возвращается 0, когда операция записи завершена или ненулевое значение, когда указывается количество байт, которое будет записано при следующем вызове потоковой функции `write_some` .
- `write_at(random_stream, offset, buf [, completion_function])` : эта функция записывает в случайный поток. Опять же не относится к сокетам.
- `async_read(sock, buf [, completion_function], handler)` : эта асинхронный двойник функции `read()` . Сигнатура обработчика следующая: `void handler(const boost::system::error_code, size_t bytes)` .
- `async_read_at(radom_stream, offset, buf [, completion_function] , handler)` : это асинхронный двойник функции `read_at()` .
- `async_read_until (sock, buf, char | string | regex | match_condition, handler)` : это асинхронный двойник функции `read_until()` .
- `async_write(sock, buf [, completion_function] , handler)` : это асинхронный двойник функции `write()` .
- `async_write_at(random_stream, offset, buf [, completion_function], handler)` : это асинхронный двойник функции `write_at()` .

Допустим, вы хотите читать до гласной буквы:

```
streambuf buf;
bool is_vowel(char c)
{
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}
size_t read_complete(boost::system::error_code, size_t bytes)
{
    const char * begin = buffer_cast<const char*>( buf.data());
    if ( bytes == 0)
        return 1;
    while ( bytes > 0)
    {
        if ( is_vowel(*begin++))
            return 0;
        else
            --bytes;
    }
    return 1;
}
```

```
...
read(sock, buf, read_complete);
```

Если вы, например, хотите использовать регулярные выражения, то это очень просто:

```
read_until(sock, buf, boost::regex("[aeiou]+") );
```

Или позвольте немного модифицировать пример, и вы сможете размещать функцию `match_condition` для работы:

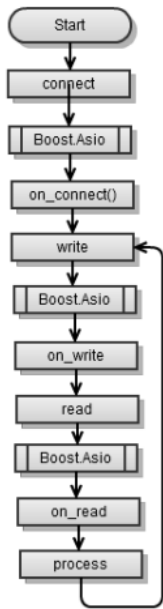
```
streambuf buf;
bool is_vowel(char c)
{
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}
typedef buffers_iterator<streambuf::const_buffers_type> iterator;
std::pair<iterator, bool> match_vowel(iterator b, iterator e)
{
    while ( b != e)
    {
        if ( is_vowel(*b++))
            return std::make_pair(b, true);
    }
    return std::make_pair(e, false);
}
...
size_t bytes = read_until(sock, buf, match_vowel);
```

Сопрограммы

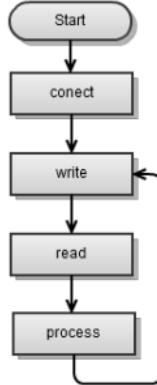
Авторы Boost.Asio, около 2009-2010 годов, реализовали очень классную идею сопрограмм, которые помогут вам создавать асинхронные приложения еще проще.

Они позволяют вам облегчить две вещи, то есть легко написать асинхронное приложение и так же легко следить за потоком управления, почти как если бы приложение было написано последовательно.

Case 1: The normal asynchronous flow



Case 2: simple asynchronous flow (sequential)



В первом случае отображается обычный подход. Используя сопрограммы, вы максимально приблизитесь ко второму случаю. Проще говоря, сопрограмма позволяет использовать множественные точки входа для приостановки и возобновления выполнения в определенных местах в пределах функции.

Если вы собираетесь использовать сопрограммы, то вам надо будет подключить два заголовочных файла, которые вы можете найти только в `boost/libs/asio/example/http/server4`: `yield.hpp` и `coroutine.hpp`. Здесь в Boost.Asio определены два макроса и класс:

- `coroutine` : этот класс есть производная от вашего или используемый вами `connection` класс в целях реализации сопрограмм.
- `reenter(entry)` : это тело сопрограммы. Входящий аргумент это указатель на подпрограмму, например, для использования в качестве блока внутри целой функции.
- `yield code`: выполняет инструкции как часть сопрограммы.

Чтобы лучше разобраться, рассмотрим несколько примеров. Мы будем повторно реализовывать приложение из 4 главы, которое представляет собой простой клиент, который входит в систему, пингуется и может сказать вам, какие другие клиенты занесены в журнал.

Основной код похож на:

```

class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr>
    , public coroutine, boost::noncopyable
{
    ...
    void step(const error_code & err = error_code(), size_t bytes = 0)
    {
        reenter(this)
        {
            for (;;)
            {
                yield async_write(sock_, write_buffer_,
  
```



```

        MEM_FN2(step, _1, _2) );
        yield async_read_until( sock_, read_buffer_, "\n", MEM_
            FN2(step, _1, _2));
        yield service.post( MEM_FN(on_answer_from_server));
    }
}
};

```

Первое, что изменилось — это пропало большое число функций-членов, таких как `connect()`, `on_connect()`, `on_read()`, `do_read()`, `on_write()`, `do_write()` и так далее, теперь у нас есть одна вызываемая функция `step()`. Тело функции находится внутри `reenter(this) { for (;;) { }}`. Вы можете думать о `reenter(this)` как о коде, который мы выполняли в последний раз, так что мы можем сейчас вызвать следующий код.

Внутри блока `reenter` вы можете увидеть несколько текущих вызовов. Первый раз при входе функцию запускается на выполнение функция `async_write`, при втором входе — функция `async_read_until`, при третьем — функция `service.post`, при четвертом — опять `async_write` и так далее.

Вы никогда не должны забывать про экземпляр `for(;;) {}`. Посмотрим на следующий код:

```

void step(const error_code & err = error_code(), size_t bytes = 0)
{
    reenter(this)
    {
        yield async_write(sock_, write_buffer_, MEM_FN2(step, _1, _2) );
        yield async_read_until( sock_, read_buffer_, "\n", MEM_FN2(step, _1, _2));
        yield service.post( MEM_FN(on_answer_from_server));
    }
}

```

Если бы мы использовали предыдущий фрагмент кода в третий раз, мы бы вошли в функцию и выполнили `service.post`. В четвертый раз мы бы прошли мимо `service.post` и ничего не выполнили. То же самое произойдет и на пятый раз и на все последующие:

```

class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr>
, public coroutine, boost::noncopyable
{
    talk_to_svr(const std::string & username) : ... {}
    void start(ip::tcp::endpoint ep)
    {
        sock_.async_connect(ep, MEM_FN2(step, _1, 0) );
    }
    static ptr start(ip::tcp::endpoint ep, const std::string & username)
    {
        ptr new_(new talk_to_svr(username));
        new_->start(ep);
        return new_;
    }
    void step(const error_code & err = error_code(), size_t bytes = 0)
    {
        reenter(this)

```

```

        {
            for (;;)
            {
                if ( !started_)
                {
                    started_ = true;
                    std::ostream out(&write_buf_);
                    out << "login " << username_ << "\n";
                }
                yield async_write(sock_, write_buf_, MEM_FN2(step,_1,_2) );
                yield async_read_until( sock_, read_buf_, "\n", MEM_FN2(step,_1,_
2)));

                yield service.post( MEM_FN(on_answer_from_server));
            }
        }
    }

    void on_answer_from_server()
    {
        std::istream in(&read_buf_);
        std::string word;
        in >> word;
        if ( word == "login")
            on_login();
        else if ( word == "ping")
            on_ping();
        else if ( word == "clients")
            on_clients();
        read_buf_.consume( read_buf_.size());
        if (write_buf_.size() > 0)
            service.post( MEM_FN2(step,error_code(),0));
    }
    ...
private:
    ip::tcp::socket sock_;
    streambuf read_buf_, write_buf_;
    bool started_;
    std::string username_;
    deadline_timer timer_;
};

```

Когда мы начинаем подключение, вызывается функция `start()`, которая асинхронно подключается к серверу. Когда соединение установлено, мы входим в `step()` в первый раз. Это когда мы отправляем сообщение с нашим логином. После этого мы используем `async_write`, затем `async_read_until` и обрабатываем сообщение (`on_answer_from_server`).

В функции `on_answer_from_server` мы обрабатываем входящие сообщения; мы читаем первое слово и направляем в соответствующую функцию, а остальную часть сообщения мы игнорируем (в любом случае):

```

class talk_to_svr : ...
{
    ...
    void on_login()

```

```

{
    do_ask_clients();
}
void on_ping()
{
    std::istream in(&read_buf_);
    std::string answer; in >> answer;
    if ( answer == "client_list_changed")
        do_ask_clients();
    else
        postpone_ping();
}
void on_clients()
{
    std::ostringstream clients;
    clients << &read_buf_;
    std::cout << username_ << ", new client list:" << clients.
    str();
    postpone_ping();
}
void do_ping()
{
    std::ostream out(&write_buf_);
    out << "ping\n";
    service.post( MEM_FN2(step,error_code(),0));
}
void postpone_ping()
{
    timer_.expires_from_now(boost::posix_time::millisec(rand() % 7000));
    timer_.async_wait( MEM_FN(do_ping));
}
void do_ask_clients()
{
    std::ostream out(&write_buf_);
    out << "ask_clients\n";
}
}
};

```

Пример немного более сложный, так как мы должны проверять связь с сервером в случайный момент времени. Чтобы сделать это, мы откладываем операцию пинговки после того, как успешно запросили список клиентов в первый раз. Затем на каждый ответный пинг от сервера мы откладываем другую операцию пинговки. Чтобы запустить все это, используйте следующий фрагмент кода:

```

int main(int argc, char* argv[])
{
    +22      115      22,3k      3      Поделиться
    ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
    talk to our server (on "127.0.0.1");
}

```



68,5

Карма

0,0

Рейтинг



Подписаться

Кузьминых Василий @Vasilui

Пользователь

ПОХОЖИЕ ПУБЛИКАЦИИ

22 августа 2019 в 18:19

Пишем API на Python (с Flask и RapidAPI)

9 46,3k 232 6 +6

20 июня 2015 в 15:17

Как мы сделали из JSON язык программирования

5 20,6k 90 31 +31

11 июля 2014 в 04:16

О бедном C++ API замолвите слово!

20 35,9k 174 71 +71

ВАКАНСИИ

Rust Developer

от 3 500 до 4 500 € • VelvetFormula • Можно удаленно

Senior Backend Engineer (Rest API, English, NodeJS, Python)

от 3 500 до 4 500 \$ • DataDirect Networks Inc. (DDN) • Можно удаленно

Программист C/C++ (протоколы мобильной связи 4G/5G)

до 150 000 Р • ПроВайд Лабс • Новосибирск • Можно удаленно

Программист встроенных систем (C/C++, Junior/Middle)

от 50 000 Р • Fort Telecom • Пермь

Программист C/C++ (протоколы мобильной связи 4G/5G)

до 150 000 Р • ПроВайд Лабс • Новосибирск • Можно удаленно

[Больше вакансий на Хабр Карьере](#)

Комментарии 3

Отслеживать новые в ☐ почте ☐ трекере



v1ctor

9 октября 2013 в 11:10

+1

Огромное спасибо! С нетерпением ждем новых статей :)

[Ответить](#)



DZhon

9 октября 2013 в 22:19

+1

И снова восхитительно, я в предвкушении!

[Ответить](#)

klirichek

17 августа 2019 в 13:20



0

Отправить данные человека по сети так же просто, как показано ниже:

Это точно вы переводите? Очень похоже на чуть-чуть поправленный машинный перевод.
(но всё равно спасибо!)

[Ответить](#)

Написать комментарий

B / u         *

Предпросмотр

Отправить

☐ Markdown (?)

САМОЕ ЧИТАЕМОЕ

Сутки

Неделя

Месяц

Собеседование в Яндекс: театр абсурда :/

+564 182k 449 1103 +1076

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

+42 23k 45 80 +54

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

+57 25k 40 257 +257

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

+101 14,6k 55 92 +92

И солнце светит ярче, и веселей пейзаж, когда читаешь про Azure, .NET и партнёрские программы с Microsoft

Подборка

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

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

