1. **项目目的**

利用编码技术等为用户提供一个远程安全存储图片的服务。

1. **系统功能概述**

客户端：

（1）登录、注册：用户根据用户名和密码登录，没有账号的用户可以用未注册的用户名进行注册，用户名是登录唯一标识。

（2）选择上传的图片文件：用户单击选择文件后弹出文件选择框选择图片文件进行上传，此时已选择的文件会展示在界面列表展示框中。

（3）上传图片：用户单击上传图片后，客户端自动清楚路径目录信息并以文件名传输所有图片到远端服务端，并清空列表。

（4）刷新云列表：用户单击刷新后展示已有云端的图片列表信息到用户界面。

（5）查看图片：双击云列表某张图片直接在窗口中展示该图片内容。

服务端：

（1）验证登录信息：查询来自客户端的登录请求是否符合数据库用户表要求

（2）验证注册信息：验证用户的用户名是否已被使用，若未使用啧在数据库新建用户信息

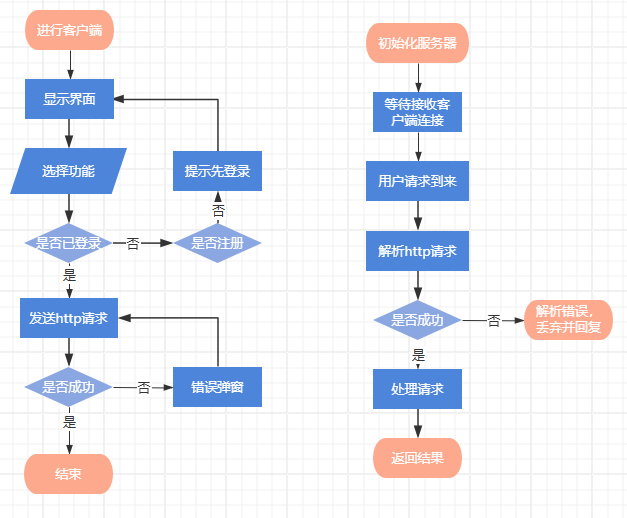
（3）处理图片上传请求：建立用户和数据库的连接，把远端用户的图片信息存储入数据库

（4）查询用户图片列表：通过数据库连接查询数据库用户的图片信息并返回客户端

（5）查询用户图片：根据用户双击图片名字返回用户点击图片内容

1. **整体框架流程**

框架流程图（左客户端、右服务端）：

****

1. **需求梳理**

**客户端主界面：**

用户打开界面，底层建立tcp连接，若连接超过两秒未打开或服务端没打开会进行弹窗提示服务端离线。

**客户端用户登录/注册：**

用户界面需提供登录注册按钮和用户名和密码输入框，用户输入用户名和密码后点击登录或注册按钮，此时底层读取用户界面的输入数据并进行json数据的建立，而后对得到的数据进行base64编码，编码完成后向底层的socket写入数据发向服务端。

**服务端用户登录/注册校验：**

服务端接受数据后从连接池获取数据库连接，查询数据库是否存在该用户名的账户，并解析两者的密码是否相等，如若不相等或无用户，则返回密码错误或用户不存在信息，如用户名密码正确，返回登录成功消息；注册时若查询数据库中存在重复账户名，则返回注册信息已存在的消息

**客户端选择待上传文件：**

用户单击选择文件按钮后，会首先检测是否登录，若未登录则不允许继续操作，否则弹出文件选择框，用户可选择一个或多个后缀为png或jpg格式的图片文件，客户端会读取图片路径信息并反馈于用户界面的已选择文件列表

**客户端上传图片：**

用户在已选择文件的前提下点击上传图片，客户端会提取出所有的文件名和图片数据并进行aes128编码再生成对应的json数据后进行base64编码，将数据发送到服务端

**服务端存储用户图片：**

服务端根据用户的请求消息解析码base64并解析json，并通过连接池获取数据库连接，从json中提取用户名、图片名、图片数据等内容，并通过数据连接池把图片信息写入数据库，并建立相应的图片用户与图片数据索引表。

**客户端获取云端列表：**

客户端提供刷新按钮，在单击刷新时会先检验是否登录，若未登录则弹窗提示并终止请求，若已登录则准备好请求json数据向服务端发送

**服务端处理获取云列表请求：**

服务端接收到一个云列表获取请求后，将获取连接池中的连接，并查询数据库的用户图片索引表，返回与申请用户相关的所有图片名

**客户端请求图片信息：**

客户端双击云端列表的某图片的信息，客户端读取选取的索引，将用户名信息和图片名打包成json数据经过base64编码后发送给服务端。

**服务端返回图片：**

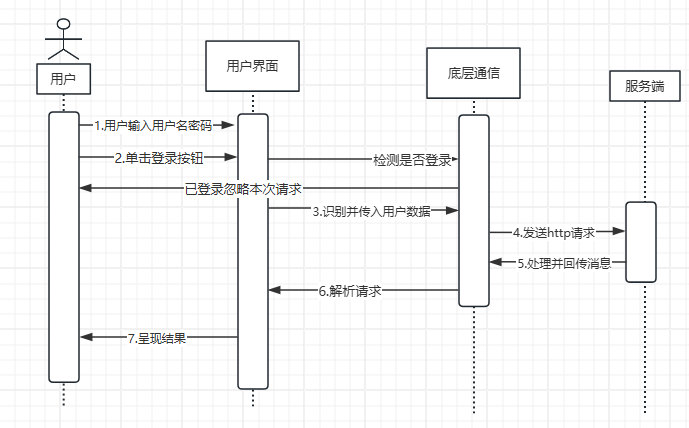
服务端接受到用户图片显示请求后，连接数据库查询相应的用户信息，将得到的图片信息打包成json数据并编码后返回

**客户端展示图片**：

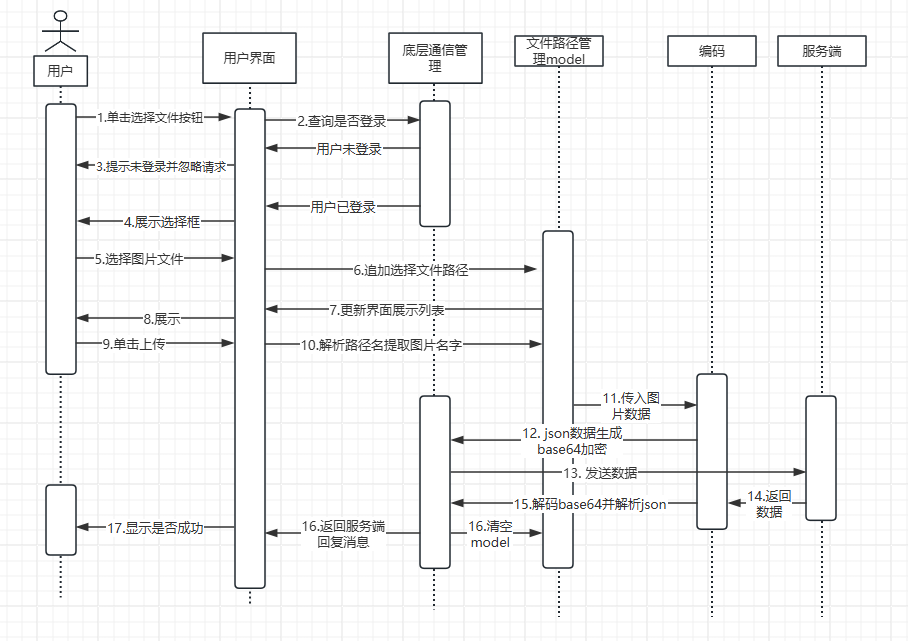
在客户端双击后得到了图片的回复信息，客户端对图片数据进行base64解码再使用key进行aes128进行解码得到图片数据，此时利用openGL展示图片。

1. **功能时序图**

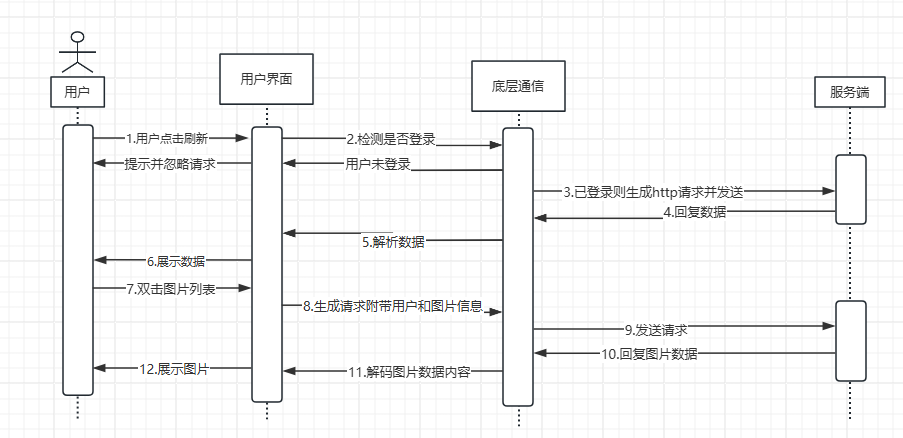
登录/注册时序图



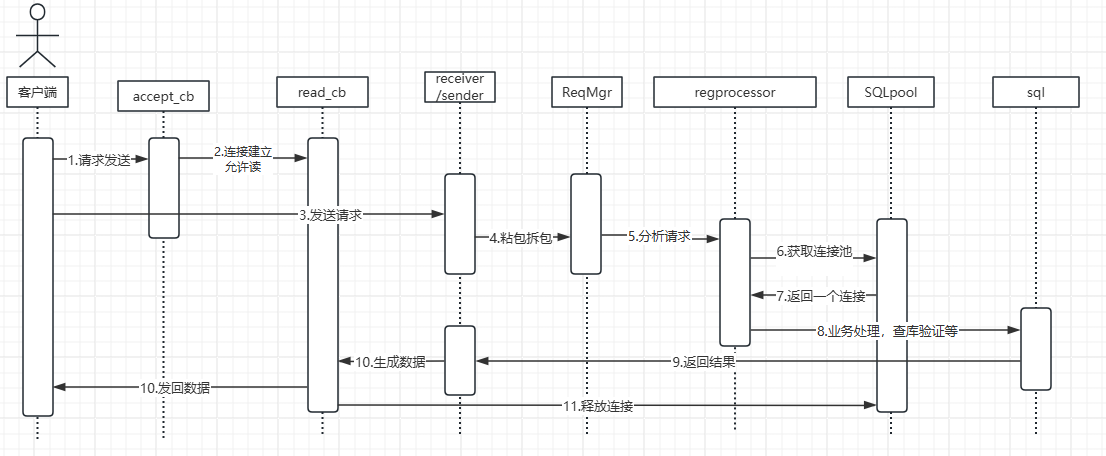
选择文件并上传图片时序图：



获取云列表并显示图片时序图：



服务端处理请求时序图：



1. **通信数据协议：**

登录或注册请求：

POST /login或/signup HTTP/1.1

Content-Type: 数据类型

{

"username":"",

"password":""

}

登录成功/注册成功：

HTTP/1.1 200 OK

登录失败：用户名不存在或者密码错误 注册失败：用户名已存在：

HTTP/1.1 403 Forbidden

Content-Type application/json

{

错误

}

登录出错：json格式不对

HTTP/1.1 400 Bad Request

上传图片请求：

POST /upload HTTP/1.1

Content-Type: 数据类型

{

"username":"",

"imgname":"",

"imgdata":""

}

服务端回复上传图片成功：

HTTP/1.1 200 OK

上传图片失败：

HTTP/1.1 500 Internal Server Error

HTTP/1.1 400 Bad Request

Content-Type application/json

{

Errormsg

}

用户请求云端列表：

GET /getlist?username=user HTTP/1.1

服务端云端列表回复：

HTTP/1.1 200 OK

Imglist

请求失败：用户未登录：

HTTP/1.1 401 Unauthorized

用户请求图片数据：

GET /getimg?username=user&imgname=img HTTP/1.1

服务端回复成功：

HTTP/1.1 200 OK

Content-Type image/png

Data

失败：

HTTP/1.1 Bad Request

Aes128图片加密key：

"aes128key":"FileStoreService"

1. **技术参考**

|  |  |
| --- | --- |
| 网络协议 | http通信 / epoll多路复用 |
| 数据库 | Mysql  Mysqlcppconn库：处理数据库连接 |
| 后端 | Libevent事件驱动库  c++ windows/linux  Cmake 3.22.1 |
| 前端 | Qt6.7.2  QTcpSocket |
| 加密库： | Base64编码  Aes128加密  Cryptopp加密库 |
| 其他 | Jsoncpp库：解析编制Json数据  QJsonObject |
| 部署 | Docker  Docker compose  Gitlab-ci/cd自动化部署 |

1. **客户端框架及主要接口描述**

Ui界面：



主窗口类设计：

完成了以下几件事情：

1. 负责把ui界面绑定到本窗口类并使得main能展示

2. 与服务端建立tcp连接

3. 配置ui界面：

1. 设置密码掩码：·

2. 绑定列表的双击事件

QJsonObject getInput()

获取用户在ui界面的输入和输出，返回的是json数据包

按钮以及事件：

void on\_pushButtonLogin\_clicked();              // 按下登录按钮

void on\_pushButtonSignUp\_clicked();             // 按下注册按钮

void on\_pushButtonSelectFile\_clicked();         // 按下选择文件按钮

void on\_pushButtonUpload\_clicked();             // 按下上传文件按钮

void on\_pushButtonFlush\_clicked();              // 按下刷新按钮

对于登录和注册按钮回调：其伪代码如下

on\_pushButtonLogin\_clicked():

获取ui界面输入:json = getInput()

检验输入格式;

if json == null:

    提示错误

记录姓名

调用底层socket处理数据

对于选择文件按钮回调，其伪代码如下

on\_pushButtonSelectFile\_clicked()

if 未登录

    警告并return

获取当前路径位置和当前已选择的路径列表

文件框弹出选择文件框

将选择的文件路径追加到路径列表

更新ui显示

选中ui最后一行

对于上传按钮回调，其伪代码如下

on\_pushButtonUpload\_clicked():

if 未登录

    警告并return

for i in 路径列表:

    打开 路径[i] 的文件

    读取图片数据

    aes128数据加密

    准备并发送json包

清空ui列表

对于刷新云列表按钮，其伪代码如下

on\_pushButtonFlush\_clicked():

if 未登录:

    警告并返回

向服务端请求云端图片列表

客户端接收到回复后发送信号函数 以下函数触发

void MainWindow::updateCloudList(QStringList imgs)

将获取的列表展示到ui

选中最后一行

用户双击图片获取图片数据：

doubleClickOnCloudListView():

收集图片用户数据

底层socket发送

底层socket类ClientWebSocket

其主要复制：

生成http 请求体

发送数据

解决tcp粘包和拆包问题

发数据流程：

1. 绑定处理数据包的事件
2. 生成http数据包
3. 数据包头处理
4. 发送数据
5. QObject::disconnect(m\_tcpsock, &QTcpSocket::readyRead, this, &ClientWebSocket::processImgDataMsg);

建立起信号槽函数的连接，用于处理数据

1. QString httpReq = QString("POST /upload HTTP/1.1\r\nHost127.0.0.1:8080\r\nContent-Type: application/json\r\n\r\n");

QJsonObject httpBody;

httpBody["imgdata"] = QString::fromUtf8(msgdata);

httpBody["username"] = username;

httpBody["imgname"] = imgname;

QJsonDocument jsonDocument(httpBody);

httpReq.append(jsonDocument.toJson());

准备类似的http数据包

1. 包头处理

// 加入魔数和数据大小头部

uint32\_t size = str.size();

size = htonl(size);

uint32\_t magic = htonl(17171717);

str.prepend(reinterpret\_cast<const char\*>(&size), sizeof(size));

str.prepend(reinterpret\_cast<const char\*>(&magic), sizeof(magic));

流程为：

计算数据大小

插入数据包头大小+魔数

1. Tcp发送数据

数据处理流程：

1. 读取数据
2. 解析http返回包
3. 展示或忽略

QbyteArray readMsg()

当tcp端口有数据可读时，读取数据并负责拆包分包，读取足够的一次返回数据提高给上层使用

伪代码如下：

while(可读数据<12)：

    等待

读取12个数据作包头

if 前8个数据 != 魔数

    丢包

读取包头中的数据长度大小datalen

while(可读数据长度<datalen):

    等待

读取datalen字节数据

base64解码

返回数据包

图片数据处理:

void ClientWebSocket::processImgDataMsg()

伪代码

调用读数据包读取数据 readMsg()

if status != 0

    图片获取有错

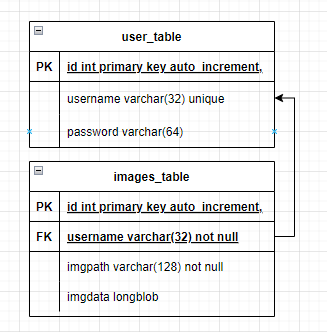
获取图片数据

base64解码

aes128解码

利用imgbox显示图片

1. **数据库设计**



User\_table为用户表，用户信息通过此表查询

id和username为唯一标识

create table if not exists user\_table(

    id int primary key auto\_increment,

    username varchar(32) unique,

    password varchar(64)

);

Images\_table为图像表，用户的所有图片信息存储于该表，imgpath为图像存储的路径

Id为图像唯一标识

create table if not exists images\_table(

    id int primary key auto\_increment,

    username varchar(32) not null,

    imgpath varchar(128) not null,

    imgdata longblob,

    foreign key (username) references user\_table(username)

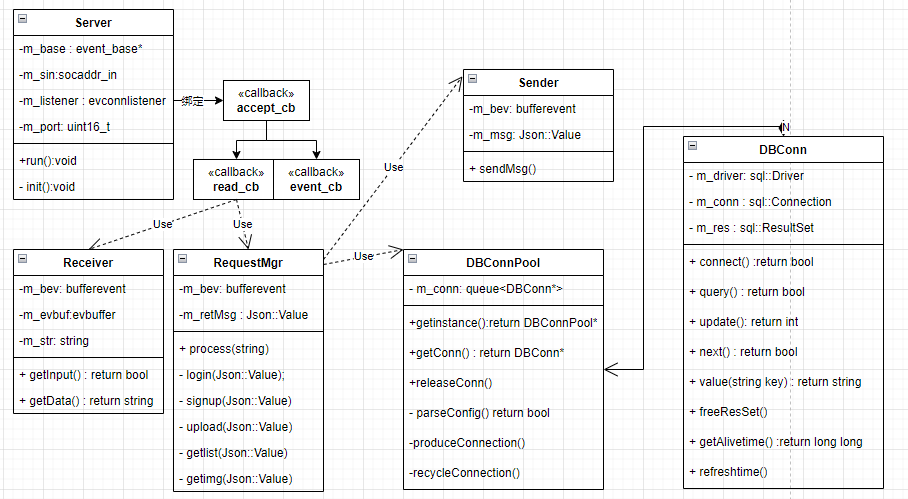
);

服务端默认连接数据库配置

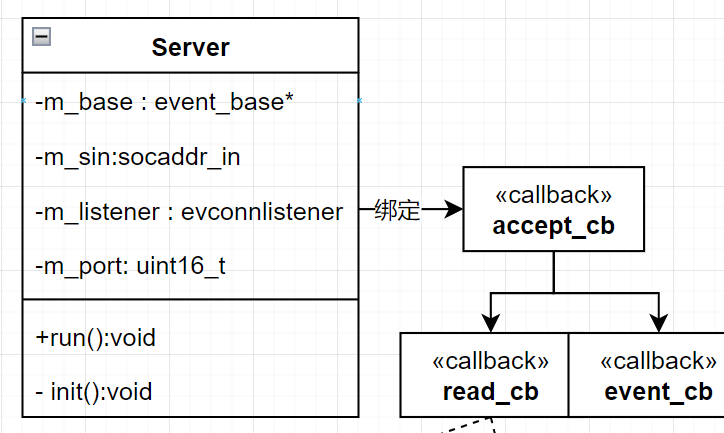
用户名:mgr

密码:123456

1. **服务端UML类图及接口描述**



服务器类：



/\*\*

 \* @class 服务器

 \*/

class Server

{

    friend void accept\_cb(evconnlistener \*listener, evutil\_socket\_t fd,

                            sockaddr \*addr, int socklen, void \*arg);

public:

    /\*\*

     \* @brief 默认构造函数，以默认配置构造服务器

     \*/

    Server();

    ~Server();

    /\*\*

     \* @brief 服务器对象运行

     \*/

    void run();

private:

    /\*\*

     \* @brief 服务器对象初始化

     \*/

    void init();

private:

    struct event\_base \*m\_base=nullptr;              // 事件集合

    ThreadPool \*m\_pool=nullptr;                     // 线程池指针

    struct sockaddr\_in m\_sin={0};                   // 服务端socket

    struct evconnlistener \*m\_listener=nullptr;      // 监听器指针

    uint16\_t m\_port = 8080;

};

Accept\_cb：监听器监听到连接的回调：

/\*\*

 \* @brief 接受回调函数

 \* @param[in] listener 监听器

 \* @param[in] evutil\_socket\_t socket描述符

 \* @param[in] addr socket结构指针

 \* @param[in] socklen socket长度

 \* @param[in] arg 将要传给读写回调函数的参数

 \*/

void accept\_cb(evconnlistener \*listener, evutil\_socket\_t fd, sockaddr \*addr, int socklen, void \*arg);

其主要负责创建对应客户端的eventbuffer以及设置对应的回调函数

Read\_cb：客户端发送数据，服务端检测到可写的回调

/\*\*

 \* @brief 读回调函数

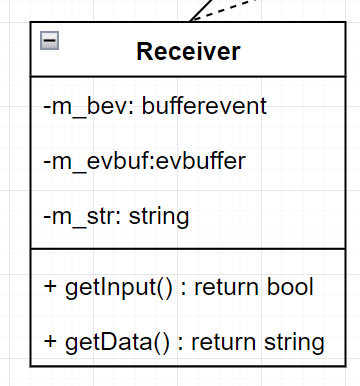
 \* @param[in] bev 事件

 \* @param[in] arg 接收回调函数所传参数

 \*/

void read\_cb(bufferevent \*bev, void \*arg);

其主要负责创建接收器接收并解码数据，然后创建需求管理器处理用户的需求



Receiver类：

/\*\*

 \* @class 接收器类

 \* @brief 负责接收并解码socket的数据

 \*/

class Receiver

{

public:

    /\*\*

     \* @brief 有参构造

     \*/

    Receiver(struct bufferevent \*bev);

    ~Receiver();

    /\*\*

     \* @brief 从底层读取输入

     \* @return 读取成功返回true，仍未能读取返回false

     \*/

    bool getInput();

    /\*\*

     \* @brief 获取读取的数据

     \* @return 解码的数据串

     \*/

    std::string getData();

private:

    /\*\*

     \* @brief 检查包头魔数信息

     \* @param[in] header 包头数组指针

     \* @return 包头正确可读返回true

     \*/

    bool checkHeader(const char \*header);

    /\*\*

     \* @brief 检查包头数据长度，判断数据是否达到可读条件

     \* @param[in] header 包头数组指针

     \* @param[in] len 已有数据长度

     \* @return 数据可读后读入m\_str并返回true

     \*/

    bool checkLen(const char \*header, size\_t len);

private:

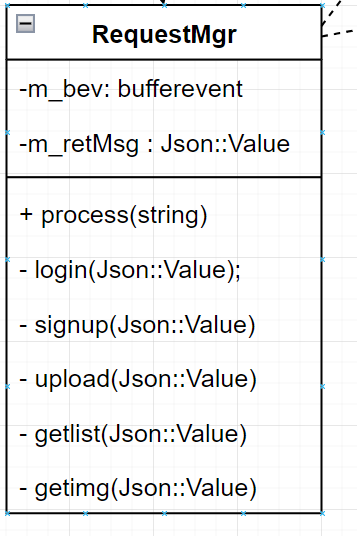
    struct bufferevent \*m\_bev;      // 事件集

    struct evbuffer \*m\_evbuf;       // 输入缓冲区

    std::string m\_str;              // 解析出来的字符串

};

RequestMgr类：



/\*\*

 \* @class 请求管理

 \* @brief 在读回调函数中处理来自客户的请求，并执行相应函数

 \*/

class RequestMgr

{

public:

    /\*\*

     \* @brief 构造函数

     \* @param[in] bev 事件集

     \* @param[in] str 来自客户的未解析字符串

     \*/

    RequestMgr(bufferevent \*bev);

    ~RequestMgr();

    /\*\*

     \* @fn 处理

     \* @brief 解析并处理用户的请求

     \*/

    void process(std::string reqstr);

private:

    /\*\*

     \* @brief 处理登录请求

     \* @param[in] data 已解析用户附带数据集

     \*/

    void login(Json::Value data);

    /\*\*

     \* @brief 处理注册请求

     \* @param[in] data 已解析用户附带数据集

     \*/

    void signup(Json::Value data);

    /\*\*

     \* @brief 处理上传图片

     \* @param[in] data 已解析用户附带数据集

     \*/

    void upload(Json::Value data);

    /\*\*

     \* @brief 处理用户获取图像列表请求

     \* @param[in] username 申请的用户名

     \*/

    void getlist(const std::string &username);

    /\*\*

     \* @brief 处理用户获取图片请求

     \* @param[in] data 已解析用户附带数据集

     \*/

    void getimg(Json::Value data);

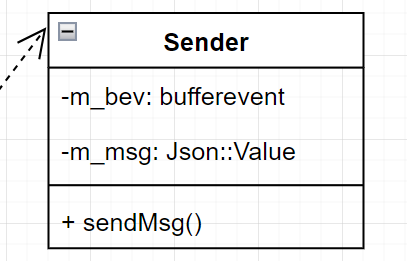
private:

    struct bufferevent \*m\_bev = nullptr;          // 事件集

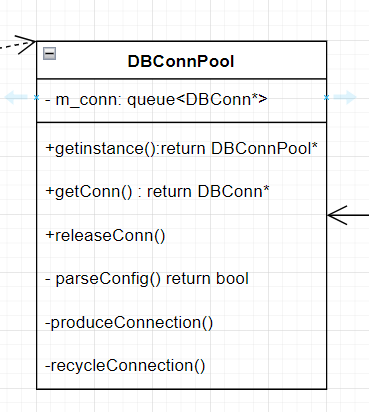
    Json::Value m\_retMsg;

};

RequestMgr内有Sender发送器负责编码并回复客户端数据：



数据库连接池类DBConnPool:

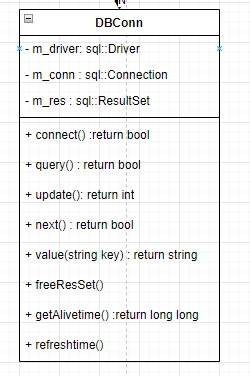


单例模式

生产连接线程

回收连接线程

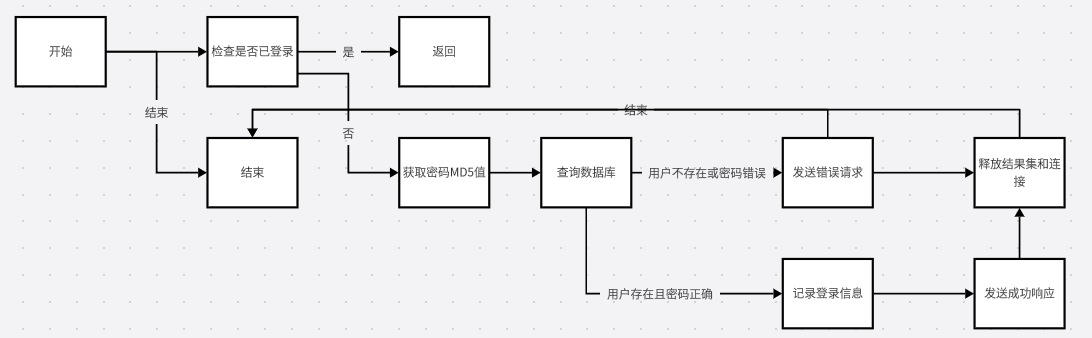
数据库连接类：



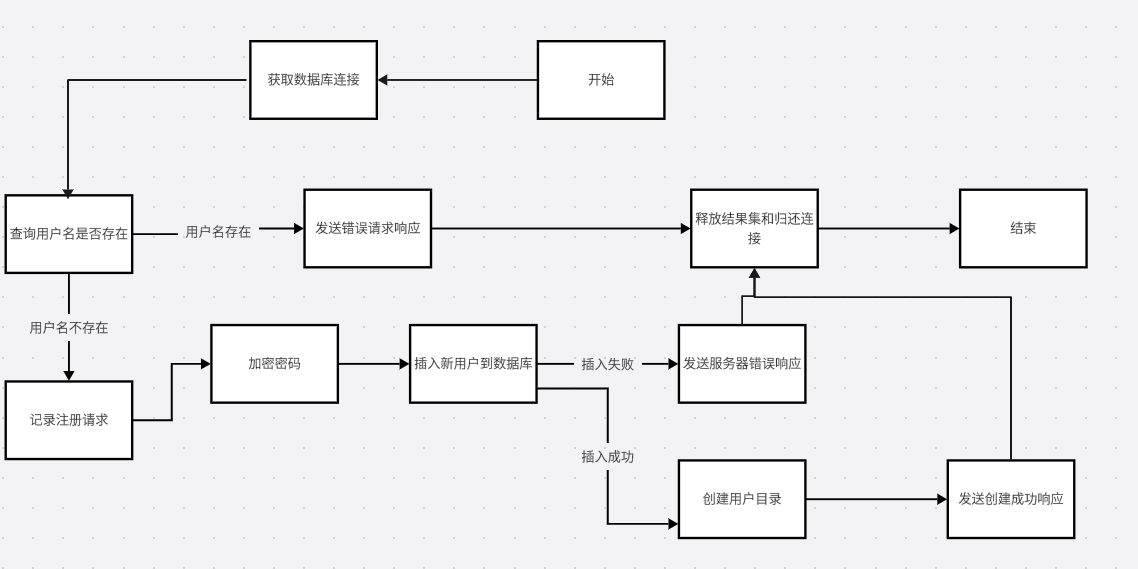
封装mysqlcppconn连接

1. **服务端业务处理流程图**

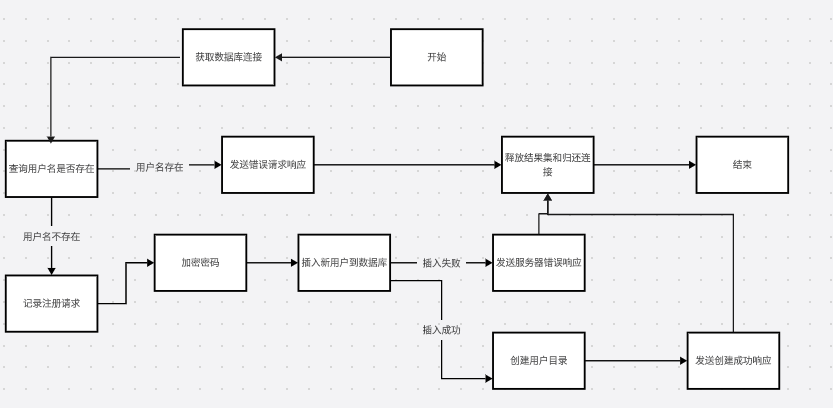
登录：



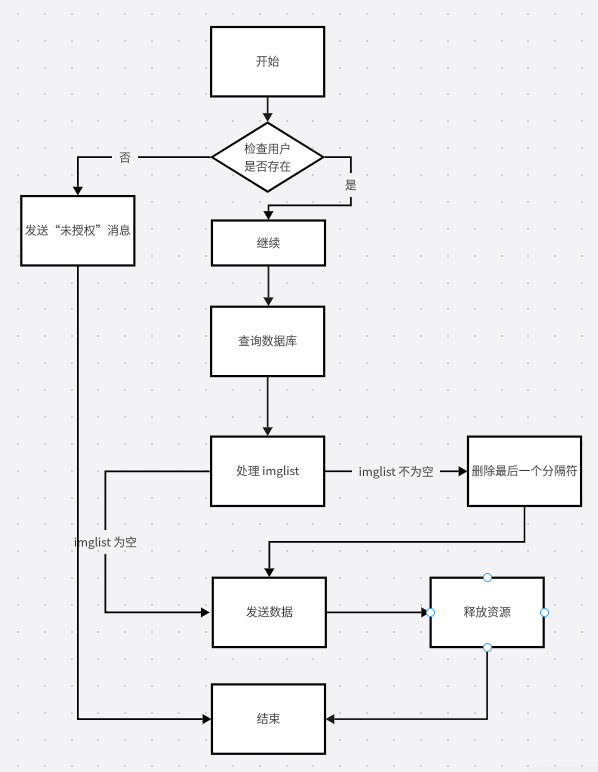
注册：



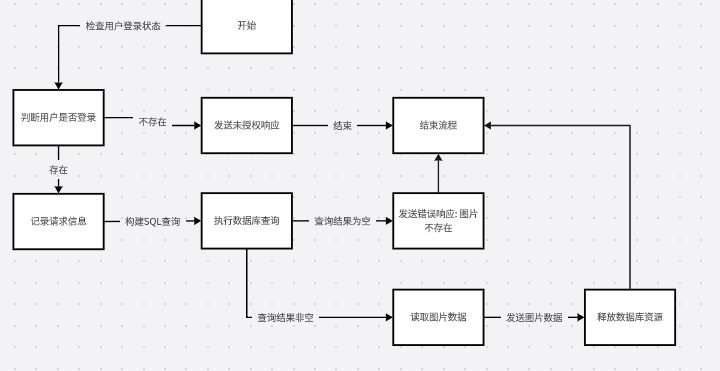
上传图片：



获取云端列表：



获取图片流程图：



1. **项目部署**

Docker部署gitlab私服：

Docker安装运行命令：

sudo docker run --detach \

--hostname 192.168.245.130 \

--publish 443:443 --publish 80:80 --publish 2222:22 \

--name gitlab \

--restart always \

--volume /home/ubuntu/project/gitlab/config:/etc/gitlab \

--volume /home/ubuntu/project/gitlab/logs:/var/log/gitlab \

--volume /home/ubuntu/project/gitlab/data:/var/opt/gitlab \

--shm-size 256m \

registry.gitlab.cn/omnibus/gitlab-jh:latest

随后进入/etc/gitlab/gitlab.rb:

修改extern\_url:http://192.168.245.130/

Gitlab\_shell\_ssh\_port:2222

登入192.168.245.130：

修改密码：初始密码在/etc/gitlab/initial\_root\_password中

Gitlab-runner安装：

curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh" | sudo bash

sudo apt-get install gitlab-runner

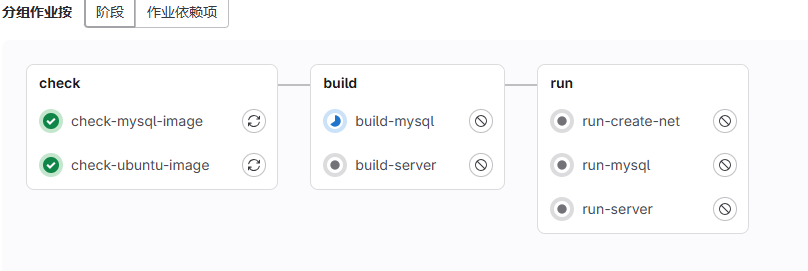
注册runner：

gitlab-runner register

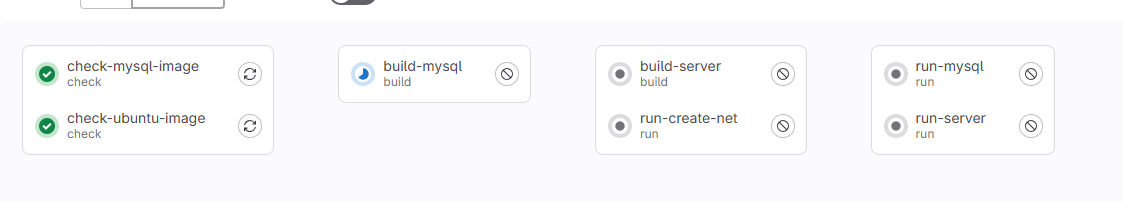
url:http://127.0.0.1

token:在gitlab网页上申请创建runner可以获取token

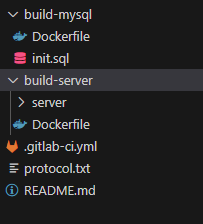
执行器:shell

流水线作业流程图：

作业依赖关系：



部署结构：



build-mysql是构建mysql的文件夹：

Dockerfile是构建mysql镜像的docker文件，它以mysql:8为镜像，把init.sql加入到mysql的初始化中，在启动mysql时会自动初始化

build-server是构建ubuntu-server的文件夹

Dockerfile是构建server镜像时会使用其中的docker文件

server/下是服务端的源代码，由docker build时加入到镜像中

protocol.txt是客户端和服务单进行的通信协议

.gitlab-ci.yml即流水线的脚本文件，当我们对代码进行修改进行一次push上传到仓库时会触发本地gitlab-ci/cd仓库的流水线部署，流水线就会自动进行镜像的拉取、镜像的构建、镜像网络的创建、mysql和服务器的启动

以下是流水线中定义的阶段和其任务

check阶段：

检测本机是否有mysql:8 以及ubuntu:latest镜像，若有则会跳过，若没有则会拉取，后续会以这两个镜像为基础构建mysql镜像

build阶段：

创建mysql镜像，初始化mysql表格，以及初始化账号密码 tags指定runner

build-mysql:

  stage: build

  image:

    name: mysql:8

    pull\_policy: if-not-present

  tags:

    - testrunner

  script:

    - echo "构建 mysql 镜像"

- docker build -t mysql:v1 ./build-mysql

其Dockerfile：

FROM mysql:8

ENV MYSQL\_ROOT\_PASSWORD=123456

ADD ./init.sql /docker-entrypoint-initdb.d/

EXPOSE 3306

下一步：

构建服务器的ubuntu镜像：

build-server:

  stage: build

  image:

    name: ubuntu:latest

    pull\_policy: if-not-present

  tags:

    - testrunner

  script:

    - echo "构建 ubuntu 镜像"

- docker build -t server:v2 ./build-server

**Dockerfile文件：**

FROM ubuntu:latest

# 服务端文件

ADD ./server/ /home/server/

# 安装依赖工具及库

RUN apt-get update && \

    apt-get install -y \

    cmake \

    g++ \

    libevent-dev \

    libjsoncpp-dev \

    libmysqlcppconn-dev \

    libcrypto++-dev \

    libspdlog-dev

# 清楚包，减轻镜像大小

RUN apt-get clean && rm -rf /var/lib/apt/lists/\*

# 切换目录

WORKDIR /home/server/build

# 编译

RUN cmake .. && make

# 创建图片文件夹

RUN mkdir /home/Image

RUN mkdir /home/Image/test1 /home/Image/test2

# 工作目录

WORKDIR /home/server/bin

# 暴露端口

EXPOSE 8080

CMD ./server

run阶段：

首先构建服务端和mysql的组网：

run-create-net:

  stage: run

  tags:

    - shellrunner

  script:

    - echo "创建网络tiki"

    - docker network create --subnet=172.20.0.0/24 tiki

  needs:

    - build-mysql

然后启动mysql：

# 运行数据库

run-mysql:

  stage: run

  tags:

    - shellrunner

  script:

    - echo "mysql镜像启动,初始化中"

    - docker run -d --name mysql --network tiki --ip 172.20.0.3 -p 3306:3306 -e MYSQL\_ROOT\_PASSWORD='123456' --restart always mysql:v1

  after\_script:

    - echo "mysql初始化完成"

  needs:

    - run-create-net

再启动服务端：

# 运行服务器：

run-server:

  stage: run

  tags:

    - shellrunner

  script:

    - echo "server启动"

    - docker run -d --name server --network tiki --ip 172.20.0.2 -p 8081:8081 --restart always server:v2

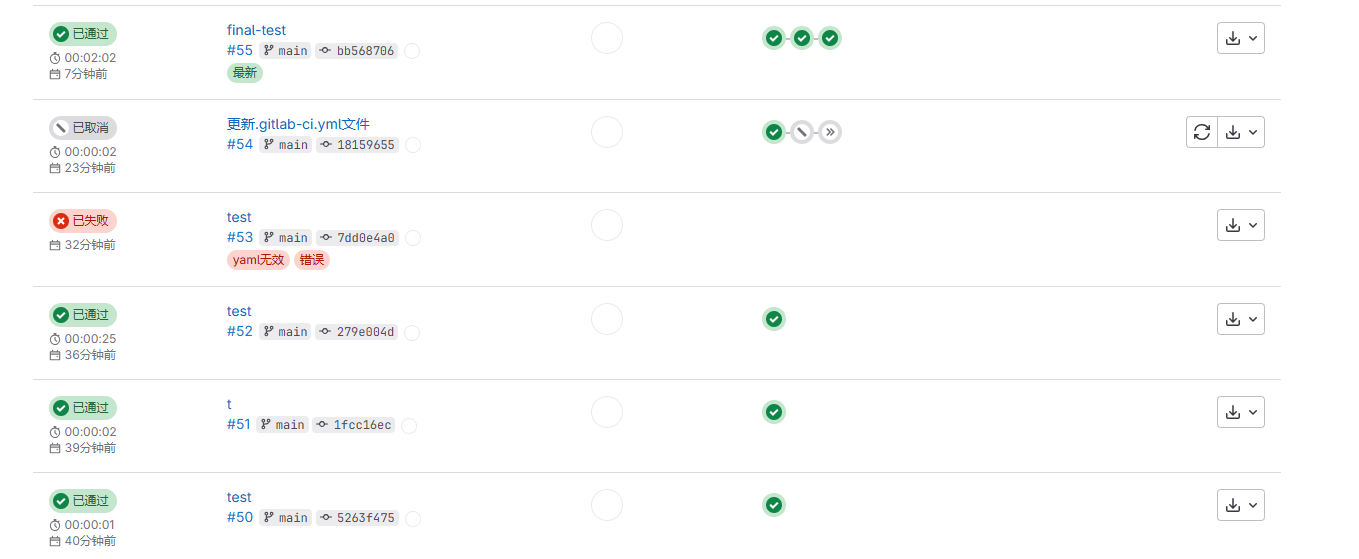
    - echo "启动完成"

  needs:

    - run-create-net

    - build-server

每当项目重新构建并提交就可以执行流水线



成功失败都会体现在流水线中

当版本回退重新提交时，也可以触发流水线进行执行