
《Linux 系统编程》大作业 CIYACA 开发 文档

发布 1.0

北京理工大学 CIYACA 团队

2020 年 06 月 08 日

1	简介	1
1.1	CIYACA 是什么	1
1.2	为什么叫 CIYACA	1
2	特色与创新之处	4
3	编译环境和使用方式	5
3.1	编译环境	5
3.2	目录结构	5
3.3	使用方式	7
3.3.1	下载项目	7
3.3.2	编译运行	7
3.3.2.1	server	7
3.3.2.2	client	7
3.3.3	开发前的配置要求	8
4	功能介绍	9
4.1	基本功能	9
4.2	具体功能	10
4.2.1	登录、注册界面	10
4.2.2	处理好友申请	11
4.2.3	单聊与群聊	12
4.2.4	BBS 论坛	13
4.2.5	消息备份与历史记录	15
4.2.6	文件传送	15
5	架构设计	16
5.1	整体架构	16
5.2	模块设计	17
5.2.1	界面设计	17
5.2.1.1	编译环境	17

5.2.1.2	注册界面设计思路	17
5.2.1.3	主体界面设计思路	17
5.2.2	客户端设计	18
5.2.2.1	功能区设计	18
5.2.2.2	帖子打包	18
5.2.2.3	帖子解析	19
5.2.3	网络通信——FeverRPC-ng	20
5.2.3.1	功能	20
5.2.3.1.1	支持的功能	20
5.2.3.1.2	不支持的功能	20
5.2.3.2	设计目标	20
5.2.3.3	模块架构	21
5.2.3.4	绑定-执行机制设计	22
5.2.3.5	任意类型不定长参数设计	22
5.2.3.6	任意长数据传送设计	22
5.2.3.7	Debug 与错误处理设计	23
5.2.3.8	双向调用设计	23
5.2.3.9	依赖	23
5.2.3.9.1	运行依赖	23
5.2.3.9.2	涉及的 Linux 系统调用	24
5.2.3.10	开发者注意事项	24
5.2.3.10.1	为什么你需要它	24
5.2.3.10.2	解决双向调用的身份识别问题	24
5.2.3.11	鸣谢	24
5.2.4	服务端设计	24
5.2.4.1	编译环境	25
5.2.4.2	功能模块	25
5.2.4.2.1	主控模块	25
5.2.4.2.2	账户管理模块	25
5.2.4.2.3	通讯模块	25
5.2.4.2.4	bbs 服务模块	25
5.2.4.2.5	数据库通讯模块	26
5.2.5	数据库的设计与建立	26
5.2.5.1	User 表	26
5.2.5.2	single_chat_info 表	26
5.2.5.3	single_chat_history 表	27
5.2.5.4	group_chat_info 表	27
5.2.5.5	group_chat_history 表	27
5.2.5.6	group_chat_management 表	28
5.2.5.7	friend_apply 表	28
6	困难与解决办法	29
6.1	解决双向 RPC 设计中可变长模板参数的设计	29
6.1.1	难点	29
6.1.2	解决方法	30

6.2	解决双向 RPC 调用中身份认证的问题	30
6.2.1	难点	30
6.2.2	解决方法	30
6.3	数据库设计方面	30
6.4	对接问题	31
6.4.1	难点	31
6.4.2	解决方法	31
6.5	BBS 呈现问题	31
6.5.1	难点	31
6.5.2	解决方案	31
6.6	服务端设计问题	32
7	分工和人员合作	33
7.1	分工	33
7.2	合作	34
7.2.1	会议记录	34
7.2.1.1	三月九日	34
7.2.1.2	三月十六日	34
7.2.1.3	四月中旬	34
7.2.1.4	五月上旬	35

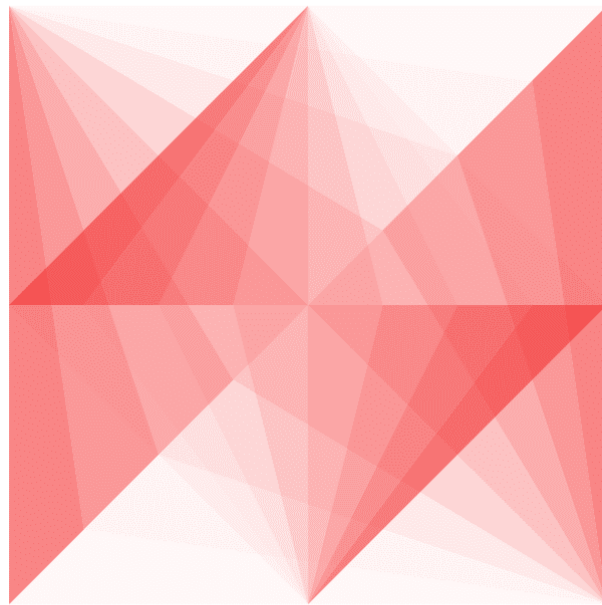
1.1 CIYACA 是什么

CIYACA 是为北京理工大学《2020 年春季 Linux 系统编程》大作业而开发的聊天 + BBS 软件，包括服务端和客户端，均采用 C/CPP 实现。

我们的 Github Organization: <https://github.com/kiyaca>

1.2 为什么叫 CIYACA

CIYACA 采用递归的命名方式，*CIYACA Is Yet Another Chat App*。



CIYACO

Is Yet Another Chat App

图 1.1: 我们设计的 LOGO

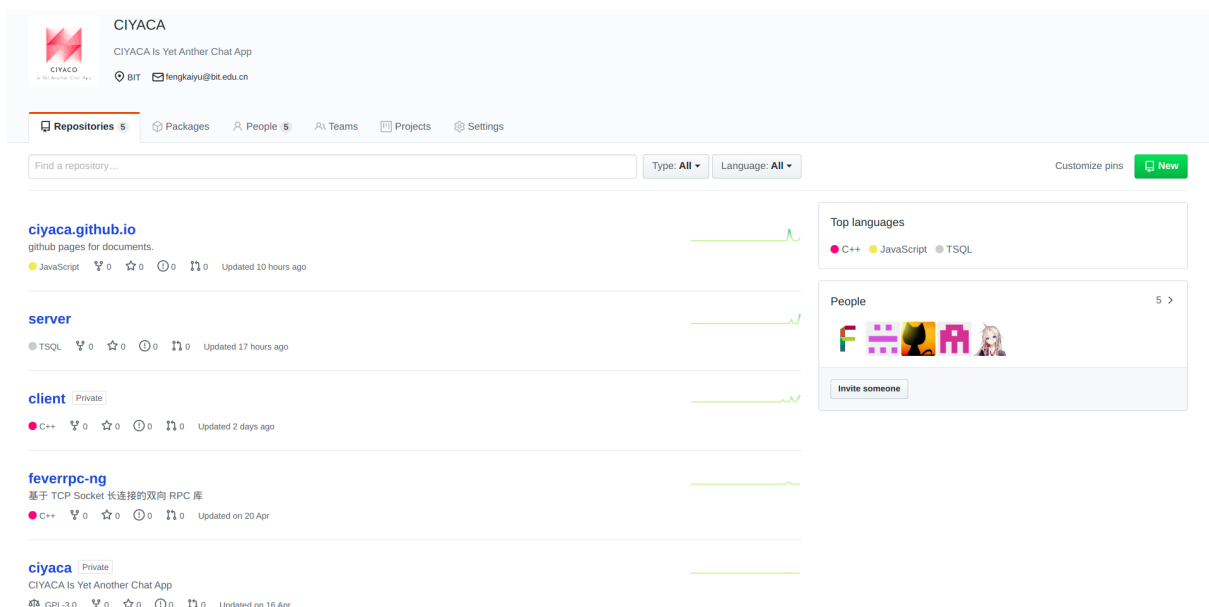


图 1.2: 代码仓库

特色与创新之处

- 专门设计了 Logo、名称
- 采用 Github 进行团队协作
- 采用 Sphinx + reStructuredText 进行文档写作
- 采用石墨文档进行共享编辑、整理前期内容
- 设计了美观的界面 - 有聊天气泡
- 使用 RPC 的方式封装了底层通信，提供耦合度极低的双向调用方式

3.1 编译环境

最低版本:

- Ubuntu 18.04
- g++ 7.5.0
- make 4.1
- qt 5.14.0
- MySQL Ver 14.14 Distrib 5.7.30
- msgpack 3.2.1

3.2 目录结构

```
..
├── client
│   └── src
│       ├── addfriend.cpp//添加好友
│       ├── bbsnewpostdialog.cpp//bbs 发帖
│       ├── bbspstlistwidget.cpp//bbs 帖子显示结构
│       ├── BBSPostReceiver.cpp//收贴
│       ├── chatface.cpp//聊天
│       ├── chatmessage.cpp//聊天信息
│       └── chatmessage.h
```

(下页继续)

(续上页)

```

|   |   |— client.cpp//登录
|   |   |— client.h
|   |   |— come_message.cpp//来信
|   |   |— come_message.h
|   |   |— common.cpp//通用数据结构
|   |   |— common.h
|   |   |— contactitem.cpp//联系人
|   |   |— contactitem.h
|   |   |— controller.cpp//主控函数
|   |   |— controller.h
|   |   |— creategroup.cpp//创建群组
|   |   |— emoji.cpp//emoji 表情
|   |   |— emojiwidget.cpp//显示 emoji 的框
|   |   |— feverrpc//底层通信 rpc 模块
|   |   |   |— feverrpc-client.cpp
|   |   |   |— feverrpc-client.hpp
|   |   |   |.....
|   |   |— friendrequest.cpp//好友请求
|   |   |— main.cpp
|   |   |— mainwindow.cpp//主界面
|   |   |— mainwindow.h
|— images
|   |— logo.png
|— LICENSE
|— README.md
|— server
|   |— account.cpp    // 账户、群组、好友管理
|   |— account.hpp
|   |— bbs.cpp        // bbs 服务、云文件实现
|   |— bbs.hpp
|   |— im.cpp         //单聊、群聊、历史消息实现
|   |— im.hpp
|   |— makefile
|   |— net_disk       //云文件的存储目录
|   |   |— addfriend.cpp
|   |   |— addfriend.o
|   |   |— outTestFile.txt
|   |   |— Screenshot from 2020-05-27 19-07-21.png
|   |— README.md
|   |— server_main.cpp    //主程序
|   |— sql
|   |   |— ciyacaSQL.sql    //数据库文件
|   |   |— sql.cpp        //数据库初始化与访问实现
|   |   |— sql.hpp
|   |— test //功能测试程序
|       |— file.txt
|       |— testCiyacaSQL

```

(下页继续)

(续上页)

```
├─ testCiyacaSQL.cpp
├─ testFileTransmit
└─ testFileTransmit.cpp
```

3.3 使用方式

3.3.1 下载项目

```
git clone https://github.com/ciyaca/ciyaca.git
cd ciyaca
git submodule update --init --recursive # 递归获取子模块
```

或者下载压缩包

3.3.2 编译运行

首先编译运行 server，然后再启动客户端。

3.3.2.1 server

1. 安装并运行 MySQL
2. 在项目根目录中使用 `make all` 命令，所有源码会被编译，得到的 server 主程序的可执行文件置于 `bin/server`
3. 对于 `sql/ciyacaSQL.sql`，在 MySQL 里创建一个名为 `ciyacaSQL` 的数据库，并使用 `source` 命令将其导入
4. 在数据库导入完毕后，将 `sql/sql.cpp` 中的 `mysql_real_connect` 函数连接参数填写正确（MySQL 登录账户、登录口令、端口、数据库名等）
5. 执行 `bin/server` 即可启动服务端程序

3.3.2.2 client

1. 下载 Qt 5.14.0
2. 安装时选择全部组件
3. 使用 qt 打开工程，选择 `client/serc/test.pro` 进行加载
4. 运行即可

3.3.3 开发前的配置要求

1. 安装 msgpack
1. 请访问 https://github.com/msgpack/msgpack-c/tree/cpp_master 自行编译安装
2. 安装 qt5.14.0
3. 安装 mysql 5.7.30
4. 依据 server 说明配置数据库

功能介绍

4.1 基本功能



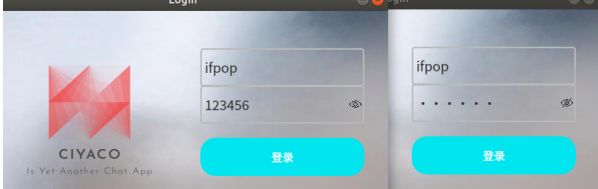
表 4.1: 基本功能介绍

功能名称	功能介绍
联系人树状显示	将所有联系人按照好友与群组分两类以树状结构进行呈现。
好友请求处理	主要用于处理好友申请，选择同意或拒绝，并与服务器完成信息交互后删除该请求。
单聊	进行单人聊天。
历史消息	登录后，服务端会将历史消息发送至客户端，客户端重新渲染聊天记录。
表情包发送	可发送动态表情包。
文件发送	可发送图片等文件，发送时聊天界面中间有进度条显示进度。
添加好友/群组	输入好友名称或群组名称，服务器将返回处理结果，存在该用户则可发送好友请求。
创建群组	输入群组名称，服务器查询无重复即可创建群组。
群聊	进行群组聊天。
BBS 发帖	可进行论坛聊天。
BBS 评论	可进行论坛评论。
BBS 文件上传	可以上传文件到论坛。
BBS 文件下载	可以从论坛上下载文件。

4.2 具体功能

4.2.1 登录、注册界面

表 4.2: 登录、注册界面

截图	描述
	登录界面
	注册界面
	密码的隐藏与显示

4.2.2 处理好友申请

表 4.3: 处理好友申请

截图	描述
	添加好友（或者群组）
	同意好友请求
	左侧好友列表中，新的好友已经添加

4.2.3 单聊与群聊

表 4.4: 单聊与群聊

截图	描述
	单聊
	创建群组
	联系人树状显示

4.2.4 BBS 论坛

表 4.5: BBS 论坛 (1)

截图	描述
	进入 BBS 论坛
	点击 <i>new post</i> 按钮调出编辑窗口, 编辑窗口中的 <i>attach file</i> 按钮允许用户添加附件。编辑之后点击发送按钮即可在 BBS 界面看到新发的帖子以及附件。
	同上

表 4.6: BBS 论坛 (2)

The image is a composite of three screenshots illustrating a file-sharing forum workflow:

- Top Screenshot:** A forum post interface. A user profile picture is on the left. The post text says "hello world" and "addfriend.cpp". Below the text is a "Comment" button. A red box highlights the "comment" button.
- Middle Screenshot:** A terminal window showing a file list. The file list includes "addfriend.cpp" and "addfriend.o". A red box highlights the "addfriend.cpp" file. To the right of the file list is a "Download" button.
- Bottom Screenshot:** A terminal window showing a file list. The file list includes "addfriend.cpp" and "addfriend.o". A red box highlights the "addfriend.cpp" file. To the right of the file list is a "Download" button.

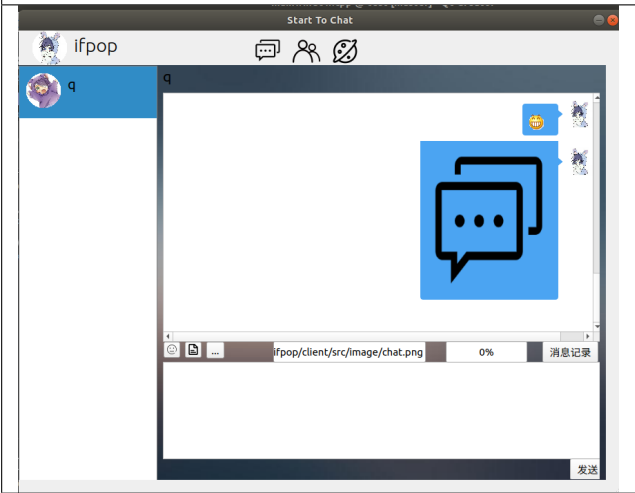
4.2.5 消息备份与历史记录

表 4.7: 消息备份历史记录

截图	描述
	在 13: 01 分 用户 <i>q</i> 向 用户 <i>w</i> 发送了四条消息, 此时 用户 <i>w</i> 没有登录上线。
	用户 <i>w</i> 在 13:21 登录。
	用户 <i>w</i> 能够点击消息记录按钮查看历史消息。

4.2.6 文件传送

表 4.8: 文件传送

截图	描述
	传送一个图片

5.1 整体架构

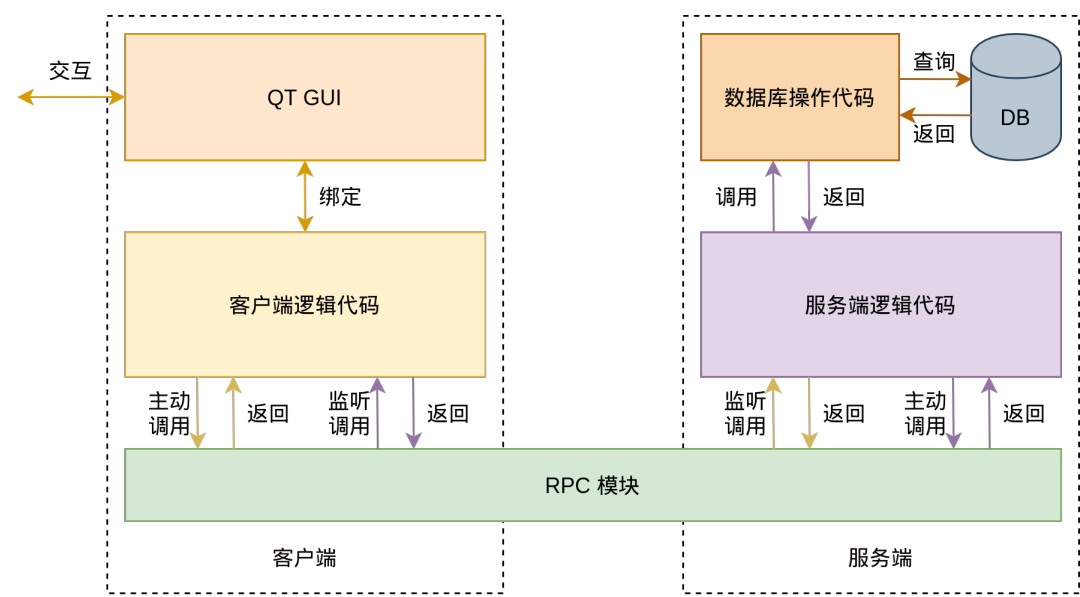


图 5.1: 系统架构

如 图 5.1 所示，系统架构分为五个主要的部分（每个模块的具体设计将在下面章节详述）：

- QT GUI 模块，与用户交互并调用底层通讯。详见[界面设计](#)。
- 客户端逻辑代码模块，提供客户端应用逻辑。详见[客户端设计](#)。
- 双向 RPC 调用模块，将网络通信封装成函数调用。详见[网络通信——FeverRPC-ng](#)。
- 服务端逻辑代码模块，提供服务端应用逻辑。详见[服务端设计](#)。

- 数据库调用模块, 提供数据持久化支持。详见[数据库的设计与建立](#)。

注意模块之间的通讯形式各不相同。

5.2 模块设计

5.2.1 界面设计

本小节讨论客户端的界面设计。

5.2.1.1 编译环境

操作系统 Ubuntu 18.04

工具 Qt 5.14.1

5.2.1.2 注册界面设计思路

该界面模仿与 TIM 的登录界面, 元素包含 logo, 登录账号输入框, 登录密码输入框, 登录按钮, 注册按钮, 注册账号输入框, 注册密码输入框、注册密码确认框以及返回按钮, 使用 Qt 自带的 ui 编辑器对其进行整理。

细节设计:

- 为了避免出现两个窗口, 使用 `QStackedWidget` 控件来生成两个页面, 使用按钮进行切换。点击注册按钮, 则出现注册界面, 点击返回按钮, 则返回到登录界面。
- 为了界面美化, 对输入框样式进行了设计, 鼠标越过以及点击出现不同颜色, 表示选中, 同时将框体的四个角变成圆弧, 更加接近 TIM 界面。
- 密码隐藏与展示。在密码输入框的右侧设置一个按钮, 来完成密码的隐藏与展示的切换。
- Logo 显示, 使用 `label` 在界面左边显示 logo。

5.2.1.3 主体界面设计思路

为了做出一个比较好的主体界面, 这里也是模仿 TIM 进行相关设计, 主界面需要展现用户的名字以及头像等信息, 然后需要三个子界面: 聊天界面, 联系人界面以及 BBS 界面。这里使用 `QtabWidget` 进行分页, 分成三个主页面。

头像以及名字 当用户成功登陆后, 服务器会返回一系列信息, 其中包括用户的账户以及头像标签, 当前端收到这些信息, 将会使用对应 `Qlabel` 进行呈现。

聊天界面设计 类似 TIM, 聊天界面左边一栏将会出现一个简略的来信信息, 右边则是聊天主体界面。对于左边来信消息的实现则是使用 `QlistWidget` 类进行呈现, 每出现一个新的用户来信, 则会在左边新建一行, 随即建立对应的聊天主体界面。

在聊天界面中, 最上面一行则是交互信息用户名, 然后接着便是消息呈现界面, 最下面则是消息发送框, 其中增设消息记录, 表情包以及文件传输三个按钮, 中间还有文件传输进度条的呈现, 显示当前文件传输状态。

联系人界面 类似 TIM，左边是联系人树状显示，另加上一个添加好友的按钮，在界面实现过程中没有像图中一样分为好友以及群组抽屉，而是放一起，也就是说，左边只有两个分组，一个是“我的好友”，一个是“我的群组”，使用 `button` 空间生成一个按钮，完成添加好友/群组功能。对于界面右边，则呈现的是加好友信息的来信提醒。

添加好友或群组 该界面仅有一个输入框，当输入需要添加的用户名称，服务器将会处理请求。

创建群聊 该界面也是仅有一个输入框，输入需要创建的群聊名称，服务器处理请求后返回结果。

好友申请界面 该界面与用户联系人界面在一起，使用 `QListWidget` 维护着一个请求列表。对于每一个好友请求，将呈用户头像、用户名以及申请理由，同时设置有同意、拒绝按钮，这些按钮绑定对应的函数与服务端进行交互，一同处理请求。值得注意的是，当点击同意或拒绝后，该请求将从列表中删除，同时更新请求列表。

5.2.2 客户端设计

本小节讨论客户端的界面逻辑设计。

5.2.2.1 功能区设计

客户端设计分为三大块：发帖和刷新按钮功能区、BBS 显示区、帖子编辑区。

功能区实现的是发帖和刷新功能，点击刷新可以获取最新发布的帖子并在 BBS 显示区显示。点击发帖按钮可以调出编辑区。

在 Qt 中，得益于 `connect` 通信机制，点击按钮可以自动新建编辑窗口。编辑窗口由 `send` 和 `attach file` 两个按钮组成的功能区和 `textEditor` 组成的编辑区组成。在编辑区运行输入任意文本；点击 `attach file` 可以调用文件系统窗口进行浏览搜索需要发送的文件。在点击 `send` 按钮之后，先发送文件到服务器，再发送帖子内容到服务器，完成整个发送过程。

点击 `refresh` 按钮之后，要求服务器返回帖子内容，BBS 客户端对内容进行解析，将其显示到界面上。客户端界面使用了 `scroll area` 实现，如果帖子数目过长过多，可通过鼠标滚轮向下翻滚。

5.2.2.2 帖子打包

由于一个帖子涉及用户名、帖子内容、附件、评论等信息，对帖子的解析就很关键。

规定一个帖子的基本内容封装如下，受益于 `html` 格式的封装的启发，依照规定进行封装。

```
<li> // 每个帖子使用 li 进行封装
    <div>1024</div> // post id
    <div>pipixia</div> // 发帖人
    <div>为了在文本框中显示字符串，我们常用这样两个函数实现</div> // 帖子内容
</div>1</div> // 附件数量
    <div></div> // 附件 ID 号
<div>测试.txt</div> // 附件文件名
    <div>pipixia</div> // 评论者昵称
    <div></div> // 评论者回复的对象
```

(下页继续)

(续上页)

```

        <div>今天好开心呀</div> //评论内容
        <div>0</div> //评论者发送的附件数量
        <div>董斌</div> //第二个评论者昵称
        <div></div> //第二个评论者回复的对象
        <div>我喜欢你</div> //评论内容
        <div>1</div> //第二个评论者发送的附件的数量
        <div></div> //附件 ID 号
        <div>爱心.gif</div> //附件文件名
    </li>
    <li>
        ... //第二个帖子的内容
    </li>

```

5.2.2.3 帖子解析

帖子内容是格式化的, 因此给帖子内容解析带来了方便。下面对帖子内容进行解析:

1. 将获取到的数个帖子 `` 封装格式进行拆分为单个帖子 * 为每个帖子生成一个 `QWidget *post`, 用于存储接下来的内容
2. 获取用户昵称, 第二个 `<div></div>` 的内容 * 每个用户信息生成一个 `QWidget *user_info` 包含用户名、用户头像 * 在用户信息的右侧添加 `comment` 按钮, 以便于对该帖子进行评论
3. 获取帖子内容, 第三个 `<div></div>` 的内容 * 生成一个 `QLabel` 保存帖子内容
4. 查询帖子附件个数, 第四个 `<div></div>` 的内容
5. 根据附件个数查询附件, 每两个 `<div></div>` 的内容表示一个附件 * 每查询到一个附件, 就生成一个 `QPushButton`, 用于响应下载请求。
6. 接下来的 `<div></div>` 的内容就是评论内容, 其解析方式与帖子解析类似。

5.2.3 网络通信——FeverRPC-ng

FeverRPC-ng 是一个基于 Socket 长连接双向 RPC 框架。

注解：FeverRPC-ng 为 FeverRPC 的重构版，FeverRPC 初版在大三上的时候完成的。

5.2.3.1 功能

5.2.3.1.1 支持的功能

- 使用 TCP/Socket 长连接
- 双向 RPC
- 支持任意长度、类型参数绑定
- 基于 MsgPack，可自定义序列化类型
- Socket 支持任意大小传输功能 (不超过 int 表示范围的字节数)
- 抛出异常

5.2.3.1.2 不支持的功能

- void 返回值
- 绑定非静态成员函数

5.2.3.2 设计目标

FeverRPC 的设计目标是解决 Socket 通信中客户端和服务端相互调用中的序列化问题。

对于一般程序来说，传统的做法是使用序列化库，每增加一个接口，就需要增加相应的转换代码和序列化、反序列化代码。然而这对于程序开发显然是个重复而低效的过程，我们可以使用更抽象的方式完成这一功能。

因此，FeverRPC 旨在为客户端和服务端提供如下的通信接口：

```
// Client.cpp

#include "feverrpc.hpp"

int main() {
    FeverRPC::Client rpc("127.0.0.1");

    // 调用远程的方法
    int ans = rpc.call<int>("add", 1, 2, 3);
}
```



```
// Server

#include "feverrpc.hpp"

int add(int a, int b, int c){
    return a + b + c;
}

int main(){
    FeverRPC::Server rpc;

    // 绑定方法
    rpc.bind("add", add);
    // 监听调用
    rpc.c2s();
}
```

可以见到, 对于服务端和客户端来说, 只需要将自己给对方提供的逻辑绑定, 然后调用者就可以像使用普通函数调用一样, 调用远程的逻辑。

为了满足丰富的函数和变量类型, 我们也需要提供 **可变长** 的参数调用。

5.2.3.3 模块架构

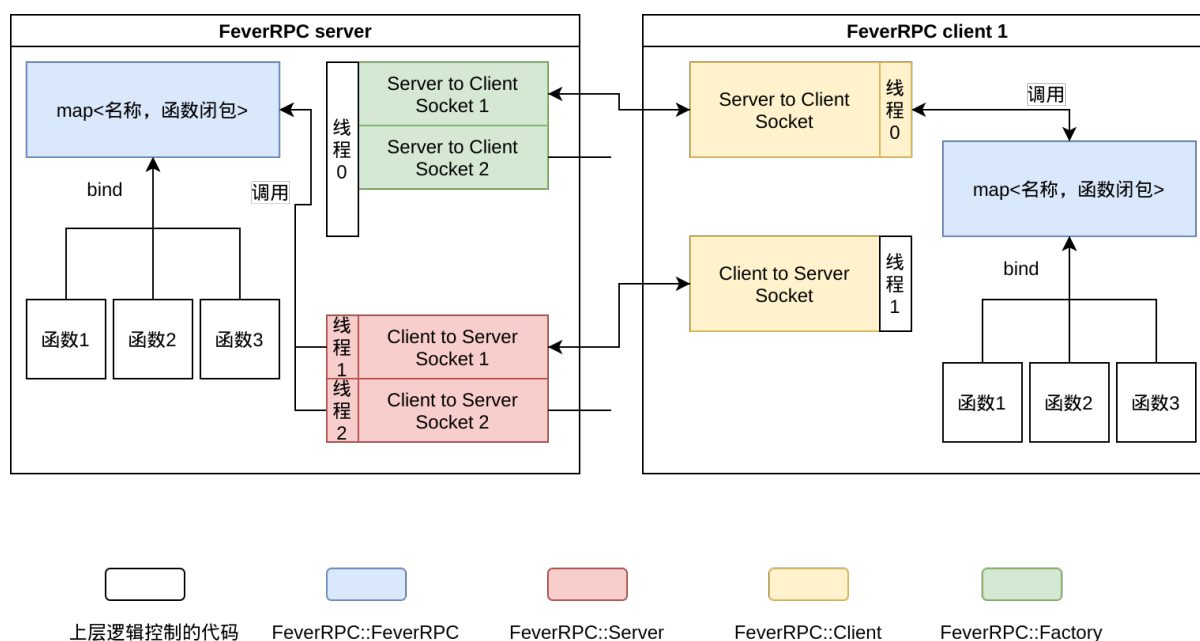


图 5.2: FeverRPC 模块架构

FeverRPC::FeverRPC 处理函数绑定和函数闭包调用的数据结构和逻辑代码

RPC 通信的本质在于事先绑定需要调用的函数, 以便在以后的任意时间取出并调用。不过如果仅仅是绑定固定参数的函数还不够, 我们需要用 C++ 来提供支持任意参数函数绑定的功能。这些都由 *FeverRPC::FeverRPC* 实现。

FeverRPC::Server 服务端用来监听所有来此客户端的调用的模块

封装了服务端监听所用客户端发起通信、主动调用的 Socket 代码。通过多线程来处理多个客户端。内部调用 *FeverRPC::FeverRPC*。

FeverRPC::Client 客户端用来监听/调用服务端的模块

封装了客户端连接服务端的 Socket 代码，并且使用一个线程监听来自服务端的调用。同时也可以主动调用服务端函数。内部调用 *FeverRPC::FeverRPC*。

FeverRPC::Factory 服务端主动调用客户端函数的句柄

封装了客户端连接服务端的 Socket 代码，并且返回给上层调用一个可以调用客户端函数的句柄。具体的使用方式由上层代码实现。

上层逻辑控制代码 客户端/服务端的逻辑代码

这些代码又上层实现，在实现的过程中，完全可以屏蔽底层的 Socket 的传输机制。在不影响功能的情况下，最大程度上解耦模块。

5.2.3.4 绑定-执行机制设计

RPC 通信需要双方首先绑定可以使用的函数，然后延迟调用。简单来说，就是使用一个类似 *map<String, Function>* 的映射来存储这些函数。

在执行的时候，使用根据相应的名称从这个映射中取出函数，并传入参数执行就可以了。

5.2.3.5 任意类型不定长参数设计

如果仅仅需要绑定和执行，直接使用强制类型转换将函数参数统一成 *void ** 的方式就行了。但是我们有一个问题出现了，在双方通信时，变量的类型是无法传递的。如果采用 *void ** 的方式只能由上层代码将类型强制转换成需要的类型——这样增加了上层代码的工作。

所以我们需要能够在无法泛型的 *map* 中存储函数的类型，也就是让类型变成可以存储的数据。这就要使用 *c++11*、*c++14* 引入的 *tuple*、变长参数模板、函数参数绑定等功能。简单来说，这些功能能让我们将参数和返回值的数据类型一同存入 *map* 中。并在需要取回的时候，通过这部分信息来反序列化传入的参数（以及序列化返回的返回值），保证了 RPC 过程中函数类型的自动转换。

注解：具体的代码比较长，详见源码。

5.2.3.6 任意长数据传送设计

为了传输不定长的数据，我们需要对于每一个发送的数据，首先传送数据大小，然后按照 1024 字节分块传送数据。接收端也按照同样的逻辑接受。

5.2.3.7 Debug 与错误处理设计

为了方便调试，我们可以采用宏定义 + 宏函数的方式自定义 Debug 函数。

```
#ifndef DEBUG
#define dbgprintf(format, args...) \
    printf("[%s,%s,%d] " format "\n", __FILE__, __FUNCTION__, __LINE__, args)
#define dbgputs(string) \
    printf("[%s,%s,%d] " string "\n", __FILE__, __FUNCTION__, __LINE__)
#else
#define dbgprintf(format, args...) \
    {}
#define dbgputs(string) \
    {}
#endif
```

这样当我们在 make 的时候传入 `-DDEBUG` flag，就能编译出打印调试信息的代码，并且每个输出都有文件名、行号、函数名。

5.2.3.8 双向调用设计

为了满足交互需求，RPC 不仅仅需要类似 HTTP 的单向无状态请求的方式；还需要提供从服务端调用客户端的能力（并且在此之上保证能够记忆客户端的身份）。

经管对于 Socket 协议来说，这一切是很好实现的，但是不要忘记我们将 Socket 完全封装了。因此，上层代码在编译期是无法提前知道这个 RPC 和哪个客户端对应。

双向调用 这需要采用两个 Socket 进行通信，分别负责客户端主动发起的调用和服务端主动发起的调用。当然，如果采用异步 IO 也可以使用一个 Socket 解决，但是这样需要严格设计消息缓冲的序列，会增加较大的工作量、增加调试难度。

有状态的服务 详见解决双向调用的身份识别问题

5.2.3.9 依赖

5.2.3.9.1 运行依赖

- msgpack
- c++17 (因为使用了 `std::apply`)
- Linux 操作系统

5.2.3.9.2 涉及的 Linux 系统调用

5.2.3.10 开发者注意事项

警告：如果你是一名开发者，这里的内容需要你认真阅读。

5.2.3.10.1 为什么你需要它

why should I use it 你能获得的优势：

在开发阶段，不在需要协商底层通信内容，只需要协商回调函数的接口（可以完全忽略网络传输的问题）。如果有新增的数据类型，不需要对底层的通信接口进行改动，只需要双方协商好字段和类型就可以。

5.2.3.10.2 解决双向调用的身份识别问题

在使用双向的 RPC 中会遇到一个棘手的问题，就是在服务端作为 Caller 的代码没有办法知道对应的 Callee 是哪个。比如当设计即时通信的服务时，自己 bind 的代码被调用了以后，并不能分清楚要将消息发给哪些 RPC 的对端。

在想了很长时间后我发现这个问题还是需要交给通信协议（交互逻辑）去实现。简单来说，只要满足以下两条要求就可以获得足够的信息，保证双向调用不会遇到身份认证问题。

1. 客户端除了 login，其他的方法必须首先传入能够标识唯一 ID 的参数信息。
2. 服务端在获得 S2C 的 RPC 连接后，会立刻调用 getID 来获取上一个要求中同样的 ID 信息。

这样，在业务上足够完成身份的绑定，以便后续的操作。

5.2.3.11 鸣谢

- 感谢 [button-rpc](#) 给予了我最关键的知识
- 其他能在搜索引擎上找到的相关教程

5.2.4 服务端设计

本小节讨论服务端的逻辑设计。

5.2.4.1 编译环境

OS Arch Linux x86_64 Kernel 5.4.2

编译工具 g++、make

5.2.4.2 功能模块

服务端含有如下功能模块

- 主控模块 - server_main.cpp
- 账户管理模块 - account.hpp - account.cpp
- 通讯模块 - im.hpp - im.cpp
- bbs 服务模块 - bbs.hpp - bbs.cpp
- 数据库通讯模块 - sql.hpp - sql.cpp

5.2.4.2.1 主控模块

采用了一个 c2s 的线程, 将提供给服务端的所有函数绑定, 并交由此线程处理来自客户端的 rpc 请求。

采用了一个线程监听客户端的 s2c 请求, 并在一个新的客户端注册/登录完成后, 请求连接时, 获取其账户名与 FeverRPC::Caller 的映射关系。以此达成 rpc call 特定客户端函数的目的。

5.2.4.2.2 账户管理模块

实现了相关函数, 处理来自客户端用户的注册、登录, 群组的创建、加入、退出, 好友的添加请求。

5.2.4.2.3 通讯模块

实现了相关函数, 处理来自客户端的单聊、群聊、发送文件、请求历史消息、请求离线消息的请求。同时, 在处理消息转发时, 自动将消息与离线消息存放于数据库中, 并提供给客户端查询接口。

5.2.4.2.4 bbs 服务模块

实现了相关函数, 处理来自客户端的发帖、回帖、查看帖子, 上传文件至云端、自云端下载特定文件的请求。同时, 对于客户端传送而来的帖子数据, 自动打包为约定的 xml 格式, 进行传输与存放至数据库。对于文件传输, 借助于 FeverRPC-ng 中采用的 msgpack, 将文件数据存放于 vector 中并以参数与返回值的进行传输。上传的文件默认保存在 server/net_disk/ 中。

5.2.4.2.5 数据库通讯模块

提供了与 ciyacaSQL 数据库的初始化与连接。处理 SQL 请求，并在请求异常时打印错误信息。

5.2.5 数据库的设计与建立

根据最初的需求分析，我们的数据库需要记录好友的申请信息、好友信息、群聊聊天记录、群聊信息、群聊的申请信息、头像信息、私聊记录、私聊信息、未读消息以及每一个账号的信息等，故在设计数据库时设计了 friend_apply、friend_info 等表，下面将叙述数据库部分重要表的详细信息。

5.2.5.1 User 表

Field	Type	Null	Key	Default	Extra
userid	int(5) unsigned zerofill	NO	PRI	NULL	auto_increment
user_name	varchar(45)	NO	UNI	NULL	
password	varchar(45)	NO		NULL	
nickname	varchar(45)	YES		NULL	
signature	varchar(60)	YES		NULL	
online_status	tinyint(1)	YES		0	
head_portrait_id	varchar(64)	YES		NULL	
VIP	tinyint(1)	NO		0	
token	varchar(32)	YES		NULL	
password_protect_id	int(1) unsigned	NO		NULL	
answer	varchar(32)	NO		NULL	

User_id 代表着账号的唯一索引，有了这一项便能够允许用户给账号创建昵称时能够任意创建而不需考虑重名的问题。

User_name 是账号的用户名，同样是账号的唯一索引。

Password 是账号的密码，最高支持 45 位字符。

Signature 是账号的个人签名，可以置空。

Online_status 是账号的在线状态，可以用来标注好友列表中的在线状态。

Head_portrait_id 记录着账号的头像，目前头像仅仅是服务器端提供一些预设的图片，不支持用户自定义。

5.2.5.2 single_chat_info 表

Field	Type	Null	Key	Default	Extra
single_chat_info_id	int(11) unsigned zerofill	NO	PRI	NULL	auto_increment
member_id1	int(5)	NO		NULL	
member_id2	int(5)	NO		NULL	

该表利用 single_chat_info_id 记录了所有有聊天记录的私聊信息以及私聊双方的 user_id，便于之后检索聊天信息。

5.2.5.3 single_chat_history 表

Field	Type	Null	Key	Default	Extra
chat_info_id	int(11)	NO		NULL	
time	varchar(64)	NO		NULL	
photo	tinyint(1)	NO		NULL	
read	tinyint(1)	NO		0	
poster_id	int(5) unsigned zerofill	NO		NULL	
content	varchar(200)	NO		NULL	

该表利用 chat_info_id 与 single_chat_info 进行连接, 可以获得聊天双方的 user_id。

time、read、poster_id、content 记录了该条信息的时间、已读情况、发送者 id 以及内容, 内容长度被设定为不超过 200 位字符。

5.2.5.4 group_chat_info 表

Field	Type	Null	Key	Default	Extra
group_chat_info_id	int(11)	NO		NULL	
member_id	int(5)	NO		NULL	
group_chat_name	varchar(45)	YES		NULL	
group_chat_admin	tinyint(1)	NO		0	

该表设置了 group_chat_info_id 作为群聊的唯一索引, 群聊管理者可以更改群聊的名称以及任命管理者。

该表将通过记录加入者的 user_id, 与 user 表进行连接后便能够获取到所有群聊成员的信息。

5.2.5.5 group_chat_history 表

Field	Type	Null	Key	Default	Extra
group_chat_info_id	int(11)	NO		NULL	
time	varchar(64)	NO		NULL	
read	varchar(45)	NO		0	
poster_id	int(5)	NO		NULL	
content	varchar(200)	NO		NULL	

该表同样利用 group_chat_info_id 作为消息记录的检索方式, 使用 group_chat_info_id 可以查询到该群聊的所有历史信息。

time、read、poster_id、content 记录了某一条信息的时间、已读情况、发送者 id 和信息的内容。

5.2.5.6 group_chat_management 表

Field	Type	Null	Key	Default	Extra
applicant_id	int(5)	NO		NULL	
group_chat_id	varchar(45)	NO	PRI	NULL	
group_chat_management_message	varchar(200)	YES		""	

该表用来记录每个群聊的申请者信息, 申请者可以通过 group_chat_management_message 来向群聊的管理者发出加入群聊的申请信息, 管理者审核通过后便将该申请者加入群聊。

5.2.5.7 friend_apply 表

Field	Type	Null	Key	Default	Extra
object_id	int(5)	NO	PRI	NULL	
applicant_id	int(5)	NO		NULL	
friend_apply_message	varchar(200)	YES		NULL	

该表用来记录用户好友申请信息, object_id 代表被申请者的 user_id, applicant_id 代表申请者的 user_id。

申请者可以通过 friend_apply_message 向被申请者发出好友申请信息, 被申请者审核通过后便可以添加为好友。

6.1 解决双向 RPC 设计中可变长模板参数的设计

6.1.1 难点

如何不仅传递数值信息，并且传递类型信息以及根据类型信息自动反序列化，是这个 RPC 的最大难点之一。

可变长参数模板的概念实际上是超出一个普通 C/CPP 使用者的认知的。因为对于传统的编译型语言来说，人们的认知都是：“除了虚函数以外，很难实现运行时多态”。

比如 C 语言中的 `printf`，虽然支持可变长参数，并且看上去类型任意。但其实这些参数的类型信息在传递的时候已经被丢弃了，只按照大小压栈。这也是为什么前面还需要用控制字符串来说明当前变量的信息类型。

而只有在动态语言中，才会有 `print(1, "a", [1,2,3])` 这种不需要提供参数类型信息的方式。（`cout << 1 << "a";` 这种相当与多次调用同一个函数，不在我们讨论范围之内）。

所以一开始为了实现我们的动态的 RPC 调用接口，我花了很长时间去考虑该选择怎样的技术来完成它。也就是，类型信息怎么在网络通信中传输。

原始的方式之所以总需要根据接口修改，就是因为双方通信的时候并没有传递类型信息；或者说这部分类型信息没有在接收方被正确利用，所以需要双方实现约定好格式的序列化和反序列换。但是我们的希望是：传递类型信息，并且在接收方动态地反序列化（我们不手动判断）。

6.1.2 解决方法

还好有可变长参数模板这一概念, 以及 C++11/14/17 中对类型提供的支持, 它能够让我们使用 `std::tuple` 类型来装载 `Template` 中的类型信息。并且通过多级的模板参数展开, 我们能够还原其中的类型。

然而这一系列知识以及代码实现还是非常复杂, 我用了大概 3 天 × 4 个小时的时间写了两遍, 并且参考了一些网上对于这部分知识的实验性代码, 才最终有了今天这版 RPC 的动态参数调用实现。

6.2 解决双向 RPC 调用中身份认证的问题

6.2.1 难点

这版 FeverRPC-ng 是基于大三上完成的初版的进一步重构。在我大三上完成整体设计的时候, 就已经因为这个问题十分头痛。

身份认证的问题是這樣的: RPC 调用依赖实现注册的函数, 服务端和客户端相互调用属于两个线程。而如果想在服务端绑定的函数中调用某个具体的 Server to Client 的 RPC 句柄, 就会遇到无法确认具体是哪个句柄的问题。

最初为了解决这个问题。我曾经试图把认证逻辑封装在了 RPC 之中, 但是事实证明这样限制了灵活性, 也让 RPC 承担了更复杂而多余的逻辑功能。

6.2.2 解决方法

既然代码层面不好约束, 在很长时间思考以后, 我最终选择了从协议层面入手。

也就是说使用这个 RPC 的上层代码要满足以下的协议要求:

1. 客户端除了 login, 其他的方法必须首先传入能够标识唯一 ID 的参数信息。
2. 服务端在获得 S2C 的 RPC 连接后, 会立刻调用 getID 来获取上一个要求中同样的 ID 信息。

这样能够保证, 首先建立连接的时候服务端就能将句柄和 ID 一一对应, 然后在之后的通讯中, 客户端都会提供 ID 来保证服务端能够区分。

这样这个问题就能够比较完美的解决 (除了增加性能开销, 不过不大)。

6.3 数据库设计方面

作为一名数据库设计与开发者, 体会到了在实际的项目开发中, 数据库作为最为基础的一个功能, 在各个功能的实现中都有涉及。在数据库设计的过程中, 我遇到了各种各样的问题, 比如各个表间同一字段取名不同导致使用不方便、主键设置不当导致无法插入多条聊天记录……让我体会到设计出一个稳定的数据库是一件非常困难的事情。

6.4 对接问题

6.4.1 难点

在客户端设计与后端设计分离的情况下验证客户端开发是否正确就成了一个问题，同时当后端开发完毕在进行对接过程中会存在细节对接不上情况。

6.4.2 解决方法

采用软件测试中打桩的方法，在需要借助后端进行通信的模块一律转为对文件的操作，这样可以向外留有对接接口，在与后端的对接的过程中，只需要将数据流从文件转为与后端的 socket 连接的数据流即可。同时，在设计过程中会先于后端商议好信息交流的方向和过程，这样即使在实际开发中由于开发实际情况原因使得开发接口改变也只需简单修改接口即可，无需对内部实现函数进行大的修改，因为实际的处理内容是一致的。

6.5 BBS 呈现问题

6.5.1 难点

一般的 BBS 都是由网页开发的。由于老师限制了必须使用 C 进行开发，因此在数据的客户端显示过程中遇到了很多困难。比方说如何显示内容，如何实现数据解析，如何使大量数据在窗口中通过滚动实现顺滑呈现等。

6.5.2 解决方案

由于使用了 QT 的框架进行开发，在一定程度上减小了困难程度。

1. 限制呈现的内容

起初考虑呈现富文本，但是不使用 HTML 开发渲染的话工作量巨大。因此将数据降维，仅呈现简单的文本信息，附件也以文本链接的形式呈现。

2. 良好的数据封装

BBS 具有丰富的信息：发帖人、评论人、附件、内容等，因此需要对数据进行良好的组织。对数据进行严格的 xml 格式封装，确保了在信息处理过程中不会出现失误。同时提供了良好的拓展性

3. 帖子与附件分离

帖子与附件采用不同的数据库进行存储，帖子仅包含指向对应文件的唯一标识，这样当需要下载附件时，通过该唯一标识到数据库检索下载即可。

4. scroll area

由于帖子内容可能比较多，但是窗口有限，必须要实现翻页功能。起初设计时采用 *QListView* 组件呈现，但是具有明显的撕裂感，观感很差且实际向下翻动效果不好。后更换为 *QScrollArea* 组件进行呈现，达到了类似于网页向下翻滚的效果。

6.6 服务端设计问题

由于将 socket 的维护与客户端的连接请求全部交由了 FeverRPC 管理, 致使服务端上层代码难以获取到客户端的用户标识 (username)。经过组内商议, 最后的解决方案是, 在用户注册或登录, 与数据库交互返回成功后, 客户端申请与服务端的连接时, 服务端主动 rpc call 客户端函数, 获取客户端的用户标识, 并将其与 FeverRPC::Caller 的映射保存于一个 map 中, 作为在线用户的管理结构。

分工和人员合作

7.1 分工

表 7.1: 成员分工

姓名	学号	分工
董斌	1120173585	负责客户端界面的设计和客户端的开发工作，相关文档撰写
冯开宇	1120171224	负责网络通信框架 FeverRPC-ng 的开发和维护，相关文档撰写和文档排版，以及项目计划的制定和督促工作
李昌昊	1120173304	李昌昊负责数据库的设计与建立，以及相关 Mysql 语句编写，相关文档撰写
王占坤	1120170124	负责部分客户端界面的开发和全部客户端接口的编写工作，相关文档撰写
张剑威	1120173586	张剑威负责后端接口的编写以及各个成员之间的协调工作，相关文档撰写

7.2 合作

我们小组在项目开始通过小组会议的方式实现了对项目的大体分工。

在确定好分工后, 会定期开展会议来分享大家的进度以及遇到的问题。同时, 小组内部部分成员会不定期讨论来决定一些细节问题, 比如接口设计问题, 并通过及时沟通解决开发上遇到的冲突。我们项目实现了成员职责的明确界定, 项目的每个部分都有对应的负责人。我们重视人际协调和沟通, 每个人遇到问题, 我们大家都会想办法一起解决。通过我们良好的人员合作, 我们最终圆满完成了这个项目。

7.2.1 会议记录

7.2.1.1 三月九日

主要内容 讨论项目的技术栈, 以及明确需求。

讨论结果 我们主要对技术栈进行了分析和选择, 比如

- 考虑使用 Rust 来作为后端服务器, 稳定性更好
- 使用 Github 进行代码协作
- 使用 QT 进行客户端开发
- 使用 doxygen 进行 C++ 文档管理
- 使用 PostgreSQL

在使用 Rust 的方面我们进行了比较多的讨论, 主要是后端的 Rust 其实封装 Linux API 比较多, 不太好直接操作 api (因而不是很符合老师的要求); 但是如果采用 C++ + WebSocket 的话, 其实自己操作 IO 的部分也很少 (会使用大量的库), 所以感觉 Rust 并没有不让用。

7.2.1.2 三月十六日

主要内容 确定分工、各自的任务以及下一步的结果。

讨论结果 一开始还是在后端语言的问题上纠结, 然后再次向老师确认以后, 发现只能用 C/C++ 了。

7.2.1.3 四月中旬

进度

- GUI: 已经有大致的窗体
- 数据库: 已经有表了
- RPC: 差不多了, 可以进行双向的调用

更多的分析

- 可以尝试 SQLite 减小运行的要求
- 文档可以开始做了

- 前后端的逻辑代码可以开始完成了

7.2.1.4 五月上旬

主要内容 是分享目前进度。

客户端与服务端统一接口。

后续的会议大多以两、三位成员的规模进行召开，涉及到比较细节的问题。因此不再记录。