第八周周报

组号	班级	姓名	学号
27	2022211305	胡宇杭	2022212408
27	2022211305	孟林	2022210484
27	2022211305	陈炳璇	2022211479

前端部分

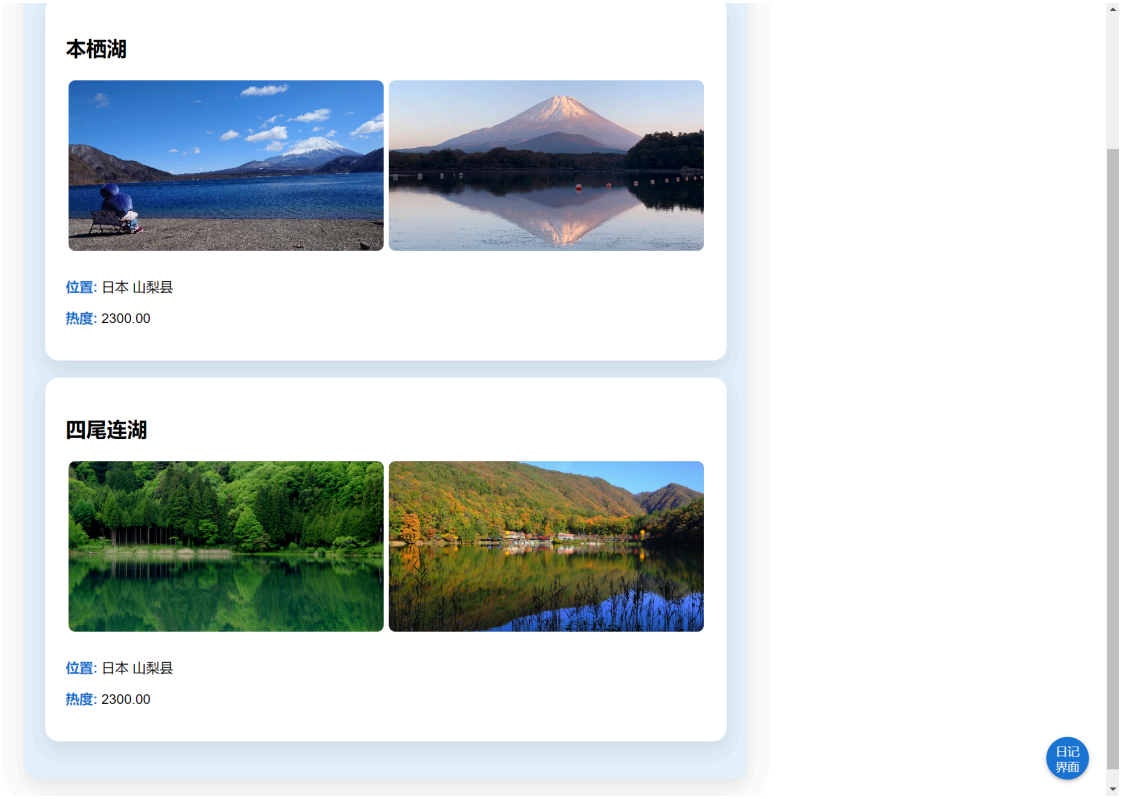
概述

在过去的一周中，我们的前端开发团队取得了显著的进步，成功实现了多个关键功能，进一步提升了用户体验和界面互动性。特别地，我们完成了日记模块的全流程开发，涵盖了路由优化、导航逻辑的细化以及界面的美观设计。

详细进展

1. 游学系统首页的全新体验

本周，日记首页经历了一次全面的更新，现在能够根据日记的“热度”展示内容，这种热度是通过一个考虑了点赞、用户评分及评论数量的复杂算法计算得出的。新增的删除功能让用户能更好地管理自己的日记，提升了应用的个性化和安全性。用户现在可以从首页直接进入任何日记的详细页面，这不仅显示了完整日记内容，还包括了互动丰富的评论区以及点赞和打分功能。



2. 游学系统详情页的功能增强

详情页新增了多种互动元素，用户可以查看日记的发布时间、内容，并参与点赞、评论和打分。特别新增的功能允许用户删除自己的评论，这提供了更大的控制权和自由度，使互动变得更自然和方便。

3. 页面的视觉设计提升

我们继续选择了天蓝色和白色作为主页面色彩，使页面不仅看起来更加清新舒适，还提高了内容的可读性。界面布局经过精细调整，确保了按钮和组件的美观与实用性，页面的现代化和优雅风格显著提升了界面的一体化和专业感。

4. 下周的工作预告

展望未来，我们计划继续扩展前端的功能。下周，我们将开始设计网站的主页，该主页将展示按热度排序的景点，每个景点会附带位置信息、详细描述和预览图片等。我们的目标是通过提供丰富的视觉内容和详细信息，来增强用户的浏览体验和互动性。

总结

本周的工作使我们在前端开发方面取得了重要进展，不仅优化了用户的互动体验，还显著提升了界面的视觉效果。我们期待持续推动项目发展，确保应用能够满足用户的需求并提供卓越的服务。

后端

后端上周计划任务完成情况

- [x] 实现 InputStream 模块
- [x] 实现 Processor 模块
- [x] 实现 Cursor 类
- [ ] 实现基本的 *Row*、*Page*、*Table* 类

> 未完成原因：我们发现在实际编写 Page、Table 类的时候发现其实现与 B+ 树紧密联系，因此决定将其放到 B+ 树实现之后再做编写

已完成部分

InputStream 模块

```
class InputStream
{
  private:
    std::string inputBuffer;
    size_t bufferLength;

  public:
    InputStream();
    std::string getInputBuffer() const { return inputBuffer; }
    void readInputStream();
};
```

InputStream 类用于从控制台中读入并存储数据，提供了控制台读取和获得输入两个方法

```
void InputStream::readInputStream()
{
    std::getline(std::cin, inputBuffer);
    bufferLength = inputBuffer.length();

    if (bufferLength <= 0) {
        throw READ_INPUT_FAIL;
    }
}
```

readInputStream 使用 getline 方法从控制台读入一行输入并将其存储在 std::string 容器中

Processor 模块

```
class Parser
{
    friend Executor;
  private:
    bool isMetaCmd = false;
    bool isStatement = false;
    META_COMMAND_TYPE metaCmdType = META_COMMAND_UNDEFINED;
    STATEMENT_TYPE statementType = STATEMENT_UNDEFINED;
    std::string conditions;

    static META_COMMAND_TYPE parseMetaCmdType(std::string& inputBuffer);
    static STATEMENT_TYPE parseStatementType(std::string& inputBuffer);

  public:
    Parser() = default;
    ~Parser() = default;

    void parseInputStream(InputStream& inputStream);
    void clear();
};
```

Parser 类接受一个 InputStream 类，并将其解析后的结果存储在成员中，目前支持的指令如下：

```
enum META_COMMAND_TYPE : uint16_t
{
    META_COMMAND_UNDEFINED,
    META_COMMAND_HELP,
    META_COMMAND_SHOW_DATABASES,
    META_COMMAND_SHOW_TABLES,
    META_COMMAND_QUIT,
    META_COMMAND_MODE,
    META_COMMAND_OPEN,
};

enum STATEMENT_TYPE : uint16_t
{
    STATEMENT_UNDEFINED,
    STATEMENT_INSERT,
    STATEMENT_DELETE,
    STATEMENT_UPDATE,
    STATEMENT_SELECT,
};
```

我们提供了两个私有成员函数 parseStatementType 和 parseMetaCmdType

```
META_COMMAND_TYPE Parser::parseMetaCmdType(std::string& inputBuffer)
{
    auto buffer = split(inputBuffer, ' ', 0);
    std::transform(buffer.begin(), buffer.end(), buffer.begin(),
                   [](unsigned char c) { return std::tolower(c); });

    if (buffer.compare(".help")) {
        return META_COMMAND_HELP;
    } else if (buffer.compare(".quit")) {
        return META_COMMAND_QUIT;
    } else if (buffer.compare(".mode")) {
        return META_COMMAND_MODE;
    } else if (buffer.compare(".open")) {
        return META_COMMAND_OPEN;
    } else if (buffer.compare(".databases")) {
        return META_COMMAND_SHOW_DATABASES;
    } else if (buffer.compare(".tables")) {
        return META_COMMAND_SHOW_TABLES;
    } else {
        return META_COMMAND_UNDEFINED;
    }
}

STATEMENT_TYPE Parser::parseStatementType(std::string& inputBuffer)
{
    auto buffer = split(inputBuffer, ' ', 0);
    std::transform(buffer.begin(), buffer.end(), buffer.begin(),
                   [](unsigned char c) { return std::tolower(c); });

    if (buffer.compare("select")) {
        return STATEMENT_SELECT;
    } else if (buffer.compare("insert")) {
        return STATEMENT_INSERT;
    } else if (buffer.compare("update")) {
        return STATEMENT_UPDATE;
    } else if (buffer.compare("delete")) {
        return STATEMENT_DELETE;
    }
}
```

其作用为当已知指令基本类型之后将其具体指令种类解析出来，通过上层方法 parseInputStream 调用

```
void Parser::parseInputStream(InputStream& inputStream)
{
    auto buffer = inputStream.getInputBuffer();

    if (buffer[0] == '.') {
        isMetaCmd = true;
        metaCmdType = parseMetaCmdType(buffer);
    } else {
        isStatement = true;
        statementType = parseStatementType(buffer);
    }

    if ((isMetaCmd && metaCmdType == META_COMMAND_UNDEFINED) ||
        (isStatement && statementType == STATEMENT_UNDEFINED)) {
        throw ORDER_UNDEFINED;
    }

    conditions = split(buffer, ' ', 1, 1);
}
```

在 parseInputStream 类中，我们对指令的基本类型进行辨识并调用上述相应方法解析，并将指令后面的条件语句拆分并存入 conditions 中，其中，split 方法实现如下

```cpp
std::vector<std::string> split(const std::string buffer, const char syntax)
{
    auto preLocation = buffer.begin();
    auto curLocation = buffer.begin();
    std::vector<std::string> result;

    while (true) {
        if (curLocation == buffer.end()) {
            result.push_back(std::string(preLocation, curLocation));
            return result;
        }

        if (*(curLocation) != syntax) {
            curLocation ++ ;
            continue;
        }

        result.push_back(std::string(preLocation, curLocation));
        preLocation = ++ curLocation;
    }
}
```

Cursor 类

```
class Cursor
{
    using Tokens = std::vector<std::tuple<int32_t, std::string, std::string>>;
  private:
    Table* tablePtr;
    std::vector<std::string> conditions;

    int32_t numConditions;
    std::vector<int32_t> conditionPositions;
    std::vector<PrimKey> conditionValues;
    std::string conditionCmpSign;

    int32_t primKeyPos;
    int32_t primKeyLowerBound;
    int32_t primKeyUpperBound;

    int32_t numParams;
    std::vector<uint32_t> paramPositions;
    std::vector<std::string> paramValues;

    Tokens extractFrom(const std::string& main, const std::string& pattern) const;
    bool compare(const PrimKey& a, const PrimKey& b, char cmpType) const;

  public:
    Cursor(Table* table);
    ~Cursor() = default;

    void extractConditions(const std::string& conditions);
    void extractParams(const std::string& params);
    bool checkConditions(const Row* row) const;

};
```

Cursor 类会将 Parser 类中保存的 conditions 进一步解析、拆分成可以直接进行判断的形式，并保存在类内成员中

我们提供了私有成员 extractFrom 供公有方法 extractConditions 和 extractParams

```
Tokens Cursor::extractFrom(const std::string& main, const std::string& pattern) const
{
    Tokens result;

    if (main.empty()) {
        return result;
    }

    size_t curLocation = 0;
    size_t length = main.length();

    while (curLocation < length) {
        auto endLocation = main.find(pattern, curLocation);
        if (endLocation == std::string::npos) {
            endLocation = main.length();
        }

        auto singlePhase = split(main.substr(curLocation, endLocation), ' ');
        auto& paramName = singlePhase[0];
        auto& cmpSign = singlePhase[1];
        auto& paramValue = singlePhase[2];

        // TODO: 先按字符存到 update 的值中，再根据condition转移
        auto index = tablePtr->findParamPos(paramName);
        if (index == -1) {
            throw PARAM_NOT_FOUND;
        }

        if (cmpSign != "<" || cmpSign != ">" || cmpSign != "=" ) {
            throw OPERATION_UNSUPPORTED;
        }

        result.emplace_back(index, cmpSign, paramValue);
        curLocation = endLocation + pattern.length() + 1;
    }

    return result;
}
```

Tokens 是一个 std::vector 容器，单位元素为一个三元组 std::tuple，存放每一个 condition 的对象 操作符 和 值，extractFrom 会根据提供的模式串将主串分割，并将结果存入 Tokens 中

```
void Cursor::extractConditions(const std::string& conditions)
{
    auto tokens = extractFrom(conditions, "and");
    if (tokens.empty()) {
        numConditions = 0;
        return;
    }

    for (auto& [index, cmpSign, value] : tokens) {
        conditionPositions.push_back(index);
        conditionCmpSign += cmpSign;
        conditionValues.push_back(PrimKey(value));

        if (index == tablePtr->primKeyPos) {
            // Update the range of primKey
            if (cmpSign == "<") {
                if (primKeyLowerBound == -1 || conditionValues[primKeyLowerBound] < co
                    primKeyLowerBound = numConditions;
                }
            } else if (cmpSign == ">") {
                if (primKeyUpperBound == -1 || conditionValues[primKeyUpperBound] > co
                    primKeyUpperBound = numConditions;
                }
            } else {
                // have a condition that primKey = value
                primKeyPos = numConditions;
            }
        }
        numConditions ++ ;
    }
}

void Cursor::extractParams(const std::string& params)
{
    auto tokens = extractFrom(params, ",");
    if (tokens.empty()) {
        numParams = 0;
        return;
    }

    for (auto& [index, cmpSign, value] : tokens) {
        if (cmpSign != "=") {
            numParams = 0;
```

```
            throw OPERATION_UNSUPPORTED;
        }
        paramPositions.push_back(index);
        paramValues.push_back(value);
        numParams ++ ;
    }
}
```

extractConditions 函数会将 extractFrom 函数返回的三元组存入类内成员中，并且在遍历 Tokens 时单独提取主键相关的 conditions

extractParams 用于 update 操作，将 对象 和 值存入类内成员中

```
bool Cursor::checkConditions(const Row* row) const
{
    if (numConditions == 0) {
        return true;
    }

    for (int i = 0; i < numConditions; i ++ ) {
        auto& index = conditionPositions[i];
        auto& cmpSign = conditionCmpSign[i];
        auto& value = conditionValues[i];

        PrimKey temp;
        temp.modifyValue(row->contents[index], tablePtr->paramTypes[index]);
        if (compare(value, temp, cmpSign)) {
            continue;
        } else {
            return false;
        }
    }

    return true;
}
```

checkCondition 方法会将 Cursor 内记录的条件与实际的行进行比较进行筛选满足所有条件的行，因为需要满足所有的条件，所以不需要单独对主键进行判断

其中，compare 方法的定义如下

```
bool Cursor::compare(const PrimKey& a, const PrimKey& b, char cmpType) const
{
    switch (cmpType) {
        case '<':
            return a < b;
        case '>':
            return a > b;
        case '=':
            return a == b;
        default:
            throw OPERATION_UNSUPPORTED;
    }

}
```

compare 会根据提供的比较方式比较两个 param 的大小并返回相应的 bool 值

下周目标

- [ ] 完成 B+ 树的编写
- [ ] 有空的话完成 page 和 table 的编写