Vasilui 30 сентября 2013 в 15:07

«Boost.Asio C++ Network Programming». Глава 4: Клиент и Сервер

C++, API

Tutorial

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

Содержание:

В этой главе мы собираемся углубиться в создание нетривиальных клиент/серверных приложений с использованием Boost.Asio. Вы можете запускать и тестировать их, и как только вы разберетесь в них, вы сможете использовать их как основу для создания собственных приложений.

В следующих приложениях:

- Клиент заходит на сервер с именем пользователя (без пароля)
- Все соединения инициируются клиентом, где клиент запрашивает ответ от сервера
- Все запросы и ответы на них заканчиваются символом '\n'
- Сервер отключает любого клиента, который не пингуется в течение 5 секунд

Клиент может делать следующие запросы:

- Получить список всех подключенных клиентов
- Клиент может пинговаться, и когда он припингуется сервер ответить либо `ping_ok` либо `ping client_list_chaned` (в последнем случае клиент повторно запрашивает список подключенных клиентов).

Для интереса добавим несколько выкрутасов:

- В каждое клиентское приложение входит 6 подключаемых пользователей, таких как Джон, Джеймс, Люси, Трейси Франк и

Эбби.

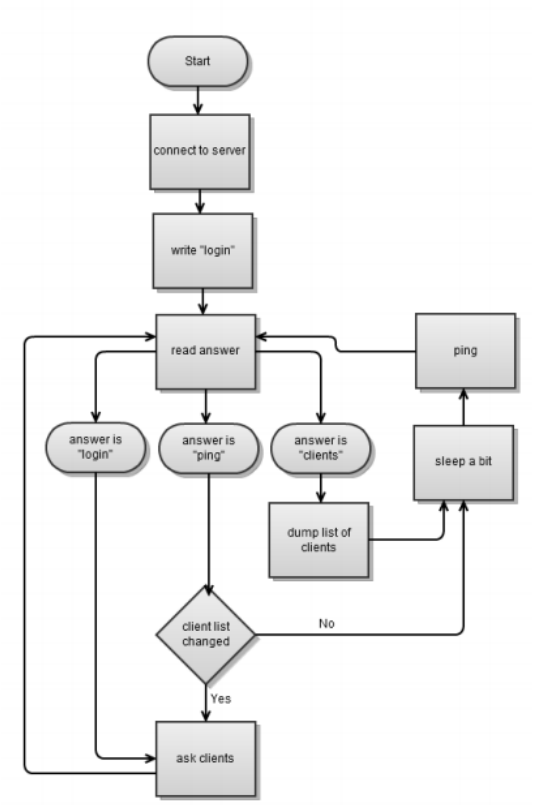
- Каждый клиент проверяет связь с сервером в случайный момент времени (раз в 1-7 секунд, таким образом, время от времени соединение с сервером будет разрываться)

Синхронные сервер/клиент

Во-первых, мы реализуем синхронное приложение. Вы увидите, что код является простым и понятным для усваивания. Тем не менее, сетевая часть должна выполняться в отдельном потоке, так как все сетевые вызовы блокируются.

Синхронный клиент

Синхронный клиент, как вы и ожидали, делает все последовательно; подключается к серверу, заходит на него, а затем выполняет цикл связи, а именно заснуть, сделать запрос, прочитать ответ сервера, опять заснуть и так далее.



Так как мы делаем синхронный вариант, то это позволяет делать некоторые вещи более простыми. Во-первых, подключение к серверу; сделаем это в виде цикла, например, так:

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
void run_client(const std::string & client_name)
{
        talk_to_svr client(client_name);
        try
        {
                client.connect(ep);
```

```
                client.loop();
        }
        catch(boost::system::system_error & err)
        {
                std::cout << "client terminated " << std::endl;
        }
}
```

Следующий пример это класс `talk_to_svr` :

```
struct talk_to_svr
{
        talk_to_svr(const std::string & username): sock_(service), started_(true), username_(use
rname) {}
        void connect(ip::tcp::endpoint ep)
        {
                sock_.connect(ep);
        }
        void loop()
        {
                write("login " + username_ + "\n");
                read_answer();
                while ( started_)
                {
                        write_request();
                        read_answer();
                        boost::this_thread::sleep(millisec(rand() % 7000));
                }
        }
        std::string username() const { return username_; }
        ...
private:
        ip::tcp::socket sock_;
        enum { max_msg = 1024 };
        int already_read_;
        char buff_[max_msg];
        bool started_;
        std::string username_;
};
```

В цикле мы просто пингуемся, читаем ответ от сервера и засыпаем. Засыпаем мы на неопределенное время (иногда более 5 секунд), так что в определенный момент сервер будет нас отключать:

```
void write_request()
{
        write("ping\n");
}
void read_answer()
{
        already_read_ = 0;
```

```
        read(sock_, buffer(buff_),
        boost::bind(&talk_to_svr::read_complete, this, _1, _2));
        process_msg();
}
void process_msg()
{
        std::string msg(buff_, already_read_);
        if ( msg.find("login ") == 0) on_login();
        else if ( msg.find("ping") == 0) on_ping(msg);
        else if ( msg.find("clients ") == 0) on_clients(msg);
        else std::cerr << "invalid msg " << msg << std::endl;
}
```

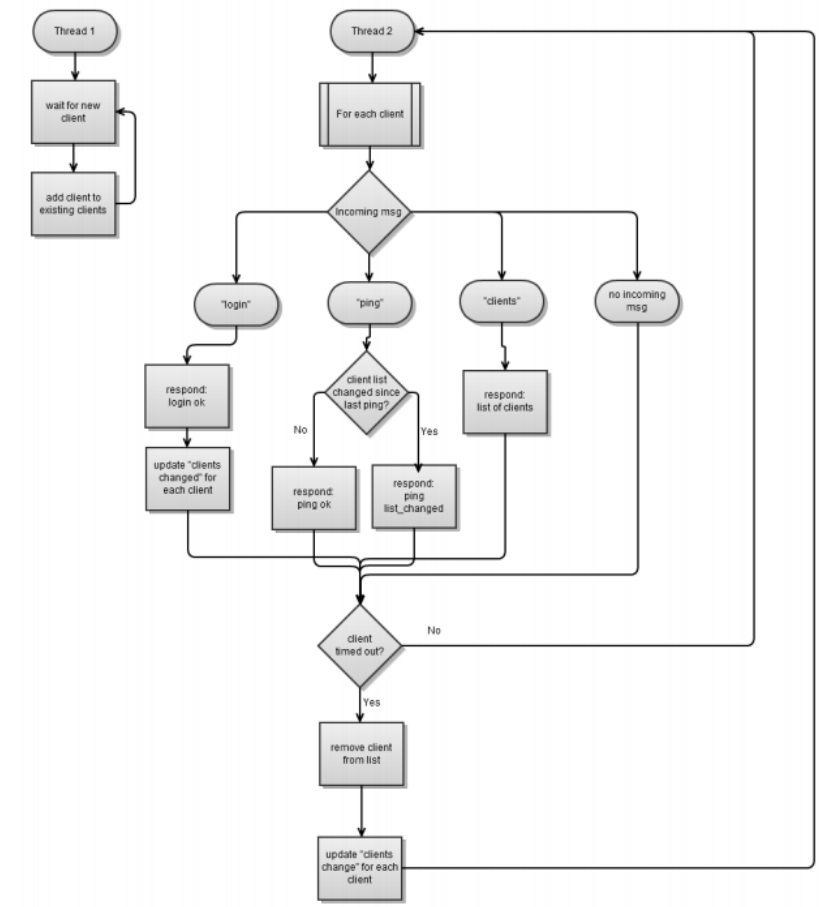
Для чтения ответа мы используем `read_complete` (о которой много говорилось в прошлой главе), чтобы убедиться, что мы дочитали до символа '\n'. Логика заключена в функции `process_msg()`, где мы читаем ответ клиента и направляем в правильную функцию:

```
void on_login() { do_ask_clients(); }
void on_ping(const std::string & msg)
{
        std::istringstream in(msg);
        std::string answer;
        in >> answer >> answer;
        if ( answer == "client_list_changed")
        do_ask_clients();
}
void on_clients(const std::string & msg)
{
        std::string clients = msg.substr(8);
        std::cout << username_ << ", new client list:" << clients;
}
void do_ask_clients()
{
        write("ask_clients\n");
        read_answer();
}
void write(const std::string & msg) { sock_.write_some(buffer(msg)); }
size_t read_complete(const boost::system::error_code & err, size_t bytes)
{
        // ... same as before
}
```

При чтении ответа от сервера в нашем пинге, если мы получим `client_list_changed`, то мы снова делаем запрос на получение листа клиентов.

Синхронный сервер

Синхронный сервер так же достаточно прост. Он нуждается в двух потоках, один для прослушивания новых клиентов, другой для обработки существующих. Он не может использовать один поток, ожидание нового клиента является блокирующей операцией, таким образом нам нужен дополнительный поток для обработки существующих клиентов.



Как и ожидалось, сервер писать немного сложнее, чем клиента. С одной стороны он должен управлять всеми подключенными клиентами. Так как мы пишем синхронный вариант сервера, то нам необходимо, по крайней мере, два потока, один из которых принимает новых клиентов (так как `accept()` блокирующая операция), а другой отвечает за уже существующих:

```
void accept_thread()
{
ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::v4(), 8001));
while ( true)
{
        client_ptr new_( new talk_to_client);
        acceptor.accept(new_->sock());
        boost::recursive_mutex::scoped_lock lk(cs);
        clients.push_back(new_);
}
}
void handle_clients_thread()
{
        while ( true)
        {
                boost::this_thread::sleep( millisec(1));
                boost::recursive_mutex::scoped_lock lk(cs);
                for(array::iterator b = clients.begin(),e = clients.end(); b != e; ++b)
                (*b)->answer_to_client();
```

```
                // erase clients that timed out
                clients.erase(std::remove_if(clients.begin(), clients.end(),
                boost::bind(&talk_to_client::timed_out,_1)), clients.end());
        }
}
int main(int argc, char* argv[])
{
        boost::thread_group threads;
        threads.create_thread(accept_thread);
        threads.create_thread(handle_clients_thread);
        threads.join_all();
}
```

Нам нужен список клиентов, чтобы обрабатывать входящие запросы от них.
У каждого экземпляра `talk_to_client` есть сокет. У него нет копирующего конструктора, таким образом, если вы хотите запихнуть его в `std::vector`, то вам понадобиться завести shared pointer на него. Есть два способа сделать это: либо внутри `talk_to_client` завести shared pointer на сокет, а затем сделать массив из экземпляров `talk_to_client` или когда есть экземпляр talk_to_client с сокетом по значению и завести массив shared pointer-ов на talk_to_client. Я выбрал последнее, но вы можете пойти и другим путем:

```
typedef boost::shared_ptr<talk_to_client> client_ptr;
typedef std::vector<client_ptr> array;
array clients;
boost::recursive_mutex cs; // thread-safe access to clients array
```

Основной код `talk_to_client` выглядит следующим образом:

```
struct talk_to_client : boost::enable_shared_from_this<talk_to_client>
{
        talk_to_client() { ... }
        std::string username() const { return username_; }
        void answer_to_client()
        {
        try
        {
                read_request();
                process_request();
        }
        catch ( boost::system::system_error&)
        {
                stop();
        }
        if ( timed_out())
                stop();
        }
        void set_clients_changed() { clients_changed_ = true; }
        ip::tcp::socket & sock() { return sock_; }
        bool timed_out() const
        {
                ptime now = microsec_clock::local_time();
```

```
                long long ms = (now - last_ping).total_milliseconds();
                return ms > 5000 ;
        }
        void stop()
        {
                boost::system::error_code err; sock_.close(err);
        }
        void read_request()
        {
                if ( sock_.available())
                        already_read_ += sock_.read_some(
                buffer(buff_ + already_read_, max_msg - already_read_));
        }
        ...
private:
        // ... same as in Synchronous Client
        bool clients_changed_;
        ptime last_ping;
};
```

Приведенный выше код довольно очевиден. Наиболее важная функция это `read_request()` . Чтение будет происходить только если есть данные, таким образом, сервер никогда не будет заблокирован:

```
void process_request()
{
        bool found_enter = std::find(buff_, buff_ + already_read_, '\n') < buff_ + already_read
_;
        if ( !found_enter)
                return; // message is not full
        // process the msg
        last_ping = microsec_clock::local_time();
        size_t pos = std::find(buff_, buff_ + already_read_, '\n') - buff_;
        std::string msg(buff_, pos);
        std::copy(buff_ + already_read_, buff_ + max_msg, buff_);
        already_read_ -= pos + 1;
        if ( msg.find("login ") == 0) on_login(msg);
        else if ( msg.find("ping") == 0) on_ping();
        else if ( msg.find("ask_clients") == 0) on_clients();
        else std::cerr << "invalid msg " << msg << std::endl;
}
void on_login(const std::string & msg)
{
        std::istringstream in(msg);
        in >> username_ >> username_;
        write("login ok\n");
        update_clients_changed();
}
void on_ping()
{
        write(clients_changed_ ? "ping client_list_changed\n" : "ping ok\n");
        clients_changed_ = false;
```

```
}
void on_clients()
{
        std::string msg;
        {
                boost::recursive_mutex::scoped_lock lk(cs);
                for( array::const_iterator b = clients.begin(), e = clients.end() ;b != e; ++b)
                        msg += (*b)->username() + " ";
        }
        write("clients " + msg + "\n");
}
void write(const std::string & msg) { sock_.write_some(buffer(msg)); }
```

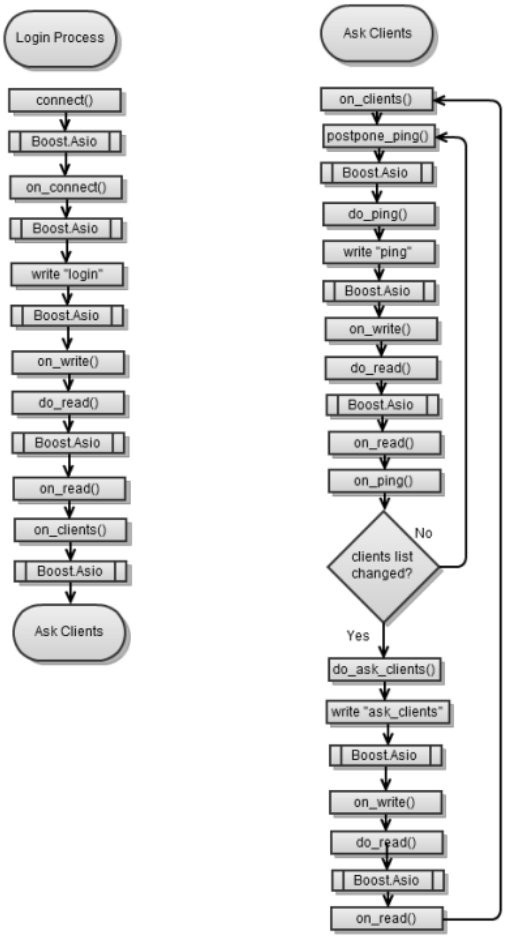
Взгляните на `process_request()` . После того как мы считали те данные, которые были доступны, мы должны проверить считали ли мы сообщение до конца (если да, то `found_enteris` установится в true). Если это так, то мы защищаем себя от чтения, может быть, больше чем одного сообщения (после символа '\n' сохраняться в буфер ничего не будет), а затем мы интерпретируем полностью прочитанное сообщение. Остальная часть кода довольно проста.

Асинхронные сервер/клиент

А теперь самая интересная (и сложная) часть, пойдем асинхронным путем.

Асинхронный клиент

Вещи сейчас будут рассматриваться немного сложнее, но, безусловно, управляемые. И у нас будет приложение, которое не блокируется.



Вам уже должен быть понятен следующий код:

```
#define MEM_FN(x) boost::bind(&self_type::x, shared_from_this())
#define MEM_FN1(x,y) boost::bind(&self_type::x, shared_from_this(),y)
#define MEM_FN2(x,y,z) boost::bind(&self_type::x, shared_from_this(),y,z)
class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr>, boost::noncopyable
{
        typedef talk_to_svr self_type;
        talk_to_svr(const std::string & username) : sock_(service), started_(true), username_(us
ername), timer_(service) {}
        void start(ip::tcp::endpoint ep)
        {
                sock_.async_connect(ep, MEM_FN1(on_connect,_1));
        }
public:
        typedef boost::system::error_code error_code;
        typedef boost::shared_ptr<talk_to_svr> ptr;
        static ptr start(ip::tcp::endpoint ep, const std::string & username)
        {
                ptr new_(new talk_to_svr(username));
                new_->start(ep);
```

```
                return new_;
        }
        void stop()
        {
                if ( !started_) return;
                started_ = false;
                sock_.close();
        }
        bool started() { return started_; }
        ...
private:
        size_t read_complete(const boost::system::error_code & err, size_t bytes)
        {
                if ( err) return 0;
                bool found = std::find(read_buffer_, read_buffer_ + bytes, '\n') < read_buffer_
+ bytes;
                return found ? 0 : 1;
        }
private:
        ip::tcp::socket sock_;
        enum { max_msg = 1024 };
        char read_buffer_[max_msg];
        char write_buffer_[max_msg];
        bool started_;
        std::string username_;
        deadline_timer timer_;
};
```

Вы увидите дополнительную функцию-таймер `deadline_timer` для осуществления пинга сервера; и опять же, мы будем проверять связь с сервером в случайный момент времени.
Ну а теперь посмотрим, как выглядит основная логика класса:

```
void on_connect(const error_code & err)
{
        if ( !err) do_write("login " + username_ + "\n");
        else stop();
}
void on_read(const error_code & err, size_t bytes)
{
        if ( err) stop();
        if ( !started() ) return;
        // process the msg
        std::string msg(read_buffer_, bytes);
        if ( msg.find("login ") == 0) on_login();
        else if ( msg.find("ping") == 0) on_ping(msg);
        else if ( msg.find("clients ") == 0) on_clients(msg);
}
void on_login()
{
        do_ask_clients();
}
```

```
void on_ping(const std::string & msg)
{
        std::istringstream in(msg);
        std::string answer;
        in >> answer >> answer;
        if ( answer == "client_list_changed") do_ask_clients();
        else postpone_ping();
}
void on_clients(const std::string & msg)
{
        std::string clients = msg.substr(8);
        std::cout << username_ << ", new client list:" << clients ;
        postpone_ping();
}
```

В `on_read()` в первых двух строчках кода сделано все очень красиво. В первой строке мы проверяем, есть ли ошибка, если да, то останавливаемся. Во второй строке мы проверяем, остановились ли мы (до этого или только что), если да, то возвращаемся. В противном случае, если все хорошо, мы обрабатываем входящее сообщение.
И наконец, функции `do_*` следующие:

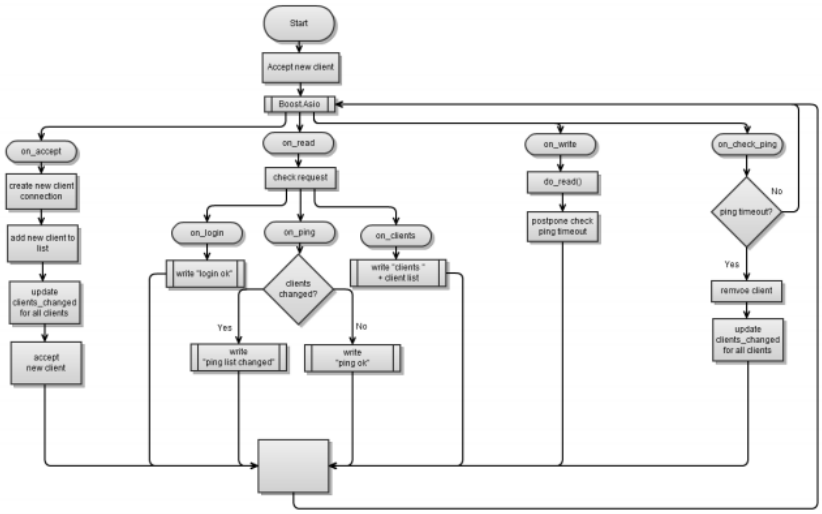
```
void do_ping() { do_write("ping\n"); }
void postpone_ping()
{
        timer_.expires_from_now(boost::posix_time::millisec(rand() % 7000));
        timer_.async_wait( MEM_FN(do_ping));
}
void do_ask_clients() { do_write("ask_clients\n"); }
void on_write(const error_code & err, size_t bytes) { do_read(); }
void do_read()
{
        async_read(sock_, buffer(read_buffer_), MEM_FN2(read_complete,_1,_2), MEM_FN2(on_read,_
1,_2));
}
void do_write(const std::string & msg)
{
        if ( !started() ) return;
        std::copy(msg.begin(), msg.end(), write_buffer_);
        sock_.async_write_some( buffer(write_buffer_, msg.size()), MEM_FN2(on_write,_1,_2));
}
```

Обратите внимание, что каждая операция `read` вызывает пинг:

- Когда операция `read` завершится, вызовется `on_read()`
- `on_read()` перенаправляется в `on_login()`, `on_ping()`, или `on_clients()`
- Каждая из функций либо откладывает пинг, либо запрашивает клиентов
- Если мы запросим клиентов, когда их получила операция read, она отложит пинг

Асинхронный сервер

Схема довольно сложна, вы видите, что от Boost.Asio отходит четыре стрелки к `on_accept, on_read, on_write` и `on_check_ping` . В основном это означает, что вы никогда не узнаете вызовом какой из этих асинхронных операций все закончится, но вы точно знаете, что это будет одна из них.



Итак, мы работаем асинхронно, поэтому можем работать в одном потоке. Прием клиентов это самая легкая часть, как показано в следующем фрагменте кода:

```
ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::v4(), 8001));
void handle_accept(talk_to_client::ptr client, const error_code & err)
{
        client->start();
        talk_to_client::ptr new_client = talk_to_client::new_();
        acceptor.async_accept(new_client->sock(),
        boost::bind(handle_accept,new_client,_1));
}
int main(int argc, char* argv[])
{
        talk_to_client::ptr client = talk_to_client::new_();
        acceptor.async_accept(client->sock(), boost::bind(handle_accept,client,_1));
        service.run();
}
```

Приведенный выше код будет всегда асинхронно ждать новых клиентов (каждое новое подключение клиента будет вызывать другое асинхронное ожидание).
Мы должны следить за событием `client list changed` (подключился новый клиент или один из клиентов получил список и отключился) и уведомить остальных клиентов, когда это произойдет. Таким образом, мы должны хранить массив клиентов, в противном случае не было бы никакой необходимости в этом массиве, если вы не хотели бы знать всех подключенных клиентов в данный момент времени:

```
class talk_to_client; typedef boost::shared_ptr<talk_to_client> client_ptr;
typedef std::vector<client_ptr> array;
array clients;
```

Скелет класса `connection` выглядит следующим образом:

```
class talk_to_client : public boost::enable_shared_from_this<talk_to_
client>, boost::noncopyable
{
        talk_to_client() { ... }
public:
        typedef boost::system::error_code error_code;
        typedef boost::shared_ptr<talk_to_client> ptr;
        void start()
        {
                started_ = true;
                clients.push_back( shared_from_this());
                last_ping = boost::posix_time::microsec_clock::local_time();
                do_read(); // first, we wait for client to login
        }
        static ptr new_() { ptr new_(new talk_to_client); return new_; }
        void stop()
        {
                if ( !started_) return;
                started_ = false;
                sock_.close();
                ptr self = shared_from_this();
                array::iterator it = std::find(clients.begin(), clients.end(), self);
                clients.erase(it);
                update_clients_changed();
        }
        bool started() const { return started_; }
        ip::tcp::socket & sock() { return sock_;}
        std::string username() const { return username_; }
        void set_clients_changed() { clients_changed_ = true; }
        ...
private:
        ip::tcp::socket sock_;
        enum { max_msg = 1024 };
        char read_buffer_[max_msg];
        char write_buffer_[max_msg];
        bool started_;
        std::string username_;
        deadline_timer timer_;
        boost::posix_time::ptime last_ping;
        bool clients_changed_;
};
```

Я вызываю `talk_to_client` или `talk_to_server` из класса `connection`, чтобы сделать более ясным то, что я говорю. Мы должны будем использовать предыдущий код сейчас; он подобен тому, что мы использовали для клиентского приложения. У нас есть дополнительная функция `stop()`, которая удаляет подключенного клиента из массива клиентов.
Сервер непрерывно ожидает асинхронных операций чтения:

```
void on_read(const error_code & err, size_t bytes)
{
```

```
        if ( err) stop();
        if ( !started() ) return;
        std::string msg(read_buffer_, bytes);
        if ( msg.find("login ") == 0) on_login(msg);
        else if ( msg.find("ping") == 0) on_ping();
        else if ( msg.find("ask_clients") == 0) on_clients();
}
void on_login(const std::string & msg)
{
        std::istringstream in(msg);
        in >> username_ >> username_;
        do_write("login ok\n");
        update_clients_changed();
}
void on_ping()
{
        do_write(clients_changed_ ? "ping client_list_changed\n" : "ping ok\n");
        clients_changed_ = false;
}
void on_clients()
{
        std::string msg;
        for(array::const_iterator b =clients.begin(),e =clients.end(); b != e; ++b)
        msg += (*b)->username() + " ";
        do_write("clients " + msg + "\n");
}
```

Код довольно прост; одна вещь состоит в том, что, когда новый клиент входит в систему, мы вызываем `update_clients_changed()`, которая устанавливает `clients_changed_` в `true` для всех клиентов. Как только он получает запрос, он сразу же отвечает на него, как показано в следующем фрагменте кода:

```
void do_ping() { do_write("ping\n"); }
void do_ask_clients() { do_write("ask_clients\n"); }
void on_write(const error_code & err, size_t bytes) { do_read(); }
void do_read()
{
        async_read(sock_, buffer(read_buffer_), MEM_FN2(read_complete,_1,_2), MEM_FN2(on_read,_
1,_2));
        post_check_ping();
}
void do_write(const std::string & msg)
{
        if ( !started() ) return;
        std::copy(msg.begin(), msg.end(), write_buffer_);
        sock_.async_write_some( buffer(write_buffer_, msg.size()), MEM_FN2(on_write,_1,_2));
}
size_t read_complete(const boost::system::error_code & err, size_t bytes)
{
        // ... as before
}
```

68,5
Карма

0,0
Рейтинг

Подписаться

Кузьминых Василий @Vasilui
Пользователь

ПОХОЖИЕ ПУБЛИКАЦИИ

8 апреля 2019 в 12:03
Operating Systems: Three Easy Pieces. Part 3: Process API (перевод)
5 1,7k 16 1 +1

28 января 2014 в 12:17
User Timing API
18 12,9k 111 11 +11

16 мая 2013 в 10:56
Новые функции Google для разработчиков: игровые API, перевод и тестирование
31 13,5k 72 15 +15

Комментарии 6

Отслеживать новые в ☐ почте ☐ трекере

begezavr 30 сентября 2013 в 16:49 +4

Странно, что такая отличная серия переводов так мало плюсуется. На хабре, по-моему, вообще ещё не было ни одного доведённого до конца перевода целой книги. А тут по всему видно, что дело будет до конца доведено, круто.

Ответить

nekipelov 30 сентября 2013 в 18:56 **–1**

Рука не поднимается плюс ставить по причинам:

1. Макросы MEM_FN. Перевод для начинающих, ну и не надо начинающих программистов приучать к макросам. Это же C++.
2. Форматирование исходных текстов ужасно. Я понимаю, что это нудно и неудобно… но код должен быть оформлен красиво.
3. Ну и содержимое на столько простое и очевидное, что лично я не вижу смысла в этом переводе. Документация у boost не на столько хороша, как у Qt, но все же имеется.

Разумеется минус я тоже не стал ставить.

Ответить

ElleSolomina 1 октября 2013 в 22:22 **0**

1) расскажите пожалуйста подробнее, что плохого в таких макросах:

```
#define MEM_FN(x) boost::bind(&self_type::x, shared_from_this())
#define MEM_FN1(x,y) boost::bind(&self_type::x, shared_from_this(),y)
#define MEM_FN2(x,y,z) boost::bind(&self_type::x, shared_from_this(),y,z)
```

Это же простые однострочники, при том безопасные, и даже контроль типов нигде не пропущен из-за простоты. Для реализации этого «по сиплюсплюсному» потребовался бы шаблон, который был бы гораздо толще и лишь помешал бы восприятию кода.
2) это тема вечного холивара, по мне так нормальное вполне, что конкретно Вам не понравилось? И да, прошу осознать, что это всё таки перевод.
3) no comment.

Ответить

nekipelov 2 октября 2013 в 15:42 **–2**

1. Плох сам факт использования макросов. Это как если бы вместо for/while использовать метку и goto, вроде такого:

```
    int i = 0;
label:
    // code
    if (++i < 100)
        goto label;
```

Казалось бы, ну что тут плохого? Код на C++, компилируется и работает без ошибок. Значит код правильный и использовать можно. Примерно такими же соображениями руководствовался программист, сделавший в где-то в WinAPI макрос с именем **max** и мне уже много лет приходится писать #ifdef max\n#undef max\n#endif. Чем именно плохи макросы, по моему, написано в любой книге по программированию на C++ и также дана рекомендация не использовать их. Но давайте я не буду учить вас программированию.

2. Тема для холивара — это какой стиль лучше. Тут же, откровенно говоря, стиля нет вообще. Код написан небрежно.

Вот например:

```
void on_read(const error_code & err, size_t bytes)
{
    if ( err) stop();
    if ( !started() ) return;
    std::string msg(read_buffer_, bytes);
```

```
    if ( msg.find("login ") == 0) on_login(msg);
    else if ( msg.find("ping") == 0) on_ping();
    else if ( msg.find("ask_clients") == 0) on_clients();
}
```

и тут уже ниже

```
void on_clients()
{
    std::string msg;
    for(array::const_iterator b =clients.begin(),e =clients.end(); b != e; ++b)
    msg += (*b)->username() + " ";
    do_write("clients " + msg + "\n");
}
```

Глядя на первый кусок уже и не кажется, что строка `msg += (*b)->username() + " ";` является телом цикла. Ведь даже отступа нет.

```
void read_answer()
{
    already_read_ = 0;
    read(sock_, buffer(buff_),
    boost::bind(&talk_to_svr::read_complete, this, _1, _2));
    process_msg();
}
```

Тут вообще аргумент функции на новой строке.

```
typedef std::vector<client_ptr> array;
array clients;
```

Вот зачем специально запутывать людей, создавая тип `array`, который на самом деле `vector`? Это ведь разные типы данных, один динамический массив, другой использует память на стеке. Кстати говоря, члены класса talk_to_client:

```
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
```

должны быть объявлены вот так:

```
    boost::array<char, max_msg> read_buffer_;
    boost::array<char, max_msg> write_buffer_;
```

Ну и все остальное в том же духе. Конечно кому-то кажется, что человек может писать так, как ему хочется. Использовать макросы и goto, привидение типов в стиле C, free и delete вместо RAII… Надо проработать программистом не один год и в разных командах, чтобы осознать важность этих правил.

И да, я вижу, что это перевод. И даже потрудился заглянуть в оригинал и выяснить, что макросы и такой вот нехороший код идут от туда, а не от переводчика. Но вопрос был совсем не в том, хорош перевод или нет. Я отвечал на комментарий объяснив причину, по которой я стал ставить плюс.

Ответить

 **ElleSolomina** 3 октября 2013 в 05:07 **+1**

1. На макросы будет много ругани при компиляции если макрос с таким именем уже присутствует.

> Примерно такими же соображениями руководствовался программист, сделавший в где-то в WinAPI макрос с именем max и мне уже много лет приходится писать #ifdef max\n#undef max\n#endif.

Вообще то WinAPI, как и огромное количество другого системного кода, это C, а не C++, а стало быть наличие там данных макросов — нормальная ситуация. И да, в ОС используются гораздо более серьёзные макросы, в том числе и глобальные. К слову, в C++ до сих пор не реализован даже if этапа компиляции, и это огромная проблема, зато реализованы монструозные конструкции sfinae на которых и предлагается что то делать по «сиплюсплюсному» для реализации парадигмы «if этапа компиляции».
2.
void on_read(...) — нормальный код, хотя если бы его писал я, то написал бы со скобками под каждым условием,
а вообще

как то так:

Пусть код со скобками для меня привычнее, но это нисколько не мешает воспринять условия в исходном примере кода с однострочными ифами — так очень многие пишут.

С void on_clients() скорее согласен, отступ там желателен, но почему бы не сообщить о таком допущении в личку?.. До кучи я бы, к примеру, использовал там сокращённую запись из C++11: for(const auto &b: clients), однако даже этот факт не мешает нормальному восприятию кода, к тому же это перевод, и судя по другим участкам кода — в оригинале код C++03.

с read_answer() проблем не наблюдаю, перенос длинных строк с параметрами — общепринятая практика, пусть там и не хватает отступа, и возможно этот перенос вообще не нужен, но это не мешает, и это ещё один недочёт сродни опечатки, о котором стоит сообщить в личку. Можно было даже в личку отправить предложение об использовании Astyle, но смысла обсуждать сие нет совсем никакого.

Ситуация с

```
typedef std::vector<client_ptr> array;
```

это ещё один привет из C++03.

> Кстати говоря, члены класса talk_to_client:
>
> ```
>     char read_buffer_[max_msg];
>     char write_buffer_[max_msg];
> ```
>
> должны быть объявлены вот так:
>
> ```
>     boost::array<char, max_msg> read_buffer_;
>     boost::array<char, max_msg> write_buffer_;
> ```

У меня к Вам претензия: на дворе 2013 год, а Вы почему то предлагаете boost::array а не std::array на замену! :) Ладно шучу, просто это действительно не то из-за чего могут быть проблемы с восприятием примера кода! Это пример использования, и

никто не заставляет данный код копипастить и сразу же использовать в промышленном применении, это просто пример :) К слову у Александреску, в примерах, вообще местами адский-ад.

> Ну и все остальное в том же духе. Конечно кому-то кажется, что человек может писать так, как ему хочется. Использовать макросы и goto, привидение типов в стиле C, free и delete вместо RAII… Надо проработать программистом не один год и в разных командах, чтобы осознать важность этих правил.

Пожалуй соглашусь только с привидением типов в стиле C при использовании в C++, а вот всё остальное: макросы, goto, и сырые указатели использовать можно если это действительно необходимо. К слову C++ из-за отсутствия make_unique даже в C++11 не полностью гарантирует безопасность использования умных указателей, а C++03 толком вообще этих возможностей не имеет (умолчим тут про auto_ptr, с которым были проблемы, о которых надо было знать, и неоднозначность с которой вообще ничего поделать было нельзя, и который ныне, к счастью — deprecated). Безусловно в Boost умные указатели появились раньше C++11, но факт остаётся фактом.

p.s: руководствуясь написанным у Вас в пункте 3 вообще не понимаю зачем было писать что либо в данном топике, если данная тема Вам не интересна.

Ответить

nekipelov 3 октября 2013 в 17:23 **0**

> Вообще то WinAPI, как и огромное количество другого системного кода, это C, а не C++, а стало быть наличие там данных макросов — нормальная ситуация.

Вот в том то и дело, что там код на C, поэтому макросы, хоть и вызывают неудобства, но без них не обойтись. А мы говорим о C++, и о коде, который макросов не требует вообще.

> Пусть код со скобками для меня привычнее, но это нисколько не мешает воспринять условия в исходном примере кода с однострочными ифами — так очень многие пишут.

Я конечно опять нарываюсь на минусы, но все же нормальный код не должен так выглядеть. В книгах так часто пишут чтобы сократить объем. Но там и примечания ставят соответствующие. Но в статье на сайте нет смысла экономить строки.

> С void on_clients() скорее согласен, отступ там желателен, но почему бы не сообщить о таком допущении в личку?

Следуя логике большинства — код компилируется? Да, значит он правильный. Так зачем что-то сообщать? Мои претензии к коду — исключительно мое личное дело, я никому не навязываю свой стиль. Тем более, что стилей очень много. Мне вот венгерская нотация не нравится, но не сообщать же теперь об этом в личку каждому автору, посмевшему ее использовать? :-)

> Ситуация с
>
> > typedef std::vector<client_ptr> array;
>
> это ещё один привет из C++03.

Я специально показал следующий пример с array из boost. Asio появился в boost начиная с версии 1.35, открываю первый попавшийся пример и вижу вот такие строки:

```
#include <boost/array.hpp>
```

```
...
boost::array<char, 1024> data_;
```

Пруф: www.boost.org/doc/libs/1_35_0/doc/html/boost_asio/example/allocation/server.cpp

Сомнительно, что автор пишет о boost asio и не знает о контейнере array. А если знает, то зачем вводить потенциально конфликтный тип данных?

> Пожалуй соглашусь только с привидением типов в стиле C при использовании в C++, а вот всё остальное: макросы, goto, и сырые указатели использовать можно если это действительно необходимо.

Я и не говорил, что это нельзя использовать. Разумеется можно, но только там, где это действительно необходимо. Но ведь не сокращать каждый вызов макросом.

> p.s: руководствуясь написанным у Вас в пункте 3 вообще не понимаю зачем было писать что либо в данном топике, если данная тема Вам не интересна.

Тема как раз интересна, иначе бы я не стал открывать этот топик и тем более пролистывать до комментариев.

Ну и предлагаю завершить эту дискуссию :-)

Ответить

Написать комментарий

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

Как перейти в онлайн за 2 месяца, если ты крупный ритейлер и всегда работал по старинке

Мегапост