基于 C++ 的博客系统

重要知识点

1. 简单的Web服务器设计能力
2. C/C++ 操作 MySQL 数据库
3. 数据库设计
4. Restful 风格 API
5. json 和 jsoncpp 的使用
6. 强化 HTTP 协议的理解
7. cpp-httplib 的使用和原理
8. 正则表达式
9. Postman 工具的使用
10. boost 的使用
11. 前端页面的开发方法和技巧(免费模板 + bootstrap + Vue.js)
12. 使用 Editor.md Markdown编辑器
13. 软件测试的基本思想和方法

整体架构

博客服务器

1. 对博客的增删改查管理能力
2. 对标签的增删查改能力

博客客户端

1. 博客标题列表页面
2. 博客内容展示页面
3. 博客管理页面
4. 博客内容编辑页面

关于数据存储和交互

1. 服务器存储 markdown 格式的数据
2. 前端通过 editor.md 这个工具将markdown的文档内容提交给服务器, 或者解析服务器返回的 markdown 作为前端页面内容.

授课预估

共需五节课左右.

第一节课: 介绍整体框架和业务流程, 设计数据库并建好数据表, 讲解使用 MySQL C API 操作数据库.

第二节课: 根据 MySQL C API 封装数据库操作, 实现服务器的数据存储层, 并进行测试. 介绍 cpp-httplib, 介绍正则表达式. 设计服务器提供的 API.

第三节课: 介绍 jsoncpp. 基于 cpp-httplib 实现实现服务器 API, 并使用 Postman 测试.

第四节课: 简要介绍前端技术, 使用 免费模板 + Vue 构建出一个博客列表页面.

第五节课: 介绍 editor.md, 实现博客展示页面和博客编辑页面.

数据库设计

创建数据库

```
create database if not exists blog_system;
use blog_system;
```

创建博客表

```
drop table if exists `blog_table`
create table `blog_table`(blog_id int not null primary key auto_increment,
                          title varchar(50),
                          content text,
                          tag_id int,
                          create_time varchar(50))
```

创建标签表

```
drop table if exists `tag_table`
create table `tag_table`(tag_id int not null primary key auto_increment,
                         tag_name varchar(50))
```

使用 MySQL C API 操作数据库

插入数据库

```
////////////////////////////////////////////////////
// 编译选项
// -L /usr/lib64/mysql -lmysqlclient
// 删除, 更新 和插入同理, 只是拼装成不同的 SQL 而已
////////////////////////////////////////////////////

#include <cstdio>
#include <cstdlib>
#include <mysql/mysql.h>

int main() {
  // 1. 初始化句柄
  MYSQL* connect_fd = mysql_init(NULL);
  // 2. 建立链接
  // mysql_init 返回的指针
  // 主机地址
  // 用户名
  // 密码
  // 数据库名
```

```
  // 端口号
  // unix_socket
  // client_flag
  if (mysql_real_connect(connect_fd, "127.0.0.1", "root", "",
                                                "blog_system", 3306, NULL, 0) == NULL) {
    printf("连接失败! %s\n", mysql_error(connect_fd));
    return 1;
  }
  // 3. 设置编码格式
  mysql_set_character_set(connect_fd, "utf8");
  // 4. 拼装 SQL 语句
    char sql[1024 * 4] = {0};
  char title[] = "立一个 flag";
  char content[] = "我要拿 30w 年薪";
  int tag_id = 2;
  char datetime[] = "2019/05/14 12:00";
    sprintf(sql, "insert into blog_table values(null, '%s', '%s', %d, '%s')",
      title, content, tag_id, datetime);
  // 5. 执行 SQL 语句
  int ret = mysql_query(connect_fd, sql);
  if (ret < 0) {
    printf("执行 sql 失败! %s\n", mysql_error(connect_fd));
    return 1;
  }
  // 6. 关闭句柄
  mysql_close(connect_fd);
  printf("执行成功!\n");
  return 0;
}
```

查找数据库

mysql_select.cc

```
//////////////////////////////////////////////////////
// 编译选项
// -L /usr/lib64/mysql -lmysqlclient
// 删除, 更新 和插入同理, 只是拼装成不同的 SQL 而已
//////////////////////////////////////////////////////

#include <cstdio>
#include <cstdlib>
#include <mysql/mysql.h>

int main() {
  // 1. 初始化句柄
  MYSQL* connect_fd = mysql_init(NULL);
  // 2. 建立链接
  // mysql_init 返回的指针
  // 主机地址
  // 用户名
  // 密码
  // 数据库名
```

```
  // 端口号
  // unix_socket
  // client_flag
  if (mysql_real_connect(connect_fd, "127.0.0.1", "root", "",
                                                "blog_system", 3306, NULL, 0) == NULL) {
    printf("连接失败! %s\n", mysql_error(connect_fd));
    return 1;
  }
  // 3. 设置编码格式
  mysql_set_character_set(connect_fd, "utf8");
  // 4. 拼装 SQL 语句
    char sql[1024 * 4] = {0};
    sprintf(sql, "select * from blog_table");
  // 5. 执行 SQL 语句
  int ret = mysql_query(connect_fd, sql);
  if (ret < 0) {
    printf("执行 sql 失败! %s\n", mysql_error(connect_fd));
    return 1;
  }
  // 6. 遍历查询结果
  MYSQL_RES* result = mysql_store_result(connect_fd);
  if (result == NULL) {
    printf("获取结果失败! %s\n", mysql_error(connect_fd));
    return 1;
  }
  //  a) 获取行数和列数
  int rows = mysql_num_rows(result);
  int fields = mysql_num_fields(result);
  printf("rows: %d, fields: %d\n", rows, fields);
  //  b) 打印结果
  for (int i = 0; i < rows; ++i) {
    MYSQL_ROW row = mysql_fetch_row(result);
    for (int j = 0; j < fields; ++j) {
      printf("%s\t", row[j]);
    }
    printf("\n");
  }
  // 7. 释放结果集
  mysql_free_result(result);
  // 8. 关闭句柄
  mysql_close(connect_fd);
  printf("执行成功!\n");
  return 0;
}
```

服务器 API 设计

博客管理 API 设计

新增博客

请求:

```
POST /blog
{
    "title": "我的第一篇博客",
    "content": "博客的 markdown 格式内容",
    "tag_id": 1,
    "create_time": "2019/05/14 12:00",
}

响应:
HTTP/1.1 200 OK
{
    "ok": true,
}
```

获取所有博客标题

```
请求(使用 tag_id 参数筛选指定标签下的博客):
GET /blog?tag_id=1

响应:
HTTP/1.1 200 OK
[
    {
        "blog_id": 1,
        "blog_url": "/blog/1",
        "title": "我的第一篇博客",
        "tag_id": 1,
        "create_time": "2019/05/14 12:00"
    },
    {
        "blog_id": 2,
        "blog_url": "/blog/2",
        "title": "C 语言的类型",
        "tag_id": 1,
        "create_time": "2019/05/14 12:00"
    }
]
```

删除博客

```
请求:
DELETE /blog/:blog_id

响应:
HTTP/1.1 200 OK
{
    "ok": true,
}
```

修改博客

```
请求:
PUT /blog/:blog_id
{
    "title": "我的第一篇博客",
    "content": "博客的 markdown 格式内容",
    "tag_id": 1
}

响应:
HTTP/1.1 200 OK
{
    "ok": true,
}
```

获取博客详细内容

```
请求:
GET /blog/:blog_id

响应:
HTTP/1.1 200 OK
{
    "blog_id": 1,
    "title": "我的第一篇博客",
    "content": "博客的 markdown 格式内容",
    "tag_id": 1
}
```

标签管理 API 设计

新增标签

```
请求:
POST /tag
{
    "tag_name": "C 语言",
}

响应:
HTTP/1.1 200 OK
{
    "ok": true,
}
```

删除标签

```
请求:
DELETE /tag/:tag_id

响应:
HTTP/1.1 200 OK
{
    "ok": true,
}
```

获取所有标签

```
请求:
GET /tag

响应:
HTTP/1.1 200 OK
[
    {
        "tag_id": 1,
        "tag_name": "C语言"
    },
    {
        "tag_id": 2,
        "tag_name": "数据结构"
    }
]
```

客户端设计

博客标题列表页面

显示一共有哪些博客

博客内容展示页面

点击某个博客标题, 进入内容页, 显示博客内容

博客管理页面

对博客进行增删

博客内容编辑页面

编辑修改博客内容

服务器端实现

数据库操作整体结构

db.hpp

```
namespace blog_system {

static MYSQL* MySQLInit() {
  MYSQL* connect_fd = mysql_init(NULL);
  if (mysql_real_connect(connect_fd, "127.0.0.1", "root", "",
                                                "blog_system", 3306, NULL, 0) == NULL) {
    printf("连接失败! %s\n", mysql_error(connect_fd));
    return NULL;
  }
  mysql_set_character_set(connect_fd, "utf8");
  return connect_fd;
}

static void MySQLRelease(MYSQL* mysql) {
  mysql_close(mysql);
}

class BlogTable {
public:
  BlogTable(MYSQL* mysql) : mysql_(mysql) {  }

  bool Insert(const Json::Value& blog) {

  }

  bool SelectAll(Json::Value* blogs, const std::string& tag_id = "") {

  }

  bool SelectOne(int32_t blog_id, Json::Value* blog) {

  }

  bool Update(const Json::Value& blog) {

  }

  bool Delete(int blog_id) {

  }
private:
  MYSQL* mysql_;
};

class TagTable {
public:
  TagTable(MYSQL* mysql) : mysql_(mysql) {  }

  bool SelectAll(Json::Value* tags) {

  }

  bool Insert(const Json::Value& tag) {
```

```
  }

  bool Delete(int tag_id) {

  }
private:
  MYSQL* mysql_;
};
}
```

使用 Json 作为参数

json 出自 JavaScript, 是一种非常方便的键值对数据组织格式, 目前被业界广泛使用.

C++ 中可以使用 jsoncpp 这个库来解析和构造 json 数据

```
yum install jsoncpp-devel
```

实现插入博客

```
  bool Insert(const Json::Value& blog) {
    // 如果拿一个完整的课件做实验，会发现:
    // 由于博客内容中可能包含一些特殊字符(\n, '', "" 等)，会导致拼装出的 sql 语句有问题.
    // 应该使用 mysql_real_escape_string 对 content 字段来进行转义
    // 转义只是为了让 SQL 语句拼接正确. 实际上插入成功后数据库的内容已经自动转义回来了.
    const std::string& content = blog["content"].asString();
    // 文档上要求转义的缓冲区长度必须是之前的 2 倍 + 1
    // 使用 unique_ptr 管理内存
    std::unique_ptr<char> content_escape(new char[content.size() * 2 + 1]);
    mysql_real_escape_string(mysql_, content_escape.get(), content.c_str(), content.size());

    // 插入的博客内容可能较长，需要搞个大点的缓冲区(根据用户请求的长度自适应)，
    std::unique_ptr<char> sql(new char[content.size() * 2 + 4096]);
    sprintf(sql.get(), "insert into blog_table values(null, '%s', '%s', %d,'%s')",
        blog["title"].asCString(), content_escape.get(),
        blog["tag_id"].asInt(), blog["create_time"].asCString());
    int ret = mysql_query(mysql_, sql.get());
    if (ret != 0) {
      printf("执行 sql 失败! sql=%s, %s\n", sql.get(), mysql_error(mysql_));
      return false;
    }
    return true;
  }
```

实现查询所有博客

```
  bool SelectAll(Json::Value* blogs, const std::string& tag_id = "") {
    char sql[1024 * 4] = {0};
    // 可以根据 tag_id 来筛选结果
    if (tag_id.empty()) {
      sprintf(sql, "select blog_id, title, tag_id, create_time from blog_table");
```

```
  } else {
    sprintf(sql, "select blog_id, title, tag_id, create_time\
        from blog_table where tag_id = '%s'", tag_id.c_str());
  }
  int ret = mysql_query(mysql_, sql);
  if (ret != 0) {
    printf("执行 sql 失败! %s\n", mysql_error(mysql_));
    return false;
  }
  MYSQL_RES* result = mysql_store_result(mysql_);
  if (result == NULL) {
    printf("获取结果失败! %s\n", mysql_error(mysql_));
    return false;
  }
  int rows = mysql_num_rows(result);
  for (int i = 0; i < rows; ++i) {
    MYSQL_ROW row = mysql_fetch_row(result);
    Json::Value blog;
    blog["blog_id"] = atoi(row[0]);
    blog["title"] = row[1];
    blog["tag_id"] = atoi(row[2]);
    blog["create_time"] = row[3];
    // 遍历结果依次加入到 dishes 中
    blogs->append(blog);
  }
  return true;
}
```

实现查询单个博客

```
bool SelectOne(int32_t blog_id, Json::Value* blog) {
  char sql[1024 * 4] = {0};
  sprintf(sql, "select * from blog_table where blog_id = %d", blog_id);
  int ret = mysql_query(mysql_, sql);
  if (ret != 0) {
    printf("执行 sql 失败! %s\n", mysql_error(mysql_));
    return false;
  }
  MYSQL_RES* result = mysql_store_result(mysql_);
  if (result == NULL) {
    printf("获取结果失败! %s\n", mysql_error(mysql_));
    return false;
  }
  int rows = mysql_num_rows(result);
  if (rows != 1) {
    printf("查找结果不为 1 条. rows = %d!\n", rows);
    return false;
  }
  MYSQL_ROW row = mysql_fetch_row(result);
  (*blog)["blog_id"] = atoi(row[0]);
  (*blog)["title"] = row[1];
  (*blog)["content"] = row[2];
  (*blog)["tag_id"] = atoi(row[3]);
```

```
    (*blog)["create_time"] = row[4];
    return true;
  }
```

实现更新博客

```
bool Update(const Json::Value& blog) {
  // 如果拿一个完整的课件做实验，会发现：
  // 由于博客内容中可能包含一些特殊字符(\n, '', "" 等)，会导致拼装出的 sql 语句有问题.
  // 应该使用 mysql_real_escape_string 对 content 字段来进行转义
  // 转义只是为了让 SQL 语句拼接正确. 实际上插入成功后数据库的内容已经自动转义回来了.
  const std::string& content = blog["content"].asString();
  // 文档上要求转义的缓冲区长度必须是之前的 2 倍 + 1
  // 使用 unique_ptr 管理内存
  std::unique_ptr<char> content_escape(new char[content.size() * 2 + 1]);
  mysql_real_escape_string(mysql_, content_escape.get(), content.c_str(), content.size());

  // 插入的博客内容可能较长，需要搞个大点的缓冲区(根据用户请求的长度自适应)，
  std::unique_ptr<char> sql(new char[content.size() * 2 + 4096]);
  sprintf(sql.get(), "update blog_table SET title='%s', content='%s',\
      tag_id=%d where blog_id=%d",
      blog["title"].asCString(),
      content_escape.get(),
      blog["tag_id"].asInt(),
      blog["blog_id"].asInt());

  // DEBUG 用于调试
  // printf("[SQL] %s\n", sql);

  int ret = mysql_query(mysql_, sql.get());
  if (ret != 0) {
    printf("执行 sql 失败! sql=%s, %s\n", sql.get(), mysql_error(mysql_));
    return false;
  }
  return true;
}
```

实现删除博客

```
bool Delete(int blog_id) {
  char sql[1024 * 4] = {0};
  sprintf(sql, "delete from blog_table where blog_id=%d", blog_id);
  int ret = mysql_query(mysql_, sql);
  if (ret != 0) {
    printf("执行 sql 失败! sql=%s, %s\n", sql, mysql_error(mysql_));
    return false;
  }
  return true;
}
```

实现新增标签

```
bool Insert(const Json::Value& tag) {
  char sql[1024 * 4] = {0};
  // 此处 dish_ids 需要先转成字符串(本来是一个对象,
  // 形如 [1, 2, 3]. 如果不转, 是无法 asCString)
  sprintf(sql, "insert into tag_table values(null, '%s')",
      tag["tag_name"].asCString());
  int ret = mysql_query(mysql_, sql);
  if (ret != 0) {
    printf("执行 sql 失败! sql=%s, %s\n", sql, mysql_error(mysql_));
    return false;
  }
  return true;
}
```

实现删除标签

```
bool Delete(int tag_id) {
  char sql[1024 * 4] = {0};
  sprintf(sql, "delete from tag_table where tag_id = %d", tag_id);
  int ret = mysql_query(mysql_, sql);
  if (ret != 0) {
    printf("执行 sql 失败! sql=%s, %s\n", sql, mysql_error(mysql_));
    return false;
  }
  return true;
}
```

实现查看所有标签

```
bool SelectAll(Json::Value* tags) {
  char sql[1024 * 4] = {0};
  sprintf(sql, "select * from tag_table");
  int ret = mysql_query(mysql_, sql);
  if (ret != 0) {
    printf("执行 sql 失败! %s\n", mysql_error(mysql_));
    return false;
  }
  MYSQL_RES* result = mysql_store_result(mysql_);
  if (result == NULL) {
    printf("获取结果失败! %s\n", mysql_error(mysql_));
    return false;
  }
  int rows = mysql_num_rows(result);
  for (int i = 0; i < rows; ++i) {
    MYSQL_ROW row = mysql_fetch_row(result);
    Json::Value tag;
    tag["tag_id"] = atoi(row[0]);
    tag["tag_name"] = row[1];
    tags->append(tag);
  }
```

```
    return true;
  }
```

测试数据库操作

db_test.cc

```
void TestBlogTable() {
  bool ret = false;
  // 更友好的格式化显示 Json
  Json::StyledWriter writer;
  MYSQL* mysql = MySQLInit();

  Json::Value blog;
  blog["title"] = "初识 C 语言";
  std::string content;
  FileUtil::ReadFile("./test_data/1.md", &content);
  blog["content"] = content;
  blog["tag_id"] = 1;
  blog["create_time"] = "2019/05/14 12:00";

  std::cout << "==============测试插入====================" << std::endl;
  BlogTable blog_table(mysql);
  ret = blog_table.Insert(blog);
  std::cout << "Insert: " << ret << std::endl;

  std::cout << "==============测试查找====================" << std::endl;
  Json::Value blogs;
  ret = blog_table.SelectAll(&blogs);
  std::cout << "SelectAll: " << ret << std::endl
            << writer.write(blogs) << std::endl;

  std::cout << "==============测试更新====================" << std::endl;
  blog["blog_id"] = 1;
  blog["title"] = "测试更新博客";
  blog["content"] = content;
  blog["tag_id"] = 2;
  blog["create_time"] = "2019/05/20 12:00";
  Json::Value blog_out;
  ret = blog_table.Update(blog);
  std::cout << "Update: " << ret << std::endl;
  ret = blog_table.SelectOne(1, &blog_out);
  std::cout << "SelectOne: " << ret << std::endl
            << writer.write(blog_out) << std::endl;

  std::cout << "==============测试删除====================" << std::endl;
  int blog_id = 6;
  ret = blog_table.Delete(blog_id);
  std::cout << "Delete: " << ret << std::endl;

  MySQLRelease(mysql);
```

```
}

void TestTagTable() {
  bool ret = false;
  Json::StyledWriter writer;
  MYSQL* mysql = MySQLInit();
  TagTable tag_table(mysql);

  std::cout << "==============测试插入====================" << std::endl;
  Json::Value tag;
  tag["tag_name"] = "Java";
  ret = tag_table.Insert(tag);
  std::cout << "Insert: " << ret << std::endl;

  std::cout << "==============测试查看====================" << std::endl;
  Json::Value tags;
  ret = tag_table.SelectAll(&tags);
  std::cout << "SelectAll: " << ret << std::endl
            << writer.write(tags) << std::endl;

  std::cout << "==============测试删除状态====================" << std::endl;
  ret = tag_table.Delete(5);
  std::cout << "ChangeState ret:" << ret << std::endl;

  MySQLRelease(mysql);
}

int main() {
  // TestBlogTable();
  TestTagTable();
  return 0;
}
```

实现服务器接口

使用 httplib 搭建服务器框架

使用 cpp-httplib 实现一个 hello world

```
#include "httplib.h"

int main() {
  using namespace httplib;
  Server server;
  server.Get("/", [](const Request& req, Response& resp) {
    (void)req;
    resp.set_content("<html>hello</html>", "text/html");
  });
  server.set_base_dir("./wwwroot");
  server.listen("0.0.0.0", 9092);
  return 0;
}
```

编译选项

```
g++ main.cc -lpthread -std=c++11
```

搭建服务器整体框架

blog_server.cc

```
MYSQL* mysql = NULL;

int main() {
  using namespace httplib;
  using namespace blog_system;
  Server server;

  // 1. 数据库客户端初始化和释放
  mysql = MySQLInit();
  signal(SIGINT, [](int) { MySQLRelease(mysql); exit(0);});
  BlogTable blog_table(mysql);
  TagTable tag_table(mysql);

  // 2. 设置路由
  // 新增博客
  server.Post("/blog", [&blog_table](const Request& req, Response& resp) {

  });
  // 查看所有博客(可以按标签筛选)
  server.Get("/blog", [&blog_table](const Request& req, Response& resp) {

  });

  // 查看一篇博客内容
  server.Get(R"(/blog/(\d+))", [&blog_table](const Request& req, Response& resp) {

  });

  // 删除博客
  // raw string(c++ 11), 转义字符不生效. 用来表示正则表达式正好合适
  // 关于正则表达式, 只介绍最基础概念即可. \d+ 表示匹配一个数字
  // http://help.locoy.com/Document/Learn_Regex_For_30_Minutes.htm
  server.Delete(R"(/blog/(\d+))", [&blog_table](const Request& req, Response& resp) {

  });
  // 修改博客
  server.Put(R"(/blog/(\d+))", [&blog_table](const Request& req, Response& resp) {

  });
  // 新增标签
  server.Post("/tag", [&tag_table](const Request& req, Response& resp) {

  });
```

```
  // 删除标签
  server.Delete(R"(/tag/(\d+))", [&tag_table](const Request& req, Response& resp) {

  });

  // 获取所有标签
  server.Get("/tag", [&tag_table](const Request& req, Response& resp) {

  });

  // 设置静态文件目录
  server.set_base_dir("./wwwroot");

  server.listen("0.0.0.0", 9093);
  return 0;
}
```

实现新增博客

```
  server.Post("/blog", [&blog_table](const Request& req, Response& resp) {
    LOG(INFO) << "新增博客: " << req.body << std::endl;
    Json::Reader reader;
    Json::FastWriter writer;
    Json::Value req_json;
    Json::Value resp_json;
    // 1. 请求解析成 Json 格式
    bool ret = reader.parse(req.body, req_json);
    if (!ret) {
      // 请求解析出错，返回一个400响应
      resp_json["ok"] = false;
      resp_json["reason"] = "parse Request failed!\n";
      resp.status = 400;
      resp.set_content(writer.write(resp_json), "application/json");
      return;
    }
    // 2. 进行参数校验
    if (req_json["title"].empty() || req_json["content"].empty()
        || req_json["tag_id"].empty() || req_json["create_time"].empty()) {
      resp_json["ok"] = false;
      resp_json["reason"] = "Request fields error!\n";
      resp.status = 400;
      resp.set_content(writer.write(resp_json), "application/json");
      return;
    }
    // 3. 调用数据库接口进行操作数据
    ret = blog_table.Insert(req_json);
    if (!ret) {
      resp_json["ok"] = false;
      resp_json["reason"] = "Insert failed!\n";
      resp.status = 500;
      resp.set_content(writer.write(resp_json), "application/json");
      return;
    }
```

```
  // 4. 封装正确的返回结果
  resp_json["ok"] = true;
  resp.set_content(writer.write(resp_json), "application/json");
  return;
});
```

实现查看所有博客

```
// 查看所有博客(可以按标签筛选)
server.Get("/blog", [&blog_table](const Request& req, Response& resp) {
  LOG(INFO) << "查看所有博客" << std::endl;
  Json::Reader reader;
  Json::FastWriter writer;
  Json::Value resp_json;
  // 如果没传 tag_id, 返回的是空字符串
  const std::string& tag_id = req.get_param_value("tag_id");
  // 对于查看博客来说 API 没有请求参数, 不需要解析参数和校验了, 直接构造结果即可
  // 1. 调用数据库接口查询数据
  Json::Value blogs;
  bool ret = blog_table.SelectAll(&blogs, tag_id);
  if (!ret) {
    resp_json["ok"] = false;
    resp_json["reason"] = "SelectAll failed!\n";
    resp.status = 500;
    resp.set_content(writer.write(resp_json), "application/json");
    return;
  }
  // 2. 构造响应结果
  resp.set_content(writer.write(blogs), "application/json");
  return;
});
```

实现查看指定博客内容

```
// 查看一篇博客内容
server.Get(R"(/blog/(\d+))", [&blog_table](const Request& req, Response& resp) {
  Json::Value resp_json;
  Json::FastWriter writer;
  // 1. 解析获取 blog_id
  int blog_id = std::stoi(req.matches[1]);
  LOG(INFO) << "查看指定博客: " << blog_id << std::endl;
  // 2. 调用数据库接口查看博客
  bool ret = blog_table.SelectOne(blog_id, &resp_json);
  if (!ret) {
    resp_json["ok"] = false;
    resp_json["reason"] = "SelectOne failed!\n";
    resp.status = 500;
    resp.set_content(writer.write(resp_json), "application/json");
    return;
  }
  // 3. 包装正确的响应
  resp_json["ok"] = true;
```

```
  resp.set_content(writer.write(resp_json), "application/json");
  return;
});
```

实现删除博客

```
// 删除博客
// raw string(c++ 11), 转义字符不生效. 用来表示正则表达式正好合适
// 关于正则表达式, 只介绍最基础概念即可. \d+ 表示匹配一个数字
// http://help.locoy.com/Document/Learn_Regex_For_30_Minutes.htm
server.Delete(R"(/blog/(\d+))", [&blog_table](const Request& req, Response& resp) {
  Json::Value resp_json;
  Json::FastWriter writer;
  // 1. 解析获取 blog_id
  // 使用 matches[1] 就能获取到 blog_id
  // LOG(INFO) << req.matches[0] << "," << req.matches[1] << "\n";
  int blog_id = std::stoi(req.matches[1]);
  LOG(INFO) << "删除指定博客: " << blog_id << std::endl;
  // 2. 调用数据库接口删除博客
  bool ret = blog_table.Delete(blog_id);
  if (!ret) {
    resp_json["ok"] = false;
    resp_json["reason"] = "Delete failed!\n";
    resp.status = 500;
    resp.set_content(writer.write(resp_json), "application/json");
    return;
  }
  // 3. 包装正确的响应
  resp_json["ok"] = true;
  resp.set_content(writer.write(resp_json), "application/json");
  return;
});
```

实现修改博客

```
// 修改博客
server.Put(R"(/blog/(\d+))", [&blog_table](const Request& req, Response& resp) {
  Json::Reader reader;
  Json::FastWriter writer;
  Json::Value req_json;
  Json::Value resp_json;
  // 1. 获取到博客 id
  int blog_id = std::stoi(req.matches[1]);
  LOG(INFO) << "修改博客 " << blog_id << "|" << req.body << std::endl;
  // 2. 解析博客信息
  bool ret = reader.parse(req.body, req_json);
  if (!ret) {
    // 请求解析出错, 返回一个400响应
    resp_json["ok"] = false;
    resp_json["reason"] = "parse Request failed!\n";
    resp.status = 400;
    resp.set_content(writer.write(resp_json), "application/json");
```

```
    return;
  }
  // [注意!!] 一定要记得补充上 dish_id
  req_json["blog_id"] = blog_id;
  // 3. 校验博客信息
  if (req_json["title"].empty() || req_json["content"].empty()
      || req_json["tag_id"].empty()) {
    // 请求解析出错，返回一个400响应
    resp_json["ok"] = false;
    resp_json["reason"] = "Request has no name or price!\n";
    resp.status = 400;
    resp.set_content(writer.write(resp_json), "application/json");
    return;
  }
  // 4. 调用数据库接口进行修改
  ret = blog_table.Update(req_json);
  if (!ret) {
    resp_json["ok"] = false;
    resp_json["reason"] = "Update failed!\n";
    resp.status = 500;
    resp.set_content(writer.write(resp_json), "application/json");
    return;
  }
  // 5. 封装正确的数据
  resp_json["ok"] = true;
  resp.set_content(writer.write(resp_json), "application/json");
  return;
});
```

实现新增标签

```
// 新增标签
server.Post("/tag", [&tag_table](const Request& req, Response& resp) {
  LOG(INFO) << "新增订单: " << req.body << std::endl;
  Json::Reader reader;
  Json::FastWriter writer;
  Json::Value req_json;
  Json::Value resp_json;
  // 1. 请求解析成 Json 格式
  bool ret = reader.parse(req.body, req_json);
  if (!ret) {
    // 请求解析出错，返回一个400响应
    resp_json["ok"] = false;
    resp_json["reason"] = "parse Request failed!\n";
    resp.status = 400;
    resp.set_content(writer.write(resp_json), "application/json");
    return;
  }
  // 2. 校验标签格式
  if (req_json["tag_name"].empty()) {
    // 请求解析出错，返回一个400响应
    resp_json["ok"] = false;
    resp_json["reason"] = "Request has no table_id or time or dish_ids!\n";
```

```
    resp.status = 400;
    resp.set_content(writer.write(resp_json), "application/json");
    return;
  }
  // 3. 调用数据库接口，插入标签
  ret = tag_table.Insert(req_json);
  if (!ret) {
    resp_json["ok"] = false;
    resp_json["reason"] = "TagTable Insert failed!\n";
    resp.status = 500;
    resp.set_content(writer.write(resp_json), "application/json");
    return;
  }
  // 4. 返回正确的结果
  resp_json["ok"] = true;
  resp.set_content(writer.write(resp_json), "application/json");
});
```

实现删除标签

```
// 删除标签
server.Delete(R"(/tag/(\d+))", [&tag_table](const Request& req, Response& resp) {
  Json::Value resp_json;
  Json::FastWriter writer;
  // 1. 解析出 tag_id
  int tag_id = std::stoi(req.matches[1]);
  LOG(INFO) << "删除指定标签: " << tag_id << std::endl;
  // 2. 执行数据库操作删除标签
  bool ret = tag_table.Delete(tag_id);
  if (!ret) {
    resp_json["ok"] = false;
    resp_json["reason"] = "TagTable Insert failed!\n";
    resp.status = 500;
    resp.set_content(writer.write(resp_json), "application/json");
    return;
  }
  // 3. 包装正确的结果
  resp_json["ok"] = true;
  resp.set_content(writer.write(resp_json), "application/json");
  return;
});
```

实现获取所有标签

```
// 获取所有标签
server.Get("/tag", [&tag_table](const Request& req, Response& resp) {
  (void) req;
  LOG(INFO) << "查看所有订单" << std::endl;
  Json::Reader reader;
  Json::FastWriter writer;
  Json::Value resp_json;
  // 1. 调用数据库接口查询数据
```

```
    Json::Value tags;
    bool ret = tag_table.SelectAll(&tags);
    if (!ret) {
      resp_json["ok"] = false;
      resp_json["reason"] = "SelectAll failed!\n";
      resp.status = 500;
      resp.set_content(writer.write(resp_json), "application/json");
      return;
    }
    // 2. 构造响应结果
    resp.set_content(writer.write(tags), "application/json");
    return;
  });
```

使用 Postman 测试接口

使用 Postman 构造请求测试即可.

客户端实现

实现博客标题列表页面

使用免费模板

使用 amaze ui 提供的免费模板

http://tpl.amazeui.org/

基于免费模板进行修改

复制 lw-index.html 为 index.html, 并进行修改.

去掉不必要的组件(过程略)

使用 Vue.js

网页中需要动态交互的部分, 使用 JS 来实现. 但是原生的基于 Dom API 的交互方式比较麻烦, 因此 Vue.js 提供了更简单更方便的做法.

Vue.js 官网: https://cn.vuejs.org/v2/guide/

参考官网引入 CDN, 并复制官方的 hello world 代码

在 data 中新增一个属性 author, 并在页面合适的位置绑定

```
data: {
   author: '汤老湿',
}
```

刷新页面, 能看到页面上出现了 "汤老湿", 通过控制台修改 author 属性, 页面会同步发生变化.

构造 blogs 和 tags 数据并绑定到页面

数据参考 API 接口返回的数据格式

```
    data: {
        author: '汤老湿',
        blogs: [
            {
                "blog_id": 1,
                "title": "我的第一篇博客",
                "tag_id": 1,
                "create_time": "2019/05/14 12:00"
            },
            {
                "blog_id": 2,
                "title": "C 语言的类型",
                "tag_id": 1,
                "create_time": "2019/05/14 12:00"
            }
        ],
        tags: [
            {
                "tag_id": 1,
                "tag_name": "C语言"
            },
            {
                "tag_id": 2,
                "tag_name": "数据结构"
            }
        ]
    },
```

在页面合适的位置绑定数据.

```
<article class="am-g blog-entry-article" v-for="blog in blogs">
    <div class="am-u-lg-6 am-u-md-12 am-u-sm-12 blog-entry-text">
        <span><a href="" class="blog-color">{{blog.tag_id}}  </a></span>
        <span> @汤老湿  </span>
        <span>{{blog.create_time}}</span>
        <!-- 注意, 此时 get_blog 函数还没实现呢!!! -->
        <h1><a v-on:click="get_blog(blog.blog_id)">{{blog.title}}</a></h1>
        <p>我是摘要信息</p>
        <p><a href="" class="blog-continue">continue reading</a></p>
    </div>
</article>
```

此时刷新网页, 应该就能看到和构造数据匹配的界面了.

实现展示标签名字

当前展示的只是标签 id, 需要根据 id 转为 标签名.

在 methods 中添加一个方法

```
    methods: {
        tag_id2name(tag_id) {
            // 注意, 要访问 tags 和 blogs 需要带 this
            for(var index in this.tags) {
                if (this.tags[index].tag_id == tag_id) {
                    return this.tags[index].tag_name;
                }
            }
            return null;
        },
    },
```

然后在需要用到标签 id 的位置调用函数获取标签名

```
<span><a href="" class="blog-color">{{tag_id2name(blog.tag_id)}}  </a></span>
```

通过网络获取服务器上的 blogs 和 tags 数据

使用 JQuery 的 ajax 方法, 文档参见 http://www.w3school.com.cn/jquery/ajax_ajax.asp

实现获取博客列表

```
get_blogs(tag_id) {
    var url = "blog";
    if (tag_id != null) {
        url = "/blog?tag_id=" + tag_id;
    }
    $.ajax({
        url: url,
        type: "get",
        context: this,
        success: function(data, status) {
            this.blogs = data
        }
    })
},
```

实现获取标签列表

要保证先获取到 tag 列表, 再获取博客列表. 思考一下为什么?

```
init() {
    $.ajax({
        url: "/tag",
        type: "get",
        context: this,
        success: function(data, status) {
            this.tags = data;
            // 在获取到 tags 之后再获取 blogs，防止时序上出现问题
            this.get_blogs();
        }
    });
},
```

在合适的位置调用 初始化 函数

```
app.init()
```

部署到服务器上进行测试

在 服务器 目录下创建 wwwroot 目录, 将客户端代码拷贝进去. 并且在服务器中 `set_base_dir` 来指定 http 服务器的根目录.

之后通过浏览器访问, 如果页面显示不正确, 通过 chrome 控制台查看错误信息.

使用 cloak 优化显示

此时在网速较慢的情况下, 能够看到渲染前的插值表达式. 为了解决这个问题, 可以使用 Vue.js 提供的 cloak 功能

cloak 原意为 "披风" 能够把丑陋的插值表达式给遮盖住.

```
<style>
    [v-cloak] {
        display: none;
    }
</style>
```

然后在需要用到差值表达式的标签上加上 v-cloak 属性即可

```
<div class="am-u-md-8 am-u-sm-12" v-cloak>
```

实现博客内容展示页面

新增 blog 对象

在 data 中新增一个 blog 对象

```
blog: {
    title: "",
    content: "",
    blog_id: null,
    tag_id: null,
}, // 表示当前博客信息，是一个对象
```

并且将之前的博客列表使用 v-show 控制起来. 如果 blog_id 为 null, 则显示博客列表. 如果 blog_id 为非 null, 则显示博客内容.

创建博客显示容器

```
    <div class="am-u-md-8 am-u-sm-12" v-show="blog.blog_id != null">
        <article id="test-editormd-view">

        </article>
    </div
```

介绍 Editor.md 的用法(参考附录)

使用 Editor.md 将 markdown 转为 html

这个代码复制自 `editormd\examples\html-preview-markdown-to-html.html` 并稍加修改.

先引入 Editor.md. 照着官方示例复制粘贴即可.

```
<!-- 引入 Editor.md 完成 markdown 的操作，注意路径-->
<script src="/editor.md/lib/marked.min.js"></script>
<script src="/editor.md/lib/prettify.min.js"></script>
<script src="/editor.md/lib/raphael.min.js"></script>
<script src="/editor.md/lib/underscore.min.js"></script>
<script src="/editor.md/lib/sequence-diagram.min.js"></script>
<script src="/editor.md/lib/flowchart.min.js"></script>
<script src="/editor.md/lib/jquery.flowchart.min.js"></script>
<script src="/editor.md/editormd.js"></script>
```

编写点击博客标题的响应函数.

```
get_blog(blog_id) {
    $.ajax({
        url: "/blog/" + blog_id,
        type: "get",
        context: this,
        async: false,
        success: function(data, status) {
            this.blog = data;
            // [注意!] 需要先清理上次的结果，再填充新结果
            $('#test-editormd-view').empty();
            testEditormdView = editormd.markdownToHTML("test-editormd-view", {
                markdown        : this.blog.content ,//+ "\r\n" + $("#append-test").text(),
                //htmlDecode      : true,       // 开启 HTML 标签解析，为了安全性，默认不开启
```

```
                htmlDecode      : "style,script,iframe",  // you can filter tags decode
                //toc             : false,
                tocm            : true,    // Using [TOCM]
                //tocContainer    : "#custom-toc-container", // 自定义 ToC 容器层
                //gfm             : false,
                //tocDropdown     : true,
                // markdownSourceCode : true, // 是否保留 Markdown 源码，即是否删除保存源码的
Textarea 标签
                emoji           : true,
                taskList        : true,
                tex             : true,  // 默认不解析
                flowChart       : true,  // 默认不解析
                sequenceDiagram : true,  // 默认不解析
            });
        }
    });
}
```

此时就可以验证看博客内容是否展示正确了. 注意观察控制台中的错误日志.

制作一个返回列表的按钮

```
    <div class="am-u-sm-2">
        <button type="button" class="am-btn am-btn-default am-round" v-
on:click="main_page()">主页</button>
    </div>
```

实现响应函数

```
main_page() {
    this.blog.blog_id = null
    this.blog.tag_id = null
    this.blog.title = ""
    this.blog.content = ""
}
```

实现博客管理页面

复制 index.html 为 admin.html, 并修改标题.

注意!!! 博客的管理页面和之前的展示页面不再共用同一份页面了.

引入 status 状态

为了更好的管理页面状态变换, 引入一个 status 变量, 取值为

```
blog_list: 表示展示博客列表状态
blog_edit: 表示编辑博客状态
```

初始化设定为 `blog_list` 状态

此时需要调整 `main_page` 函数

```
main_page() {
    this.status = 'blog_list';
},
```

实现删除博客功能

每个博客标题下方新增删除按钮

```
<button type="button" class="am-btn am-btn-danger am-round" v-
on:click="delete_blog(blog.blog_id)">删除</button>
```

新增删除方法

```
delete_blog(blog_id) {
    $.ajax({
        url: "/blog/" + blog_id,
        type: "delete",
        context: this,
        success: function(data, status) {
            this.status = 'blog_list';
            this.get_blogs(); // 重新从服务器获取数据
            alert("删除成功!");
        }
    });
}
```

实现编辑博客功能1

修改 博客标题 点击事件的响应

```
<h1><a v-on:click="edit_blog(blog)">{{blog.title}}</a></h1>
```

新增 edit_blog 方法

```
edit_blog(blog_id) {
    this.blog = {}
    $.ajax({
        url: "/blog/" + blog_id,
        type: "get",
        context: this,
        success: function(data, status) {
            this.blog = data;
            editor = editormd("test-editor", {
                width  : "100%",
                height : "740px",
                path: "editor.md/lib/"
            });
            this.status = 'blog_edit'
        }
```

```
    });
},
```

新增一个编辑区(替换掉原有的展示博客内容区域)

```
<div class="am-u-md-12 am-u-sm-12" v-show="status == 'blog_edit'" v-cloak>
    <div id="test-editor">
        <textarea>{{blog.content}}</textarea>
    </div>
</div>
```

设定侧边栏也能自动隐藏

```
<div class="am-u-md-4 am-u-sm-12 blog-sidebar" v-show="status == 'blog_list'">
```

注意!!!!! , 此时页面中需要引入 Editor.md 的对应的 CSS, 否则编辑框显示不正确.

```
<link rel="stylesheet" href="editor.md/css/editormd.min.css" />
```

此时就可以验证博客是否能进行编辑了

实现编辑博客功能2

再新增一个提交按钮

```
<button type="button" class="am-btn am-btn-default am-round" v-show="status == 'blog_edit'"
v-on:click="update_blog()" v-cloak>提交</button>
```

实现 `update_blog` 函数

```
update_blog() {
    // 1. 先获取到博客内容
    var content = editor.getMarkdown()
    var body = {
        title: this.blog.title,
        content: content,
        tag_id: this.blog.tag_id,
    }
    // 2. 构造 http 请求发送给服务器即可
    $.ajax({
        url: "/blog/" + this.blog.blog_id,
        type: "put",
        contentType: "application/json;charset=utf-8",
        data: JSON.stringify(body),
        context: this,
        success: function(data, status) {
            this.get_blogs()
            alert("提交成功!");
            this.status = 'blog_list'
        }
```

```
    });
}
```

验证能否正确修改博客.

注意!!! 博客修改功能实现完毕后, 可能会发现 Editor.md 存在一点小 bug. 当编辑一篇博客提交之后, 再打开其他博客, 会发现其他博客没有立刻进入预览状态. 这个影响不大, 只要编辑一下博客内容就会出现预览结果.

实现新增博客功能

交给同学们自行完成(提示, 新增一个页面完成插入).

实现对标签的增删查改

交给同学们自行完成

附录

使用 Editor.md Markdown编辑器

官方文档

https://pandao.github.io/editor.md/

安装使用

```
# 在 client_code 目录中
wget https://github.com/pandao/editor.md/archive/master.zip
unzip master.zip
mv editor.md-master editor.md
```

创建 test.html, 将 github 下载的内容放到 test.html 同级目录下. 复制官网的代码到 test.html 中, 并稍微调整路径即可.

将 markdown 转为 html

核心操作

```
testEditormdView = editormd.markdownToHTML(...)
```

参考 `editormd\examples\html-preview-markdown-to-html.html` 这个示例代码.

前端获取 markdown 内容

```
editor.getMarkdown()
```

使用 JS 进行 base64 编码和解码

如果存在一些特殊字符引起问题, 可以使用 base64 算法进行转码

```
var encodedStr = window.btoa(“Hello world”);//字符串编码
var decodedStr = window.atob(encodedStr);//字符串解码
```