河南工業大學

软件体系结构
作品设计说明书

题　　目：基于 C/S 架构的聊天室设计与实现

院系名称：信息学院　专业班级：软件 1601

学生姓名：高天　学　　号：201616030213

教师姓名：刘灿　课程名称：软件体系结构

一．需求

1. 需求概述

本程序为基于 C/S 的网络聊天室系统，使用 Linux 网络编程作为服务器，使用 QT 编程作为客户端。

客户端通过输入 IP 地址、端口号、Email、聊天名称、聊天组号连接到服务器，用户通过客户端发送消息，同时接收来自相同组其他客户端发送的消息，获取当前在线用户信息，通知新用户的上线和用户的下线，实现群聊功能。

服务器负责管理用户的连接、发送消息与退出，有新用户建立连接时，记录新用户信息，并向同组其他客户端广播；用户退出时，清除用户信息，并向同组其他客户端广播离开信息；当有用户发送消息时，向同组其他客户端广播。

2. 服务器功能需求

服务器主要功能有：管理连接的客户端、接收客户端发送数据、向客户端转发数据、向客户端发送数据。

用例图如下所示：

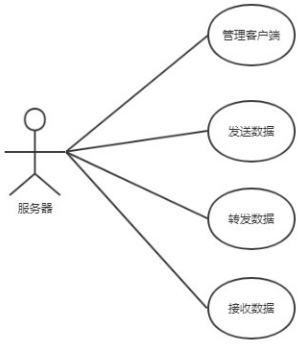

图 1 服务器用例图

客户端登录服务器时序图如下所示：

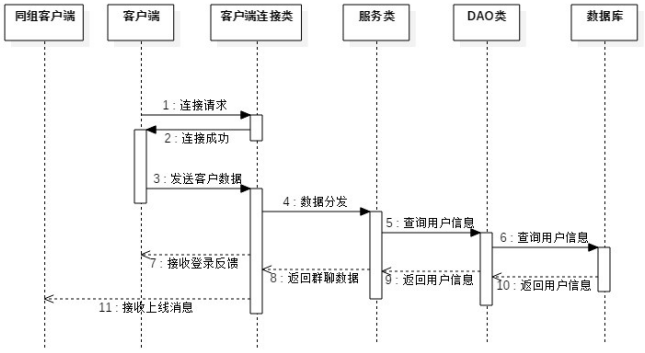

图 2 客户端登录服务器时序图

服务器接收客户端数据时序图如下所示：

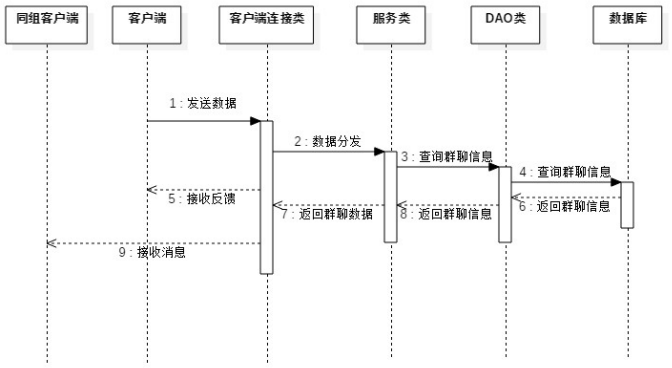

图 3 服务器接收客户端数据时序图

3. 服务器非功能需求

（1）高性能

作为 C/S 架构的聊天程序，首要解决高并发需求，服务器可以同时相应多个客户端，并且不会因为某个客户端而阻塞，具有较高的吞吐量和较低的相应时延，满足实时聊天的需求。

此系统需要选取合适的网络开发模型，网络 IO 模型，可以满足高并发高性能需求，代码实现方面，需选择高效的数据结构与算法，对系统性能进行全面提升。

（2）可修改性可读性

代码实现方面应满足可修改性与可读性，便于以后进行维护及扩展。

系统应满足可读性，代码风格统一，符合编程语言规范，符合编程语言惯用法，命名规范、明了易懂，文档齐全，具有良好的注释，代码结构规整，逻辑清晰，符合“高内聚-低耦合”原则。

系统应满足模块化，系统应是多个模块组合，每个模块具有较高的内聚性，每个模块都便于测试，模块间应具有较低的耦合。

系统应满足可重用性，对于公共的函数、模块、组件将其提出出来进行封装，供未来的重用，节省开发成本，提高开发效率。

系统应满足可维护性，可以高效的对系统进行更改、升级。

（3）可测试性

可以较为简便的对系统进行测试，快速暴露于发现系统的错误，并进行修改。

容易编写桩程序替换底层模块（如数据库等）进行测试。

可以方便的进行白盒测试、黑盒测试、性能测试、安全性测试等。

（4）可扩展性

系统能够满足相应的纵向扩展与横向扩展。

可以通过在一个计算节点中增加资源或更好利用资源进行纵向扩展，如选择合适的并发方式与模型、满足高性能等。

可以通过增加更能多的计算节点进行扩展，如将长连接与短连接分布在不同服务器上，将数据库分离开来，使用 RPC 进行扩展等。

客户端可跨平台，可方便修改通信协议，增加新功能。

4. 客户端功能需求

客户端主要功能：输入服务器 IP、端口、用户基本信息、群组 id，发送登录请求，登录成功后，需要发送信息时发送信息，有群组消息时接收并显示。

客户端用例图如下所示：

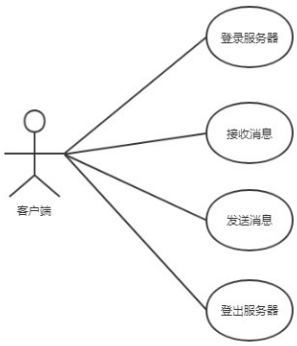

图 4 客户端用例图

登录服务器用例文本：

名称：登录服务器 概述：用户输入服务器 IP、端口号、Email、姓名、聊天组 id 连接并登录服务器 事件流： 1. 客户端根据 IP、端口号连接服务器 2. 连接服务器成功，客户端发送用户信息给服务器，进行登录

3. 解析服务器返回的 JSON 数据，成功进入聊天界面

备选流：

1. 若服务器无响应、连接失败，反馈用户连接失败

2. 若服务器返回的 JSON 数据，表示登录失败，向用户反馈失败原因

3. 若服务器 IP、端口号、Email、姓名、聊天组 id 存在为空或格式不正确，向用户反馈错误

前置条件：

无

后置条件：

进入到聊天界面进行操作

接收消息用例文本：

名称：接收消息

概述：客户端接收到服务器的 JSON 数据，进行相应操作，如新用户上线、用户下线、群聊消息、聊天组信息等

事件流：

1. 当服务器 Socket 可读时，进行事件响应

2. 解析服务器的 JSON 格式数据

3. 根据 JSON 数据中的 CODE 字段，调用相应处理函数

备选流：

若 JSON 数据解析失败，用例结束

前置条件：

客户端成功连接服务器

后置条件：

进入相应处理函数进行处理

5. 客户端非功能需求

（1）高性能

客户端与服务器连接，应具有较小的时延，较底的响应速度，发送消息和接收消息不应相互阻塞，拥有良好的聊天体验。

选择高效的网络 IO，提升系统网络速度。

需选择高效的数据结构与算法，对系统性能进行提升。

（2）可修改性可读性

代码实现方面应满足可修改性与可读性，便于以后进行维护及扩展。

系统应满足可读性，代码风格统一，符合编程语言规范，符合编程语言惯用法，命名规范、明了易懂，文档齐全，具有良好的注释，代码结构规整，逻辑清晰，符合“高内聚-低耦合”原则。

系统应满足模块化，系统应是多个模块组合，每个模块具有较高的内聚性，每个模块都便于测试，模块间应具有较低的耦合。

系统应满足可重用性，对于公共的函数、模块、组件将其提出出来进行封装，供未来的重用，节省开发成本，提高开发效率。

系统应满足可维护性，可以高效的对系统进行更改、升级。

（3）可测试性

可以较为简便的对系统进行测试，快速暴露于发现系统的错误，并进行修改。

容易编写桩程序替换底层模块（如数据库等）进行测试。

可以方便的进行白盒测试、黑盒测试、性能测试、安全性测试等。

（4）可扩展性

客户端可跨平台，可方便进行功能扩充。

二. 架构

1. 服务器初始架构

如下图所示，整体为 C/S 架构，多个客户端与服务器通信，所有客户端仅依赖一个服务器，聊天服务器承担责任过重，具有较大的性能压力。

通信协议采用字符串拼接形式，形如 code:name:data，优点是简单，体积小，缺点是可读性差、扩展性差，若需要增加新字段，需要大量修改程序。

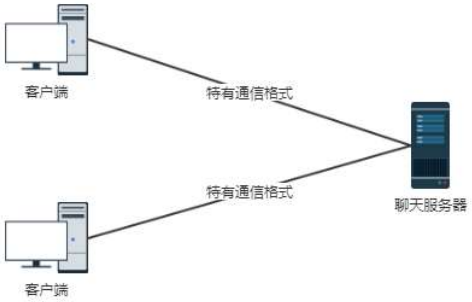

图 5 服务器初始架构

网络通信采用原始的 select I/O 多路复用方式，虽较好解决了并发性问题，但实现起来较为臃肿，且与业务代码耦合严重，难以修改维护。

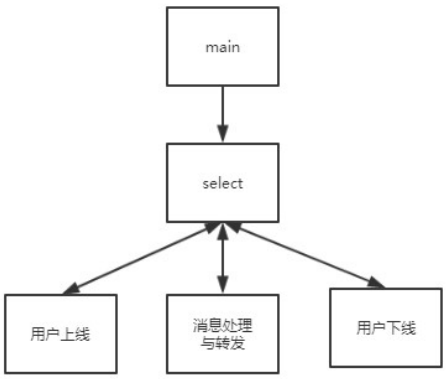

图 5 服务器初始代码结构

代码层面上，使用面向过程编程方式。代码结构较为混乱，没有明显的层次关系，业务代码与网络相关代码存在大量耦合、且大量双向耦合，如上图所示的层次结构，main 函数直接使用 select 函数，select 函数与用户上线、消息处理与转发、用户下线业务代码双向耦合，可读性、可维护性、可测试性都很差。

代码没有清晰的模块，一个文件承载了过多功能，没有较多体现分而治之思想，且代码存在注释缺失，对可修改可读性造成了更大的损害。

2. 优化后服务器架构

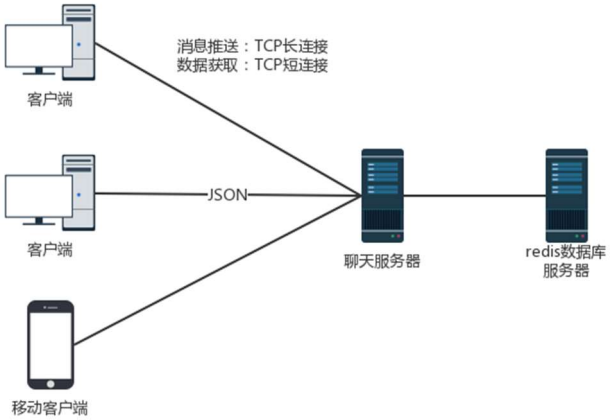

图 6 当前服务器架构

支持多种客户端，客户端与服务器使用 TCP 长连接进行消息推送，获取数据信息时使用 TCP 短连接，降低性能压力， TCP 长连接可和 TCP 短连接分离为不同服务器，TCP 短连接使用 HTTP 协议，进行横向扩展。

客户端与服务器使用 JSON 格式传输数据，方便扩展，如可以容易的进行客户端跨平台扩展，增加修改 JSON 格式字段可很少或不修改现有代码，增加新功能。

数据库使用 redis 内存数据库，具有很高的性能，用于存储各聊天室基本信息，聊天数据（使用过期策略清理聊天数据），且可独立为一高性能服务器，提供高效的数据访问能力。

网络连接层次使用基于 IO 多路复用、Reactor 模式的异步事件处理库—Libevent。

也即将网络连接层更改为事件驱动架构，Libevent 提供一个主事件循环，监听网络 IO 等事件，在检测到事件时触发具有特定参数的回调函数。通过事件驱动，将网络 IO 操作与业务层可以较好的解耦。

此外，使用现成的成熟的网络库，可以提高开发效率，提高网络并发性能，编写系统时可专注于业务代码，代码更易于维护、修改。

图 7 服务器采用分层架构

系统采用分层架构，分为接入层、业务层、DAO 持久化层、数据库层。分层结构将组件间的关注点分离，一个层中的组件只处理本层的逻辑。如接入层使用 libevent 网络库，只负责管理与客户端的网络连接，发送网络数据，接收网络数据，将接入层数据交给业务层处理。分层结构中从上往下逐层调用，不存在反向依赖与跨层依赖。

接入层：使用 libevent 事件驱动机制，负责管理与客户端的长连接，接收来自客户端数据，向各客户端发送数据。此层保存接入层客户端的标识符 sockfd 和业务层客户端标识符 email 的映射，使得下层不用考虑接入层的特有数据。

业务层：根据接入层传来的 email 和信息，解析 JSON，处理具体的业务，对于群聊业务来说，根据 emial 所在的 groupid 将消息转发给相应组员。上层接入层通过接口返回值处理相应的接入层操作。

DAO 层：用于访问 redis 数据库，进行具体的数据库访问操作，向上层返回数据访问接口。

redis 数据库层：存储用户会话信息，此系统中暂时使用内存中的 map 代替，体现了分层的好处—可测试性，基本数据结构为：

Key: email,

Value: [email, name, groupid]

3. 客户端架构

客户端负责聊天界面显示以及与服务器通信实现群聊功能，客户端结构图如下所示，mainwindow 为登录界面类，chatwindow 为聊天界面类，chattcpsocket 为聊天网络连接类，负责与服务器的连接，向服务器发送数据，以及接收来自服务器的数据等。chattcpsocket 被注入到 mainwindow 与 chatwindow 中，将业务逻辑与网络处理向分离，解除耦合，增强系统可读、可修改、可维护性。

QT 提供了一些设计优秀的结构、行为机制，可以使程序具有良好的架构。如 QT 将界面与业务逻辑相分离，可以独立方便修改界面与业务逻辑，而不用做过多修改；使用信号/槽机制来完成组件间的通信，是事件驱动编程的一种独特实现方式，很好的解除了各个组件间的耦合，使得客户端开发与维护变得简单。

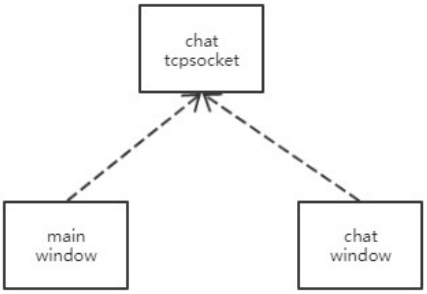

图 8 客户端代码结构

4. 服务器与客户端通信格式

客户端与服务器使用统一通用的 JSON 数据通信。

客户端发向服务器：

消息类型	消息格式

上线	{
	“code” : 0,
	“email” : “gtl10@qq.com”,
	“name” : “高天”,
	“groupid” : 0
	}

下线	{
	“code” : 1,
	“email” : “gtl10@qq.com”
	}

发送消息	{
	“code” : 2 ,
	“email” : “gtl10@qq.com”,
	“msg” : “你好，世界”
	}

获取组成员列表	{
	“code”:3,
	“email” : “gtl10@qq.com”
	}

服务器发向客户端：

消息类型	消息格式

用户上线	{
	“code” : 100,
	“email” : “gt110@qq.com”
	“name” : “高天”,
	“time” : 12345//时间戳
	}

用户下线	{
	“code” : 101,
	“email” : “gt110@qq.com”
	“time” : 12345
	}

获取聊天组成员列表	{
	“code” : 103,
	“members” : [
	{
	“email” : “gt110@qq.com”
	“name” : “高天”
	},
	{
	“email” : “test@163.com”
	“name” : “张三”
	}
	]
	“time” : 1341324
	}

加入聊天组	{
	“code” : 104,
	“status” : 1 //1 为成功，0 为失败
	“desc” : “加入成功” // 失败时为具体错误原因，如”邮箱已存在”
	“time” : 1341243
	}

群聊消息 ACK	{
	“code” : 105,
	“status” : 1 //1 为发送成功，0 为发送失败
	“time” 12321
	}

三. 代码

1. 服务器—接入层

接入客户端类定义如下所示，对于能使用 const 的场合，均使用 const 修饰，作为一种安全保证和标识。传递函数参数时，尽量使用引用，而不是值拷贝，可以得到较好的效率。使用容器时使用 STL 模板容器，拥有很好的性能，特别是使用 map 极大加快了查询的速度。

因为网络库使用了基于 C 语言的 Libevent 库，因此需要将面向过程的 Libevent API 与面向对象的类结合起来。

使用命名空间来管理类，增加系统的模块化。

```
namespace access {
    //客户端连接类，负责与各客户端的连接、IO
    class AccessClient {
    private:
        const int port_;
        const int max_data_len_;
        struct event_base *event_base_ = nullptr;
        struct evconnlistener *listener_ = nullptr;
        std::map<int, ClientAccessInfo *> clients_map_by_sockfd;
        std::map<std::string, ClientAccessInfo *> clients_map_by_email;
        service::IChatroomService *i_chatroom_service;

        struct sockaddr_in accept_all_sock_addr() const;

        static void accept_error_cb(struct evconnlistener *listener, void *ctx);

        static void
        accept_conn_cb(struct evconnlistener *listener, evutil_socket_t fd, struct
          sockaddr* addr, int socklen, void *ctx);

        static void read_cb(struct bufferevent *bev, void *ctx);

        static void write_cb(struct bufferevent *bev, void *ctx);

        static void event_cb(struct bufferevent *bev, short events, void *ctx);

        void
          user_online_op(ClientAccessInfo *client_access_info, const std::string
          &src_email, const std::string &src_data, const std::vector<std::string>
          &dst_emails, const std::string &dst_data);
```

```
        void
          group_msg_op(ClientAccessInfo *client_access_info, const std::string
          &src_email, const std::string &src_data, const std::vector<std::string>
          &dst_emails, const std::string &dst_data);

        void
          deny_enter_op(ClientAccessInfo *client_access_info, const std::string
          &src_email, const std::string &src_data, const std::vector<std::string>
          &dst_emails, const std::string &dst_data);

        void
          user_offline_op(ClientAccessInfo *client_access_info, const std::string
          &src_email, const std::string &src_data, const std::vector<std::string>
          &dst_emails, const std::string &dst_data);

        void group_members_list_op(ClientAccessInfo *client_access_info, const
          std::string &src_email, const std::string &src_data, const std::vector<std::string>
          &dst_emails, const std::string &dst_data);

        void map_sockfd_user(evutil_socket_t fd, const std::string &email);

        void send_msg_by_email(const std::string &email, const std::string &data);

        void broadcast_msg_by_emails(const std::vector<std::string> &emails,
          const std::string &data);

        void send_data(evutil_socket_t fd, const std::string &data);

        void remove_access_info_by_email(const std::string &email);

    public:
        AccessClient(int port, int max_data_len);

        ~AccessClient();

        void start() const;

        void stop() const;
    };
}
```

下面代码片段为服务器接收到客户端数据时调用的回调函数。先从缓冲区中获取数据，将接入层标识 sockfd 转换为业务层标识 email（如果是用户新用户上线，由业

务层设置），之后将数据转交给业务层，此次通过引用来是业务层修改数据，业务层处理后返回：处理的是什么类型的数据、应发送给原发送方的数据、应转发给相应用户的数据。接入层再根据返回值调用相应处理函数，进行数据转发等操作。

此处使用了 C++11 语法—auto 关键字，可自动推断变量类型。对于多重条件，使用 switch，更加清晰地处理各种情况。程序中未显示出现数字，即魔数，均使用常量来代替。

```
//当客户端写入消息时回调
void AccessClient::read_cb(struct bufferevent *bev, void *ctx) {
    //获取 this 对象
    auto *this_access_client = (AccessClient *) ctx;

    //获取输入缓冲区
    auto *input = bufferevent_get_input(bev);

    //读取输入，交给 service 层处理
    char *line;
    size_t n;

    int fd = bufferevent_getfd(bev);
    ClientAccessInfo *client_access_info = this_access_client->clients_map_by_sockfd[fd];

    while ((line = evbuffer_readln(input, &n, EVBUFFER_EOL_ANY))) {
        std::cout << "recv: " << line << std::endl;
        //交给服务层处理，根据服务层返回状态
        std::string src_email;
        if (!client_access_info->get_email().empty())
            src_email = client_access_info->get_email();
        std::string src_data = std::string(line);
        std::vector<std::string> dst_emails;
        std::string dst_data;
        int code = this_access_client->i_chatroom_service->data_dispatch(src_email,
                src_data, dst_emails, dst_data);

        switch (code) {
            case service::IChatroomService::RET_NEW_ONLINE:
                this_access_client->user_online_op(client_access_info,
                                                   src_email, src_data, dst_emails, dst_data);
                break;

            case service::IChatroomService::RET_DENY_ENTER_GROUP:
                this_access_client->deny_enter_op(client_access_info,
```

```
                                    src_email, src_data, dst_emails, dst_data);
                    break;

                case service::IChatroomService::RET_GROUP_MSG:
                    this_access_client->group_msg_op(client_access_info,
                                    src_email, src_data, dst_emails, dst_data);
                    break;

                case service::IChatroomService::RET_USER_OFFLINE:
                    this_access_client->user_offline_op(client_access_info,
                                    src_email, src_data, dst_emails, dst_data);
                    break;

                case service::IChatroomService::RET_GROUP_MEMBERS_LIST:
                    this_access_client->group_members_list_op(client_access_info,
                                    src_email, src_data, dst_emails, dst_data);
                    break;
                default:
                    break;
            }
        }
    }
```

2. 服务器—服务层

服务层接口如下所示，仅向上层—接入层提供两个接口，其中 data_dispatch 接口承担了多个相似功能，首先传入上层发送方的 email 和接收到的数据，再解析数据，根据数据格式进行相应处理，如果需要返回给上层发送方、多个接收者，则修改引用的参数即可。offline 接口负责用户下线的处理。

此接口类使用了虚析构函数，使用实现 dao 接口的对象来操纵数据库。此外，使用 const static 定义了大量常量便于使用。

```
namespace service {
    class IChatroomService {
    public:
        virtual ~IChatroomService() = default;

    protected:
        dao::IOnlineUserInfoDao *online_user_info_dao{};

    public:
        const static int NO_RETURN_DATA = 0;
        const static int USER_ONLINE = 100;
```

```
        const static int USER_OFFLINE = 101;
        const static int GROUP_MSG = 102;
        const static int GROUP_MEMBERS_LIST = 103;
        const static int ADD_GROUP = 104;

        const static int CLIENT_TO_SERVER_ONLINE_CODE = 0;
        const static int CLIENT_TO_SERVER_OFFLINE_CODE = 1;
        const static int CLIENT_TO_SERVER_SENDMSG_CODE = 2;
        const static int CLIENT_TO_SERVER_GET_GROUP_INFO = 3;

        const static int RET_NEW_ONLINE = 0;
        const static int RET_DENY_ENTER_GROUP = 1;
        const static int RET_GROUP_MSG = 2;
        const static int RET_USER_OFFLINE = 3;
        const static int RET_GROUP_MEMBERS_LIST = 4;


        virtual int data_dispatch(std::string &src_client_email, std::string &src_data,
                std::vector<std::string> &dst_clients_email, std::string &dst_data) = 0;

        virtual void offline(std::string &src_client_email, std::vector<std::string>
                                &dst_clients_email, std::string &dst_data) = 0;
    };
}
```

以下代码为业务层具体实现类（部分）。构造函数将 DAO 层接口对象组合进来使用，使用 JSON 处理类 ClientJsonMsgParser 对接收到数据进行处理，根据 CODE 字段进行不同操作。

业务层具体实现类面向接口编程，满足设计模式中合成复用原则—细节应依赖抽象，抽象不应依赖细节。

```
namespace service {
    ChatroomServiceImpl::ChatroomServiceImpl() {
        online_user_info_dao = new dao::OnlineUserInfoDaoImpl();
    }

    ChatroomServiceImpl::~ChatroomServiceImpl() {
        delete online_user_info_dao;
    }

    int ChatroomServiceImpl::data_dispatch(std::string &src_client_email, std::string
        &src_data, std::vector<std::string> &dst_clients_email, std::string &dst_data) {
        std::string ori_src_data = src_data;
```

```
std::string ori_src_client_email = src_client_email;

try {
    util::ClientJsonMsgParser json_parser(ori_src_data);

    if (json_parser.get_code() == CLIENT_TO_SERVER_ONLINE_CODE) {
        online_user_info_dao->print_map_clients_info();
        int groupid
            =
      online_user_info_dao->check_email_exists(json_parser.get_email());

        if (groupid == -1) {
            //用户未加入群组
            online_user_info_dao->add_new_user(json_parser.get_email(),
                        json_parser.get_name(), json_parser.get_groupid());
            src_client_email = json_parser.get_email();
            src_data = service::EnterGroup(ADD_GROUP, 1, "succ",
                                           time(nullptr)).to_json();

            std::vector<std::string> emails =
                    online_user_info_dao->query_emails_by_groupid(
                     json_parser.get_groupid());

            online_user_info_dao->exclude_email_from_vector(emails,
                    src_client_email);dst_clients_email.insert(
                    dst_clients_email.end(), emails.begin(), emails.end());

                dst_data = service::UserOnline(USER_ONLINE,
                        json_parser.get_email(), json_parser.get_name(),
                    time(nullptr)).to_json();

            return RET_NEW_ONLINE;
        } else {
            //已加入 groupid 群组
            src_data = service::EnterGroup(ADD_GROUP, 0,
                "fail: User owning this email has entered the group : " +
                 std::to_string(groupid), time(nullptr)).to_json();

            return RET_DENY_ENTER_GROUP;
        }

    } else if (json_parser.get_code() ==
        CLIENT_TO_SERVER_SENDMSG_CODE
        ……
```

```
                return RET_USER_OFFLINE;
            } else if (json_parser.get_code() ==
                CLIENT_TO_SERVER_GET_GROUP_INFO
                && !ori_src_client_email.empty()) {
                ……
                return RET_GROUP_MEMBERS_LIST;
            }

        } catch (util::ParseException &e) {
            std::cout << e.what() << std::endl;
        }
        return -1;
    }
}
```

3. 服务器—DAO 层

DAO 层用于与数据库交互，接口如下所示，接口设计使用了 C++纯虚函数语法，符合语言规范。

```
namespace dao {
    class IOnlineUserInfoDao {
    public:
        virtual void add_new_user(std::string email, std::string name, int groupid) = 0;

        virtual bool del_user_by_email(std::string email) = 0;

        virtual std::vector<std::string> query_emails_by_groupid(int groupid) = 0;

        virtual int check_email_exists(std::string email) = 0;

        virtual bool exclude_email_from_vector(std::vector<std::string> &emails,
                                               std::string excluded_email) = 0;

        virtual bool query_clientinfo_by_email(std::string email,
                                               dao::ClientInfo &client_info) = 0;

        virtual std::vector<dao::ClientInfo> query_clientsinfo_by_groupid(int groupid) = 0;

        virtual void print_map_clients_info() = 0;

        virtual ~IOnlineUserInfoDao() = default;
    };
}
```

以下三段代码分别为添加新用户、按照 email 删除用户、按照 groupid（聊天组 id）查询所有用户。

```
void OnlineUserInfoDaoImpl::add_new_user(std::string email, std::string name,
int groupid){
        ClientInfo c;
        c.email = email;
        c.name = name;
        c.groupid = groupid;
        map_clients_info[email] = c;
}

bool OnlineUserInfoDaoImpl::del_user_by_email(std::string email) {
    auto it = map_clients_info.find(email);
    if (it != map_clients_info.end()) {
        map_clients_info.erase(it);
        return true;
    }

    return false;
}

std::vector<std::string> OnlineUserInfoDaoImpl::query_emails_by_groupid(
int groupid) {
    std::vector<std::string> emails;

    for (auto &it : map_clients_info) {
        if (it.second.groupid == groupid)
            emails.push_back(it.second.email);
    }

    return emails;
}
```

4. 服务器—工具类

下面为解析客户端发来 JSON 数据的类的构造函数，传入 JSON 数据，将字段数组存在成员变量中。解析 JSON 库时使用 rapidjson 开源 JSON 解析库。

```
explicit ClientJsonMsgParser(const std::string &msg) {
    d.Parse(msg.c_str());

    if (d.HasParseError())
        throw ParseException();
```

```
    rapidjson::Value &code = d["code"];
    code_ = code.GetInt();

    auto it_email = d.FindMember("email");
    if (it_email != d.MemberEnd()) {
        rapidjson::Value &email = d["email"];
        email_ = email.GetString();
    }

    auto it_name = d.FindMember("name");
    if (it_name != d.MemberEnd()) {
        rapidjson::Value &name = d["name"];
        name_ = name.GetString();
    }

    auto it_groupid = d.FindMember("groupid");
    if (it_groupid != d.MemberEnd()) {
        rapidjson::Value &groupid = d["groupid"];
        groupid_ = groupid.GetInt();
    }

    auto it_msg = d.FindMember("msg");
    if (it_msg != d.MemberEnd()) {
        rapidjson::Value &msg = d["msg"];
        msg_ = msg.GetString();
    }
}
```

5. 客户端

以下为 ChatTcpSocket 类的两个函数，startRecvMsg 函数用于连接接收到数据时的信号，接收到信号时调用 transferMsg 函数，此时 transferMsg 再发送接收到数据的信号。

```
//开始接收消息
void ChatTcpSocket::startRecvMsg() {
    connect(&socket, &QTcpSocket::readyRead, this, &ChatTcpSocket::transferMsg);
}
//接收到消息时发送信号
void ChatTcpSocket::transferMsg() {
    QByteArray netdata = socket.readLine();

    emit recvMsg(QString(netdata));
}
```

以下为聊天窗口类 ChatWindow 的槽函数，当 ChatTcpSocket 类接收到数据并调用 transferMsg 发送后，ChatWindow 中的 recvMsg 函数进行接收处理。解析服务器发送来的 JSON 数据，根据不同 CODE 字段调用不同处理函数进行处理。

```
void ChatWindow::recvMsg(const QString &data) {
    serverjson::ParseServerJson p(data);

    switch (p.getCode()) {
    case serverjson::USER_ONLINE_CODE:
        newUserOnline(p.getEmail(), p.getName());
        break;
    case serverjson::USER_OFFLINE_CODE:
        if (membersMap.contains(p.getEmail())) {
            userOffline(p.getEmail(), membersMap[p.getEmail()]);
        }
        break;
    case serverjson::USER_MSG_CODE:
        if (membersMap.contains(p.getEmail())) {
            newUserMsg(p.getEmail(), membersMap[p.getEmail()],
                                                    p.getMsg(), p.getTime());
        }
        break;
    case serverjson::GROUP_MEMBER_LIST_CODE:
        disGroupMembersList(p.getMembers());
        break;
    }
}
```

以下为客户端发送在线信息和聊天信息函数，将数据封装为 JSON 数据进行发送。

```
//发送上线信息
void ChatTcpSocket::sendOnline() {
    clientjson::Online on(clientjson::ONLINE_CODE, email, name, groupid);
    sendMsgToServer(on.toJson());
}

//用户发送信息
void ChatTcpSocket::sendMsg(const QString &email, const QString &msg) {
    clientjson::SendMsg m(clientjson::SENDMSG_CODE, email, msg);
    sendMsgToServer(m.toJson());
}
```

四．质量评价

1. 高性能

作为 C/S 架构的聊天室系统，最重要的架构质量属性就是性能。此系统要求服务器的高并发能力，既是功能需求，也是非功能需求（要求更高的并发能力）。因此在设计此系统时关注点应是如何提高并发能力。

此系统中，使用 C 语言的网络库 Libevent，为轻量级、基于异步事件驱动的开源高性能网络库。底层使用主流的并发解决方案—I/O 多路复用。因此借助 Libevent 库可以实现高并发能力的 C/S 程序。实际测试中本系统具有较好的并发能力，允许多个客户端向服务器同时发送消息，并且具有较低的时延与响应速度。

代码层面，使用高效的算法和数据结构，如在进行大量查询操作时，使用 map 数据结构替换 vector 等线性结构。

数据库使用基于内存的 Redis 数据库存储在线用户信息，Redis 数据库具有优秀的性能表现。

2. 可读可修改性

系统具有一定的可读性，代码风格统一，符合编程语言规范，符合编程语言惯用法。对函数、类、模块、文件的命名较为规范，力求明了易懂。对函数、代码关键部分均添加了注释，代码结构规整，逻辑清晰，遵从“高内聚—低耦合原则”。

架构上使用分层，每一层实现较为清晰，相对独立，向上层提供接口，带来了良好的可读与可修改性。

系统使用模块化设计，使用 C++的命名空间将类分为各个不同模块。每个模块具有较高的内聚性，便于修改与测试，模块间具有较低的耦合性。

系统尽可能将公共函数、模块提取出来进行封装，以供重用，节省开发成本，提升开发效率。

3. 可测试性

系统使用标准的分层架构，带来了良好的可测试性。因为各层间通过接口来调用，可以单独测试每一层的接口。而且对于下层模块，可以通过桩程序代替测试，本系统中为测试高层模块，将最底层的数据库使用 map 变量代替，不用修改高层模块，即完成了高层模块的测试，具有较好的可测试性。

4. 可扩展性

系统使用异步技术来支持 ，具有良好的性能，也即支持良好的垂直扩展，可以通过在一个计算节点中增加资源或更好利用资源进行纵向扩展。

系统可将长连接与短连接分离在不同服务器，短连接使用 HTTP 协议，因为采用了分层结构，数据库也可以较为容易进行分离，满足水平扩展。

客户端可跨平台，可方便修改通信协议，增加新功能。

5. 设计模式

本系统设计过程中，以设计模式基本思想为指导，如使用接口满足开闭原则和依赖倒转原则，每个类满足单一职责原则，满足高内聚—低耦合原则。设计接口时，尽可能满足接口隔离原则。

五．结论

本次系统我选择了基于 C/S 架构的聊天室系统，一是因为 C/S 作为一个基础性架构，应用广泛，可以很好增强架构设计能力，二是因为之前做过类似系统，可以对其进行重构，增加新功能，认识到架构优化的重要意义。

优化过程中，我深切感受到了一个设计良好的架构带来的诸多好处。

之前的代码没有清晰的架构、层次，网络操作相关代码与业务代码严重耦合，组织混乱，一个函数包含过多内容，一个文件包含过多函数，而且没有清晰的注释。当我再次阅读代码尝试进行重构时，发现很难阅读，可读性与可修改性极差，自然可测试性、可扩展性也就很差。性能方面，虽然使用了 I/O 多路复用带来了较好的并发性，但是 I/O 多路复用函数贯穿整个代码，代码结构失调，当添加用户可以按照群组 id 进行聊天的新功能时，修改无从下手，牵一发而动全身。

面对着当初设计不友好的代码，借助于软件体系结构这门课程所学的架构知识，我开始重新设计并编写聊天室系统。在众多的架构模式中，我综合选择了最为基础性架构—分层架构，掌握好分层架构才能灵活运用其他架构模式。我参照最著名的分层设计 TCP/IP 协议，尝试着将我的系统分为四个层次—接入层、服务层、DAO 层、数据库。每一层次向上层提供相应的接口，每一层次相对独立，只为紧接着上层服务，使用紧邻的下层提供的服

务，而且不存在跨层次调用。这样，整个系统就有了大体上较为清晰的结构，我也体会到了分层带来的好处：当编写、修改某一功能时，我可以专注与该功能所在层次，而不用过多修改其他层次，更不会出现之前牵一发而动全身的现象；当我在进行测试时，我也要使用桩程序，替换掉某一层，单独测试某一层，如我使用内存中的 map 数据结构替换掉 redis 数据库，程序也具有相同的效果。

总之，未来在设计系统时，要充分考虑架构思想，设计模式思想，不断在实践中增强自己的架构运用能力。