Vasilui 24 сентября 2013 в 16:04

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

C++, API

Tutorial

Всем привет!
Продолжаю перевод книги John Torjo «Boost.Asio C++ Network Programming». В этой части второй главы мы поговорим про асинхронное программирование.

Содержание:

Этот раздел глубоко разбирает некоторые вопросы, с которыми вы столкнетесь при работе с асинхронным программированием. Прочитав один раз, я предлагаю вам вернуться к нему, по мере прохождения книги, чтобы укрепить ваше понимание этих понятий.

Необходимость работать асинхронно

Как я уже говорил, как правило, синхронное программирование гораздо проще, чем асинхронное. Потому что гораздо легче думать линейно (вызываем функцию A, после ее окончания вызываем ее обработчик, вызываем функцию B, после ее окончания вызываем ее обработчик и так далее, так что можно думать в манере событие-обработчик). В последнем случае вы можете иметь, скажем, пять событий и вы никогда не сможете узнать порядок, в котором они выполняются, и вы даже не будете знать выполнятся ли они все!
Но даже при том, что асинхронное программирование сложнее вы, скорее всего, предпочтете его, скажем, в написании серверов, которые должны иметь дело с большим количеством клиентов одновременно. Чем больше клиентов у вас есть, тем легче асинхронное программирование по сравнению с синхронным.
Скажем, у вас есть приложение, которое одновременно имеет дело с 1000 клиентами, каждое сообщение от клиента серверу и от сервера клиенту заканчивается символом '\n'.
Синхронный код, 1 поток:

```
using namespace boost::asio;
```

```
struct client
{
        ip::tcp::socket sock;
        char buff[1024]; // each msg is at maximum this size
        int already_read; // how much have we already read?
};
std::vector<client> clients;
void handle_clients()
{
        while ( true)
        for ( int i = 0; i < clients.size(); ++i)
                if ( clients[i].sock.available() ) on_read(clients[i]);
}
void on_read(client & c)
{
        int to_read = std::min( 1024 - c.already_read, c.sock.
        available());
        c.sock.read_some( buffer(c.buff + c.already_read, to_read));
        c.already_read += to_read;
        if ( std::find(c.buff, c.buff + c.already_read, '\n') < c.buff + c.already_read)
        {
                int pos = std::find(c.buff, c.buff + c.already_read, '\n') - c.buff;
                std::string msg(c.buff, c.buff + pos);
                std::copy(c.buff + pos, c.buff + 1024, c.buff);
                c.already_read -= pos;
                on_read_msg(c, msg);
        }
}
void on_read_msg(client & c, const std::string & msg)
{
        // analyze message, and write back
        if ( msg == "request_login")
                c.sock.write( "request_ok\n");
        else if ...
}
```

Одна вещь, которую вы хотите избежать при написании серверов (да и в основном любого сетевого приложения) это чтобы код перестал отвечать на запросы. В нашем случае мы хотим, чтобы функция `handle_clients()` блокировалась как можно меньше. Если функция заблокируется в какой-либо точке, то все входящие сообщения от клиента будут ждать, когда функция разблокируется и начнет их обработку.
Для того чтобы оставаться отзывчивым мы будем читать из сокета только тогда, когда в нем есть данные, то есть `if ( clients[i].sock.available() ) on_read(clients[i])` . В on_read мы будем читать только столько, сколько есть в наличии; вызов `read_until(c.sock, buffer(...),'\n')` было бы не очень хорошей идеей, так как она блокируется, пока мы не прочитаем сообщение от конкретного клиента до конца (мы никогда не узнаем когда это произойдет).
Узким местом здесь является функция `on_read_msg()` ; все входящие сообщения будут приостановлены, до тех пор, пока выполняется эта функция. Хорошо-написанная функция `on_read_msg()` будет следить, чтобы этого не произошло, но все же это может произойти (иногда запись в сокет может быть заблокирована, например, если заполнен его буфер).
Синхронный код, 10 потоков:

```
using namespace boost::asio;
struct client
```

```
 {
        // ... same as before
        bool set_reading()
        {
                boost::mutex::scoped_lock lk(cs_);
                if ( is_reading_) return false; // already reading
                else { is_reading_ = true; return true; }
        }
        void unset_reading()
        {
                boost::mutex::scoped_lock lk(cs_);
                is_reading_ = false;
        }
private:
        boost::mutex cs_;
        bool is_reading_;
};
std::vector<client> clients;
void handle_clients()
{
        for ( int i = 0; i < 10; ++i)
        boost::thread( handle_clients_thread);
}
void handle_clients_thread()
{
        while ( true)
        for ( int i = 0; i < clients.size(); ++i)
                if ( clients[i].sock.available() )
                        if ( clients[i].set_reading())
                        {
                                on_read(clients[i]);
                                clients[i].unset_reading();
                        }
}
void on_read(client & c)
 {
        // same as before
}
void on_read_msg(client & c, const std::string & msg)
{
        // same as before
}
```

Для того, чтобы использовать несколько потоков, нам нужно их синхронизировать, что и делают функции `set_reading` () и `set_unreading()` . Функция `set_reading()` является очень важной. Вы хотите, чтобы «проверить можно ли читать и начать читать» выполнялось за один шаг. Если у вас это выполняется за два шага («проверить можно ли читать» и «начать чтение»), то вы можете завести два потока: один для проверки на чтение для какого-либо клиента, другой для вызова функции `on_read` для того же клиента, в конечном итоге это может привести к повреждению данных и возможно даже к зависанию системы.

Вы заметите, что код становится все более сложным.

Возможен и третий вариант для синхронного кода, а именно иметь по одному потоку на каждого клиента. Но так как число одновременных клиентов растет, то это в значительной степени становится непозволительной операцией.

А теперь рассмотрим асинхронные варианты. Мы постоянно делали асинхронной операцию чтения. Когда клиент делает

запрос, вызывается операция `on_read` , мы отвечаем в ответ, а затем ждем, когда поступит следующий запрос (запускаем еще одну операцию асинхронного чтения).
Асинхронный код, 10 потоков:

```
using namespace boost::asio;
io_service service;
struct client
 {
        ip::tcp::socket sock;
        streambuf buff; // reads the answer from the client
}
std::vector<client> clients;
void handle_clients()
{
        for ( int i = 0; i < clients.size(); ++i)
                async_read_until(clients[i].sock, clients[i].buff, '\n', boost::bind(on_read, cl
ients[i], _1, _2));
        for ( int i = 0; i < 10; ++i)
                boost::thread(handle_clients_thread);
}
void handle_clients_thread()
{
        service.run();
}
void on_read(client & c, const error_code & err, size_t read_bytes)
{
        std::istream in(&c.buff);
        std::string msg;
        std::getline(in, msg);
        if ( msg == "request_login")
                c.sock.async_write( "request_ok\n", on_write);
        else if ...
        ...
        // now, wait for the next read from the same client
        async_read_until(c.sock, c.buff, '\n', boost::bind(on_read, c, _1, _2));
}
```

Обратите внимание, насколько проще стал код. Структура `client` имеет только два члена, `handle_clients()` просто вызывает `async_read_until` , а затем создает десять потоков, каждый из которых вызывает `service.run()` . Эти потоки будут обрабатывать все операции асинхронного чтения или записи клиенту. Еще одно нужно отметить, что функция `on_read()` будет постоянно готовиться к следующей операции асинхронного чтения (смотрите последнюю строку).

Асинхронные функции run(), run_one(), poll(), poll_one()

Для реализации цикла прослушивания класс `io_service` предоставляет четыре функции, такие как `run()`, `run_one()`, `poll()` , и `poll_one()` . Хотя большую часть времени вы будете работать с `service.run()` . Здесь вы узнаете чего можно достигнуть с помощью других функций.

Постоянно работающий

Еще раз, `run()` будет работать, пока ожидающие операции завершаются или пока вы сами не вызовете `io_service::stop()` . Чтобы сохранить экземпляр `io_service` работающим, вы, как правило, добавляете одну или

несколько асинхронных операций и когда они заканчиваются, вы продолжаете добавлять, как показано в следующем коде:

```
using namespace boost::asio;
io_service service;
ip::tcp::socket sock(service);
char buff_read[1024], buff_write[1024] = "ok";
void on_read(const boost::system::error_code &err, std::size_t bytes) ;
void on_write(const boost::system::error_code &err, std::size_t bytes)
{
        sock.async_read_some(buffer(buff_read), on_read);
}
void on_read(const boost::system::error_code &err, std::size_t bytes)
{
        // ... process the read ...
        sock.async_write_some(buffer(buff_write,3), on_write);
}
void on_connect(const boost::system::error_code &err)
{
        sock.async_read_some(buffer(buff_read), on_read);
}
int main(int argc, char* argv[])
{
        ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 2001);
        sock.async_connect(ep, on_connect);
        service.run();
}
```

Когда вызывается `service.run()`, то в ожидании находится хотя бы одна асинхронная операция. Когда сокет подключается к серверу, вызывается `on_connect`, которая добавляет еще одну асинхронную операцию. После окончания работы `on_connect` у нас остается одна запланированная операция (`read`). Когда завершается операция `on_read`, пишем ответ, добавляется еще одна запланированная операция(`write`). Когда вызывается функция `on_write`, мы читаем следующее сообщение от сервера, который будет добавлять еще одну запланированную операцию. Когда завершается функция `on_write` у нас есть одна запланированная операция (`read`). И так цикл продолжается, пока мы не решим закрыть приложение.

Функции run_one(), poll(), poll_one()

Ранее было отмечено, что обработчики асинхронных функций вызываются в том же потоке, в котором вызывался `io_service::run`. Это отмечалось для упрощения, потому что по крайней мере от 90 до 95 процентов времени это единственная функция, которую вы будете использовать. То же самое справедливо для вызовов `run_one()`, `poll()`, или `poll_one()` в потоке.

Функция `run_one()` будет выполнять и отправлять более одной асинхронной операции:

- Если нет запланированных операций, то функция сразу же завершается и возвращается 0
- Если есть отложенные операции, то выполняются функциональные блоки первой операции и возвращается 1

Вы можете рассмотреть следующий эквивалентный код:

```
io_service service;
service.run(); // OR
```

```
while ( !service.stopped()) service.run_once();
```

Вы можете использовать `run_once()` для запуска асинхронной операции, а затем ждать когда она завершится:

```
io_service service;
bool write_complete = false;
void on_write(const boost::system::error_code & err, size_t bytes)
{ write_complete = true; }
...
std::string data = "login ok";
write_complete = false;
async_write(sock, buffer(data), on_write);
do service.run_once() while (!write_complete);
```

Есть так же некоторые примеры, которые используют `run_one()` в комплекте с Boost.Asio, например `blocking_tcp_client.cpp` и `blocking_udp_client.cpp` . Функция `poll_one` запускает не более одной отложенной операции, которые готовы для запуска без блокировки:

- Если хотя бы одна отложенная операция, готовая к запуску без блокировки, то `run_one()` запустит ее и вернет 1
- В противном случае функция сразу же завершается и возвращает 0.

Отложенная операция, готовая быть запущенной без блокировки обычно является чем-то из ниже следующего:

- Таймер, который истек и должен вызваться его обработчик `async_wait`
- Операция ввода/вывода, которая завершилась (например `async_read`) и должен быть вызван ее обработчик
- Пользовательский обработчик, который был предварительно добавлен в очередь экземпляра `io_services` (это объясняется подробно в следующем разделе)

Вы можете использовать `poll_one` , чтобы убедиться, что все обработчики завершенных операций ввода/вывода запущены и перейти к следующим задачам:

```
io_service service;
while ( true)
{
        // run all handlers of completed IO operations
        while ( service.poll_one()) ;
        // ... do other work here ...
}
```

Функция `poll()` будет выполнять все операции, которые находятся в ожидании, и может быть запущена без блокировки. Следующий код эквивалентен:

```
io_service service;
service.poll(); // OR
while ( service.poll_one()) ;
```

Все предыдущие функции выбросят исключения `boost::system::system_error` в случае неудачи. Но это никогда не

должно случиться; ошибка, выброшенная здесь обычно приводит к падению, может быть это ошибка в ресурсах или один из ваших обработчиков выдал исключение. Во всяком случае, каждая из функций имеет перегрузку, которая не выбрасывает исключений, а принимает в качестве аргумента `boost::system::error_code` и устанавливает ее в качестве возвращаемого значения.

```
io_service service;
boost::system::error_code err = 0;
service.run(err);
if ( err) std::cout << "Error " << err << std::endl;
```

Асинхронная работа

Асинхронная работа это не только асинхронная обработка клиентов, подключающихся к серверу, асинхронное чтение из и запись в сокет. Это охватывает любые операции, которые могут выполняться асинхронно.
По умолчанию вы не знаете порядок, в котором вызываются обработчики всех асинхронных функций. Кроме того, обычно следующие вызовы асинхронны (исходящие от асинхронного сокет чтение/запись/прием). Вы можете использовать `service.post()` для добавления пользовательской функции, которая будет вызываться асинхронно, например:

```
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <boost/asio.hpp>
#include <iostream>
using namespace boost::asio;
io_service service;
void func(int i)
{
        std::cout << "func called, i= " << i << std::endl;
}
void worker_thread()
{
        service.run();
}
int main(int argc, char* argv[])
{
        for ( int i = 0; i < 10; ++i)
                service.post(boost::bind(func, i));
        boost::thread_group threads;
        for ( int i = 0; i < 3; ++i)
                threads.create_thread(worker_thread);
        // wait for all threads to be created
        boost::this_thread::sleep( boost::posix_time::millisec(500));
        threads.join_all();
}
```

В предыдущем примере `service.post(some_function)` добавляет асинхронный вызов функции. Эта функция сразу же завершается, после запроса экземпляра `io_service` на вызов данной `some_function` в одном из потоков, который вызывает `service.run()` . В нашем случае один из трех потоков мы создали заранее. Вы не можете быть уверены в каком порядке будут вызваны асинхронные функции. Вы не должны ожидать, что они будут вызваны в порядке их добавления (`post()`). Возможный результат работы предыдущего примера выглядит следующим образом:

```
func called, i= 0
func called, i= 2
func called, i= 1
func called, i= 4
func called, i= 3
func called, i= 6
func called, i= 7
func called, i= 8
func called, i= 5
func called, i= 9
```

Возможно будет время, когда вы захотите назначить обработчик для некоторой асинхронной функции. Скажем, вы должны пойти в ресторан (`go_to_restaurant`), сделать заказ (`order`) и поесть (`eat`). Вы хотите сначала прийти в ресторан, сделать заказ и, только потом поесть. Для этого вы будете использовать `io_service::strand` , которая будет назначать какой асинхронный обработчик вызвать. Рассмотрим следующий пример:

```
using namespace boost::asio;
io_service service;
void func(int i)
{
        std::cout << "func called, i= " << i << "/" << boost::this_thread::get_id() << std::end
l;
}
void worker_thread()
{
        service.run();
}
int main(int argc, char* argv[])
{
        io_service::strand strand_one(service), strand_two(service);
        for ( int i = 0; i < 5; ++i)
                service.post( strand_one.wrap( boost::bind(func, i)));
        for ( int i = 5; i < 10; ++i)
                service.post( strand_two.wrap( boost::bind(func, i)));
        boost::thread_group threads;
        for ( int i = 0; i < 3; ++i)
                threads.create_thread(worker_thread);
        // wait for all threads to be created
        boost::this_thread::sleep( boost::posix_time::millisec(500));
        threads.join_all();
}
```

В приведенном выше коде мы видим, что первые пять и последние пять ID потоков выводятся последовательно, а именно, `func called, i = 0` будет выведено до `func called, i = 1` , который будет выведен до `func called, i = 2` и так далее. То же самое для `func called, i = 5` , которое будет выведено до `func called, i = 6` и `func called, i = 6` будет выведено до `func called, i = 7` и так далее. Следует отметить, что даже если функции вызываются последовательно, это не означает, что они все будут вызваны в одном потоке. Возможный вариант выполнения этой программы может быть следующим:

```
func called, i= 0/002A60C8
func called, i= 5/002A6138
func called, i= 6/002A6530
func called, i= 1/002A6138
func called, i= 7/002A6530
func called, i= 2/002A6138
func called, i= 8/002A6530
func called, i= 3/002A6138
func called, i= 9/002A6530
func called, i= 4/002A6138
```

Асинхронный post() против dispatch() против wrap()

Boost.Asio предусматривает три способа добавления обработчика функции для асинхронного вызова:

- `service.post(handler)` : эта функция гарантирует, что завершится сразу же после того как сделает запрос экземпляру `io_service` на вызов заданного обработчика. Обработчик будет вызван позднее в одном из потоков, который вызвал `service.run()` .
- `service.dispatch(handler)` : это запрос экземпляру `io_service` на вызов заданного обработчика, но, кроме того, он может вызвать обработчик внутри функции, если текущий поток вызвал `service.run()` .
- `service.wrap(handler)` : эта функция создает функцию-обертку, которая будет вызывать `service.dispatch(handler)` . Это немного запутанно, вскоре я поясню что это значит.

Вы видели пример использования `service.post()` в предыдущем разделе, а также возможный результат выполнения программы. Изменим его и посмотрим как `service.dispatch` влияет на результат:

```
using namespace boost::asio;
io_service service;
void func(int i)
{
        std::cout << "func called, i= " << i << std::endl;
}
void run_dispatch_and_post()
{
        for ( int i = 0; i < 10; i += 2)
        {
                service.dispatch(boost::bind(func, i));
                service.post(boost::bind(func, i + 1));
        }
}
int main(int argc, char* argv[])
 {
        service.post(run_dispatch_and_post);
        service.run();
}
```

Прежде чем объяснить, что тут происходит, давайте посмотрим на результат, запустив программу:

```
func called, i= 0
func called, i= 2
func called, i= 4
func called, i= 6
func called, i= 8
func called, i= 1
func called, i= 3
func called, i= 5
func called, i= 7
func called, i= 9
```

Сначала пишутся четные числа, а потом нечетные. Это потому что мы используем `dispatch()` для записи четных чисел и `post()` для записи нечетных чисел. `dispatch()` вызовет обработчик прежде, чем он завершится, потому что текущий поток вызвал `service.run()`, в то время как post() завершается сразу же.
Теперь давайте поговорим об `service.wrap(handler)`. wrap() возвращает функтор, который может быть использован в качестве аргумента другой функции:

```
using namespace boost::asio;
io_service service;
void dispatched_func_1()
{
        std::cout << "dispatched 1" << std::endl;
}
void dispatched_func_2()
{
        std::cout << "dispatched 2" << std::endl;
}
void test(boost::function<void()> func)
{
        std::cout << "test" << std::endl;
        service.dispatch(dispatched_func_1);
        func();
}
void service_run()
{
        service.run();
}
int main(int argc, char* argv[])
{
        test( service.wrap(dispatched_func_2));
        boost::thread th(service_run);
        boost::this_thread::sleep( boost::posix_time::millisec(500));
        th.join();
}
```

Строка `test(service.wrap(dispatched_func_2));` будет оборачивать `dispatched_func_2` и создаст функтор, который будет передаваться в `test` в качестве аргумента. Когда вызовется `test()`, она перенаправит вызов в `dispatched_func_1()` и вызовет `func()`. На данный момент вы увидите, что вызов `func()` эквивалентен `service.dispatch(dispatched_func_2)`, так как они вызываются последовательно. Вывод программы подтверждает это:

```
test
dispatched 1
dispatched 2
```

Класс `io_service::strand` (используется для сериализации асинхронных действий) также содержит функции `poll()`, `dispatch()` и `wrap()`. Их значение такое же как и у функций `poll()`, `dispatch()` и `wrap()` из `io_service`. Тем не менее большую часть времени вы будете использовать только функцию `io_service::strand::wrap()` как аргумент для `io_service::poll()` или `io_service::dispatch()`.

Остаться в живых

Скажете вы, выполняя следующую операцию:

```
io_service service;
ip::tcp::socket sock(service);
char buff[512];
...
read(sock, buffer(buff));
```

В этом случае `sock` и `buff` оба должны пережить вызов `read()`. Другими словами, они должны быть валидными после завершения вызова `read()`. Это именно то, что вы ожидаете, все аргументы, которые вы передаете в функцию, должны быть валидными внутри нее. Все становится сложнее, когда мы идем асинхронным путем:

```
io_service service;
ip::tcp::socket sock(service);
char buff[512];
void on_read(const boost::system::error_code &, size_t) {}
...
async_read(sock, buffer(buff), on_read);
```

В этом случае `sock` и `buff` должны пережить саму операцию `read`, но мы не знаем, когда это случится, так как она асинхронна.
При использовании буферов сокета, вы можете иметь экземпляр `buffer`, который пережил асинхронный вызов (используя `boost::shared_array<>`). Здесь мы можем использовать тот же принцип, создав класс, который внутри себя содержит сокет и буферы для чтения/записи. Тогда для всех асинхронных вызовов мы передаем `boost::bind` функтор с общим указателем (shared pointer):

```
using namespace boost::asio;
io_service service;
struct connection : boost::enable_shared_from_this<connection>
{
        typedef boost::system::error_code error_code;
        typedef boost::shared_ptr<connection> ptr;
        connection() : sock_(service), started_(true) {}
        void start(ip::tcp::endpoint ep)
        {
                sock_.async_connect(ep,
```

```
                boost::bind(&connection::on_connect, shared_from_this(), _1));
        }
        void stop()
        {
                if ( !started_) return;
                started_ = false;
                sock_.close();
        }
        bool started() { return started_; }
private:
        void on_connect(const error_code & err)
        {
                // here you decide what to do with the connection: read or write
                if ( !err) do_read();
                else stop();
        }
        void on_read(const error_code & err, size_t bytes)
        {
                if ( !started() ) return;
                std::string msg(read_buffer_, bytes);
                if ( msg == "can_login")               do_write("access_data");
                else if ( msg.find("data ") == 0)    process_data(msg);
                else if ( msg == "login_fail")         stop();
        }
        void on_write(const error_code & err, size_t bytes)
        {
                do_read();
        }
        void do_read()
        {
                sock_.async_read_some(buffer(read_buffer_),
                boost::bind(&connection::on_read, shared_from_this(), _1, _2));
        }
        void do_write(const std::string & msg)
        {
                if ( !started() ) return;
                // note: in case you want to send several messages before
                // doing another async_read, you'll need several write buffers!
                std::copy(msg.begin(), msg.end(), write_buffer_);
                sock_.async_write_some(buffer(write_buffer_, msg.size()),
                boost::bind(&connection::on_write, shared_from_this(), _1, _2));
        }
        void process_data(const std::string & msg)
        {
                // process what comes from server, and then perform another write
        }
private:
        ip::tcp::socket sock_;
        enum { max_msg = 1024 };
        char read_buffer_[max_msg];
        char write_buffer_[max_msg];
        bool started_;
};
```

```
int main(int argc, char* argv[])
{
        ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
        connection::ptr(new connection)->start(ep);
}
```

Во всех асинхронных вызовах посылается функтор `boost::bind` в качестве аргумента. Этот функтор внутри себя хранит shared pointer на экземпляр `connection`. Пока асинхронная операция находится в ожидании, Boost.Asio будет хранить копию функтора `boost::bind`, который в свою очередь хранит shared pointer на `connection`. Проблема решена!
Конечно класс `connection` это только класс `skeleton`; вы должны будете приспособить его к своим потребностям (в случае сервера он будет выглядеть совсем иначе). Обратите внимание, как легко вы создаете новое подключение `connection::ptr(new connection)->start(ep)`. Это начинается (асинхронное) соединение с сервером. Если вы захотите закрыть соединение, то вы вызовите `stop()`.
Как только экземпляр начал работать (`start()`), он будет ждать подключений. Когда происходит подключение, вызывается `on_connect()`. Если ошибок нет, то вызывается операция чтения (`do_read()`). Как только операция чтения завершится, вы сможете интерпретировать сообщение; скорее всего в вашем приложении `on_read()` будет выглядеть иначе. Когда вы посылаете сообщение, вы должны скопировать его в буфер, а затем отправить, как это сделано в `do_write()`, потому что, опять же, буфер должен пережить операцию асинхронной записи. И последнее замечание – при записи помните, что вы должны указать, сколько писать, в противном случае будет посылаться весь буфер.

Резюме

Сетевое API весьма обширно. Эта глава была реализована в виде ссылки, к которой вы должны вернуться в то время, когда будете реализовывать собственное сетевое приложение.
В Boost.Asio реализована концепция конечных точек, о которых вы можете думать как об IP адресе и порте. Если вы не знаете точного IP адреса, то вы можете использовать объект `resolver` для включения имени хоста, такого как www.yahoo.com вместо одного или нескольких IP адресов.
Мы так же рассмотрели классы сокетов, которые находятся в ядре API. Boost.Asio предоставляет реализации для TCP, UDP, и ICMP, но вы можете расширить его для ваших собственных протоколов, хотя это работа не для слабонервных.
Асинхронное программирование является необходимым злом. Вы видели, почему иногда это нужно, особенно при написании серверов. Обычно вам будет достаточно вызова `service.run()` для создания асинхронного цикла, но иногда вам понадобиться пойти дальше и тогда вы сможете использовать `run_one()`, `poll()`, или `poll_one()`.
При использовании асинхронного подхода вы можете иметь свои собственные асинхронные функции, просто используйте `service.post()` или `service.dispatch()`.
Наконец для того, чтобы оба и сокет и буфер (для чтения или записи) оставались валидными в течение всего периода асинхронной операции (до завершения), мы должны принимать специальные меры предосторожности. Ваш класс `connection` должен быть производным от `enabled_shared_from_this`, содержать внутри себя все необходимые буферы и при каждом асинхронном вызове передавать shared pointer на эту операцию.
В следующей главе будет много практической работы; много прикладного кодирования при реализации таких приложений как эхо клиент/сервер.

Перевод OWASP API Security Top 10

5 2,2k 30 0

2 января 2019 в 04:57

Нет, вам не нужен Express в вашем REST API на Node.js

11 22k 84 12 +12

28 января 2014 в 12:17

User Timing API

18 12,9k 111 11 +11

ВОПРОСЫ И ОТВЕТЫ

Как правильно исопльзовать exception в api и нужно ли вообще?
PHP • Средний • 2 ответа

Для чего и как использовать sign который даёт вк при входе в мини приложение?
API • Простой • 0 ответов

Как правильно получать/добавлять данные пользователя?
API • Простой • 1 ответ

Как поэтапно вытаскивать данные из массива в json и добавлять их в следующий запрос?
API • Простой • 0 ответов

Как вывести пункты меню через API?
API • Простой • 0 ответов

Больше вопросов на Хабр Q&A

Комментарии 5

Отслеживать новые в ☐ почте ☐ трекере

Dehumanizer 24 сентября 2013 в 21:12 +4

Автор, спасибо! Пожалуйста не оставнавливайтесь на достигнутом. Передаю вам порцию вдохновения, чтоб смогли продолжить такую превосходную работу!

Ответить

Vasilui 24 сентября 2013 в 21:29 +1

Большое спасибо, буду стараться!

Ответить

Cupper 30 сентября 2013 в 18:54 –1

Автор, а вы понимаете все выше написанное? Или же это просто перевод.
Часть про run_one(), poll(), poll_one() я вообще не понял, не везде даже понятно когда вы переходите от одной к другой функции.

Замечания по тексту:

> Функция poll_one запускает не более одной отложенной операции, которые готовы для запуска без блокировки:
>
> Если хотя бы одна отложенная операция, готовая к запуску без блокировки, то run_one() запустит ее и вернет 1

> В нашем случае один из трех потоков мы создали заранее.

В коде написано другое.

> вы захотите назначить обработчик для некоторой асинхронной функции

Как это связано с dispatch/post?

Остальное не привожу, так как наверно так же и в книге написано. НО, было бы куда лучьше если бы вы делали вставки от автора для пояснение что и зачем вообще надо. Например тз написанного я не понял зачем нужен wrap. Было один раз упомянуто про strand, но зачем он нужен тоже не было сказано. Тогда смысл его упоминать?

Ответить

sanche006 3 октября 2013 в 09:20 +1

Большое спасибо за такую статью! Продолжайте в том же духе.

Ответить

RoyalDutch 23 марта 2021 в 10:40 0

Тут, как и в книге, трижды упоминается run_once(). Я не нашёл такого члена в io_service. Может быть имелось ввиду run_one?

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 40 256 +256

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