客户端应用实例

第一章：初始化和安装

设置基本类型、打开和关闭连接、发送和接收消息。

Step 1 一个基本的循环程序

一个基本的循环程序，提示用户输入命令，然后对其进行处理。在本教程中，我们将修改这个程序，以便通过WebSocket连接从远程服务器执行任务和检索数据。

Build

使用clang++来编译 `step1.cpp`：

```
clang++ step1.cpp
```

代码

注意：在git存储库中，本教程文件旁边有每个步骤的代码快照。

假设我们目前只写了这么几行代码：

```
#include <iostream>
#include <string>

int main() {
    bool done = false;
    std::string input;

    while (!done) {
        std::cout << "Enter Command: ";
        std::getline(std::cin, input);

        if (input == "quit") {
            done = true;
        } else if (input == "help") {
            std::cout
                << "\nCommand List:\n"
                << "help: Display this help text\n"
                << "quit: Exit the program\n"
                << std::endl;
        } else {
            std::cout << "Unrecognized Command" << std::endl;
        }
    }

    return 0;
}
```

可见这仅仅是一个简单的循环程序。

Step 2 添加WebSocket++包含文件（头文件）并设置终端类型

添加WebSocket++包含文件（头文件）并设置终端类型。

WebSocket++包括两种主要的对象类型。终端和连接。终端创建并启动新的连接，并维护这些连接的默认设置。终端还管理任何共享的网络资源。

连接存储每个WebSocket会话特定的信息。

注意:
一旦连接启动，终端和连接之间就没有链接了。终端将所有默认设置复制到新连接中。更改终端上的默认设置只会影响将来的连接（只会影响后面新建的连接）。
连接不会维护与其相关联终端的链接。终端不维护未完成连接的列表。如果您的应用程序需要遍历所有连接，那么它本身就需要维护它们的列表。

WebSocket++终端是通过结合终端所承担的角色（所起的作用）和终端配置来构建的。有两种不同类型的终端角色，分别用于WebSocket会话中的客户端和服务器角色。这是一个客户端教程，所以我们将使用客户端角色 `websocketpp::client` ，它由 `<websocketpp/client.hpp>` 头文件提供。

> **术语:终端配置**
>
> WebSocket++终端有一组可以在编译时通过 `config` 模板参数配置的设置。配置是一个包含类型和静态常量的结构体，用于生成具有特定属性的终端。根据使用的配置，终端将有不同的可用方法，并可能有额外的第三方依赖。

Build

添加WebSocket++给我们的程序添加了一些必须在构建系统中解决的依赖项。首先，WebSocket++和Boost库头文件必须在构建系统的include搜索路径中。这取决于你在哪里安装了WebSocket++头文件，以及你使用的是什么构建系统。

除了新的头文件，boost::asio还依赖于 `boost_system` 共享库。这需要添加(静态或动态)到链接器。有关链接到共享库的说明，请参阅构建环境文档。

使用clang++来编译 `step2.cpp` ：

```
clang++_step2.cpp -lboost_system
```

Code so far

现在的代码为：

```
//添加了websocketpp的依赖 头文件
#include <websocketpp/config/asio_no_tls_client.hpp>
#include <websocketpp/client.hpp>

#include <iostream>
#include <string>

//定义了终端类型 client
typedef websocketpp::client<websocketpp::config::asio_client> client;

int main() {
    bool done = false;
    std::string input;

    while (!done) {
        std::cout << "Enter Command: ";
        std::getline(std::cin, input);

        if (input == "quit") {
            done = true;
        } else if (input == "help") {
            std::cout
                << "\nCommand List:\n"
                << "help: Display this help text\n"
                << "quit: Exit the program\n"
                << std::endl;
        } else {
            std::cout << "Unrecognized Command" << std::endl;
        }
    }

    return 0;
}
```

Step 3 创建处理初始化和设置后台线程的终端包装器对象

创建处理初始化和设置后台线程的终端包装器对象。

为了在后台进行网络处理时处理用户输入，我们将为WebSocket++处理循环使用一个单独的线程。这使得主线程可以自由处理前台用户输入。为了为我们的线程和终端启用简单的RAII风格的资源管理，我们将使用包装器对象，在其构造函数中配置它们。

> **术语：websocketpp::lib namespace websocketpp::lib 命名空间**
>
> WebSocket++被设计成与c++ 11标准库一起使用。由于这在流行的构建系统中并不是普遍可用的，Boost库可以作为c++ 98构建环境中c++ 11标准库的补充。 `websocketpp::lib` 命名空间由库及其相

> 关示例使用，以抽象两者之间的区别。 `websocketpp::lib::shared_ptr` 在c++ 11环境中将被计算为 `std::shared_ptr` ， 否则将被计算为 `boost::shared_ptr` 。
>
> 本教程使用 `websocketpp::lib` 包装器，因为它不知道读者的构建环境是什么。对于您的应用程序，除非您对类似的可移植性感兴趣，否则可以直接使用这些类型的boost或std版本。

[TODO: link to more information about logging]

```
//清除访问日志
m_endpoint.clear_access_channels(websocketpp::log::alevel::all);
//清除错误日志
m_endpoint.clear_error_channels(websocketpp::log::elevel::all);
```

接下来，我们初始化终端基础的传输系统，并将其设置为永久模式。在永久模式下，终端的处理循环在没有连接时不会自动退出。这很重要，因为我们希望这个终端在我们的应用程序运行时保持活动状态，并在我们需要时按需处理新WebSocket连接的请求。这两种方法都是asio传输所特有的。它们在使用非asio配置的终端中是不必要的，也不存在的。

```
//初始化 基于asio的传输系统
m_endpoint.init_asio();
//以 永久模式 启动
m_endpoint.start_perpetual();
```

最后，我们启动一个线程来运行客户端终端的 `run` 方法。当终端运行时，它将处理连接任务(读取和传递传入消息、帧和发送传出消息等)。因为它是在永久模式下运行的，当没有活动的连接时，它将一直等待下一个新的连接。

```
//启动一个线程来运行run方法
m_thread.reset(new websocketpp::lib::thread(&client::run, &m_endpoint));
```

Build

现在我们的客户端终端模板已经实际实例化了，将会出现更多的连接器依赖项。特别是，WebSocket客户端需要一个加密安全的随机数生成器。WebSocket++可以使用 `boost_random` 或c++ 11标准库 `<random>` 来实现这个目的。 `<random>` 因为这个例子也使用了线程，如果我们没有c++ 11 `std::thread` 可用，我们将需要包含 `boost_thread` 。

Clang (C++98 & boost)

使用clang++来编译 `step2.cpp`

```
clang++ step3.cpp -lboost_system -lboost_random -lboost_thread
```

Clang (C++11)

```
clang++ -std=c++0x -stdlib=libc++ step3.cpp -lboost_system -D_WEBSOCKETPP_CPP11_STL_
```

G++ (C++98 & Boost)

```
g++ step3.cpp -lboost_system -lboost_random -lboost_thread
```

G++ v4.6+ (C++11)

```
g++ -std=c++0x step3.cpp -lboost_system -D_WEBSOCKETPP_CPP11_STL_
```

Code so far

目前的代码

```cpp
#include <websocketpp/config/asio_no_tls_client.hpp>
#include <websocketpp/client.hpp>

#include <websocketpp/common/thread.hpp>
#include <websocketpp/common/memory.hpp>

#include <iostream>
#include <string>

typedef websocketpp::client<websocketpp::config::asio_client> client;

class websocket_endpoint {
public:
    websocket_endpoint () {
        m_endpoint.clear_access_channels(websocketpp::log::alevel::all);
        m_endpoint.clear_error_channels(websocketpp::log::elevel::all);

        m_endpoint.init_asio();
        m_endpoint.start_perpetual();

        m_thread.reset(new websocketpp::lib::thread(&client::run, &m_endpoint));
    }
private:
    client m_endpoint;
    websocketpp::lib::shared_ptr<websocketpp::lib::thread> m_thread;
};

int main() {
    bool done = false;
    std::string input;
    websocket_endpoint endpoint;

    while (!done) {
        std::cout << "Enter Command: ";
        std::getline(std::cin, input);

        if (input == "quit") {
            done = true;
        } else if (input == "help") {
            std::cout
                << "\nCommand List:\n"
                << "help: Display this help text\n"
                << "quit: Exit the program\n"
                << std::endl;
        } else {
            std::cout << "Unrecognized Command" << std::endl;
        }
    }

    return 0;
}
```

Step 4 打开websocket连接

打开websocket连接

这个步骤向utility_client添加了两个新命令。令其有了打开新连接的能力和查看关于以前打开的连接的信息的能力。每个打开的连接都将被分配一个整数连接id，程序的用户可以使用该id与该连接进行交互。

New Connection Metadata Object 新建连接元数据对象

为了跟踪关于每个连接的信息，定义了一个 `connection_metadata` 对象。该对象存储数字连接id和一些将在处理连接时填充的字段。最初，这包括连接的状态(打开、打开、失败、关闭等)、连接到的原始URI、来自服务器的标识值，以及连接失败/关闭原因的描述。后续步骤将向此元数据对象添加更多信息。

Update `websocket_endpoint`

更新了 `websocket_endpoint`

`websocket_endpoint` 对象获得了一些新的数据成员和方法。它现在跟踪连接ID和关联元数据之间的映射，以及要分发的下一个顺序ID号。 `connect()` 方法启动一个新的连接。 `get_metadata` 方法获取给定ID的元数据。

The connect method `connect()` 方法

一个新的WebSocket连接是通过三个步骤启动的。首先，连接请求由 `endpoint::get_connection(uri)` 创建。接下来，配置连接请求。最后，通过 `endpoint::connect()` 将连接请求提交回终端，将其添加到要建立的新连接队列中。

> ### Terminology `connection_ptr`
>
> WebSocket++使用一个引用计数的共享指针来跟踪与连接相关的资源。这个指针的类型是 `endpoint::connection_ptr` 。 `connection_ptr` 允许直接访问有关连接的信息，并允许更改连接设置。由于这种直接访问和它们在库中的内部资源管理角色，终端应用程序使用 `connection_ptr` 是不安全的，除非在下面详细说明的特定情况下。
>
> #### 何时使用 `connection_ptr` 是安全的?
>
> - 在 `endpoint::get_connection(…)` 之后和 `endpoint::connect()` 之前:
>   `get_connection` 返回一个 `connection_ptr` 。使用此指针配置新连接是安全的。一旦你提交连接到 `connect` ，你可能不再使用 `connection_ptr` ，应该立即丢弃它，以优化内存管理。
> - 在处理程序期间:
>   WebSocket++允许你为在连接的生命周期内发生的特定事件注册钩子/回调/事件处理程序。在

> 调用这些处理程序中的一个时，库保证对与当前运行的处理程序相关联的连接使用 `connection_ptr` 是安全的。

> **术语 `connection_hdl`**
>
> 由于 `connection_ptr` 的线程安全性有限，库还提供了一个更灵活的连接标识符 `connection_hdl` 。 `connection_hdl` 有类型 `websocketpp::connection_hdl` ，它在 `<websocketpp/common/connection_hdl.hpp>` 中定义。注意，与 `connection_ptr` 不同，它不依赖于终端的类型或配置。简单地存储或传输 `connection_hdl` 但不使用它们的代码可以只包含上面的头文件，并可以把它的hdl当作值。

Utility Client Example Application Tutorial

Chapter 1: Initial Setup & Basics

Setting up the basic types, opening and closing connections, sending and receiving messages.

Step 1 一个基本的循环程序

A basic program loop that prompts the user for a command and then processes it. In this tutorial we will modify this program to perform tasks and retrieve data from a remote server over a WebSocket connection.

Build

使用clang++来编译 `step1.cpp` ：

```
clang++ step1.cpp
```

Code so far

note A code snapshot for each step is present next to this tutorial file in the git repository.
注意：在git存储库中，本教程文件旁边有每个步骤的代码快照。

假设我们目前只写了这么几行代码：

```cpp
#include <iostream>
#include <string>

int main() {
    bool done = false;
    std::string input;

    while (!done) {
        std::cout << "Enter Command: ";
        std::getline(std::cin, input);

        if (input == "quit") {
            done = true;
        } else if (input == "help") {
            std::cout
                << "\nCommand List:\n"
                << "help: Display this help text\n"
                << "quit: Exit the program\n"
                << std::endl;
        } else {
            std::cout << "Unrecognized Command" << std::endl;
        }
    }

    return 0;
}
```

可见这仅仅是一个简单的循环程序。

Step 2 添加WebSocket++包含文件（头文件）并设置终端类型

Add WebSocket++ includes and set up an endpoint type.

添加WebSocket++包含文件（头文件）并设置终端类型。

WebSocket++ includes two major object types. The endpoint and the connection. The endpoint creates and launches new connections and maintains default settings for those connections. Endpoints also manage any shared network resources.

WebSocket++包括两种主要的对象类型。终端和连接。终端创建并启动新的连接，并维护这些连接的默认设置。终端还管理任何共享的网络资源。

The connection stores information specific to each WebSocket session.

连接存储每个WebSocket会话特定的信息。

> **Note:** Once a connection is launched, there is no link between the endpoint and the connection. All default settings are copied into the new connection by the endpoint. Changing default settings on an endpoint will only affect future connections.
> Connections do not maintain a link back to their associated endpoint. Endpoints do not maintain a list of outstanding connections. If your application needs to iterate over all connections it will need to maintain a list of them itself.

WebSocket++ endpoints are built by combining an endpoint role with an endpoint config. There are two different types of endpoint roles, one each for the client and server roles in a WebSocket session. This is a client tutorial so we will use the client role `websocketpp::client` which is provided by the `<websocketpp/client.hpp>` header.

> **Terminology: Endpoint Config**
>
> WebSocket++ endpoints have a group of settings that may be configured at compile time via the `config` template parameter. A config is a struct that contains types and static constants that are used to produce an endpoint with specific properties. Depending on which config is being used the endpoint will have different methods available and may have additional third party dependencies.

The endpoint role takes a template parameter called `config` that is used to configure the behavior of endpoint at compile time. For this example we are going to use a default config provided by the library called `asio_client`, provided by `<websocketpp/config/asio_no_tls_client.hpp>`. This is a client config that uses boost::asio to provide network transport and does not support TLS based security. Later on we will discuss how to introduce TLS based security into a WebSocket++ application, more about the other stock configs, and how to build your own custom configs.

Combine a config with an endpoint role to produce a fully configured endpoint. This type will be used frequently so I would recommend a typedef here.

```
typedef websocketpp::client<websocketpp::config::asio_client> client
```

Build

Adding WebSocket++ has added a few dependencies to our program that must be addressed in the build system. Firstly, the WebSocket++ and Boost library headers must be in the include search path of your build system. How exactly this is done depends on where you have the WebSocket++ headers installed and what build system you are using.

In addition to the new headers, boost::asio depends on the `boost_system` shared library. This will need to be added (either as a static or dynamic) to the linker. Refer to your build environment documentation for instructions on linking to shared libraries.

除了新的头文件，boost::asio还依赖于 `boost_system` 共享库。这需要添加(静态或动态)到链接器。有关链接到共享库的说明，请参阅构建环境文档。

使用clang++来编译 `step2.cpp` ：

```
clang++_step2.cpp -lboost_system
```

Code so far

现在的代码为：

```
//添加了websocketpp的依赖 头文件
#include <websocketpp/config/asio_no_tls_client.hpp>
#include <websocketpp/client.hpp>

#include <iostream>
#include <string>

//定义了终端类型 client
typedef websocketpp::client<websocketpp::config::asio_client> client;

int main() {
    bool done = false;
    std::string input;

    while (!done) {
        std::cout << "Enter Command: ";
        std::getline(std::cin, input);

        if (input == "quit") {
            done = true;
        } else if (input == "help") {
            std::cout
                << "\nCommand List:\n"
                << "help: Display this help text\n"
                << "quit: Exit the program\n"
                << std::endl;
        } else {
            std::cout << "Unrecognized Command" << std::endl;
        }
    }

    return 0;
}
```

Step 3 创建处理初始化和设置后台线程的终端包装器对象

Create endpoint wrapper object that handles initialization and setting up the background thread.

创建处理初始化和设置后台线程的终端包装器对象。

In order to process user input while network processing occurs in the background we are going to use a separate thread for the WebSocket++ processing loop. This leaves the main thread free to process foreground user input. In order to enable simple RAII style resource management for our thread and endpoint we will use a wrapper object that configures them both in its constructor.

> **Terminology: websocketpp::lib namespace**
>
> WebSocket++ is designed to be used with a C++11 standard library. As this is not universally available in popular build systems the Boost libraries may be used as polyfills for the C++11 standard library in C++98 build environments. The `websocketpp::lib` namespace is used by the library and its associated examples to abstract away the distinctions between the two. `websocketpp::lib::shared_ptr` will evaluate to `std::shared_ptr` in a C++11 environment and `boost::shared_ptr` otherwise.
>
> This tutorial uses the `websocketpp::lib` wrappers because it doesn't know what the build environment of the reader is. For your applications, unless you are interested in similar portability, are free to use the boost or std versions of these types directly.
>
> [TODO: link to more information about websocketpp::lib namespace and C++11 setup]

Within the `websocket_endpoint` constructor several things happen:

在 `websocket_endpoint` 构造函数中会发生以下几件事:

First, we set the endpoint logging behavior to silent by clearing all of the access and error logging channels.
首先，我们通过清除所有访问和错误日志通道将终端日志记录行为设置为silent（即：什么信息都没有）。

[TODO: link to more information about logging]

```
//清除访问日志
m_endpoint.clear_access_channels(websocketpp::log::alevel::all);
//清除错误日志
m_endpoint.clear_error_channels(websocketpp::log::elevel::all);
```

Next, we initialize the transport system underlying the endpoint and set it to perpetual mode. In perpetual mode the endpoint's processing loop will not exit automatically when it has no connections. This is important because we want this endpoint to remain active while our application is running and

process requests for new WebSocket connections on demand as we need them. Both of these methods are specific to the asio transport. They will not be necessary or present in endpoints that use a non-asio config.

```
//初始化 基于asio的传输系统
m_endpoint.init_asio();
//以 永久模式 启动
m_endpoint.start_perpetual();
```

Finally, we launch a thread to run the `run` method of our client endpoint. While the endpoint is running it will process connection tasks (read and deliver incoming messages, frame and send outgoing messages, etc). Because it is running in perpetual mode, when there are no connections active it will wait for a new connection.

```
//启动一个线程来运行run方法
m_thread.reset(new websocketpp::lib::thread(&client::run, &m_endpoint));
```

Build

Now that our client endpoint template is actually instantiated a few more linker dependencies will show up. In particular, WebSocket clients require a cryptographically secure random number generator. WebSocket++ is able to use either `boost_random` or the C++11 standard library for this purpose. Because this example also uses threads, if we do not have C++11 std::thread available we will need to include `boost_thread` .

Clang (C++98 & boost)

使用clang++来编译 `step2.cpp`

```
clang++ step3.cpp -lboost_system -lboost_random -lboost_thread
```

Clang (C++11)

```
clang++ -std=c++0x -stdlib=libc++ step3.cpp -lboost_system -D_WEBSOCKETPP_CPP11_STL_
```

G++ (C++98 & Boost)

```
g++ step3.cpp -lboost_system -lboost_random -lboost_thread
```

G++ v4.6+ (C++11)

```
g++ -std=c++0x step3.cpp -lboost_system -D_WEBSOCKETPP_CPP11_STL_
```

Code so far

目前的代码

```cpp
#include <websocketpp/config/asio_no_tls_client.hpp>
#include <websocketpp/client.hpp>

#include <websocketpp/common/thread.hpp>
#include <websocketpp/common/memory.hpp>

#include <iostream>
#include <string>

typedef websocketpp::client<websocketpp::config::asio_client> client;

class websocket_endpoint {
public:
    websocket_endpoint () {
        m_endpoint.clear_access_channels(websocketpp::log::alevel::all);
        m_endpoint.clear_error_channels(websocketpp::log::elevel::all);

        m_endpoint.init_asio();
        m_endpoint.start_perpetual();

        m_thread.reset(new websocketpp::lib::thread(&client::run, &m_endpoint));
    }
private:
    client m_endpoint;
    websocketpp::lib::shared_ptr<websocketpp::lib::thread> m_thread;
};

int main() {
    bool done = false;
    std::string input;
    websocket_endpoint endpoint;

    while (!done) {
        std::cout << "Enter Command: ";
        std::getline(std::cin, input);

        if (input == "quit") {
            done = true;
        } else if (input == "help") {
            std::cout
                << "\nCommand List:\n"
                << "help: Display this help text\n"
                << "quit: Exit the program\n"
                << std::endl;
        } else {
            std::cout << "Unrecognized Command" << std::endl;
        }
    }

    return 0;
}
```

Step 4 打开websocket连接

Opening WebSocket connections

This step adds two new commands to utility_client. The ability to open a new connection and the ability to view information about a previously opened connection. Every connection that gets opened will be assigned an integer connection id that the user of the program can use to interact with that connection.

New Connection Metadata Object 新建连接元数据对象

In order to track information about each connection a `connection_metadata` object is defined. This object stores the numeric connection id and a number of fields that will be filled in as the connection is processed. Initially this includes the state of the connection (opening, open, failed, closed, etc), the original URI connected to, an identifying value from the server, and a description of the reason for connection failure/closure. Future steps will add more information to this metadata object.

Update `websocket_endpoint`

The `websocket_endpoint` object has gained some new data members and methods. It now tracks a mapping between connection IDs and their associated metadata as well as the next sequential ID number to hand out. The `connect()` method initiates a new connection. The `get_metadata` method retrieves metadata given an ID.

The connect method

A new WebSocket connection is initiated via a three step process. First, a connection request is created by `endpoint::get_connection(uri)` . Next, the connection request is configured. Lastly, the connection request is submitted back to the endpoint via `endpoint::connect()` which adds it to the queue of new connections to make.

> **Terminology `connection_ptr`**
>
> WebSocket++ keeps track of connection related resources using a reference counted shared pointer. The type of this pointer is `endpoint::connection_ptr` . A `connection_ptr` allows direct access to information about the connection and allows changing connection settings. Because of this direct access and their internal resource management role within the library it is not safe for end applications to use `connection_ptr` except in the specific circumstances detailed below.
>
> **When is it safe to use `connection_ptr` ?**

- After `endpoint::get_connection(...)` and before `endpoint::connect()`: `get_connection` returns a `connection_ptr`. It is safe to use this pointer to configure your new connection. Once you submit the connection to `connect` you may no longer use the `connection_ptr` and should discard it immediately for optimal memory management.
- During a handler: WebSocket++ allows you to register hooks / callbacks / event handlers for specific events that happen during a connection's lifetime. During the invocation of one of these handlers the library guarantees that it is safe to use a `connection_ptr` for the connection associated with the currently running handler.

Terminology `connection_hdl` (翻译到这 21-10-14)

Because of the limited thread safety of the `connection_ptr` the library also provides a more flexible connection identifier, the `connection_hdl`. The `connection_hdl` has type `websocketpp::connection_hdl` and it is defined in `<websocketpp/common/connection_hdl.hpp>`. Note that unlike `connection_ptr` this is not dependent on the type or config of the endpoint. Code that simply stores or transmits `connection_hdl` but does not use them can include only the header above and can treat its hdls like values.

Connection handles are not used directly. They are used by endpoint methods to identify the target of the desired action. For example, the endpoint method that sends a new message will take as a parameter the hdl of the connection to send the message to.

When is it safe to use `connection_hdl`?

`connection_hdl`s may be used at any time from any thread. They may be copied and stored in containers. Deleting a hdl will not affect the connection in any way. Handles may be upgraded to a `connection_ptr` during a handler call by using `endpoint::get_con_from_hdl()`. The resulting `connection_ptr` is safe to use for the duration of that handler invocation.

`connection_hdl` FAQs

- `connection_hdl`s are guaranteed to be unique within a program. Multiple endpoints in a single program will always create connections with unique handles.
- Using a `connection_hdl` with a different endpoint than the one that created its associated connection will result in undefined behavior.
- Using a `connection_hdl` whose associated connection has been closed or deleted is safe. The endpoint will return a specific error saying the operation couldn't be completed because the associated connection doesn't exist.
  [TODO: more here? link to a connection_hdl FAQ elsewhere?]

`websocket_endpoint::connect()` begins by calling `endpoint::get_connection()` using a uri passed as a parameter. Additionally, an error output value is passed to capture any errors that might occur

during. If an error does occur an error notice is printed along with a descriptive message and the -1 / 'invalid' value is returned as the new ID.

> **Terminology:** `error handling: exceptions vs error_code`
>
> WebSocket++ uses the error code system defined by the C++11 `<system_error>` library. It can optionally fall back to a similar system provided by the Boost libraries. All user facing endpoint methods that can fail take an `error_code` in an output parameter and store the error that occured there before returning. An empty/default constructed value is returned in the case of success.
>
> **Exception throwing varients**
> All user facing endpoint methods that take and use an `error_code` parameter have a version that throws an exception instead. These methods are identical in function and signature except for the lack of the final ec parameter. The type of the exception thrown is `websocketpp::exception` . This type derives from `std::exception` so it can be caught by catch blocks grabbing generic `std::exception` s. The `websocketpp::exception::code()` method may be used to extract the machine readable `error_code` value from an exception.
>
> For clarity about error handling the utility_client example uses exclusively the exception free varients of these methods. Your application may choose to use either.

If connection creation succeeds, the next sequential connection ID is generated and a `connection_metadata` object is inserted into the connection list under that ID. Initially the metadata object stores the connection ID, the `connection_hdl` , and the URI the connection was opened to.

```
int new_id = m_next_id++;
metadata_ptr metadata(new connection_metadata(new_id, con->get_handle(), uri));
m_connection_list[new_id] = metadata;
```

Next, the connection request is configured. For this step the only configuration we will do is setting up a few default handlers. Later on we will return and demonstrate some more detailed configuration that can happen here (setting user agents, origin, proxies, custom headers, subprotocols, etc).

> **Terminology: Registering handlers**
>
> WebSocket++ provides a number of execution points where you can register to have a handler run. Which of these points are available to your endpoint will depend on its config. TLS handlers will not exist on non-TLS endpoints for example. A complete list of handlers can be found at http://www.zaphoyd.com/websocketpp/manual/reference/handler-list.
>
> Handlers can be registered at the endpoint level and at the connection level. Endpoint handlers are copied into new connections as they are created. Changing an endpoint handler will affect

> only future connections. Handlers registered at the connection level will be bound to that specific connection only.
>
> The signature of handler binding methods is the same for endpoints and connections. The format is: `set_*_handler(...)`. Where * is the name of the handler. For example, `set_open_handler(...)` will set the handler to be called when a new connection is open. `set_fail_handler(...)` will set the handler to be called when a connection fails to connect.
>
> All handlers take one argument, a callable type that can be converted to a `std::function` with the correct count and type of arguments. You can pass free functions, functors, and Lambdas with matching argument lists as handlers. In addition, you can use `std::bind` (or `boost::bind`) to register functions with non-matching argument lists. This is useful for passing additional parameters not present in the handler signature or member functions that need to carry a 'this' pointer.
>
> The function signature of each handler can be looked up in the list above in the manual. In general, all handlers include the `connection_hdl` identifying which connection this even is associated with as the first parameter. Some handlers (such as the message handler) include additional parameters. Most handlers have a void return value but some (`validate`, `ping`, `tls_init`) do not. The specific meanings of the return values are documented in the handler list linked above.

`utility_client` registers an open and a fail handler. We will use these to track whether each connection was successfully opened or failed. If it successfully opens, we will gather some information from the opening handshake and store it with our connection metadata.

In this example we are going to set connection specific handlers that are bound directly to the metadata object associated with our connection. This allows us to avoid performing a lookup in each handler to find the metadata object we plan to update which is a bit more efficient.

Lets look at the parameters being sent to bind in detail:

```
con->set_open_handler(websocketpp::lib::bind(
    &connection_metadata::on_open,
    metadata,
    &m_endpoint,
    websocketpp::lib::placeholders::_1
));
```

`&connection_metadata::on_open` is the address of the `on_open` member function of the `connection_metadata` class. `metadata_ptr` is a pointer to the `connection_metadata` object associated with this class. It will be used as the object on which the `on_open` member function will be called.

`&m_endpoint` is the address of the endpoint in use. This parameter will be passed as-is to the `on_open` method. Lastly, `websocketpp::lib::placeholders::_1` is a placeholder indicating that the bound function should take one additional argument to be filled in at a later time. WebSocket++ will fill in this placeholder with the `connection_hdl` when it invokes the handler.

Finally, we call `endpoint::connect()` on our configured connection request and return the new connection ID.

Handler Member Functions

The open handler we registered, `connection_metadata::on_open`, sets the status metadata field to "Open" and retrieves the value of the "Server" header from the remote endpoint's HTTP response and stores it in the metadata object. Servers often set an identifying string in this header.

The fail handler we registered, `connection_metadata::on_fail`, sets the status metadata field to "Failed", the server field similarly to `on_open`, and retrieves the error code describing why the connection failed. The human readable message associated with that error code is saved to the metadata object.

New Commands

Two new commands have been set up. "connect [uri]" will pass the URI to the `websocket_endpoint` connect method and report an error or the connection ID of the new connection. "show [connection id]" will retrieve and print out the metadata associated with that connection. The help text has been updated accordingly.

```
} else if (input.substr(0,7) == "connect") {
    int id = endpoint.connect(input.substr(8));
    if (id != -1) {
        std::cout << "> Created connection with id " << id << std::endl;
    }
} else if (input.substr(0,4) == "show") {
    int id = atoi(input.substr(5).c_str());

    connection_metadata::ptr metadata = endpoint.get_metadata(id);
    if (metadata) {
        std::cout << *metadata << std::endl;
    } else {
        std::cout << "> Unknown connection id " << id << std::endl;
    }
}
```

Build 编译

There are no changes to the build instructions from step 3

Run 运行程序

```
Enter Command: connect not a websocket uri
> Connect initialization error: invalid uri
Enter Command: show 0
> Unknown connection id 0
Enter Command: connect ws://echo.websocket.org
> Created connection with id 0
Enter Command: show 0
> URI: ws://echo.websocket.org
> Status: Open
> Remote Server: Kaazing Gateway
> Error/close reason: N/A
Enter Command: connect ws://wikipedia.org
> Created connection with id 1
Enter Command: show 1
> URI: ws://wikipedia.org
> Status: Failed
> Remote Server: Apache
> Error/close reason: Invalid HTTP status.
```

Code so far

目前为止的代码

```cpp
#include <websocketpp/config/asio_no_tls_client.hpp>
#include <websocketpp/client.hpp>

//
#include <websocketpp/common/thread.hpp>
#include <websocketpp/common/memory.hpp>

#include <cstdlib>
#include <iostream>
#include <map>
#include <string>
#include <sstream>

typedef websocketpp::client<websocketpp::config::asio_client> client;

//
class connection_metadata {
public:
    typedef websocketpp::lib::shared_ptr<connection_metadata> ptr;

    connection_metadata(int id, websocketpp::connection_hdl hdl, std::string uri)
      : m_id(id)
      , m_hdl(hdl)
      , m_status("Connecting")
      , m_uri(uri)
      , m_server("N/A")
    {}

    void on_open(client * c, websocketpp::connection_hdl hdl) {
        m_status = "Open";

        client::connection_ptr con = c->get_con_from_hdl(hdl);
        m_server = con->get_response_header("Server");
    }

    void on_fail(client * c, websocketpp::connection_hdl hdl) {
        m_status = "Failed";

        client::connection_ptr con = c->get_con_from_hdl(hdl);
        m_server = con->get_response_header("Server");
        m_error_reason = con->get_ec().message();
    }

    friend std::ostream & operator<< (std::ostream & out, connection_metadata const & data);
private:
    int m_id;
    websocketpp::connection_hdl m_hdl;
    std::string m_status;
    std::string m_uri;
    std::string m_server;
    std::string m_error_reason;
```

```
};

std::ostream & operator<< (std::ostream & out, connection_metadata const & data) {
    out << "> URI: " << data.m_uri << "\n"
        << "> Status: " << data.m_status << "\n"
        << "> Remote Server: " << (data.m_server.empty() ? "None Specified" : data.m_server) <<
        << "> Error/close reason: " << (data.m_error_reason.empty() ? "N/A" : data.m_error_reasc

    return out;
}

class websocket_endpoint {
public:
    websocket_endpoint () : m_next_id(0) {
        m_endpoint.clear_access_channels(websocketpp::log::alevel::all);
        m_endpoint.clear_error_channels(websocketpp::log::elevel::all);

        m_endpoint.init_asio();
        m_endpoint.start_perpetual();

        m_thread.reset(new websocketpp::lib::thread(&client::run, &m_endpoint));
    }

    int connect(std::string const & uri) {
        websocketpp::lib::error_code ec;

        client::connection_ptr con = m_endpoint.get_connection(uri, ec);

        if (ec) {
            std::cout << "> Connect initialization error: " << ec.message() << std::endl;
            return -1;
        }

        int new_id = m_next_id++;
        connection_metadata::ptr metadata_ptr(new connection_metadata(new_id, con->get_handle(),
        m_connection_list[new_id] = metadata_ptr;

        con->set_open_handler(websocketpp::lib::bind(
            &connection_metadata::on_open,
            metadata_ptr,
            &m_endpoint,
            websocketpp::lib::placeholders::_1
        ));
        con->set_fail_handler(websocketpp::lib::bind(
            &connection_metadata::on_fail,
            metadata_ptr,
            &m_endpoint,
            websocketpp::lib::placeholders::_1
        ));

        m_endpoint.connect(con);
```

```
        return new_id;
    }

    connection_metadata::ptr get_metadata(int id) const {
        con_list::const_iterator metadata_it = m_connection_list.find(id);
        if (metadata_it == m_connection_list.end()) {
            return connection_metadata::ptr();
        } else {
            return metadata_it->second;
        }
    }
private:
    typedef std::map<int,connection_metadata::ptr> con_list;

    client m_endpoint;
    websocketpp::lib::shared_ptr<websocketpp::lib::thread> m_thread;

    con_list m_connection_list;
    int m_next_id;
};

int main() {
    bool done = false;
    std::string input;
    websocket_endpoint endpoint;

    while (!done) {
        std::cout << "Enter Command: ";
        std::getline(std::cin, input);

        if (input == "quit") {
            done = true;
        } else if (input == "help") {
            std::cout
                << "\nCommand List:\n"
                << "connect <ws uri>\n"
                << "show <connection id>\n"
                << "help: Display this help text\n"
                << "quit: Exit the program\n"
                << std::endl;
        } else if (input.substr(0,7) == "connect") {
            int id = endpoint.connect(input.substr(8));
            if (id != -1) {
                std::cout << "> Created connection with id " << id << std::endl;
            }
        } else if (input.substr(0,4) == "show") {
            int id = atoi(input.substr(5).c_str());

            connection_metadata::ptr metadata = endpoint.get_metadata(id);
            if (metadata) {
```

```
                std::cout << *metadata << std::endl;
            } else {
                std::cout << "> Unknown connection id " << id << std::endl;
            }
        } else {
            std::cout << "> Unrecognized Command" << std::endl;
        }
    }

    return 0;
}
```

Step 5 关闭连接

Closing connections

This step adds a command that allows you to close a WebSocket connection and adjusts the quit command so that it cleanly closes all outstanding connections before quitting.

Getting connection close information out of WebSocket++

> **Terminology: WebSocket close codes & reasons**
>
> The WebSocket close handshake involves an exchange of optional machine readable close codes and human readable reason strings. Each endpoint sends independent close details. The codes are short integers. The reasons are UTF8 text strings of at most 125 characters. More details about valid close code ranges and the meaning of each code can be found at https://tools.ietf.org/html/rfc6455#section-7.4

The `websocketpp::close::status` namespace contains named constants for all of the IANA defined close codes. It also includes free functions to determine whether a value is reserved or invalid and to convert a code to a human readable text representation.

During the close handler call WebSocket++ connections offer the following methods for accessing close handshake information:

- `connection::get_remote_close_code()` : Get the close code as reported by the remote endpoint
- `connection::get_remote_close_reason()` : Get the close reason as reported by the remote endpoint
- `connection::get_local_close_code()` : Get the close code that this endpoint sent.
- `connection::get_local_close_reason()` : Get the close reason that this endpoint sent.
- `connection::get_ec()` : Get a more detailed/specific WebSocket++ `error_code` indicating what library error (if any) ultimately resulted in the connection closure.

Note: there are some special close codes that will report a code that was not actually sent on the wire. For example 1005/"no close code" indicates that the endpoint omitted a close code entirely and 1006/"abnormal close" indicates that there was a problem that resulted in the connection closing without having performed a close handshake.

Add close handler

The `connection_metadata::on_close` method is added. This method retrieves the close code and reason from the closing handshake and stores it in the local error reason field.

```
void on_close(client * c, websocketpp::connection_hdl hdl) {
    m_status = "Closed";
    client::connection_ptr con = c->get_con_from_hdl(hdl);
    std::stringstream s;
    s << "close code: " << con->get_remote_close_code() << " ("
      << websocketpp::close::status::get_string(con->get_remote_close_code())
      << "), close reason: " << con->get_remote_close_reason();
    m_error_reason = s.str();
}
```

Similarly to `on_open` and `on_fail`, `websocket_endpoint::connect` registers this close handler when a new connection is made.

Add close method to `websocket_endpoint`

This method starts by looking up the given connection ID in the connection list. Next a close request is sent to the connection's handle with the specified WebSocket close code. This is done by calling `endpoint::close`. This is a thread safe method that is used to asynchronously dispatch a close signal to the connection with the given handle. When the operation is complete the connection's close handler will be triggered.

```
void close(int id, websocketpp::close::status::value code) {
    websocketpp::lib::error_code ec;

    con_list::iterator metadata_it = m_connection_list.find(id);
    if (metadata_it == m_connection_list.end()) {
        std::cout << "> No connection found with id " << id << std::endl;
        return;
    }

    m_endpoint.close(metadata_it->second->get_hdl(), code, "", ec);
    if (ec) {
        std::cout << "> Error initiating close: " << ec.message() << std::endl;
    }
}
```

Add close option to the command loop and help message

A close option is added to the command loop. It takes a connection ID and optionally a close code and a close reason. If no code is specified the default of 1000/Normal is used. If no reason is specified, none is sent. The `endpoint::close` method will do some error checking and abort the close request if you try and send an invalid code or a reason with invalid UTF8 formatting. Reason strings longer than 125 characters will be truncated.

An entry is also added to the help system to describe how the new command may be used.

```
else if (input.substr(0,5) == "close") {
    std::stringstream ss(input);

    std::string cmd;
    int id;
    int close_code = websocketpp::close::status::normal;
    std::string reason;

    ss >> cmd >> id >> close_code;
    std::getline(ss,reason);

    endpoint.close(id, close_code, reason);
}
```

Close all outstanding connections in `websocket_endpoint` destructor

Until now quitting the program left outstanding connections and the WebSocket++ network thread in a lurch. Now that we have a method of closing connections we can clean this up properly.

The destructor for `websocket_endpoint` now stops perpetual mode (so the run thread exits after the last connection is closed) and iterates through the list of open connections and requests a clean close for each. Finally, the run thread is joined which causes the program to wait until those connection closes complete.

```
~websocket_endpoint() {
    m_endpoint.stop_perpetual();

    for (con_list::const_iterator it = m_connection_list.begin(); it != m_connection_list.end();
        if (it->second->get_status() != "Open") {
            // Only close open connections
            continue;
        }

        std::cout << "> Closing connection " << it->second->get_id() << std::endl;

        websocketpp::lib::error_code ec;
        m_endpoint.close(it->second->get_hdl(), websocketpp::close::status::going_away, "", ec);
        if (ec) {
            std::cout << "> Error closing connection " << it->second->get_id() << ": "
                      << ec.message() << std::endl;
        }
    }

    m_thread->join();
}
```

Build

There are no changes to the build instructions from step 4

Run

```
Enter Command: connect ws://localhost:9002
> Created connection with id 0
Enter Command: close 0 1001 example message
Enter Command: show 0
> URI: ws://localhost:9002
> Status: Closed
> Remote Server: WebSocket++/0.4.0
> Error/close reason: close code: 1001 (Going away), close reason:  example message
Enter Command: connect ws://localhost:9002
> Created connection with id 1
Enter Command: close 1 1006
> Error initiating close: Invalid close code used
Enter Command: quit
> Closing connection 1
```

Step 6 收发数据

Sending and receiving messages

This step adds a command to send a message on a given connection and updates the show command to print a transcript of all sent and received messages for that connection.

> **Terminology: WebSocket message types (opcodes)**
>
> WebSocket messages have types indicated by their opcode. The protocol currently specifies two different opcodes for data messages, text and binary. Text messages represent UTF8 text and will be validated as such. Binary messages represent raw binary bytes and are passed through directly with no validation.
>
> WebSocket++ provides the values `websocketpp::frame::opcode::text` and `websocketpp::frame::opcode::binary` that can be used to direct how outgoing messages should be sent and to check how incoming messages are formatted.

Sending Messages

Messages are sent using `endpoint::send`. This is a thread safe method that may be called from anywhere to queue a message for sending on the specified connection. There are three send overloads for use with different scenarios.

Each method takes a `connection_hdl` to indicate which connection to send the message on as well as a `frame::opcode::value` to indicate which opcode to label the message as. All overloads are also available with an exception free varient that fills in a a status/error code instead of throwing.

The first overload, `connection_hdl hdl, std::string const & payload, frame::opcode::value op`, takes a `std::string`. The string contents are copied into an internal buffer and can be safely modified after calling send.

The second overload, `connection_hdl hdl, void const * payload, size_t len, frame::opcode::value op`, takes a void * buffer and length. The buffer contents are copied and can be safely modified after calling send.

The third overload, `connection_hdl hdl, message_ptr msg`, takes a WebSocket++ `message_ptr`. This overload allows a message to be constructed in place before the call to send. It also may allow a single message buffer to be sent multiple times, including to multiple connections, without copying. Whether or not this actually happens depends on other factors such as whether compression is enabled. The contents of the message buffer may not be safely modified after being sent.

> **Terminology: Outgoing WebSocket message queueing & flow control**
>
> In many configurations, such as when the Asio based transport is in use, WebSocket++ is an asynchronous system. As such the `endpoint::send` method may return before any bytes are actually written to the outgoing socket. In cases where send is called multiple times in quick

> succession messages may be coalesced and sent in the same operation or even the same TCP packet. When this happens the message boundaries are preserved (each call to send will produce a separate message).
>
> In the case of applications that call send from inside a handler this means that no messages will be written to the socket until that handler returns. If you are planning to send many messages in this manor or need a message to be written on the wire before continuing you should look into using multiple threads or the built in timer/interrupt handler functionality.
>
> If the outgoing socket link is slow messages may build up in this queue. You can use `connection::get_buffered_amount` to query the current size of the written message queue to decide if you want to change your sending behavior.

Add send method to `websocket_endpoint`

Like the close method, send will start by looking up the given connection ID in the connection list. Next a send request is sent to the connection's handle with the specified WebSocket message and the text opcode. Finally, we record the sent message with our connection metadata object so later our show connection command can print a list of messages sent.

```
void send(int id, std::string message) {
    websocketpp::lib::error_code ec;

    con_list::iterator metadata_it = m_connection_list.find(id);
    if (metadata_it == m_connection_list.end()) {
        std::cout << "> No connection found with id " << id << std::endl;
        return;
    }

    m_endpoint.send(metadata_it->second->get_hdl(), message, websocketpp::frame::opcode::text, e
    if (ec) {
        std::cout << "> Error sending message: " << ec.message() << std::endl;
        return;
    }

    metadata_it->second->record_sent_message(message);
}
```

Add send option to the command loop and help message

A send option is added to the command loop. It takes a connection ID and a text message to send. An entry is also added to the help system to describe how the new command may be used.

```
else if (input.substr(0,4) == "send") {
    std::stringstream ss(input);

        std::string cmd;
        int id;
        std::string message = "";

        ss >> cmd >> id;
        std::getline(ss,message);

        endpoint.send(id, message);
}
```

Add glue to `connection_metadata` for storing sent messages

In order to store messages sent on this connection some code is added to `connection_metadata`. This includes a new data member `std::vector<std::string> m_messages` to keep track of all messages sent and received as well as a method for adding a sent message in that list:

```
void record_sent_message(std::string message) {
    m_messages.push_back(">> " + message);
}
```

Finally the connection metadata output operator is updated to also print a list of processed messages:

```
out << "> Messages Processed: (" << data.m_messages.size() << ") \n";

std::vector<std::string>::const_iterator it;
for (it = data.m_messages.begin(); it != data.m_messages.end(); ++it) {
    out << *it << "\n";
}
```

Receiving Messages

Messages are received by registering a message handler. This handler will be called once per message received and its signature is `void on_message(websocketpp::connection_hdl hdl, endpoint::message_ptr msg)`. The `connection_hdl`, like the similar parameter from the other handlers is a handle for the connection that the message was received on. The `message_ptr` is a pointer to an object that can be queried for the message payload, opcode, and other metadata. Note that the message_ptr type, as well as its underlying message type, is dependent on how your endpoint is configured and may be different for different configs.

Add a message handler to method to `connection_metadata`

The message receiving behave that we are implementing will be to collect all messages sent and received and to print them in order when the show connection command is run. The sent messages are already being added to that list. Now we add a message handler that pushes received messages to the list as well. Text messages are pushed as-is. Binary messages are first converted to printable hexadecimal format.

```
void on_message(websocketpp::connection_hdl hdl, client::message_ptr msg) {
    if (msg->get_opcode() == websocketpp::frame::opcode::text) {
        m_messages.push_back(msg->get_payload());
    } else {
        m_messages.push_back(websocketpp::utility::to_hex(msg->get_payload()));
    }
}
```

In order to have this handler called when new messages are received we also register it with our connection. Note that unlike most other handlers, the message handler has two parameters and thus needs two placeholders.

```
con->set_message_handler(websocketpp::lib::bind(
    &connection_metadata::on_message,
    metadata_ptr,
    websocketpp::lib::placeholders::_1,
    websocketpp::lib::placeholders::_2
));
```

Build

There are no changes to the build instructions from step 5

Run

In this example run we are connecting to the WebSocket++ example echo_server. This server will repeat any message we send back to it. You can also try testing this with the echo server at `ws://echo.websocket.org` with similar results.

```
Enter Command: connect ws://localhost:9002
> Created connection with id 0
Enter Command: send 0 example message
Enter Command: show 0
> URI: ws://localhost:9002
> Status: Open
> Remote Server: WebSocket++/0.4.0
> Error/close reason: N/A
> Messages Processed: (2)
>>  example message
<<  example message
```

Step 7 使用TLS/安全的websocket

Using TLS / Secure WebSockets

- Change the includes
- link to the new library dependencies
- Switch the config
- add the `tls_init_handler`
- configure the SSL context for desired security level
- mixing secure and non-secure connections in one application.

Chapter 2: Intermediate Features 中间特性

Step 8 中级水平的功能

Intermediate level features

- Subprotocol negotiation
- Setting and reading custom headers
- Ping and Pong
- Proxies?
- Setting user agent
- Setting Origin
- Timers and security
- Close behavior
- Send one message to all connections

Misc stuff not sure if it should be included here or elsewhere?

杂项不确定是否应该包括在这里或其他地方?

core websocket++ control flow.
A handshake, followed by a split into 2 independent control strands

- Handshake
  -- use information specified before the call to endpoint::connect to construct a WebSocket handshake request.
  -- Pass the WebSocket handshake request to the transport policy. The transport policy determines how to get these bytes to the endpoint playing the server role. Depending on which transport policy your endpoint uses this method will be different.
  -- Receive a handshake response from the underlying transport. This is parsed and checked for conformance to RFC6455. If the validation fails, the fail handler is called. Otherwise the open handler is called.
- At this point control splits into two separate strands. One that reads new bytes from the transport policy on the incoming channle, the other that accepts new messages from the local application for framing and writing to the outgoing transport channel.
- Read strand
  -- Read and process new bytes from transport
  -- If the bytes contain at least one complete message dispatch each message by calling the appropriate handler. This is either the message handler for data messages, or ping/pong/close handlers for each respective control message. If no handler is registered for a particular message it is ignored.
  -- Ask the transport layer for more bytes
- Write strand
  -- Wait for messages from the application
  -- Perform error checking on message input,
  -- Frame message per RFC6455
  -- Queue message for sending
  -- Pass all outstanding messages to the transport policy for output
  -- When there are no messages left to send, return to waiting

Important observations
Handlers run in line with library processing which has several implications applications should be aware of: