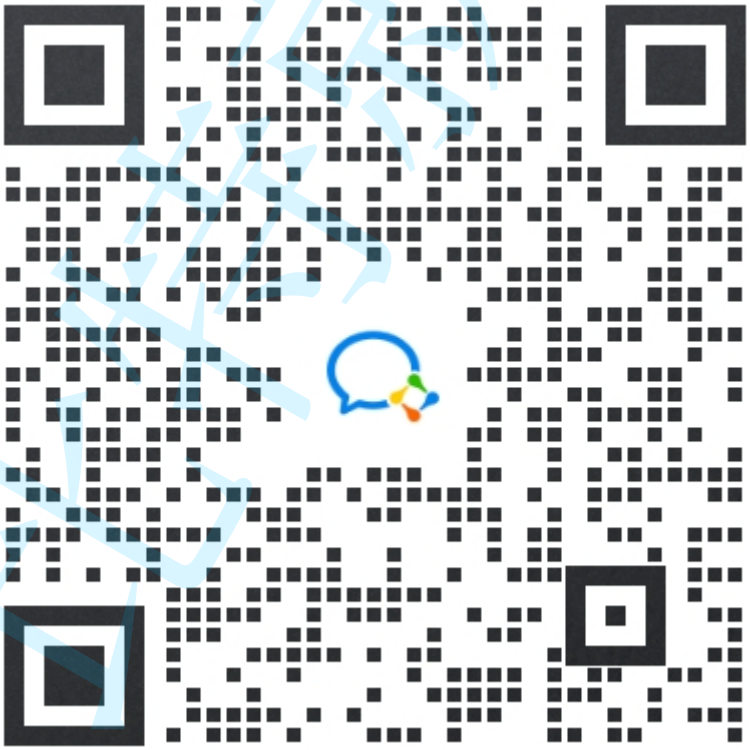
websocketpp 使用

版权说明

对比特项目感兴趣，可以联系这个微信。



代码 & 板书链接

https://gitee.com/bitedu-tech/cpp-chatsystem

安装：

```Bash
sudo apt-get install libboost-dev libboost-system-dev
libwebsocketpp-dev
```

安装完毕后，若在 /usr/include 下有了 websocketpp 目录就表示安装成功了。

```Bash
dev@dev-host:~$ ls /usr/include/websocketpp/
base64      common       connection.hpp       endpoint_base.hpp
frame.hpp  logger
....
```

介绍与使用

Websocket 协议介绍

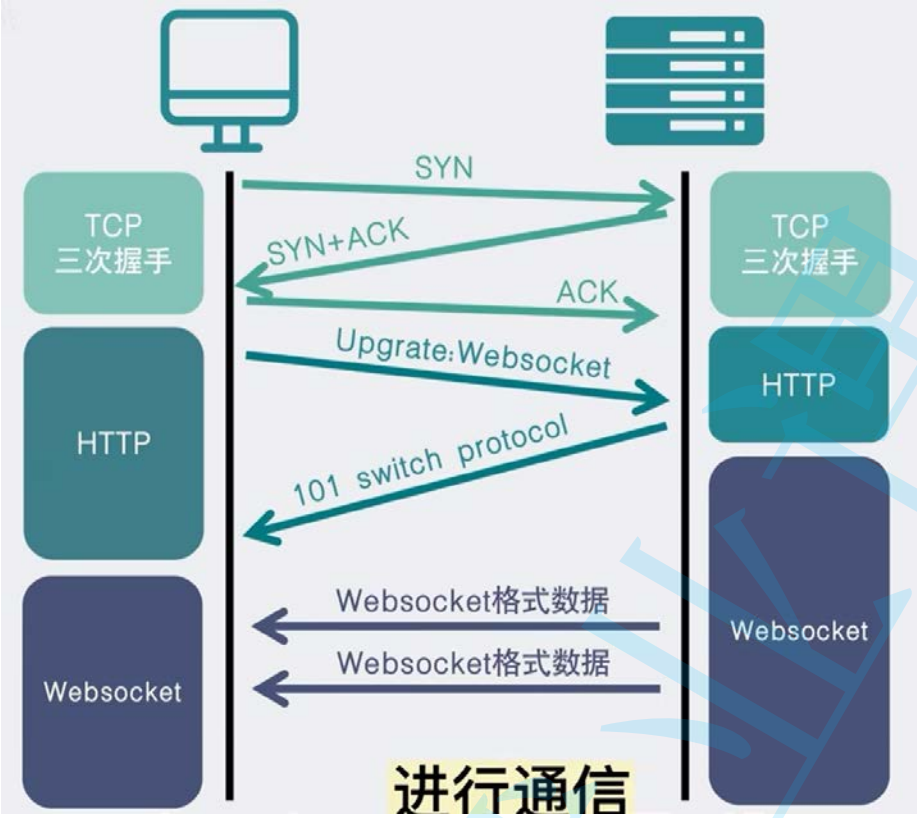
WebSocket 是从 HTML5 开始支持的一种网页端和服务端保持长连接的 消息推送机制。

- 传统的 web 程序都是属于 "一问一答" 的形式，即客户端给服务器发送了一个 HTTP 请求，服务器给客户端返回一个 HTTP 响应。这种情况下服务器是属于被动的一方，如果客户端不主动发起请求服务器就无法主动给客户端响应
- 像网页即时聊天或者我们做的五子棋游戏这样的程序都是非常依赖 "消息推送" 的，即需要服务器主动推动消息到客户端。如果只是使用原生的 HTTP 协议，要想实现消息推送一般需要通过 "轮询" 的方式实现， 而轮询的成本比较高并且也不能及时的获取到消息的响应。

基于上述两个问题， 就产生了 WebSocket 协议。WebSocket 更接近于 TCP 这种级别的通信方式，一旦连接建立完成客户端或者服务器都可以主动的向对方发送数据。

原理解析

WebSocket 协议本质上是一个基于 TCP 的协议。为了建立一个 WebSocket 连接，客户端浏览器首先要向服务器发起一个 HTTP 请求，这个请求和通常的 HTTP 请求不同，包含了一些附加头信息，通过这个附加头信息完成握手过程并升级协议的过程。



具体协议升级的过程如下:

报文格式

报文字段比较多，我们重点关注这几个字段:

- FIN: WebSocket 传输数据以消息为概念单位，一个消息有可能由一个或多个帧组成，FIN 字段为 1 表示末尾帧。
- RSV1~3：保留字段，只在扩展时使用，若未启用扩展则应置 1，若收到不全为 0 的数据帧，且未协商扩展则立即终止连接。
- opcode: 标志当前数据帧的类型
  - 0x0: 表示这是个延续帧，当 opcode 为 0 表示本次数据传输采用了数据分片，当前收到的帧为其中一个分片
  - 0x1: 表示这是文本帧
  - 0x2: 表示这是二进制帧
  - 0x3-0x7: 保留，暂未使用
  - 0x8: 表示连接断开
  - 0x9: 表示 ping 帧
  - 0xa: 表示 pong 帧
  - 0xb-0xf: 保留，暂未使用
- mask：表示 Payload 数据是否被编码，若为 1 则必有 Mask-Key，用于解码 Payload 数据。仅客户端发送给服务端的消息需要设置。
- Payload length：数据载荷的长度，单位是字节， 有可能为 7 位、7+16 位、7+64 位。假设 Payload length = x
  - x 为 0~126：数据的长度为 x 字节
  - x 为 126：后续 2 个字节代表一个 16 位的无符号整数，该无符号整数的值为数据的长度
  - x 为 127：后续 8 个字节代表一个 64 位的无符号整数（最高位为 0），该无符

号整数的值为数据的长度

- Mask-Key：当 mask 为 1 时存在，长度为 4 字节，解码规则： DECODED[i] = ENCODED[i] ^ MASK[i % 4]
- Payload data: 报文携带的载荷数据

Websocketpp 介绍

WebSocketpp 是一个跨平台的开源（BSD 许可证）头部专用 C++库，它实现了 RFC6455（WebSocket 协议）和 RFC7692（WebSocketCompression Extensions）。它允许将 WebSocket 客户端和服务器功能集成到 C++程序中。在最常见的配置中，全功能网络 I/O 由 Asio 网络库提供。

WebSocketpp 的主要特性包括：

- 事件驱动的接口
- 支持 HTTP/HTTPS、WS/WSS、IPv6
- 灵活的依赖管理 — Boost 库/C++11 标准库
- 可移植性：Posix/Windows、32/64bit、Intel/ARM
- 线程安全

WebSocketpp 同时支持 HTTP 和 Websocket 两种网络协议， 比较适用于我们本次的项目， 所以我们选用该库作为项目的依赖库用来搭建 HTTP 和 WebSocket 服务器。

下面是该项目的一些常用网站， 大家多去学习。

- github：https://github.com/zaphoyd/websocketpp
- 用户手册： http://docs.websocketpp.org/
- 官网：http://www.zaphoyd.com/websocketpp

websocketpp 常用接口介绍：

```C++
namespace websocketpp {
    typedef lib::weak_ptr<void> connection_hdl;

    template <typename config>
    class endpoint : public config::socket_type {
        typedef lib::shared_ptr<lib::asio::steady_timer>
timer_ptr;
        typedef typename connection_type::ptr connection_ptr;
        typedef typename connection_type::message_ptr message_ptr;
```

```
        typedef lib::function<void(connection_hdl)> open_handler;
        typedef lib::function<void(connection_hdl)> close_handler;
        typedef lib::function<void(connection_hdl)> http_handler;
        typedef lib::function<void(connection_hdl,message_ptr)>
message_handler;
        /* websocketpp::log::alevel::none  禁止打印所有日志*/
        void set_access_channels(log::level channels);/*设置日志打
印等级*/
        void clear_access_channels(log::level channels);/*清除指定
等级的日志*/
        /*设置指定事件的回调函数*/
        void set_open_handler(open_handler h);/*websocket 握手成功
回调处理函数*/
        void set_close_handler(close_handler h);/*websocket 连接关
闭回调处理函数*/
        void set_message_handler(message_handler h);/*websocket 消
息回调处理函数*/
        void set_http_handler(http_handler h);/*http 请求回调处理函
数*/
        /*发送数据接口*/
        void send(connection_hdl hdl, std::string& payload,
frame::opcode::value op);
        void send(connection_hdl hdl, void* payload, size_t len,
frame::opcode::value op);
        /*关闭连接接口*/
        void close(connection_hdl hdl, close::status::value code,
std::string& reason);
        /*获取 connection_hdl 对应连接的 connection_ptr*/
        connection_ptr get_con_from_hdl(connection_hdl hdl);
        /*websocketpp 基于 asio 框架实现，init_asio 用于初始化 asio 框架
中的 io_service 调度器*/
        void init_asio();
        /*设置是否启用地址重用*/
        void set_reuse_addr(bool value);
        /*设置 endpoint 的绑定监听端口*/
        void listen(uint16_t port);
        /*对 io_service 对象的 run 接口封装，用于启动服务器*/
        std::size_t run();
        /*websocketpp 提供的定时器，以毫秒为单位*/
        timer_ptr set_timer(long duration, timer_handler
callback);
    };
```

```
    template <typename config>
    class server : public endpoint<connection<config>,config> {
        /*初始化并启动服务端监听连接的 accept 事件处理*/
        void start_accept();
    };

    template <typename config>
    class connection
     : public config::transport_type::transport_con_type
     , public config::connection_base
    {
        /*发送数据接口*/
        error_code send(std::string&payload, frame::opcode::value
op=frame::opcode::text);
        /*获取 http 请求头部*/
        std::string const & get_request_header(std::string const &
key)
        /*获取请求正文*/
        std::string const & get_request_body();
        /*设置响应状态码*/
        void set_status(http::status_code::value code);
        /*设置 http 响应正文*/
        void set_body(std::string const & value);
        /*添加 http 响应头部字段*/
        void append_header(std::string const & key, std::string
const & val);
        /*获取 http 请求对象*/
        request_type const & get_request();
        /*获取 connection_ptr 对应的 connection_hdl */
        connection_hdl get_handle();
    };

    namespace http {
    namespace parser {
        class parser {
            std::string const & get_header(std::string const &
key)
            std::string const & get_body()
            typedef std::map<std::string, std::string,
utility::ci_less > header_list;
            header_list const & get_headers()
        }
```

```
        class request : public parser {
            /*获取请求方法*/
            std::string const & get_method()
            /*获取请求 uri 接口*/
            std::string const & get_uri()
        };
    }};

    namespace message_buffer {
        /*获取 websocket 请求中的 payload 数据类型*/
        frame::opcode::value get_opcode();
        /*获取 websocket 中 payload 数据*/
        std::string const & get_payload();
    };

    namespace log {
        struct alevel {
            static level const none = 0x0;
            static level const connect = 0x1;
            static level const disconnect = 0x2;
            static level const control = 0x4;
            static level const frame_header = 0x8;
            static level const frame_payload = 0x10;
            static level const message_header = 0x20;
            static level const message_payload = 0x40;
            static level const endpoint = 0x80;
            static level const debug_handshake = 0x100;
            static level const debug_close = 0x200;
            static level const devel = 0x400;
            static level const app = 0x800;
            static level const http = 0x1000;
            static level const fail = 0x2000;
            static level const access_core = 0x00003003;
            static level const all = 0xffffffff;
        };
    }

    namespace http {
    namespace status_code {
        enum value {
            uninitialized = 0,

            continue_code = 100,
            switching_protocols = 101,
```

```
        ok = 200,
        created = 201,
        accepted = 202,
        non_authoritative_information = 203,
        no_content = 204,
        reset_content = 205,
        partial_content = 206,

        multiple_choices = 300,
        moved_permanently = 301,
        found = 302,
        see_other = 303,
        not_modified = 304,
        use_proxy = 305,
        temporary_redirect = 307,

        bad_request = 400,
        unauthorized = 401,
        payment_required = 402,
        forbidden = 403,
        not_found = 404,
        method_not_allowed = 405,
        not_acceptable = 406,
        proxy_authentication_required = 407,
        request_timeout = 408,
        conflict = 409,
        gone = 410,
        length_required = 411,
        precondition_failed = 412,
        request_entity_too_large = 413,
        request_uri_too_long = 414,
        unsupported_media_type = 415,
        request_range_not_satisfiable = 416,
        expectation_failed = 417,
        im_a_teapot = 418,
        upgrade_required = 426,
        precondition_required = 428,
        too_many_requests = 429,
        request_header_fields_too_large = 431,

        internal_server_error = 500,
        not_implemented = 501,
        bad_gateway = 502,
```

```
            service_unavailable = 503,
            gateway_timeout = 504,
            http_version_not_supported = 505,
            not_extended = 510,
            network_authentication_required = 511
        };}}
        namespace frame {
        namespace opcode {
        enum value {
            continuation = 0x0,
            text = 0x1,
            binary = 0x2,
            rsv3 = 0x3,
            rsv4 = 0x4,
            rsv5 = 0x5,
            rsv6 = 0x6,
            rsv7 = 0x7,
            close = 0x8,
            ping = 0x9,
            pong = 0xA,
            control_rsvb = 0xB,
            control_rsvc = 0xC,
            control_rsvd = 0xD,
            control_rsve = 0xE,
            control_rsvf = 0xF,
        };}}
}
```

Websocketpp 使用

Simple http/websocket 服务器

使用 Websocketpp 实现一个简单的 http 和 websocket 服务器

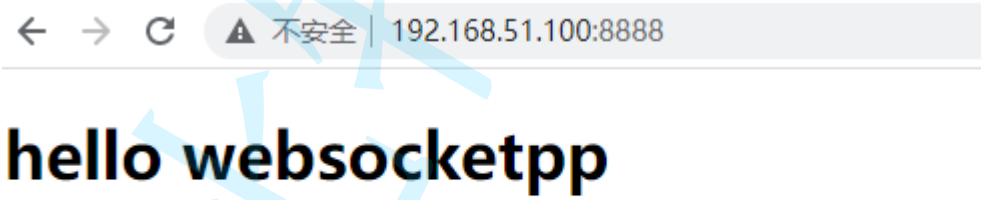
```
C++
#include <iostream>
#include <websocketpp/config/asio_no_tls.hpp>
#include <websocketpp/server.hpp>

using namespace std;

typedef websocketpp::server<websocketpp::config::asio>
websocketsvr;
typedef websocketsvr::message_ptr message_ptr;
```

```
using websocketpp::lib::placeholders::_1;
using websocketpp::lib::placeholders::_2;
using websocketpp::lib::bind;

// websocket 连接成功的回调函数
void OnOpen(websocketsvr *server,websocketpp::connection_hdl hdl){
    cout<<"连接成功"<<endl;
}

// websocket 连接成功的回调函数
void OnClose(websocketsvr *server,websocketpp::connection_hdl
hdl){
    cout<<"连接关闭"<<endl;
}

// websocket 连接收到消息的回调函数
void OnMessage(websocketsvr *server,websocketpp::connection_hdl
hdl,message_ptr msg){
    cout << "收到消息" << msg->get_payload() << endl;
    // 收到消息将相同的消息发回给 websocket 客户端
    server->send(hdl, msg->get_payload(),
websocketpp::frame::opcode::text);
}

// websocket 连接异常的回调函数
void OnFail(websocketsvr *server,websocketpp::connection_hdl hdl){
    cout<<"连接异常"<<endl;
}

// 处理 http 请求的回调函数 返回一个 html 欢迎页面
void OnHttp(websocketsvr *server,websocketpp::connection_hdl hdl){
    cout<<"处理 http 请求"<<endl;
    websocketsvr::connection_ptr con = server-
>get_con_from_hdl(hdl);
    std::stringstream ss;
    ss << "<!doctype html><html><head>"
        << "<title>hello websocket</title><body>"
        << "<h1>hello websocketpp</h1>"
        << "</body></head></html>";
    con->set_body(ss.str());
    con->set_status(websocketpp::http::status_code::ok);
}
```

```
int main(){
    // 使用 websocketpp 库创建服务器
    websocketsvr server;
    // 设置 websocketpp 库的日志级别
    // all 表示打印全部级别日志
    // none 表示什么日志都不打印
    server.set_access_channels(websocketpp::log::alevel::none);
    /*初始化 asio*/
    server.init_asio();
    // 注册 http 请求的处理函数
    server.set_http_handler(bind(&OnHttp, &server, ::_1));
    // 注册 websocket 请求的处理函数
    server.set_open_handler(bind(&OnOpen, &server, ::_1));
    server.set_close_handler(bind(&OnClose, &server, _1));
    server.set_message_handler(bind(&OnMessage,&server,_1,_2));
    // 监听 8888 端口
    server.listen(8888);
    // 开始接收 tcp 连接
    server.start_accept();
    // 开始运行服务器
    server.run();
    return 0;
}
```

Http 客户端

使用浏览器作为 http 客户端即可， 访问服务器的 8888 端口。



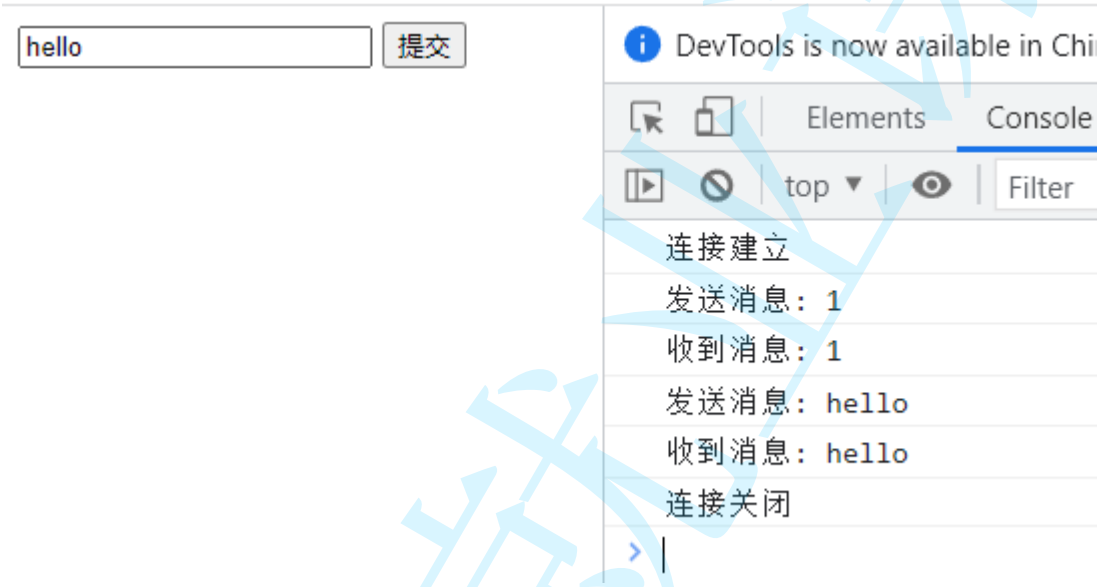
WS 客户端

```HTML
    <!DOCTYPE html>
```

```
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width,
initial-scale=1.0">
        <title>Test Websocket</title>
    </head>
    <body>
        <input type="text" id="message">
        <button id="submit">提交</button>

    <script>
        // 创建 websocket 实例
        // ws://192.168.51.100:8888
        // 类比 http
        // ws 表示 websocket 协议
        // 192.168.51.100 表示服务器地址
        // 8888 表示服务器绑定的端口
        let websocket = new WebSocket("ws://192.168.51.100:8888");

        // 处理连接打开的回调函数
        websocket.onopen = function() {
            console.log("连接建立");
        }
        // 处理收到消息的回调函数
        // 控制台打印消息
        websocket.onmessage = function(e) {
            console.log("收到消息: " + e.data);
        }
        // 处理连接异常的回调函数
        websocket.onerror = function() {
            console.log("连接异常");
        }
        // 处理连接关闭的回调函数
        websocket.onclose = function() {
            console.log("连接关闭");
        }

        // 实现点击按钮后, 通过 websocket 实例 向服务器发送请求
        let input = document.querySelector('#message');
        let button = document.querySelector('#submit');
        button.onclick = function() {
```

```
            console.log("发送消息: " + input.value);
            websocket.send(input.value);
        }
    </script>
</body>
</html>
```

在控制台中我们可以看到连接建立、客户端和服务器通信以及断开连接的过程(关闭服务器就会看到断开连接的现象)



注: 通过 f12 或者 fn + f12 打开浏览器的调试模式