Vasilui 7 октября 2013 в 09:30

«Boost.Asio C++ Network Programming». Глава 5: Синхронное против асинхронного

Автор оригинала: Tim Packt Publishing

Программирование, C++, API

Перевод Tutorial

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

Содержание:

Авторы Boost.Asio сделали замечательную работу, давая нам возможность выбрать то, что больше удовлетворяет нашим приложениям, выбрав синхронный или асинхронный путь.
В предыдущей главе мы видели каркасы для всех типов приложений, таких как синхронный клиент, синхронный сервер, а так же их асинхронные варианты. Вы можете использовать каждый из них в качестве основы для вашего приложения. Если же возникнет необходимость вникать в подробности о каждом типе приложения, то читаем дальше.

Смешивание синхронного и асинхронного программирования

Библиотека Boost.Asio позволяет смешивать синхронное и асинхронное программирование. Лично я думаю, что это плохая идея, но Boost.Asio, как и C++ в целом, позволяет выстрелить себе в ногу, если вы того захотите.
Вы легко можете попасть в ловушку, особенно если ваше приложение работает асинхронно. Например, в ответ на асинхронную операцию записи вы, скажем, делаете асинхронную операцию чтения:

```
io_service service;
ip::tcp::socket sock(service);
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
```

```
void on_write(boost::system::error_code err, size_t bytes)
{
        char read_buff[512];
        read(sock, buffer(read_buff));
}
async_write(sock, buffer("echo"), on_write);
```

Наверняка синхронная операция чтения будет блокировать текущий поток, таким образом, любые другие незавершенные асинхронные операции будут находиться в режиме ожидания (для этого потока). Это плохой код и может привести к тому, что приложение начнет тормозить или вообще заблокируется (весь смысл использования асинхронного подхода — это чтобы избежать блокировок, поэтому используя синхронные операции, вы отрицаете это). Если у вас есть синхронное приложение, то маловероятно, что вы будете использовать асинхронные операции чтения или записи, так как думать синхронно уже значит думать линейным образом (сделать A, потом B, потом C и так далее).
Единственный случай, по моему мнению, когда синхронные и асинхронные операции могут работать вместе, это когда они полностью отделены друг от друга, например, синхронная сеть и асинхронные операции ввода и вывода из базы данных.

Доставка сообщений от клиента серверу и наоборот

Очень важная часть хорошего клиент/серверного приложения это доставка сообщений туда и обратно (от сервера к клиенту и от клиента к серверу). Необходимо указать, что идентифицирует сообщение. Другими словами, когда происходит чтение входящего сообщения, как мы можем узнать, что сообщение прочитано полностью?
Вам необходимо определить конец сообщения (начало определить легко, это первый пришедший байт после конца последнего сообщения), но вы убедитесь, что это не так легко.
Вы можете:

- Сделать сообщение фиксированного размера (это не очень хорошая идея; что делать, когда вам понадобится отправить больше данных?)
- Сделать конкретный символ, завершающий сообщение, такой как ‘\n’ или ‘\0’
- Указать длину сообщения в качестве префикса сообщения и так далее

На протяжении всей книги, я решил использовать «символ ‘\n’ в качестве конца каждого сообщения». Так, чтение сообщений будет демонстрировать следующий фрагмент кода:

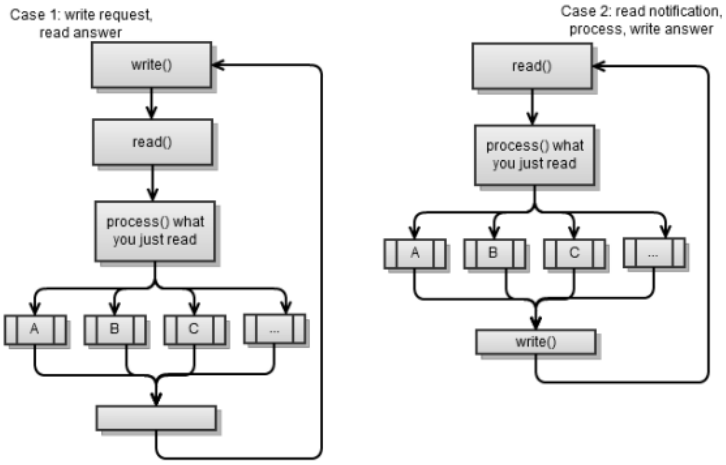
```
char buff_[512];
// synchronous read
read(sock_, buffer(buff_), boost::bind(&read_complete, this, _1, _2));
// asynchronous read
async_read(sock_   buffer(buff_),MEM_FN2(read_complete,_1,_2),
        MEM_FN2(on_read,_1,_2));
size_t read_complete(const boost::system::error_code & err, size_t bytes)
{
        if ( err)
                return 0;
        already_read_ = bytes;
        bool found = std::find(buff_, buff_ + bytes, '\n') < buff_ + bytes;
        // we read one-by-one until we get to enter, no buffering
        return found ? 0 : 1;
}
```

Оставим указание длины в качестве префикса сообщения в качестве упражнения для читателя, это довольно легко.

Синхронный ввод/вывод в клиентских приложениях

Синхронный клиент, как правило, бывает двух видов:

- Он запрашивает что-то от сервера, читает и обрабатывает ответ. Затем запрашивает что-то еще и так далее. Это, по сути, синхронный клиент, который рассматривался в предыдущей главе.
- Читает входящее сообщение от сервера, обрабатывает его и пишет ответ. Затем читает следующее входящее сообщение и так далее.



Оба сценария используют следующую стратегию: сделать запрос – прочитать ответ. Другими словами, одна сторона делает запрос, на который другая сторона отвечает в ответ. Это простой способ реализации клиент/серверного приложения и это то, что я рекомендую вам.
Вы всегда можете создать Mambo Jambo клиент/сервер, где каждая сторона пишет когда угодно, но, весьма вероятно, что этот путь приведет к катастрофе (как вы узнаете что произошло, когда клиент или сервер заблокируется?).
Предыдущие сценарии могут выглядеть одинаково, но, они очень разные:

- В первом случае сервер реагирует на запросы (сервер ждет запросы от клиентов и отвечает на них). Это дерганное (pull-like) соединение, когда клиент получает по запросу от сервера то, что ему необходимо.
- В последнем случае сервер посылает клиенту события, на которые тот реагирует. Это толчковое (push-like) соединение, когда сервер проталкивает уведомления/события клиентам.

В основном вы будете сталкиваться с pull-like клиент/серверными приложениями, которые облегчают разработку, а так же, как правило, являются нормой.
Вы можете смешивать эти два подхода: получить по запросу (клиент-сервер) и протолкнуть запрос (сервер-клиент), однако, это сложно и лучше этого избежать. Есть проблема смешивания двух этих подходов, если вы используете стратегию сделать запрос – прочитать ответ; может произойти следующее:

- Клиент пишет (делает запрос)
- Сервер пишет (посылает уведомление клиенту)
- Клиент читает то, что написал сервер и интерпретирует это в качестве ответа на свой запрос
- Сервер блокирует ожидание ответа от клиента, который придет, когда клиент сделает новый запрос
- Клиент пишет (делает новый запрос)
- Сервер будет интерпретировать этот запрос в качестве ответа, который он ждал
- Клиент блокируется (сервер не посылает обратного ответа, потому что он интерпретировал запрос клиента в качестве

ответа на свое уведомление).

В pull-like клиент/серверном приложении можно было легко избежать предыдущего сценария. Вы можете моделировать push-like поведение путем реализации процесса пинговки, когда клиент проверяет связь с сервером, скажем, каждые 5 секунд. Сервер может ответить что-то типа ping_ok, если нечего сообщить или ping_[event_name], если есть событие для оповещения. Потом клиент может инициировать новый запрос для обработки этого события.
Повторим, предыдущий сценарий иллюстрирует синхронный клиент из предыдущей главы. Его основной цикл:

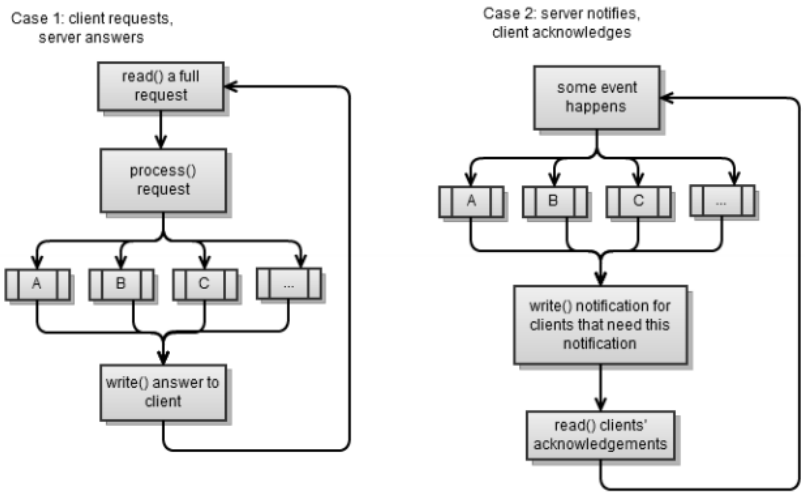
```
void loop()
{
        // read answer to our login
        write("login " + username_ + "\n");
        read_answer();
        while ( started_)
        {
                write_request();
                read_answer();
                ...
        }
}
```

Позвольте изменить его, чтобы соответствовать последнему сценарию:

```
void loop()
{
        while ( started_)
        {
                read_notification();
                write_answer();
        }
}
void read_notification()
{
        already_read_ = 0;
        read(sock_, buffer(buff_), boost::bind(&talk_to_svr::read_complete, this, _1, _2));
        process_notification();
}
void process_notification()
{
        // ... see what the notification is, and prepare answer
}
```

Синхронный ввод/вывод в серверных приложениях

Серверы, как и клиенты, бывают двух типов, они соответствуют двум сценариям из предыдущего раздела. Опять же, оба сценария используют стратегию: создать запрос – прочитать ответ.



Первый сценарий это синхронный сервер, который мы реализовали в предыдущей главе. Прочитать запрос полностью не легко, если вы работаете синхронно, так как вы хотите избежать блокировок (вы всегда читаете столько, сколько можете).

```
void read_request()
{
        if ( sock_.available())
                already_read_ += sock_.read_some(buffer(buff_ + already_read_,
                        max_msg -already_read_));
}
```

После того как сообщение было полностью прочитано, просто обрабатываем его и отвечаем клиенту:

```
void process_request()
{
        bool found_enter = std::find(buff_, buff_ + already_read_, '\n') < buff_ + already_read
_;
        if ( !found_enter)
                return; // message is not full
        size_t pos = std::find(buff_, buff_ + already_read_, '\n') - buff_;
        std::string msg(buff_, pos);
        ...
        if ( msg.find("login ") == 0)
                on_login(msg);
        else if ( msg.find("ping") == 0)
                on_ping();
        else
                ...
}
```

Если бы мы хотели, чтобы наш сервер стал push-like сервером, то мы изменили бы его следующим образом:

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

```
array clients;
array notify;
std::string notify_msg;
void on_new_client()
{
        // on a new client, we notify all clients of this event
        notify = clients;
        std::ostringstream msg;
        msg << "client count " << clients.size();
        notify_msg = msg.str();
        notify_clients();
}
void notify_clients()
{
        for ( array::const_iterator b = notify.begin(), e = notify.end(); b != e; ++b)
        {
                (*b)->sock_.write_some(notify_msg);
        }
}
```

Функция `on_new_client()` – функция одного события, где мы должны уведомить о нем всех клиентов. `notify_clients` это функция, которая будет уведомлять клиентов, которые подписаны на данное событие. Сервер посылает сообщение, но не ждет ответа от каждого клиента, так как это может привести к блокировке. Когда от клиента приходит ответ, то клиент может сказать нам, что это именно ответ на наше уведомление (и мы сможем обработать его правильно).

Потоки в синхронном сервере

Это очень важный фактор: сколько потоков мы выделим для обработки клиентов?
Для синхронного сервера нам понадобится как минимум один поток, который будет обрабатывать новые подключения:

```
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_);
        }
}
```

Для существующих клиентов:

- Мы можем пойти одно-поточным путем. Это самый простой путь и именно его мы выбрали, когда реализовывали синхронный сервер в 4 главе. Он с легкостью справляется со 100-200 одновременными подключениями, а иногда может и больше, что достаточно для подавляющего большинства случаев.
- Мы можем сделать поток для каждого клиента. Это очень редко бывает хорошим вариантом, он будет тратить много потоков, делая отладку иногда затруднительной, и, хотя, вероятно, будет обрабатываться более 200 одновременно работающих пользователей, вскоре после этого он достигнет своего предела.
- Мы можем сделать фиксированное число потоков для обработки существующих клиентов.

Третий вариант очень сложно реализовать в синхронном сервере. Весь класс `talk_to_client` стал потоко-безопасным. Затем вы будете нуждаться в специальном механизме, чтобы знать какие потоки обрабатывают каких клиентов. Для этого у вас есть два варианта:

- Назначить конкретного клиента конкретному потоку; например, первый поток обрабатывает первые 20 клиентов, второй поток обрабатывает клиентов с 21 по 40 и так далее. Когда клиент используется, мы извлекаем его из множества существующих клиентов. После того, как мы поработали с этим клиентом, помещаем его обратно в список. Каждый поток будет циклически обходить всех существующих клиентов, и брать на обработку первого клиента с полным запросом (мы полностью прочитали входящее сообщение от клиента) и отвечать на него.

- Сервер может перестать отвечать на запросы:
  - В первом случае несколько клиентов, обрабатываемые в одном потоке одновременно создают запросы, а один поток может обработать только один запрос за раз. Тем не менее, мы ничего не можем сделать в этом случае.
  - Во втором случае мы одновременно получаем больше запросов, чем у нас есть потоков. В этом случае мы просто можем создать новые потоки, чтобы справиться с нагрузкой.

Следующий код, который похож на исходную функцию `answer_to_client` , показывает, как последний сценарий может быть реализован:

```
struct talk_to_client : boost::enable_shared_from_this<talk_to_client>
{
        ...
        void answer_to_client()
        {
                try
                {
                        read_request();
                        process_request();
                }
                catch ( boost::system::system_error&)
                {
                        stop();
                }
        }
};
```

Мы будем изменять его, как показано ниже:

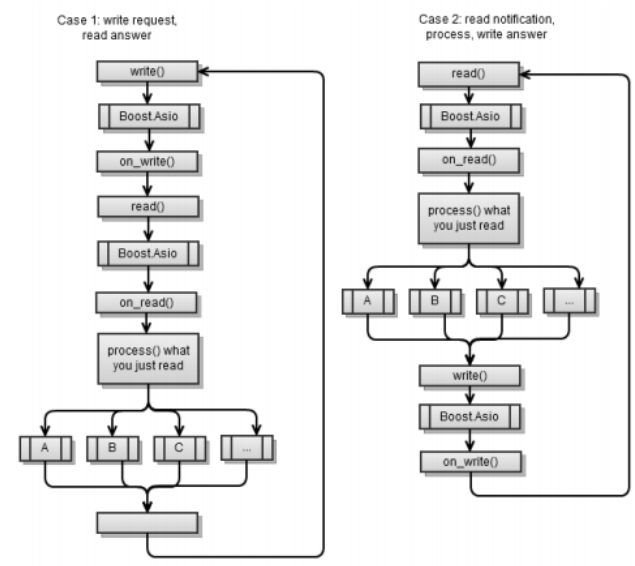
```
struct talk_to_client : boost::enable_shared_from_this<talk_to_client>
{
        boost::recursive_mutex cs;
        boost::recursive_mutex cs_ask;
        bool in_process;
        void answer_to_client()
        {
                {
                        boost::recursive_mutex::scoped_lock lk(cs_ask);
                        if ( in_process)
```

```
                        return;
                in_process = true;
        }
        {
                boost::recursive_mutex::scoped_lock lk(cs);
                try
                {
                        read_request();
                        process_request();
                }
                catch ( boost::system::system_error&)
                {
                        stop();
                }
        }
        {
                boost::recursive_mutex::scoped_lock lk(cs_ask);
                in_process = false;
        }
    }
};
```

Пока мы будем обрабатывать клиента, его экземпляр `in_process` будет установлен в `true`, и другие потоки будут игнорировать этого клиента. Дополнительным бонусом является то, что функция `handle_clients_thread()` не может быть модифицирована; вы можете просто создать столько функций `handle_clients_thread()` сколько вам захочется.

Асинхронный ввод/вывод в клиентских приложениях

Основной рабочий процесс чем-то напоминает этот же процесс в синхронном клиентском приложении, с той разницей, что Boost.Asio находится между каждым запросом `async_read` и `async_write`.

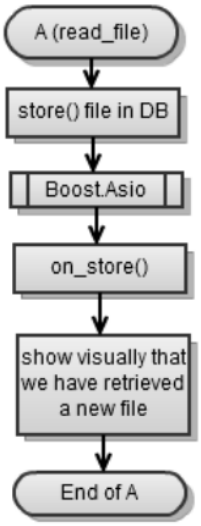


Первый сценарий тот же, как был реализован асинхронный клиент в главе 4. Помните, что в конце каждой асинхронной операции необходимо начинать другую асинхронную операцию, чтобы функция `service.run()` не завершала своей деятельности.

Чтобы привести первый сценарий ко второму нам понадобится использовать следующий фрагмент кода:

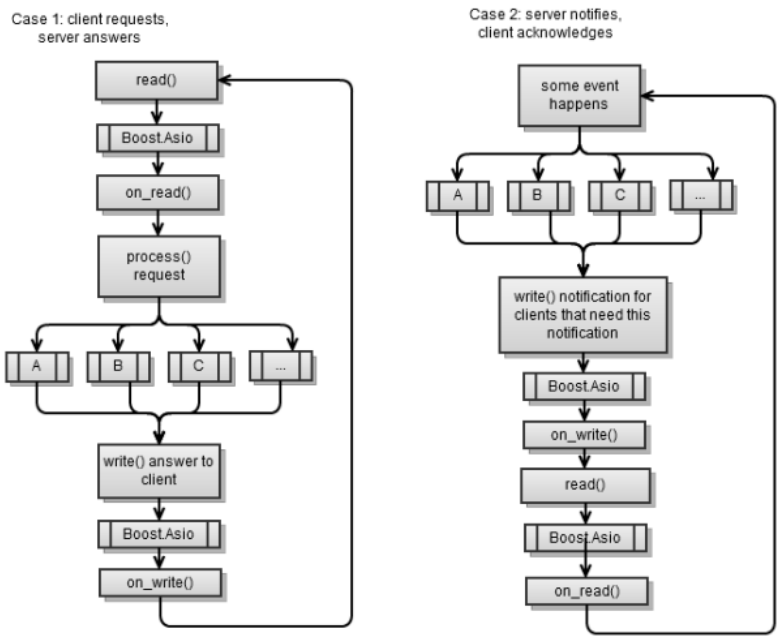
```
void on_connect()
{
        do_read();
}
void do_read()
{
        async_read(sock_, buffer(read_buffer_), MEM_FN2(read_complete,_1,_2),
                MEM_FN2(on_read,_1,_2));
}
void on_read(const error_code & err, size_t bytes)
{
        if ( err)
                stop();
        if ( !started() )
                return;
        std::string msg(read_buffer_, bytes);
        if ( msg.find("clients") == 0)
                on_clients(msg);
        else
                ...
}
void on_clients(const std::string & msg)
{
        std::string clients = msg.substr(8);
        std::cout << username_ << ", new client list:" << clients ;
        do_write("clients ok\n");
}
```

Обратите внимание, что как только вы успешно подключаетесь, вы начинаете читать с сервера. Каждая функция `on_[event]` заканчивает ее и пишет ответ серверу.
Красота асинхронного подхода в том, что вы можете смешивать сетевые операции ввода/вывода с любыми другими асинхронными операциями используя Boost.Asio, чтобы организовать все это. Даже при том, что поток не так ясен как синхронный поток, вы, практически, можете думать о нем как об синхронном.
Скажем, вы читаете файл с веб-сервера и сохраняете его в базу данных (асинхронно). Вы, практически, можете думать об этом, как показано в следующей блок-схеме:



Асинхронный ввод/вывод в серверных приложениях

Опять вездесущие два случая, первый сценарий (pull) и второй сценарий (push):



Первый сценарий асинхронного сервера был реализован в предыдущей главе. В конце каждой асинхронной операции необходимо начинать другую асинхронную операцию, чтобы `service.run()` не прекращала своего действия.
Вот каркас кода, который урезан. Ниже приведены все члены класса `talk_to_client` :

```
void start()
{
        ...
        do_read(); // first, we wait for client to login
```

```
}
void on_read(const error_code & err, size_t bytes)
{
        std::string msg(read_buffer_, bytes);
        if ( msg.find("login ") == 0)
                on_login(msg);
        else if ( msg.find("ping") == 0)
                on_ping();
        else
                ...
}
void on_login(const std::string & msg)
{
        std::istringstream in(msg);
        in >> username_ >> username_;
        do_write("login ok\n");
}
void do_write(const std::string & msg)
{
        std::copy(msg.begin(), msg.end(), write_buffer_);
        sock_.async_write_some( buffer(write_buffer_, msg.size()),
                MEM_FN2(on_write,_1,_2));
}
void on_write(const error_code & err, size_t bytes)
{
        do_read();
}
```

В двух словах, мы всегда ждем операцию чтения, как только она завершится, мы обрабатываем сообщение и отвечаем обратно клиенту.
Преобразуем предыдущий код в push сервер:

```
void start()
{
        ...
        on_new_client_event();
}
void on_new_client_event()
{
        std::ostringstream msg;
        msg << "client count " << clients.size();
        for ( array::const_iterator b = clients.begin(), e = clients.end();b != e; ++b)
                (*b)->do_write(msg.str());
}
void on_read(const error_code & err, size_t bytes)
{
        std::string msg(read_buffer_, bytes);
        // basically here, we only acknowledge
        // that our clients received our notifications
}
void do_write(const std::string & msg)
```

```
{
        std::copy(msg.begin(), msg.end(), write_buffer_);
        sock_.async_write_some( buffer(write_buffer_, msg.size()),
                MEM_FN2(on_write,_1,_2));
}
void on_write(const error_code & err, size_t bytes)
{
        do_read();
}
```

Когда происходит событие, скажем, `on_new_client_event`, всем клиентам, которые должны быть проинформированы об этом событии, будут отправлены сообщения. Когда они ответят, мы поймем, что они обработали полученное событие. Обратите внимание, что мы никогда не закончим асинхронно ждать событий (поэтому `service.run()` не закончит работать), так как мы всегда ждем новых клиентов.

Потоки в асинхронном сервере

Асинхронный сервер был показан в главе 4, он одно-поточный, так как там все происходит в функции `main()`:

```
int main()
{
        talk_to_client::ptr client = talk_to_client::new_();
        acc.async_accept(client->sock(), boost::bind(handle_accept,client,_1));
        service.run();
}
```

Красота асинхронного подхода заключается в простоте перехода от одно-поточного к много-поточному варианту. Вы всегда можете пойти одно-поточным путем, по крайней мере, пока ваших клиентов не будет более 200 одновременно или около того. Тогда, чтобы перейти от одного потока к 100 потокам, вам надо будет использовать следующий фрагмент кода:

```
boost::thread_group threads;
void listen_thread()
{
        service.run();
}
void start_listen(int thread_count)
{
        for ( int i = 0; i < thread_count; ++i)
                threads.create_thread( listen_thread);
}
int main(int argc, char* argv[])
{
        talk_to_client::ptr client = talk_to_client::new_();
        acc.async_accept(client->sock(), boost::bind(handle_accept,client,_1));
        start_listen(100);
        threads.join_all();
}
```

Конечно, как только вы начинаете использовать много-поточность, вы должны думать о потоко-безопасности. Даже если вы

вызовете `async_*` в потоке A, то процедура ее завершения может быть вызвана в потоке B (до тех пор пока поток B вызывает `service.run()`). Само по себе это не является проблемой. До тех пор, пока вы будете следовать логической последовательности, то есть, от `async_read()` к `on_read()`, от `on_read()` к `process_reques` t, от `process_request` к `async_write()`, от `async_write()` к `on_write()`, от `on_write()` к `async_read()` и нет никаких `public` функций, которые вызывали бы ваш класс `talk_to_client`, хотя разные функции могут быть вызваны в разных потоках, они все равно будут вызваны последовательно. Таким образом, мьютексы не нужны.
Это, однако, означает, что для клиента может быть только одна асинхронная операция в ожидании. Если в какой-то момент у клиента имеется две отложенные асинхронные функции, то вам понадобятся мьютексы. Потому что две отложенные операции могут завершиться примерно в одно время и в конечном итоге мы могли бы вызвать их обработчики одновременно в двух разных потоках. Таким образом, есть необходимость в потоко-безопасности, таким образом, в мьютексах.
В нашем асинхронном сервере на самом деле есть одновременно две отложенные операции:

```
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 post_check_ping()
{
        timer_.expires_from_now(boost::posix_time::millisec(5000));
        timer_.async_wait( MEM_FN(on_check_ping));
}
```

При выполнении операции чтения мы будем асинхронно ждать ее завершения в течение некоторого периода. Таким образом, существует необходимость в потоко-безопасности. Мой совет, если вы планируете, что вы выберете много-поточный вариант, то сделайте ваш класс потоко-безопасным с самого начала. Это, как правило, не повредит производительности (вы, конечно, можете проверить это). Кроме того, если вы планируете пойти много-поточным путем, то идите по нему с самого начала. Таким образом, вы столкнетесь с возможными проблемами на ранней стадии. Как только вы обнаружите проблему, первое, что вы должны проверить происходит ли это при одном запущенном потоке? Если да, то это легко, просто отладьте. В противном случае вы, вероятно, забыли залочить (mutex) какую-то функцию.
Поскольку наш пример нуждается в потоко-безопасности, мы изменили `talk_to_client`, используя мьютексы. Кроме того, у нас есть массив клиентов, на который мы ссылаемся несколько раз в коде, который также нуждается в собственном мьютексе. Избежать дедлока и повреждения памяти не так просто. Вот как надо было изменить функцию `update_clients_changed()`:

```
void update_clients_changed()
{
        array copy;
        {
                boost::recursive_mutex::scoped_lock lk(clients_cs);
                copy = clients;
        }
        for( array::iterator b = copy.begin(), e = copy.end(); b != e; ++b)
                (*b)->set_clients_changed();
}
```

То чего мы хотим избежать это, чтобы два мьютекса были заблокированы в одно и то же время (что может привести к ситуации взаимной блокировки). В нашем случае, мы не хотим, чтобы `clients_cs` и клиентский `cs_` мьютекс были заблокированы одновременно.

Асинхронные операции

Boost.Asio так же позволяет выполнять любые ваши функции асинхронно. Просто используйте следующий фрагмент кода:

```
void my_func()
{
        ...
}
service.post(my_func);
```

Можете убедиться, что `my_func` вызывается в одном из потоков, которые вызывают `service.run()`. Вы также можете запустить асинхронную функцию и сделать завершающий обработчик, который сообщит вам, когда функция завершится. Псевдокод будет выглядеть следующим образом:

```
void on_complete()
{
        ...
}
void my_func()
{
        ...
        service.post(on_complete);
}
async_call(my_func);
```

Здесь нет функции `async_call`, вам придется создать свою собственную. К счастью это не так сложно. Смотрите следующий фрагмент кода:

```
struct async_op : boost::enable_shared_from_this<async_op>, ...
{
        typedef boost::function<void(boost::system::error_code)> completion_func;
        typedef boost::function<boost::system::error_code ()> op_func;
        struct operation { ... };
        void start()
        {
                {
                        boost::recursive_mutex::scoped_lock lk(cs_);
                        if ( started_)
                                return;
                        started_ = true;
                 }
                boost::thread t( boost::bind(&async_op::run,this));
        }
        void add(op_func op, completion_func completion, io_service &service)
        {
                self_ = shared_from_this();
                boost::recursive_mutex::scoped_lock lk(cs_);
                ops_.push_back( operation(service, op, completion));
                if ( !started_)
```

```
                        start();
        }
        void stop()
        {
                boost::recursive_mutex::scoped_lock lk(cs_);
                started_ = false;
                ops_.clear();
        }
private:
        boost::recursive_mutex cs_;
        std::vector<operation> ops_;
        bool started_;
        ptr self_;
};
```

В структуре `async_op` создается фоновый поток, который будет работать (`run()`) со всеми асинхронными функциями, которые вы добавляете (`add()`) к нему. Для меня это не представляется чем-то сложным, так как для каждой операции выполняется следующее:

- Функция вызывается асинхронно.
- `completion` функция вызывается при первом завершении функции
- Экземпляр `io_service` , который будет выполнять `completion` функцию. Это место, где вы будете уведомлены о завершении. Смотрите следующий фрагмент кода:

```
struct async_op : boost::enable_shared_from_this<async_op> , private boost::noncopyable
{
        struct operation
        {
                operation(io_service & service, op_func op, completion_func
                        completion): service(&service),
                        op(op)completion(completion), work(new o_service::work(service)){}
                operation() : service(0) {}
                io_service * service;
                op_func op;
                completion_func completion;
                typedef boost::shared_ptr<io_service::work> work_ptr;
                work_ptr work;
        };
        ...
};
```

Обратите внимание, что в то время пока операция не закончена, мы конструируем экземпляр `io_service::work` , поэтому `service.run()` не заканчивает своей работы на этом, пока мы не закончили наш асинхронный вызов (в то время, пока жив экземпляр `io_service::work, service.run()` будет считать, что у него есть работа). Посмотрите на следующий код:

```
struct async_op : ...
{
```

```
        typedef boost::shared_ptr<async_op> ptr;
        static ptr new_()
        {
                return ptr(new async_op);
        }
        ...
        void run()
        {
                while ( true)
                {
                        {
                                boost::recursive_mutex::scoped_lock lk(cs_);
                                if ( !started_)
                                        break;
                        }
                        boost::this_thread::sleep( boost::posix_time::millisec(10));
                        operation cur;
                        {
                                boost::recursive_mutex::scoped_lock lk(cs_);
                                if ( !ops_.empty())
                                {
                                        cur = ops_[0];
                                        ops_.erase( ops_.begin());
                                }
                        }
                        if ( cur.service)
                                cur.service->post(boost::bind(cur.completion, cur.op() ));
                }
                self_.reset();
        }
};
```

Функция `run()` работающая в фоновом потоке смотрит, есть ли работа, чтобы ее сделать; если да, то выполняет асинхронные функции по очереди. В конце каждого вызова она вызывает соответствующую функцию завершения.
Чтобы проверить это, мы создадим функцию `compute_file_checksum` , которая будет выполняться асинхронно:

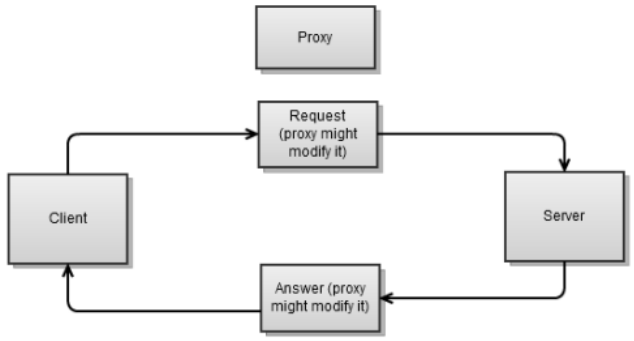
```
size_t checksum = 0;
boost::system::error_code compute_file_checksum(std::string file_name)
{
        HANDLE file = ::CreateFile(file_name.c_str(), GENERIC_READ, 0, 0,
                OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, 0);
        windows::random_access_handle h(service, file);
        long buff[1024];
        checksum = 0;
        size_t bytes = 0, at = 0;
        boost::system::error_code ec;
        while ( (bytes = read_at(h, at, buffer(buff), ec)) > 0)
        {
                at += bytes;
                bytes /= sizeof(long);
                for ( size_t i = 0; i < bytes; ++i)
```

```
                        checksum += buff[i];
        }
        return boost::system::error_code(0, boost::system::generic_category());
}
void on_checksum(std::string file_name, boost::system::error_code)
{
        std::cout << "checksum for " << file_name << "=" << checksum << std::endl;
}
int main(int argc, char* argv[])
{
        std::string fn = "readme.txt";
        async_op::new_()->add( service, boost::bind(compute_file_checksum,fn),
                boost::bind(on_checksum,fn,_1));
        service.run();
}
```

Обратите внимание, как я только что показал вам возможную реализацию вызова функции асинхронно. Вместо реализации фонового потока, как это сделал я, вы можете использовать внутренний экземпляр `io_service`, к которому вы отправите (`post()`) асинхронный вызов функции. Оставим это в качестве упражнение для читателя.
Так же можно расширить класс, чтобы показывать ход выполнения асинхронной работы (например, в процентах). В таком случае, вы могли бы показать прогресс в прогресс-баре, в основном потоке.

Реализация прокси

Прокси обычно находится между клиентом и сервером. Он принимает запрос от клиента, может его изменить, и направляет его на сервер. Затем он берет ответ от сервера, может его изменить, и направляет его клиенту.



В чем же особенность прокси-сервера? Для наших целей в том, что для каждого соединения мы будем иметь два сокета, один для клиента, другой для сервера. Это совсем немного усложняет реализацию прокси.
Реализация прокси для синхронного приложения будет более сложной, чем для асинхронного; данные могут идти от обоих концов (от клиента и сервера), в то же время данные могут направляться к ним обоим (клиенту и серверу). Это означает, что если мы выберем синхронный вариант, то мы можем закончить тем, что заблокируем чтение или запись с одной стороны, в то время когда нам нужно прочитать или записать с другой стороны, что означает, что мы перестанем реагировать на одном из концов.
Рассмотрим следующие пункты простого примера асинхронного прокси:

- В нашем случае мы знаем, что оба соединения установлены. Это не всегда так, например, для веб-прокси клиент говорит нам адрес сервера.

- Ради простоты, рассмотрим следующий фрагмент кода, он не потоко-безопасный:

```
class proxy : public boost::enable_shared_from_this<proxy>
{
        proxy(ip::tcp::endpoint ep_client, ip::tcp::endpoint ep_server) : ... {}
public:
        static ptr start(ip::tcp::endpoint ep_client, ip::tcp::endpoint ep_svr)
        {
                ptr new_(new proxy(ep_client, ep_svr));
                // ... connect to both endpoints
                return new_;
        }
        void stop()
        {
                // ... stop both connections
        }
        bool started()
        {
                return started_ == 2;
        }
private:
        void on_connect(const error_code & err)
        {
                if ( !err)
                {
                        if ( ++started_ == 2)
                                on_start();
                }
                else
                        stop();
        }
        void on_start()
        {
                do_read(client_, buff_client_);
                do_read(server_, buff_server_);
        }
        ...
private:
        ip::tcp::socket client_, server_;
        enum { max_msg = 1024 };
        char buff_client_[max_msg], buff_server_[max_msg];
        int started_;
};
```

Это очень простой прокси. При подключении на обоих концах он начинает читать на обоих соединениях (функция `on_start()`):

```
class proxy : public boost::enable_shared_from_this<proxy>
{
        ...
```

```
        void on_read(ip::tcp::socket & sock, const error_code& err, size_t bytes)
        {
                char * buff = &sock == &client_ ? buff_client_ : buff_server_;
                do_write(&sock == &client_ ? server_ : client_, buff, bytes);
        }
        void on_write(ip::tcp::socket & sock, const error_code &err, size_t bytes)
        {
                if ( &sock == &client_)
                        do_read(server_, buff_server_);
                else
                        do_read(client_, buff_client_);
        }
        void do_read(ip::tcp::socket & sock, char* buff)
        {
                async_read(sock, buffer(buff, max_msg),
                        MEM_FN3(read_complete,ref(sock),_1,_2),
                        MEM_FN3(on_read,ref(sock),_1,_2));
        }
        void do_write(ip::tcp::socket & sock, char * buff, size_t size)
        {
                sock.async_write_some(buffer(buff,size),
                        MEM_FN3(on_write,ref(sock),_1,_2));
        }
        size_t read_complete(ip::tcp::socket & sock, const error_code & err, size_t bytes)
        {
                if ( sock.available() > 0)
                        return sock.available();
                return bytes > 0 ? 0 : 1;
        }
};
```

После каждого успешного чтения (on_read) он передает сообщение другой стороне. Как только сообщение было успешно передано (on_write), мы снова начинаем читать.
Чтобы это работало, используйте следующий фрагмент кода:

```
int main(int argc, char* argv[])
{
        ip::tcp::endpoint ep_c( ip::address::from_string("127.0.0.1"), 8001);
        ip::tcp::endpoint ep_s( ip::address::from_string("127.0.0.1"), 8002);
        proxy::start(ep_c, ep_s);
        service.run();
}
```

Вы заметили, что я повторно использую буферы (`buff_client_` и `buff_server_`) для чтения и записи. Это повторное использование нормально, потому что прочитанное сообщение от клиента написано серверу прежде, чем новое сообщение будет прочитано от клиента и наоборот. Это также означает, что эта конкретная реализация страдает от проблемы живого отклика. В то время, когда мы находимся в процессе записи на стороне B, мы не читаем от стороны A (мы перезапускаем

Пользователь

ПОХОЖИЕ ПУБЛИКАЦИИ

14 декабря 2020 в 00:13

Руководство Google по стилю в C++. Часть 4

3 4,1k 40 3 +3

8 апреля 2019 в 12:03

Operating Systems: Three Easy Pieces. Part 3: Process API (перевод)

5 1,7k 16 1 +1

11 июля 2014 в 04:16

О бедном C++ API замолвите словцо!

20 35,9k 174 71 +71

Комментарии 4

Отслеживать новые в ☐ почте ☐ трекере

sba 7 октября 2013 в 12:44 0

В теле функции async_op::start() экземпляр потока boost::thread t будет разрушен до завершения самого потока.

Ответить

PSIAlt 7 октября 2013 в 12:51 +1

Ничего плохого не будет — он перейдёт в detached-состояние

Ответить

antoshkka 7 октября 2013 в 14:18 +1

Если используется версия 4 Boost.Thread (выставляется с помощью макроса), то вызовется std::terminate и все приложение навернется. Аналогичная ситуация будет при использовании std::thread вместо boost::thread.

Чтобы std::terminate() не вызывался, надо явно вызывать t.detach()

Ответить

DZhon 8 октября 2013 в 14:25 +1

Прекрасно, жду самого вкусного (сопрограмм).
Большое вам спасибо!

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

Когда старый подход хуже новых двух: коллекция про нетривиальную разработку

Мегапост