

YatHA 开发日志

TODO

1. 规范命名变量(采用驼峰命名法)，引入 Xmake 支持跨平台编译
2. 配置外部化，避免 `hardcode`
3. 敏感词过滤（放弃，不是没眼看就是没法看）
4. 词性过滤
5. 动态滑动窗口大小
6. 交互式可视化
7. 服务部署（找不到可用的服务器/(T o T)/~~）
8. CI/CD（同上）
9. 滚动查询与图表支持

v0.1 完成基础滑动窗口热词统计功能

遇到的问题

1. 输出文件每次都被覆盖的问题

```
std::ofstream out(outputFile, std::ios::binary);
countTopKWords(out);
```

- **现象：**`countTopKWords()` 函数在每次执行 ACTION 命令时都会创建新的 `std::ofstream` 对象，导致文件被覆盖，只能看到最后一次的输出
- **原因：**在 `cutWord()` 中每次遇到 ACTION 时都执行 `std::ofstream out(outputFile, std::ios::binary);`，默认模式为覆盖模式 查阅资料得到以下解决方案：
- **解决方案：**
 - 方案1：使用追加模式 `std::ios::binary | std::ios::app`
 - 方案2：将 `ofstream` 作为类成员变量，在构造函数中打开，析构函数中关闭
 - 方案3：使用静态变量标记首次写入，首次覆盖，后续追加
- **方案分析：**
 - 方案1：如果 `output.txt` 已存在，那么每次运行程序后 `output.txt` 都会变长，显然不是我们想要的结果
 - 方案2：将文件流作为类成员，在构造函数中打开一次，整个对象生命周期内保持打开状态，符合面向对象设计原则，且确保每次运行都是新文件
 - 方案3：需要额外的静态变量维护状态，增加了复杂度，且如果创建多个 `HaEngine` 对象会出现问题 故选择方案2

```
class haEngine
{
    private:
        ...
        std::ofstream out;
```

```
};

// 初始化 jieba 和 swmanager
HaEngine::HaEngine(int window, int k, const std::string &i, const
std::string &o) :
jieba("../data/dict/jieba.dict.utf8",
      "../data/dict/hmm_model.utf8",
      "../data/dict/user.dict.utf8",
      "../data/dict/idf.utf8",
      "../data/dict/stop_words.utf8"),
swManager("../data/dict/stop_words.utf8"),
max_window_size(window), top_k(k), inputFile(i), outputFile(o) {
out.open(outputFile, std::ios::binary); }
```

2. 同一时间戳多条数据触发重复删除导致词频异常

```
// 错误代码
else
{
    curr_time = timeSec;          // 更新当前时间
    remove_outdate_words();      // 每条弹幕都删除一次
}
```

- **现象**：程序运行后，后期查询的热词频率都变成了1次，热词统计失效
- **原因**：当窗口满了（达到600秒）后，同一时间戳的多条弹幕会重复触发删除逻辑
- **问题分析**：
 - 假设第600秒有100条弹幕（同一时间戳）
 - 第1条弹幕：timeSec = 600，curr_time 从599更新为600，调用 remove_outdate_words() 删除第0秒的所有词
 - 第2条弹幕：timeSec = 600，但 curr_time 已经是600，cur_window_size >= max_window_size 仍然成立，再次进入 else 分支
 - 又调用 remove_outdate_words()，这次删除第1秒的所有词
 - 第3-100条弹幕：继续删除第2秒、第3秒...的词
 - 结果：窗口被疯狂缩小，大量高频词被误删，导致剩余词频率都变成1
- **解决方案**：只在时间戳变化时触发删除操作

```
// 修正后代码
if (cur_window_size < max_window_size)
{
    if (curr_time != timeSec)
    {
        curr_time = timeSec;
        ++cur_window_size;
    }
}
else
{
}
```

```
if (curr_time != timeSec) // 增加时间戳变化判断
{
    curr_time = timeSec;
    remove_outdate_words();
}
}
```

v0.2 规范化命名，引入 Xmake 支持跨平台编译，支持基本 CLI

人生苦短，我选 Xmake

xmake.lua :

```
add_rules("mode.debug", "mode.release")

target("yatha")
    set_kind("binary")
    add_files("src/*.cpp")

    add_includedirs("include")
    add_includedirs("third_party/cppjieba")

    set_targetdir("bin")
    set_rundir("${projectdir}")
    set_runargs("input1.txt", "output.txt")

    if is_plat("linux", "macosx") then
        add_syslinks("pthread", "m")
    end
```

完成同等功能所需的 CmakeLists.txt:

```
cmake_minimum_required(VERSION 3.10)

project(yatha CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_SOURCE_DIR}/bin)

file(GLOB SOURCES "src/*.cpp")

add_executable(yatha ${SOURCES})
```

```
target_include_directories(yatha PRIVATE
    include
    third_party/cppjieba
)

if(UNIX OR APPLE)
    # 查找线程库 (比直接写 "pthread" 更规范)
    set(THREADS_PREFER_PTHREAD_FLAG ON)
    find_package(Threads REQUIRED)

    target_link_libraries(yatha PRIVATE
        Threads::Threads
        m
    )
endif()

add_custom_target(run
    COMMAND yatha input1.txt output.txt
    WORKING_DIRECTORY ${CMAKE_SOURCE_DIR} # 对应
set_rundir("${projectdir}")
    DEPENDS yatha
    COMMENT "Running yatha with default arguments..."
)
```


1. **可读性强** lua 是一门脚本语言，语法简洁直观比起 `CmakeLists.txt` 中一会大写一会小写，各种何意味的宏定义，xmake 的语法绝对是**小葱拌豆腐——一清二白了**
2. **构建流程简单** Cmake 通常需要

```
mkdir build
cd build
cmake ..
make
```

这一套繁琐的流程才能开始构建项目

但是对于 xmake，只要写好配置文件 `xmake.lua`（如上面所示，并不繁琐）不管在项目的哪一个目录，构建项目只需执行 `xmake`，运行可执行文件只需 `xmake run`（若任何依赖文件有更新会重新构建）~~对于正在焦头烂额赶大作业 DDL 的我来说这确实是一make my life easier子~~

Windows下使用 Xmake 遇到的编码问题

 alt text 查阅资料可知：Windows 默认代码页（操作系统中用于表示文本文件中字符的编码）通常是 GBK，这时需要为 MSVC 指定 `/utf8` 参数

```
if is_plat("windows") then
    add_cxflags("/utf-8", {tools = "cl"})
    add_cxflags("-finput-charset=UTF-8", "-fexec-charset=UTF-8", {tools
= {"gcc", "clang"}})
end
```

扩展命令行参数选项，实现词性筛选功能

1. 发现 cppjieba 库的作者在 `limonp` 文件夹中偷偷实现了很多很好用的工具函数

比如 `ArgvContext.hpp` 就是一个用来处理命令行参数的类

该类的构造函数：

```
ArgvContext(int argc, const char* const * argv) {
    for(int i = 0; i < argc; i++) {
        if(StartsWith(argv[i], "-")) {
            if(i + 1 < argc && !StartsWith(argv[i + 1], "-")) {
                mpss_[argv[i]] = argv[i+1];
                i++;
            } else {
                sset_.insert(argv[i]);
            }
        } else {
            args_.push_back(argv[i]);
        }
    }
}
```

于是在该类中添加一个 `public` 函数 `ReadArgv()` 实现命令行参数的读取

```
void ReadArgv(std::string& Input, std::string& Output, int& Windows, int&
TopK, std::unordered_set<std::string>& ftr,
std::unordered_set<std::string>& csr)
{
    if(this->HasKey("-h"))
    {
        PrintHelp();
        exit(0);
    }
    if(this->HasKey("-wc"))
    {
        PrintPOSHelp();
        exit(0);
    }
    if(this->HasKey("-i"))
    {
        Input = mpss_["-i"];
        if(args_[0] == ".\\yatha.exe")
            std::cout << "输入文件：" << "data\\" << Input << "\n";
        else
            std::cout << "输入文件：" << "data/" << Input << "\n";
    }
    else
```

```

{
    std::cout << "请指定输入文件\n";
    exit(1);
}
if(this->HasKey("-o"))
{
    Output = mpss_["-o"];
    if(args_[0] == ".\\yatha.exe")
        std::cout << "输出文件：" << "data\\" << Output << "\n";
    else
        std::cout << "输出文件：" << "data/" << Output << "\n";
}
else
{
    std::cout << "请指定输出文件\n";
    exit(1);
}
if(this->HasKey("-t"))
{
    Windows = std::stoi(mpss_["-t"]);
    std::cout << "时间窗口大小：" << Windows << "s\n";
}
if(this->HasKey("-k"))
{
    TopK = std::stoi(mpss_["-k"]);
    std::cout << "TopK：" << TopK << "\n";
}
}

```

2. 查阅cppjieba库资料得到词性对照表，根据用户输入来选择过滤/放行属于某种词性的词语

（敏感词过滤功能因为测试文本不方便生成，所以改做词性筛选）由于cppjieba库本身实现对某些词语的识别就不太准确，如“刘备”竟然被归为音译人名所以是否精准地筛选/放行某类词性的词语不在考虑范围内，这里只注重算法设计

在 `HaEngin` 类里分别维护无序集 `filter`, `chooser` 用来确定需要过滤/放行的词性，再对原来的 `cutWords()` 函数稍加修改即可实现功能

v0.3 实现词性过滤/放行功能

在 `HaEngine` 类中增加了 `cutWordFilter` 和 `cutWordChooser` 两个成员函数

- **原理：**使用 `jieba.Tag()` 替代普通的 `jieba.Cut()`，这样分词结果会包含词性信息（如 `n` 表示名词，`v` 表示动词）
- **实现逻辑：**
 - `cutWordFilter`：遍历分词结果，如果某词的词性不在 `filter` 集合中，且不是停用词，则统计该词
 - `cutWordChooser`：遍历分词结果，如果某词的词性在 `chooser` 集合中，且不是停用词，则统计该词

```
if (!swManager.isStopWord(wordWithCls[i].first) &&
    filter.find(wordWithCls[i].second) == filter.end())
{
    // 加入统计...
}
```

同时增添命令行参数选项 `-wc`，打印词性标识符对应的词性，方便用户进行选择

v0.7 实现 Web GUI 界面

使用一个轻量级的 C++ HTTP 库 `cpp-httplib` 来搭建 Web 服务器（引用一个头文件就能启动服务器，非常方便），为用户提供一个 Web GUI 界面。

1. 后端实现

- **集成 httplib**：在 `yatha.cpp` 中引入 `httplib.h`
- **新增服务器模式**：
 - 修改 `ArgvContext` 类，增加 `-s` 参数解析，用来启动 HTTP 服务器
 - 当用户输入 `./yatha -s` 时，启动 HTTP 服务器监听 8080 端口
- **API 设计**：
 - 提供 `/api/analyze` POST 接口
 - 接收前端上传的文本内容，写入临时文件 `temp_input.txt`
 - 利用 `cpp` 的 RAII 机制控制 `HaEngine` 生命周期，确保结果写入磁盘后再读取返回

```
// RAII 确保文件流正确关闭
{
    HaEngine ha(..., tempInput, tempOutput);
    ha.cutWord();
} // ha 离开域，自动析构，确保文件正确关闭并保存
```

2. 前端实现

- **界面设计**：创建一个简洁的 HTML 页面，包含文件上传区域，顺便给实验室打一个小小的广告☺
- **交互逻辑**：
 - 支持点击上传和拖拽上传 `.txt` 文件
 - 使用 `fetch` API 将文件内容发送给后端
 - 收到响应后实时在页面上显示热词分析结果，无需页面跳转

3. 遇到的坑与解决方案

- **问题1：网页端分析结果显示为空**
 - 原因：`HaEngine` 还在运行（文件流未关闭）时就尝试读取输出文件，导致读到空内容
 - 解决办法：将 `HaEngine` 的实例化放入独立作用域 `{ }` 中，强制其在读取前析构并刷新文件流
- **问题2：文件覆盖风险**

- 原因：最初使用 `input1.txt` 作为临时文件，容易覆盖用户数据
- 解决办法：改用 `temp_input.txt` 和 `temp_output.txt`

• 问题3：文件挂载问题

- 原因：前端访问静态文件的请求需要处理之后返回 如

```
svr.Get("/index.html", [](auto& req, auto& res){
    // 1. 打开 web/index.html
    // 2. 读取内容
    // 3. 设置 Content-Type 为 text/html
    // 4. 发送
});

svr.Get("/style.css", [](auto& req, auto& res){
    // 1. 打开 web/style.css
    // ... 重复
});
```

- 解决方案，使用 `svr.set_mount_point()` 函数，一行代码即可实现静态目录的挂载，无需手动处理每个文件的请求。

```
// Serve static files from web directory (assuming running from
data/ directory)
svr.set_mount_point("/", "../web");
svr.set_mount_point("/img", "../img");
```

• 问题4：拖放功能默认行为

- 原因：浏览器默认会对某一些行为做处理。如把文本文件拖进浏览器，默认行为是打开并显示。
- 解决办法：

```
e.preventDefault();
```

调用`preventDefault()`方法阻止默认行为。

v0.8 代码重构

当我准备在Web GUI 中新增滚动查询功能的时候，发现 `HaEngine` 非常臃肿，几乎所有功能都是在这个类中实现的。为了保证代码的可读性和减少未来不必要的调试麻烦，同时减少编译坐牢时间（如果函数都实现在一个类中，那么每一次小变动都会导致项目主要文件的重新编译，会极大地提升不幸福感，大作业要赶不完了），决定重构代码！

1. 原来的 `HaEngine` 类：

```

class HaEngine
{
    private:
        std::queue<Tword> historyQueue;           // 记录当前时间窗口的词语
        std::unordered_map<std::string, int> freqMap; // 记录当前时间窗口词语的频次
        std::set<Tword> rankingSet;               // 按词语出现频次升序排列集合

        std::vector<std::string> lines;
        std::vector<Tword> words;
        StopWordsManager swManager;

        const std::string inputFile{};
        const std::string outputFile{};
        std::ofstream out;                       // 在构造函数中打开文件，而不是在 countTopKWords中打开，避免内容覆盖；
        int maxWindowSize{};
        int currTime = -1;                       // 输入文件时间戳从0秒开始，初始时间戳设为 -1
        int curWindowSize{};
        int topK{};
        std::unordered_set<std::string> filter{};
        std::unordered_set<std::string> chooser{};

    public:
        cppjieba::Jieba jieba;

        HaEngine(const std::string& dictPath, const std::string& hmmPath,
            const std::string& userDictPath, const std::string& idfPath, const std::string& stopWordDictPath,
            int window, int k, std::unordered_set<std::string>& ftr, std::unordered_set<std::string>& chsr,
            const std::string &i, const std::string &o);
        void cutWordsTest();
        void cutWord();
        void cutWordFilter();
        void cutWordChooser();
        void writeOutput();
        bool readUtf8Lines(std::vector<std::string>& lines);
        void testOutput();
        void removeOutdatedWords();
        void countTopKWords(std::ofstream& out);
};

```

可以看到包括1) 时间窗口管理, 2) TopK 排行榜管理等功能的有关变量和函数统统挤在了这个类里, 看得头晕。

2. 重构代码！

HaEngine 作为总调度器，将原来的时间窗口管理、TopK排行榜管理分别分离到TimeWindowManager类和WordRanker类中

```
class WordRanker
{
    private:
        std::unordered_map<std::string, int> freqMap; // 词频映射
        std::set<Tword> rankingSet;                // 按频次排序的集合
        (freq, word)

    public:
        WordRanker() = default;

        // 添加一个词（词频+1）
        void addWord(const std::string &word);

        // 移除一个词（词频-1，如果为0则完全删除）
        void removeWord(const std::string &word);

        // 获取TopK词汇
        std::vector<std::pair<std::string, int>> getTopK(int k) const;

        // 获取排名集合（用于输出格式化）
        const std::set<Tword> &getRankingSet() const { return rankingSet; }
};
```

```
class TimeWindowManager
{
    private:
        std::queue<Tword> historyQueue; // 存储所有在窗口内的词
        int maxWindowSize;              // 最大窗口大小
        int currTime = -1;              // 当前时间戳
        int curWindowSize = 0;          // 当前窗口大小

    public:
        TimeWindowManager(int windowSize);

        // 判断是否需要移除过期词
        bool shouldRemoveOld(int newTime);

        // 获取并移除过期的词（返回需要删除的词列表）
        std::vector<Tword> getAndRemoveOutdatedWords();

        // 添加新词到窗口
        void addWord(int timestamp, const std::string& word);

        // Getter方法
        int getCurrentTime() const { return currTime; }
```

```
int getCurrentWindowSize() const { return curWindowSize; }  
bool isEmpty() const { return historyQueue.empty(); }  
};
```

重构之后的好处：

1. **单一职责** 每个类只负责一件事：**TimeWindowManager** 管理时间窗口，**WordRanker** 管理词频排名，**HaEngine** 负责协调调度。代码职责清晰，可读性更强了，易于维护
2. **降低耦合度** 各模块相对独立，接口明确。且修改一个模块的内部实现不会影响其他模块
3. **减少编译时间**（主要！！）假如想修改**WordRanker**的实现时，只需重新编译**word_ranker.cpp**及依赖它的文件，而不会触发整个项目的重新编译，节省时间（天生打工圣体，太好了剩余价值又能被充分压榨了）
4. **便于功能扩展**

v1.0 将词性过滤/放行功能整合入 Web GUI 界面

1. 提供两个新的接口 `/api/analyze-filter` 和 `/api/analyser-chooser`

因为原来已经实现了 `cutWordFiter()` 和 `cutWordChooser()` 函数，所以只需要提供两个新的接口供前端请求即可。

```
svr.Post("/api/analyze-filter", [](const httpplib::Request &req,  
httpplib::Response &res{}));  
svr.Post("/api/analyze-chooser", [](const httpplib::Request &req,  
httpplib::Response &res){});
```

2. 前端新增一个词性选择表，为用户提供词性过滤/放行功能

前端会根据所选模式来请求不同的接口：

```
// 根据模式选择API端点  
let apiUrl = '/api/analyze';  
if (posConfig.mode === 'filter' && posConfig.pos.length > 0) {  
    apiUrl = '/api/analyze-filter';  
} else if (posConfig.mode === 'allow' && posConfig.pos.length > 0) {  
    apiUrl = '/api/analyze-chooser';  
}  
  
// 准备请求体  
const fileContent = await file.text();  
  
let response;  
if(apiUrl === '/api/analyze')  
{  
    response = await fetch(apiUrl,  
    {
```

```
        method: 'POST',
        headers:
        {
            'Content-Type': 'text/plain'
        },
        body: fileContent
    });
}
else
{
    response = await fetch(apiUrl,
    {
        method: 'POST',
        headers:
        {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            content: fileContent,
            pos: posConfig.pos.join(',')
        })
    });
}
```

遇到的问题

1. JSON 文件格式处理问题

/api/analyzer-...

```
// 1. 提取 content 字段
size_t contentStart = jsonBody.find("\"content\":") + 12;
size_t contentEnd = jsonBody.find("\"", contentStart);
std::string inputContent = jsonBody.substr(contentStart, contentEnd -
contentStart);

// 2. 提取 pos 字段
size_t posStart = jsonBody.find("\"pos\":") + 7;
size_t posEnd = jsonBody.find("\"", posStart);
std::string posString = jsonBody.substr(posStart, posEnd - posStart);
```

发现前端无法分析，只提示：[xx模式：xx] 查看生成的temp_chooser_output.txt文件，发现原因出在没有正确处理字符串中的换行符\n；

解决办法：

```
std::string inputContent;
for (size_t i = contentStart; i < jsonBody.length(); i++) {
    if (jsonBody[i] == '\\') && i + 1 < jsonBody.length())
```

```
{
  if (jsonBody[i + 1] == 'n') { inputContent += '\n'; i++; }
  else if (jsonBody[i + 1] == 't') { inputContent += '\t'; i++; }
  else if (jsonBody[i + 1] == '\"') { inputContent += '\"'; i++; }
  else if (jsonBody[i + 1] == '\\') { inputContent += '\\'; i++; }
  else inputContent += jsonBody[i];
}
else if (jsonBody[i] == '\"' && (i == 0 || jsonBody[i-1] != '\\'))
  break; // 找到content字段的结束引号
else
  inputContent += jsonBody[i];
}
```

建立一个循环来正确处理 JSON 数据中的换行符。

反思：

不过滤模式下，只需要返回文本即可，格式是`text/plain`，这是纯文本文件，不需要处理换行符。但是对于 JSON 字符串，就需要处理换行符了。

2. 不过滤模式下无法正常分析文本

检查`index.html`发现，不论是过滤/放行模式还是不过滤模式，http请求头的body都变成了 JSON 字符串

```
response = await fetch(apiUrl,
  {
    method: 'POST',
    headers:
    {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      content: fileContent,
      pos: posConfig.pos.join(',')
    })
  });
```

解决办法：添加一个条件判断，若是不过滤模式，则body为`text/plain`，否则为`application/json`

```
let response;
if(apiUrl === '/api/analyze')
{
  response = await fetch(apiUrl,
  {
    method: 'POST',
    headers:
    {
      'Content-Type': 'text/plain'
    },
```

```
        body: fileContent
    });
}
else
{
    response = await fetch(apiUrl,
    {
        method: 'POST',
        headers:
        {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            content: fileContent,
            pos: posConfig.pos.join(',')
        })
    });
}
```

v1.1 引入 nlohmann/json 库，从此不用手搓 JSON 字符串解析

(之前手搓的白写了.....this is life)

```
std::string inputContent;
for (size_t i = contentStart; i < jsonBody.length(); i++) {
    if (jsonBody[i] == '\\' && i + 1 < jsonBody.length())
    {
        if (jsonBody[i + 1] == 'n') { inputContent += '\n'; i++; }
        else if (jsonBody[i + 1] == 't') { inputContent += '\t'; i++; }
        else if (jsonBody[i + 1] == '"') { inputContent += '\"'; i++; }
        else if (jsonBody[i + 1] == '\\') { inputContent += '\\'; i++; }
        else inputContent += jsonBody[i];
    }
    else if (jsonBody[i] == '"' && (i == 0 || jsonBody[i-1] != '\\'))
        break; // 找到content字段的结束引号
    else
        inputContent += jsonBody[i];
}
```

因为后续打算实现的**滚动查询**功能打算通过预处理每一秒的 TopK 来实现（生成每一秒 TopK 的 JSON 文件），所以打算提前引入方便好用的 JSON 库。

上网查阅资料后发现 nlohmann/json 库广受好评，它可以：

1. 语法直观，可读性强

```
json j = {
    {"pi", 3.141},
    {"list", {1, 0, 2}},
```

```
    {"object", {{ "currency", "USD"}, {"value", 42.99}}}
};
```

2. 只需引入一个头文件 `json.hpp` (与 `httpLib.h` 一样)

3. 与 STL 容器集成度高

```
std::unordered_set<std::string> filter;
if(j.contains("pos"))
    filter = j["pos"].get<std::unordered_set<std::string>>();
```

可以很方便地就将 JSON 数组转换成 STL 容器

将原来 JSON 字符串处理用 `json.hpp` 提供的方法重写

```
json j = json::parse(req.body);
std::string inputContent;
if(j.contains("content"))
    inputContent = j["content"];
else
{
    res.set_content("JSON数据未提供\"content\"字段", "text/plain");
    return;
}
std::unordered_set<std::string> filter;
if(j.contains("pos"))
    filter = j["pos"].get<std::unordered_set<std::string>>();
else
{
    res.set_content("JSON数据未提供\"pos\"字段", "text/plain");
    return;
}
```

代码简洁了很多，且可读性更强了

v1.7 Web GUI 增加滚动查询功能

设计构想：实现一个滚动查询功能——用户可以在上传文件之后，设定所需要的时间窗口长度、TopK和步进长度，同时可以显示前 K 个热点词汇的柱状图和词云图。

1. 技术路线的选择

与大模型沟通之后得知，要实现这个功能有三种技术方案：

- 后端预处理，前端轮询
 1. 后端先处理每 `step` 秒（`step` 为步进长度，由用户设置）的数据生成 JSON 文件，前端按需轮询这些文件。

这种方法**最简单**但是有较高的性能瓶颈，因为每 `step` 秒后端都需要处理一次，太复杂。

- SSE (Server-Sent Events)

1. 后端实时推送数据流，前端通过 `EventSource` 接收服务器推送的数据。

这种方法**单向通信**，适合服务器主动推送场景，实现相对简单，但连接单向，前端无法主动请求特定时间段数据。

- WebSocket

1. 建立双向通信通道，前端可以随时请求特定时间段的数据，后端实时响应。

这种方法**最灵活**，支持双向通信，但实现相对复杂，需要维护连接状态。

最后选择了 SSE 方法，因为滚动查询是**单向数据流**场景，服务器按时间顺序推送即可，无需双向交互。且 `httplib` 库原生支持 SSE，实现简单，前端只需一个 `EventSource` 对象即可接收数据流，避免了 WebSocket 的状态管理复杂度。

2. 后端代码的扩展

新增一个 `HaEngine` 的子类 `HaEngineSSE` 为 SSE 做适配

```
class HaEngineSSE: public HaEngine
{
    private:
        int step;
        int lineIndex{};          // 永远指向需要读取的行
    public:
        HaEngineSSE(const std::string& dictPath, const std::string&
hmmPath, const std::string& userDictPath,
                    const std::string& idfPath, const std::string&
stopWordDictPath, int window, int k, std::unordered_set<std::string>& ftr,
                    std::unordered_set<std::string>& chsr, const
std::string &i, const std::string &o, int step);
        bool cutWord() override;
        void countTopKWords(std::ofstream& out) override;
        bool cutWordFilter() override;
        bool cutWordChooser() override;

        // 本来想实现实时更新这两个参数的功能，但是由于 SSE 只能由后端向前端单向推送数
据作罢
        // void updateStep(int newStep) { step = newStep; }
        // void updateTopK(int newTopK) { topK = newTopK; }
};
```

`step`和`lineIndex`分别用来记录步进长度（每一次读取多少秒的数据）和追踪行数。

```
HaEngineSSE::HaEngineSSE
(
    const std::string &dictPath, const std::string &hmmPath, const
std::string &userDictPath,
    const std::string &idfPath, const std::string &stopWordDictPath, int
```

```

window, int k, std::unordered_set<std::string> &ftr,
    std::unordered_set<std::string> &csr, const std::string &i, const
std::string &o, int step
) :
HaEngine(dictPath, hmmPath, userDictPath, idfPath, stopWordDictPath,
window, k, ftr, csr, i, o), step(step)
{
    // 读取文件内容到lines成员变量
    if (!readUtf8Lines(lines)) {
        std::cerr << "[错误] 无法读取输入文件: " << inputFile << std::endl;
        exit(1);
    }
    std::cout << "[HaEngineSSE] 已读取 " << lines.size() << " 行数据" <<
std::endl;
}

```

1. 类构造函数会先将数据分行，存放在父类的 `lines` 中。
2. 构造函数先不打开输出文件，因为输出文件每一次都需要覆盖而不是追加，选择在每一次调用 `countTopKWords()` 时打开。
3. 将 `cutWord()` 等分词函数改成返回 `bool` 值，用来判断是否已经读取完 `lines` 中的数据。重写基类函数，按步进长度 `step` 读取 `lines` 中的数据。其它算法基本保持一致。

2. 服务器 API 端口的暴露

在 `runWebServer()` 函数中暴露了一个 `/api/stream-analyze` 接口，用来将后端生成的数据组装成 SSE 信息持续推送给前端。（还好之前引入了 JSON 库，不然这会处理 JSON 字符串就是搬起石头砸自己的脚）

3. 设计过程中遇到的问题

1. 文件写入和读取的时序问题

- 现象：SSE 推送的数据时常为空或者读取到上一次的旧数据
- 原因：`countTopKWords()` 中使用成员变量 `out` 写入文件，但流未及时刷新，导致后续读取时缓冲区数据还未写入磁盘
- 解决：在 `countTopKWords()` 中每次都创建局部流对象，写完立即关闭

```

void HaEngineSSE::countTopKWords(std::ofstream& out)
{
    // 创建本地流对象，每次都覆盖文件
    std::ofstream localOut(outputFile, std::ios::out | std::ios::trunc);

    // ... 写入JSON数据 ...

    localOut << jsonStr << std::endl;
    localOut.flush(); // 确保刷新缓冲区
    localOut.close(); // 关闭文件
}

```

这样确保每次写入都完整刷新到磁盘，避免读取到空数据或旧数据

2. 并发冲突问题

- 现象：日志显示推送顺序混乱，第10次推送（540s）和第11次推送（600s）之间插入了第56次推送（3300s）

```
[SSE] 第10次推送: timestamp=540
[SSE] 第56次推送: timestamp=3300 // 顺序错乱
[SSE] 第11次推送: timestamp=600
```

- 原因：多个用户同时发起滚动分析请求时，所有请求共享同一个临时文件 `temp_input.txt` 和 `temp_output.txt`

```
// 错误做法：固定文件名导致并发冲突
std::string tempInput = "temp_input.txt";
std::string tempOutput = "temp_output.txt";
```

多个 SSE 连接同时写入和读取同一文件，导致数据交叉覆盖

- 解决办法：使用时间戳为每个请求生成唯一的临时文件名

```
// 使用时间戳生成唯一文件名
auto timestamp =
std::chrono::system_clock::now().time_since_epoch().count();
std::string tempInput = "stream_temp_input_" + std::to_string(timestamp) +
".txt";
std::string tempOutput = "stream_temp_output_" + std::to_string(timestamp)
+ ".txt";
```

每个 SSE 连接使用独立文件，避免并发冲突。同时在推送完成后清理临时文件：

```
// 清理临时文件
std::remove(tempInput.c_str());
std::remove(tempOutput.c_str());
```

3. 客户端断开连接后仍持续推送数据

- 现象：前端点击停止按钮后，终端仍显示后端在持续推送数据
- 原因：SSE 推送循环未检测连接状态，即使客户端取消读取，后端仍在执行数据处理和写入操作
- 解决方案：在循环中检测连接状态，及时终止推送

```
while (hasMoreData) {
    // 检查连接是否已断开
    if (!sink.is_writable()) {
```

```

        std::cout << "[SSE] 客户端已断开连接，停止推送" << std::endl;
        break;
    }

    // ... 处理数据 ...

    // 推送数据，检查写入是否成功
    if (!sink.write(sseMessage.c_str(), sseMessage.size())) {
        std::cout << "[SSE] 写入失败，客户端可能已断开" << std::endl;
        break;
    }
}

```

使用 `sink.is_writable()` 在每次循环开始前检测连接，并通过 `sink.write()` 的返回值判断写入是否成功。

4. Web 代码重构的问题

随着功能的增多，`index.html` 的代码越来越长，于是将样式和 JavaScript 脚本从 HTML 代码中分离出来，让 HTML 引入。

- **现象**：前端无法上传文件
- **原因**：HTML 在 `<head>` 中加载 `script.js`，js 会读取 DOM 元素时，但是此时这些 DOM 元素还不存在，因为 HTML 的 `<body>` 部分还没有开始，所以这些变量都是 `null`

```

const fileInput = document.getElementById('fileInput');
const resultDiv = document.getElementById('result');
const uploadArea = document.querySelector('.upload-area');
const fileStatus = document.getElementById('fileStatus');
const fileName = document.getElementById('fileName');
const analyzeBtn = document.getElementById('analyzeBtn');

```

- **解决办法**：引入 js 脚本时添加 `defer` 属性，确保脚本在 DOM 解析之后再执行。

```
<script src="script.js" defer></script>
```

v1.8 引入 Catch2 测试框架进行单元测试

HaEngineSSE 类中的 `cutWord()` 函数错误导致多处理一行数据

- **现象**：编写测试用例验证 `step=10` 时应该只处理 0s、5s、9s 的数据，不应处理 15s 的数据。但测试失败，葡萄（15s 时间戳）的计数为 4 而不是预期的 0

```

std::vector<std::string> lines = {
    "[0:00:00] 苹果 苹果",
    "[0:00:05] 香蕉 香蕉 香蕉",

```

```
"[0:00:09] 橙子 橙子",
"[0:00:15] 葡萄 葡萄 葡萄 葡萄" // 不应被处理
};

engine.cutWord();
REQUIRE(getWordCount(output, "葡萄") == 0); // 失败: 4 == 0
```

- **原因**：`cutWord()` 循环逻辑存在错误

```
// 错误的实现
while (TWManager.getCurrentTime() < threshold)
{
    // 先读取并处理当前行
    std::string line = lines[lineIndex++];
    // ... 处理数据 ...

    // 然后才检查时间戳（此时检查的是已处理行的时间）
}
```

循环条件检查的是上一行的时间戳，而不是即将处理的行的时间戳。导致当 `lineIndex` 指向第 4 行（15s）时，循环条件检查的是第 3 行（9s）的时间戳，判断 `9 < 10` 为真，于是处理了第 4 行，多处理了一行数据

- **解决方案**：重构循环逻辑，先预读时间戳，判断后再决定是否处理

```
// 正确的实现
while (lineIndex < lines.size())
{
    // 先预读下一行的时间戳
    std::string line = lines[lineIndex];
    std::smatch match;
    if (!std::regex_search(line, match, timeRegex)) break;

    int timeSec = parseTime(match[1].str());

    // 判断是否超出当前步进的时间范围
    if (timeSec >= threshold)
        break;

    // 确认在范围内，才处理这一行
    lineIndex++;
    // ... 处理数据 ...
}
```

同样的修复也应用到了 `cutWordFilter()` 和 `cutWordChooser()` 函数中，确保所有基于步进的分词函数都正确处理时间边界