# YatHA 热词统计分析系统设计文档

> Yet Another Tiny Hotwords Analyzer

本项目旨在构建一个高性能的实时热词分析系统，基于滑动时间窗口机制，能够从大量文本数据中快速提取和分析热门词汇以为弹幕分析、社交媒体监测、舆情分析等场景提供强大的技术支持。

## 一、背景假设与外部依赖

### （一）、输入数据假设

#### 1、直接分析功能的数据

1. 输入数据按照 `[H:MM:SS] <文本数据>` 的格式提供，每行一条文本数据。如：

```
[0:00:00] 先登！先登！
[0:00:00] 吃什么！
[0:00:00] 先登啦
```

在需要查询的位置插入一行`[ACTION] QUERY K=<整数>`。如：

```
[0:04:27] 你们三兄弟长的一点都不一样
[ACTION] QUERY K=6
[0:04:27] 诸葛瑾确实是神级管家！其他能力就一般了
```

2. 输入文件使用 **UTF-8** 编码

#### 2、滚动分析功能的数据

1. 输入的文本数据同样按照 `[H:MM:SS] <文本数据>` 的格式提供，每行一条文本数据。但不再需要在文件中提供 `[ACTION] QUERY K=<整数>` 指令，系统会按设定的步进长度逐段分析数据。

2. 输入文件同样使用 **UTF-8** 编码

### （二）、外部依赖

#### 1、编译器依赖

需要支持 **C++17** 标准的编译器，（建议 GCC 9.0 或 Clang 10.0 以上）

#### 2、第三方库

- cppjieba —— 用于中文分词处理
  - **版本**：v5.6.0
  - **许可证**：MIT license

- cpp-httplib —— 用于启动 HTTP 服务器
  - **版本**：v0.29.0
  - **许可证**：MIT license
- nlohmann-json —— 用于解析 JSON 数据
  - **版本**：v3.12.0
  - **许可证**：MIT license
- Catch2 —— 单元测试框架
  - **版本**：v3.11
  - **许可证**：BSD-1.0 license

**3、构建工具**

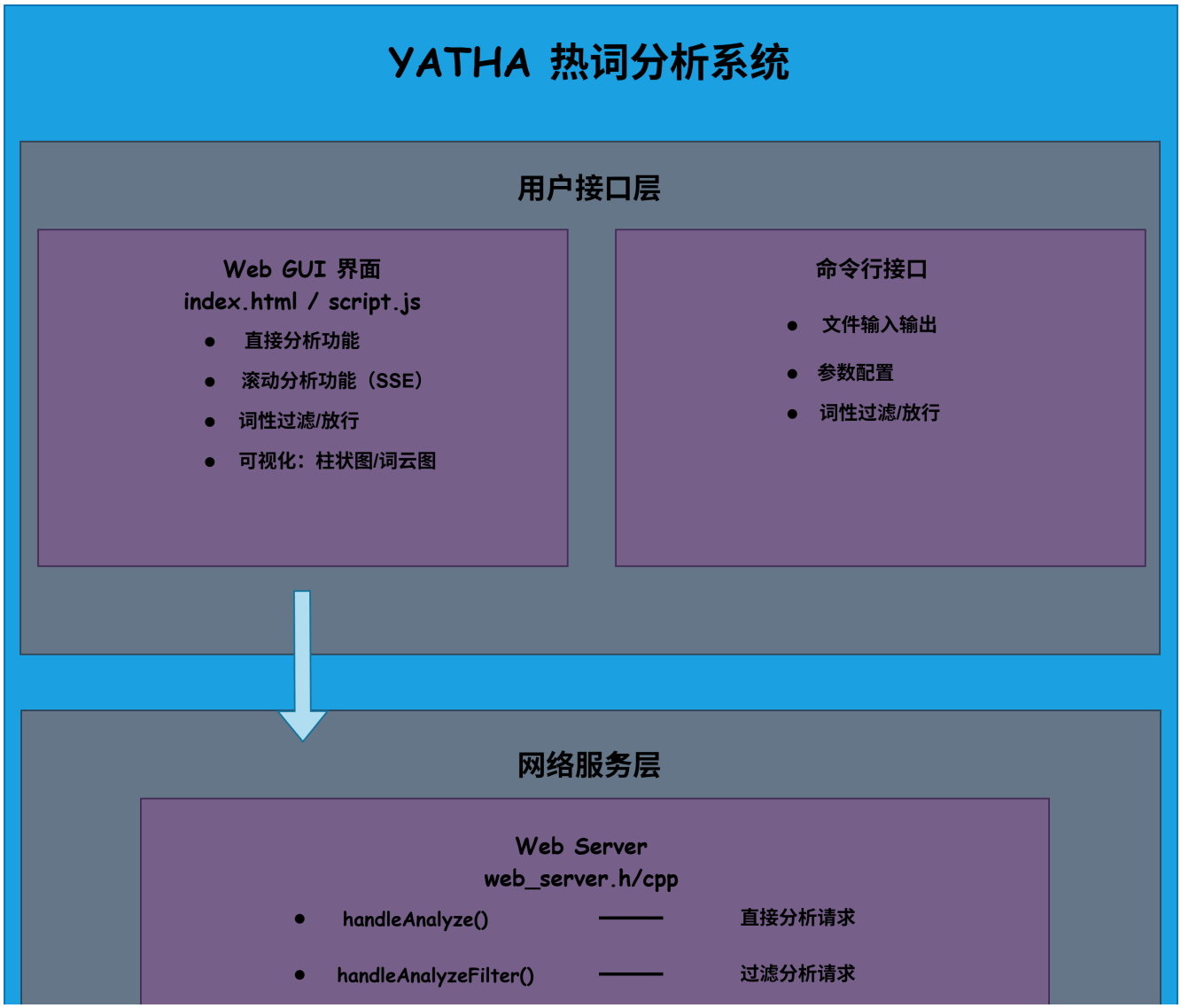
使用 Xmake 作为跨平台构建工具，需要安装 `xmake` v2.5.0 及以上版本

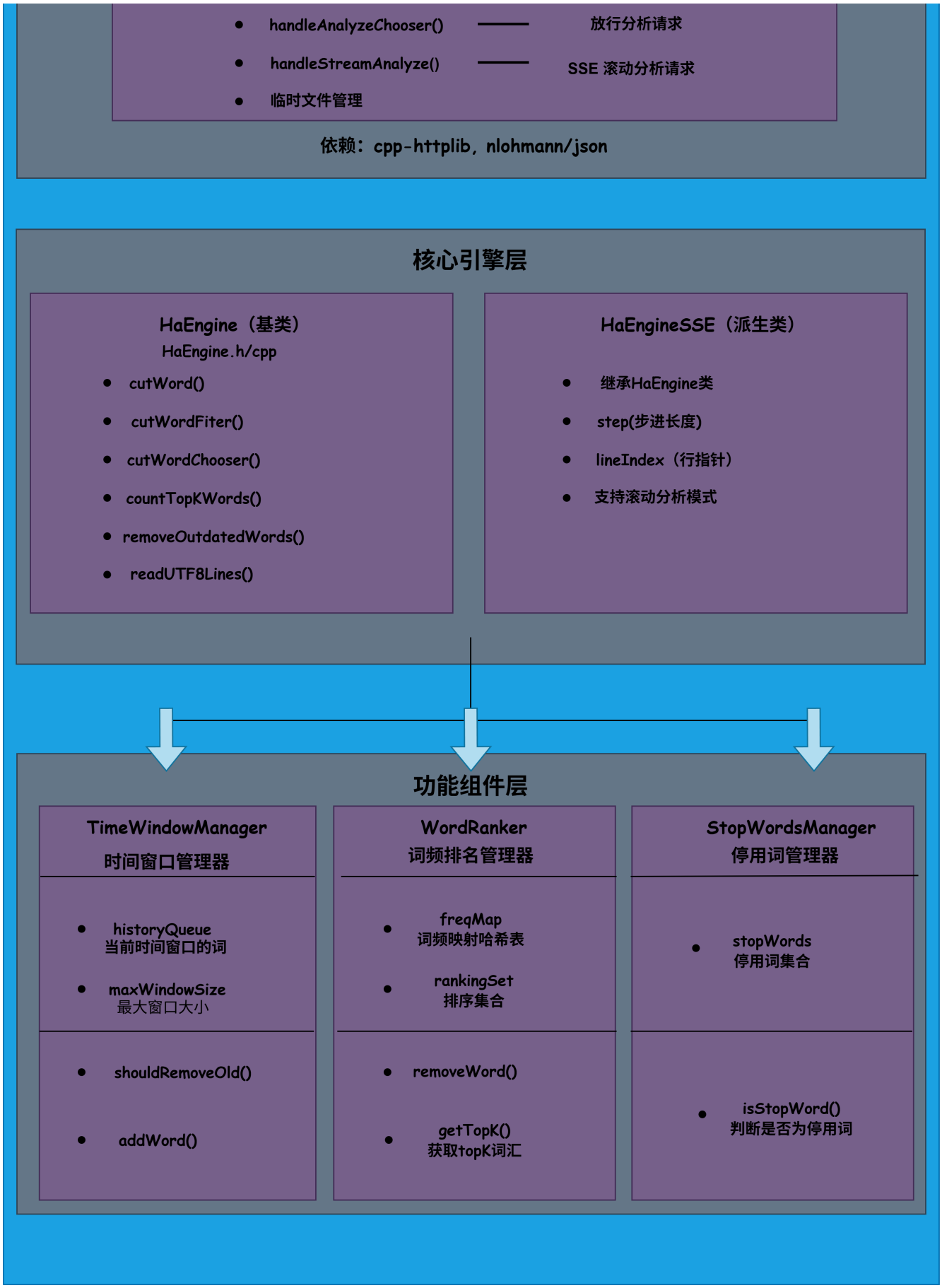
**网络要求**

**如果需要使用 Web GUI 界面，用户主机的 8080-8089 端口至少有一个未被占用**

# 二、模块/架构图与数据流

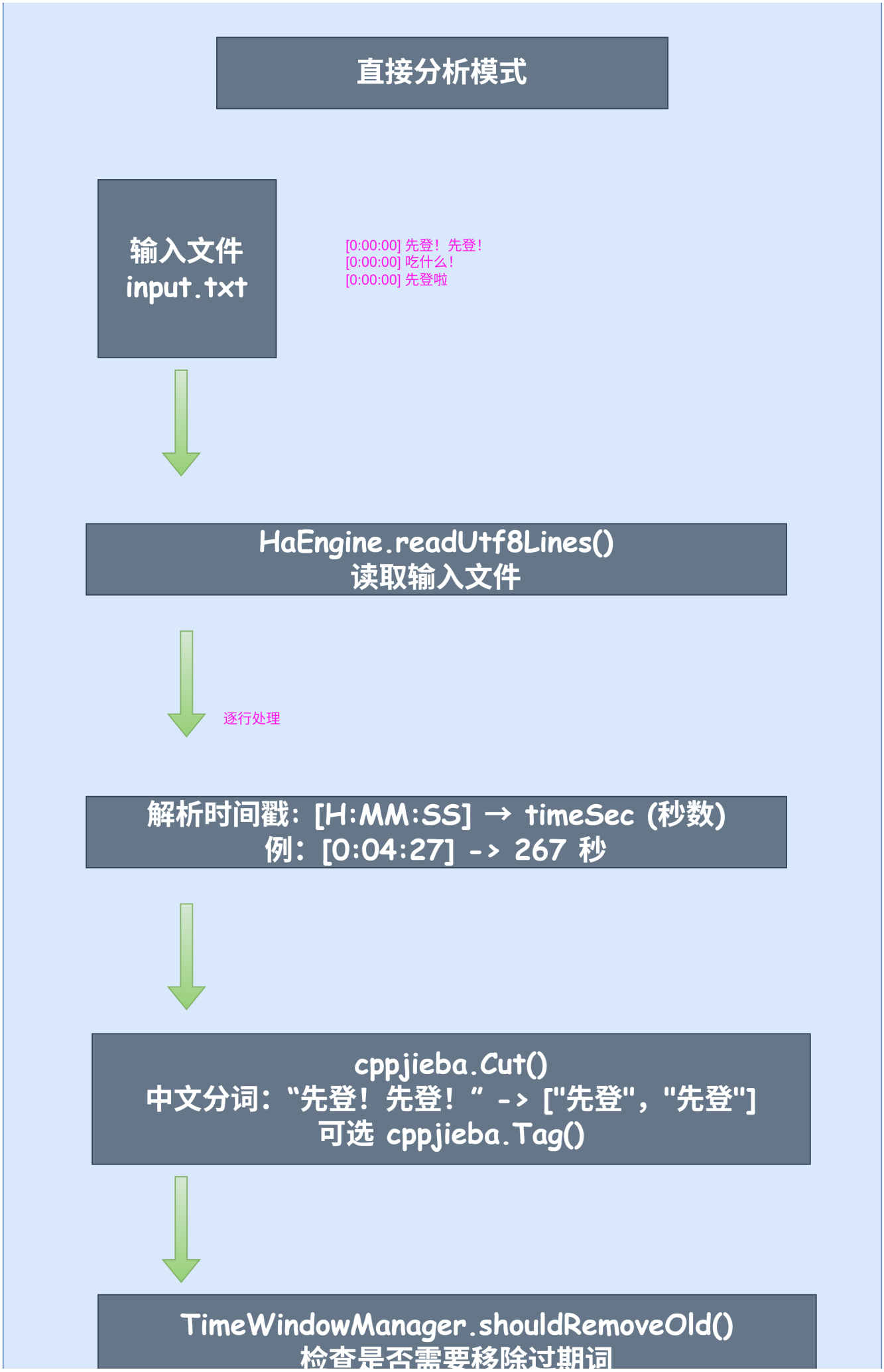
## （一）、系统架构图

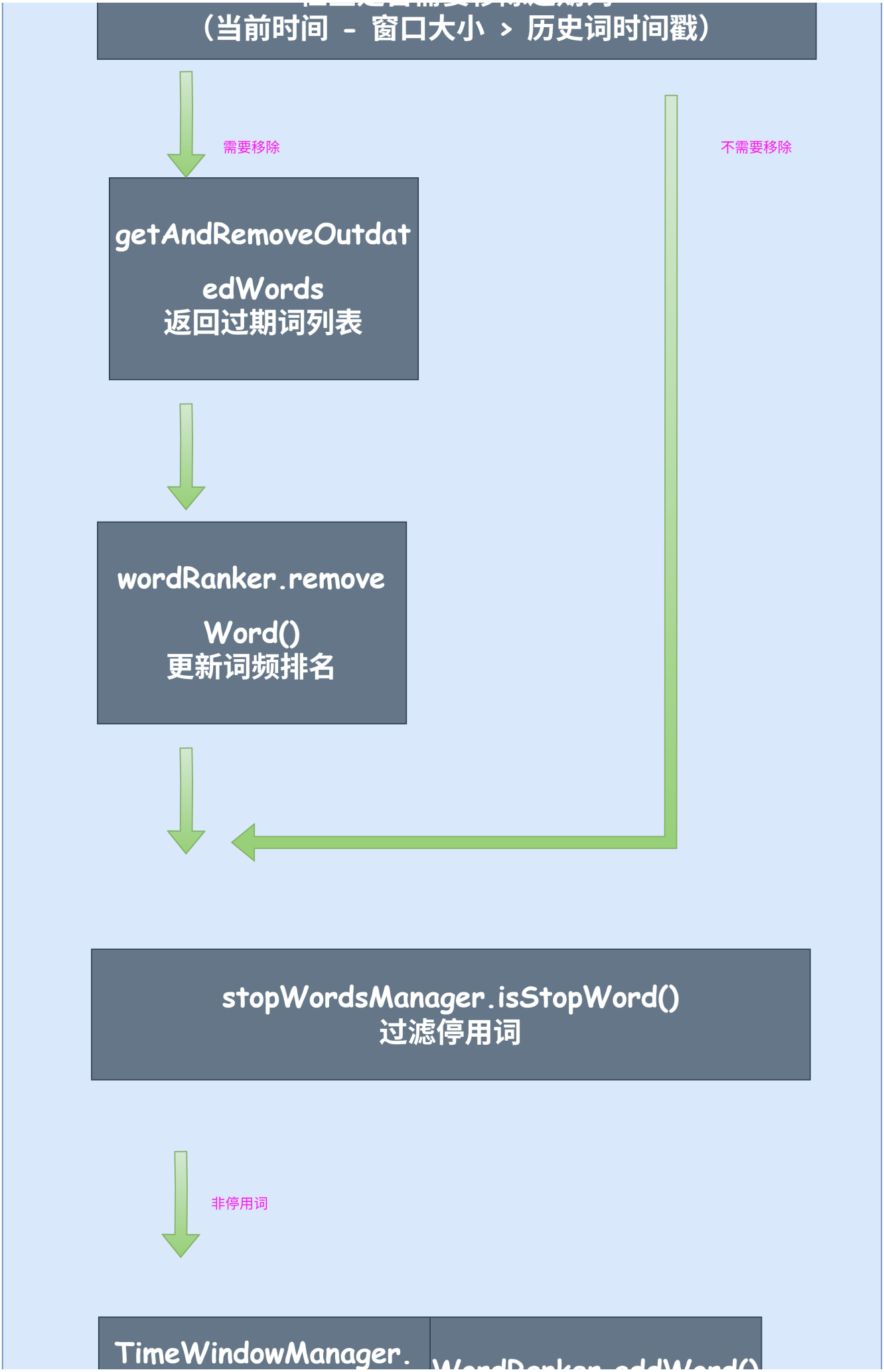




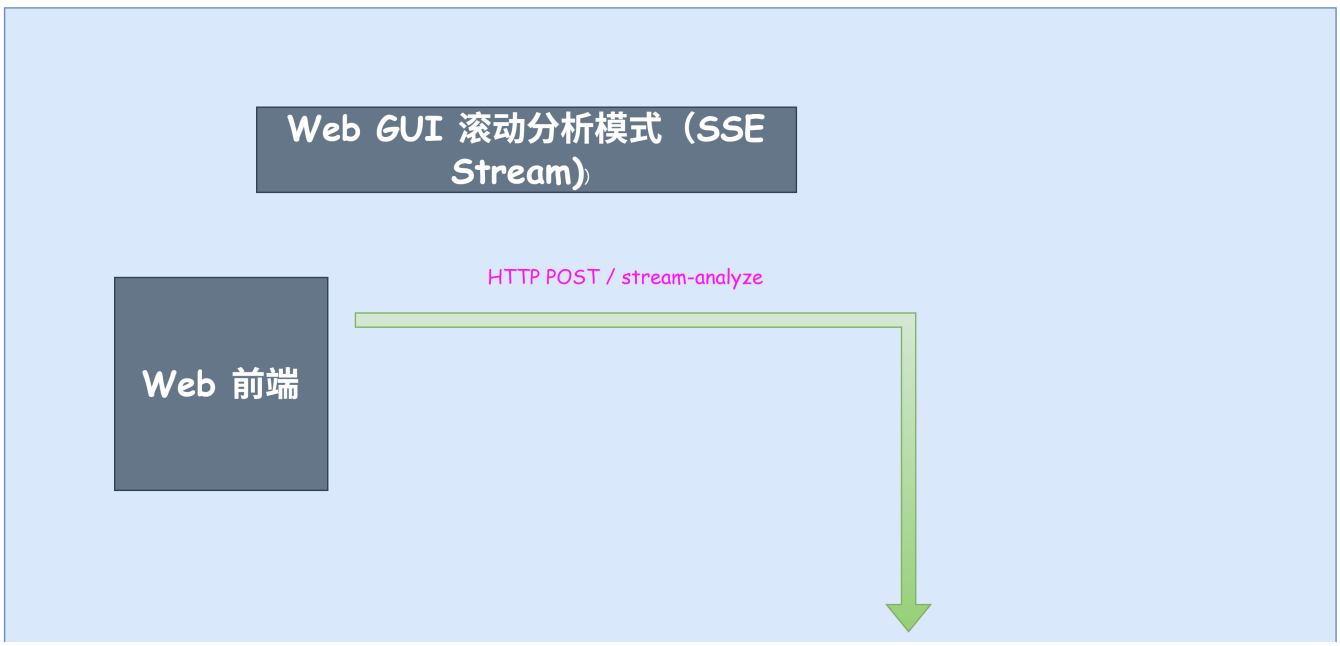

## （二）、数据流图

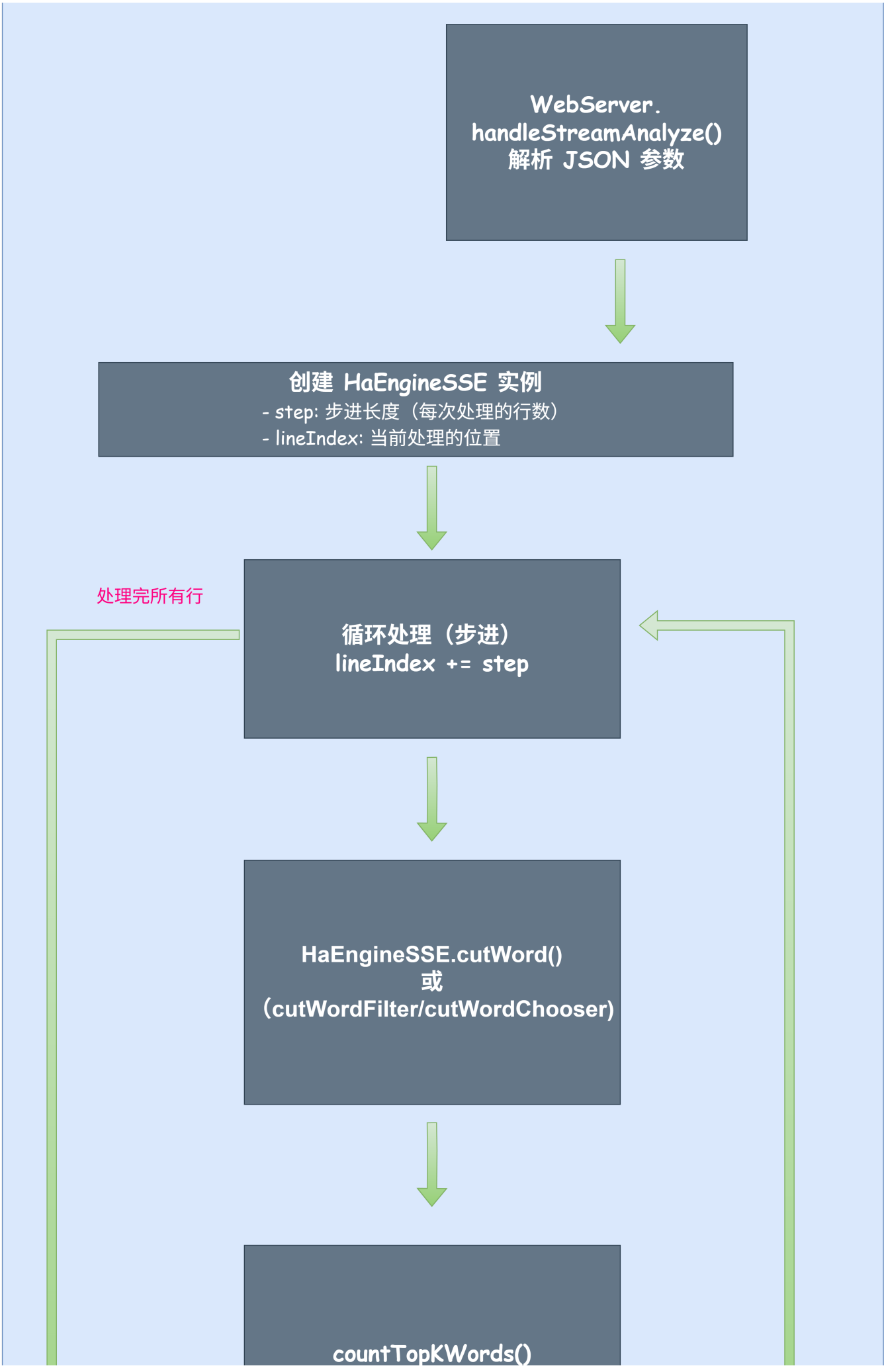
**1. 直接分析模式数据流**

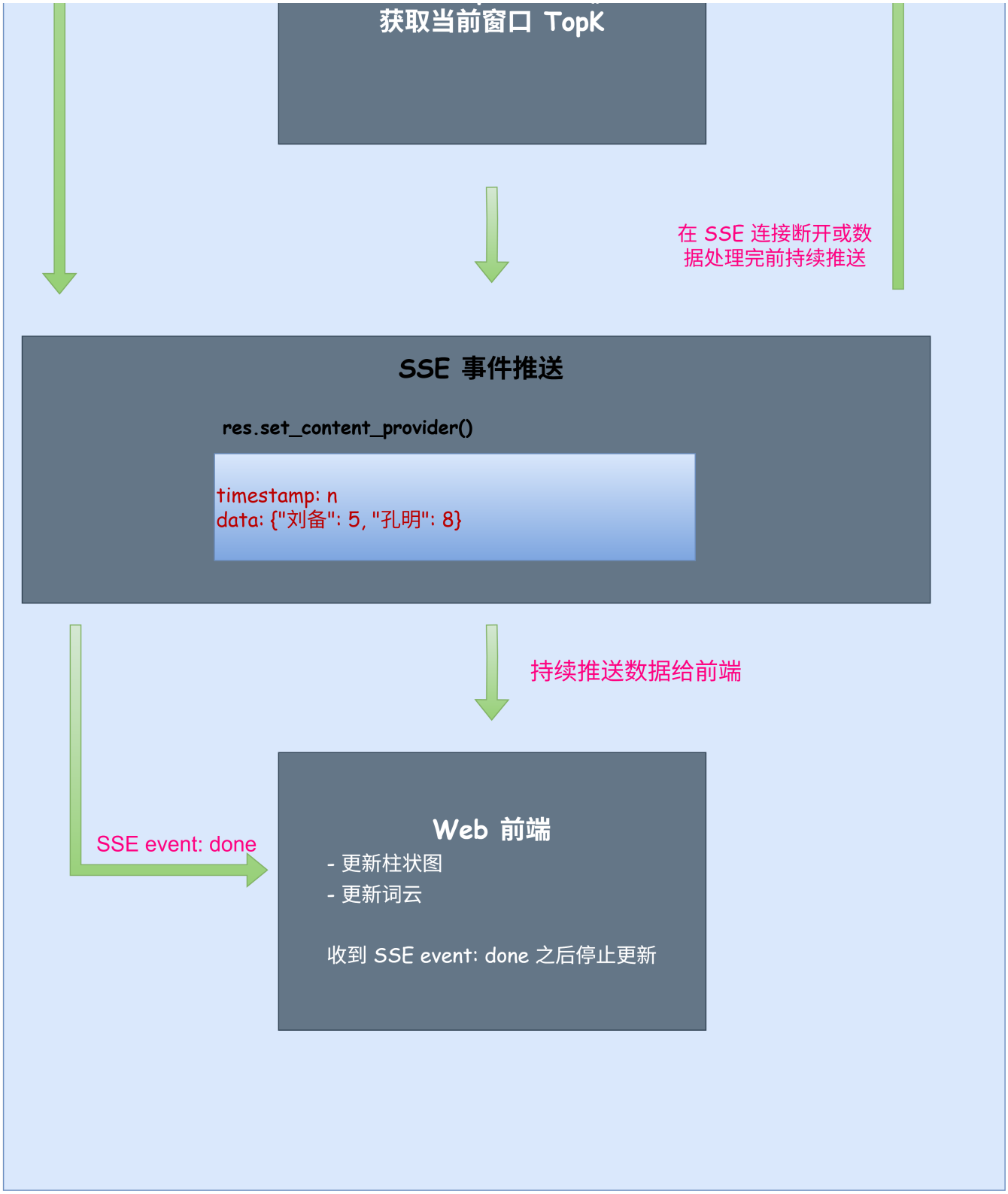

直接分析模式
输入文件
input.txt
[0:00:00] 先登！先登！
[0:00:00] 吃什么！
[0:00:00] 先登啦
HaEngine.readUtf8Lines()
读取输入文件
逐行处理
解析时间戳：[H:MM:SS] → timeSec（秒数）
例：[0:04:27] -> 267 秒
cppjieba.Cut()
中文分词："先登！先登！" -> ["先登"，"先登"]
可选 cppjieba.Tag()
TimeWindowManager.shouldRemoveOld()
检查是否需要移除过期词


（当前时间 - 窗口大小 > 历史词时间戳）
需要移除
不需要移除
getAndRemoveOutdat
edWords
返回过期词列表
wordRanker.remove
Word()
更新词频排名
stopWordsManager.isStopWord()
过滤停用词
非停用词
TimeWindowManager.


**2. Web GUI 滚动分析模式数据流 (SSE)**




WebServer.
handleStreamAnalyze()
解析 JSON 参数
创建 HaEngineSSE 实例
- step: 步进长度（每次处理的行数）
- lineIndex: 当前处理的位置
处理完所有行
循环处理（步进）
lineIndex += step
HaEngineSSE.cutWord()
或
(cutWordFilter/cutWordChooser)
countTopKWords()




## 三、核心数据结构和算法设计

### （一）、核心数据结构

#### 1. 时间窗口管理器 (TimeWindowManager)

```
class TimeWindowManager {
private:
    std::queue<Tword> historyQueue;  // 历史词队列
```

```
    int maxWindowSize;              // 最大窗口大小（秒）
    int currTime;                   // 当前时间戳
    int curWindowSize;              // 当前窗口大小
};
```

| 数据成员 | 类型 | 说明 |
| --- | --- | --- |
| historyQueue | std::queue<pair<int, string>> | FIFO 队列，存储 (时间戳, 词语) 对，按进入顺序排列 |
| maxWindowSize | int | 滑动窗口的最大时间跨度（默认 600 秒） |
| currTime | int | 当前处理的时间戳（秒） |
| curWindowSize | int | 当前窗口实际覆盖的时间长度 |

**设计理由**：使用队列实现 FIFO 语义，保证最早进入的词最先过期，与时间窗口的滑动特性完美匹配。

#### 2. 词频排名管理器 (WordRanker)

```
class WordRanker {
private:
    std::unordered_map<std::string, int> freqMap;  // 词频映射
    std::set<Tword> rankingSet;                    // 排序集合
};
```

| 数据成员 | 类型 | 说明 |
| --- | --- | --- |
| freqMap | unordered_map<string, int> | 哈希表，词语 → 词频，支持 O(1) 查找和更新 |
| rankingSet | set<pair<int, string>> | 红黑树有序集合，按 (词频, 词语) 排序，支持 O(log n) 插入/删除和 O(k) 的 TopK 查询 |

**设计理由**：双数据结构协同工作。`freqMap` 提供快速词频查询，`rankingSet` 维护全局有序性，避免每次查询时的排序开销。

#### 3. 停用词管理器 (StopWordsManager)

```
class StopWordsManager {
private:
    std::unordered_set<std::string> stopWords;  // 停用词集合
};
```

| 数据成员 | 类型 | 说明 |
| --- | --- | --- |
| stopWords | unordered_set<string> | 哈希集合，存储所有停用词，支持 O(1) 判断 |

**设计理由**：停用词判断是高频操作，使用哈希集合确保常数时间复杂度。

#### 4. 核心类型定义

```
using Tword = std::pair<int, std::string>;  // (时间戳/词频, 词语)
```

## （二）、核心算法设计

#### 1. 词频更新算法 (WordRanker)

**添加词语** `addWord(word)`：

```
1. 如果 word 已存在于 freqMap:
   a. 获取旧词频 oldCount = freqMap[word]
   b. 从 rankingSet 删除 (oldCount, word)
   c. 插入 (oldCount + 1, word) 到 rankingSet
   d. freqMap[word]++
2. 否则（新词）:
   a. freqMap[word] = 1
   b. 插入 (1, word) 到 rankingSet
```

**删除词语** `removeWord(word)`：

```
1. 如果 word 不存在于 freqMap，直接返回
2. 获取当前词频 count = freqMap[word]
3. 从 rankingSet 删除 (count, word)
4. 如果 count == 1:
   a. 从 freqMap 删除 word
5. 否则:
   a. freqMap[word]--
   b. 插入 (count - 1, word) 到 rankingSet
```

**时间复杂度分析**：

| 操作 | 时间复杂度 |
| --- | --- |
| `addWord()` | O(log n)，其中 n 为不同词的数量 |
| `removeWord()` | O(log n) |
| `getTopK()` | O(k)，从有序集合尾部反向遍历 |

#### 2. 滑动窗口过期检测算法 (TimeWindowManager)

**判断是否需要移除** `shouldRemoveOld(newTime)`：

```
1. 如果窗口未满 (curWindowSize < maxWindowSize) 且时间变化:
   a. 更新 currTime = newTime
   b. curWindowSize++
   c. 返回 false（不需要移除）
2. 如果窗口已满且时间变化:
   a. 更新 currTime = newTime
   b. 返回 true（需要移除过期词）
3. 否则返回 false
```

**获取并移除过期词 `getAndRemoveOutdatedWords()`：**

```
1. 计算过期阈值 threshold = currTime - maxWindowSize
2. 初始化过期词列表 outdated = []
3. 当队列非空且队首时间戳 <= threshold:
   a. 将队首元素加入 outdated
   b. 弹出队首
4. 返回 outdated
```

**3. TopK 热词查询算法**

```
1. 获取 rankingSet 的反向迭代器（从最大值开始）
2. 遍历前 k 个元素，收集 (词语, 词频) 对
3. 返回结果列表
```

由于 `rankingSet` 是按 (词频, 词语) 排序的 `std::set`，反向遍历自然获得降序结果。

### （三）、算法复杂度汇总

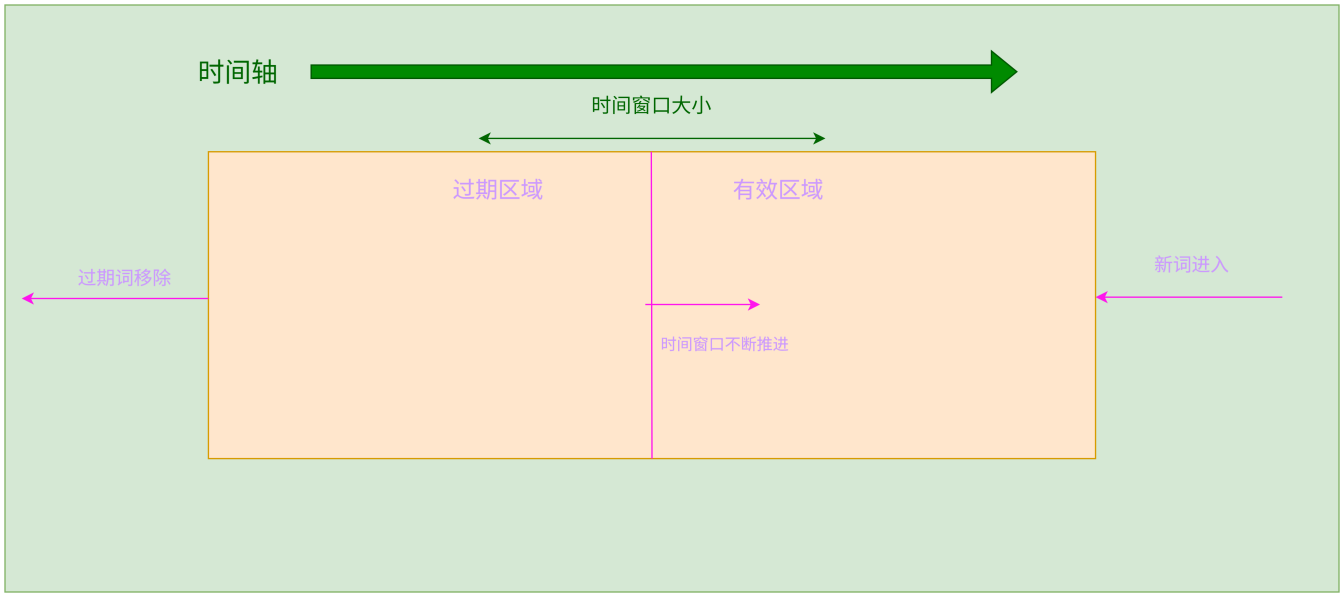
| 操作 | 时间复杂度 | 空间复杂度 |
|---|---|---|
| 添加词语到窗口 | O(log n) | O(1) |
| 移除过期词 | O(m · log n)，m 为过期词数量 | O(m) |
| TopK 查询 | O(k) | O(k) |
| 停用词判断 | O(1) | O(1) |
| 中文分词 (cppjieba) | O(L)，L 为文本长度 | O(L) |

## 四、滑动窗口和实时性保证

### （一）、滑动窗口机制

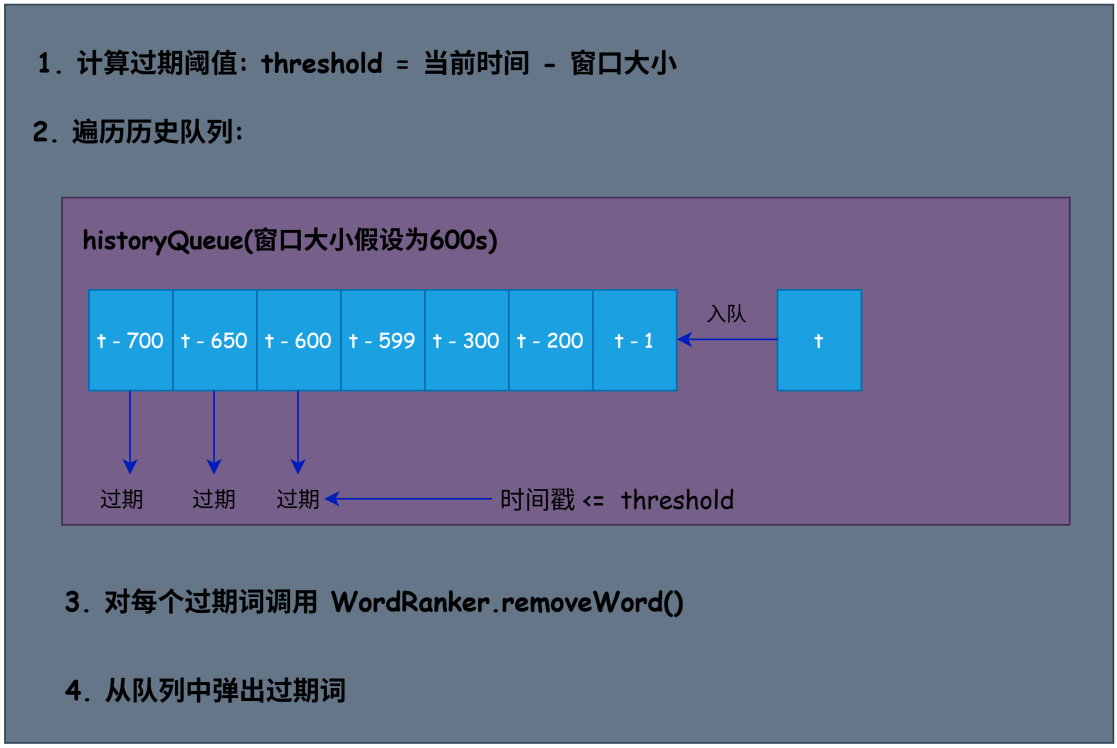
**1、 窗口模型**

本系统采用**基于时间的滑动窗口**模型，而非基于数量的固定大小窗口。




**2、 过期词处理流程**

当检测到需要移除过期词时：




## （二）、实时性保证

**1. 增量更新策略**

系统采用**增量更新**而非全量重算：

| 策略 | 每次查询复杂度 | 本系统采用 |
| --- | --- | --- |
| 全量重算 | O(n · log n) | ✗ |

| 策略 | 每次查询复杂度 | 本系统采用 |
| --- | --- | --- |
| 增量更新 | O(m · log n + k) | ✓ |

其中 n 为窗口内总词数，m 为过期词数量，k 为 TopK 的 k 值。

**2. 懒惰删除机制**

过期词的删除采用**懒惰策略**：

- 只在时间戳变化且窗口已满时触发删除
- 同一秒内的多条数据不触发删除检查
- 批量处理同一时刻过期的所有词

```
// 仅在时间变化时检查
if (TWManager.shouldRemoveOld(timeSec))
    removeOutdatedWords();  // 批量处理
```

**3、 步进控制参数**

SSE 模式支持以下实时性控制参数：

| 参数 | 说明 | 影响 |
| --- | --- | --- |
| `step` | 每次处理的时间步长（秒） | 步长越小，更新越频繁，实时性越高 |
| `speed` | 推送间隔时间（毫秒） | 间隔越短，前端更新越快 |
| `window` | 滑动窗口大小（秒） | 窗口越小，热词变化越敏感 |

**4、 性能优化措施**

1. **预加载数据**：`HaEngineSSE` 构造时一次性读取所有输入行到内存
2. **行索引管理**：使用 `lineIndex` 追踪处理进度，避免重复解析
3. **批量处理**：每个步进周期内批量处理多行数据，减少 I/O 次数
4. **异步推送**：使用 `set_chunked_content_provider` 实现非阻塞推送

## （三）、实时性指标

在典型配置下（窗口 600 秒，步进 10 秒）：

| 指标 | 数值 |
| --- | --- |
| 单次步进处理延迟 | < 50ms（取决于文本量） |
| TopK 查询延迟 | < 1ms |
| 过期词清理延迟 | < 10ms |
| 端到端响应时间 | < 100ms |

# 五、性能优化与资源评估方法

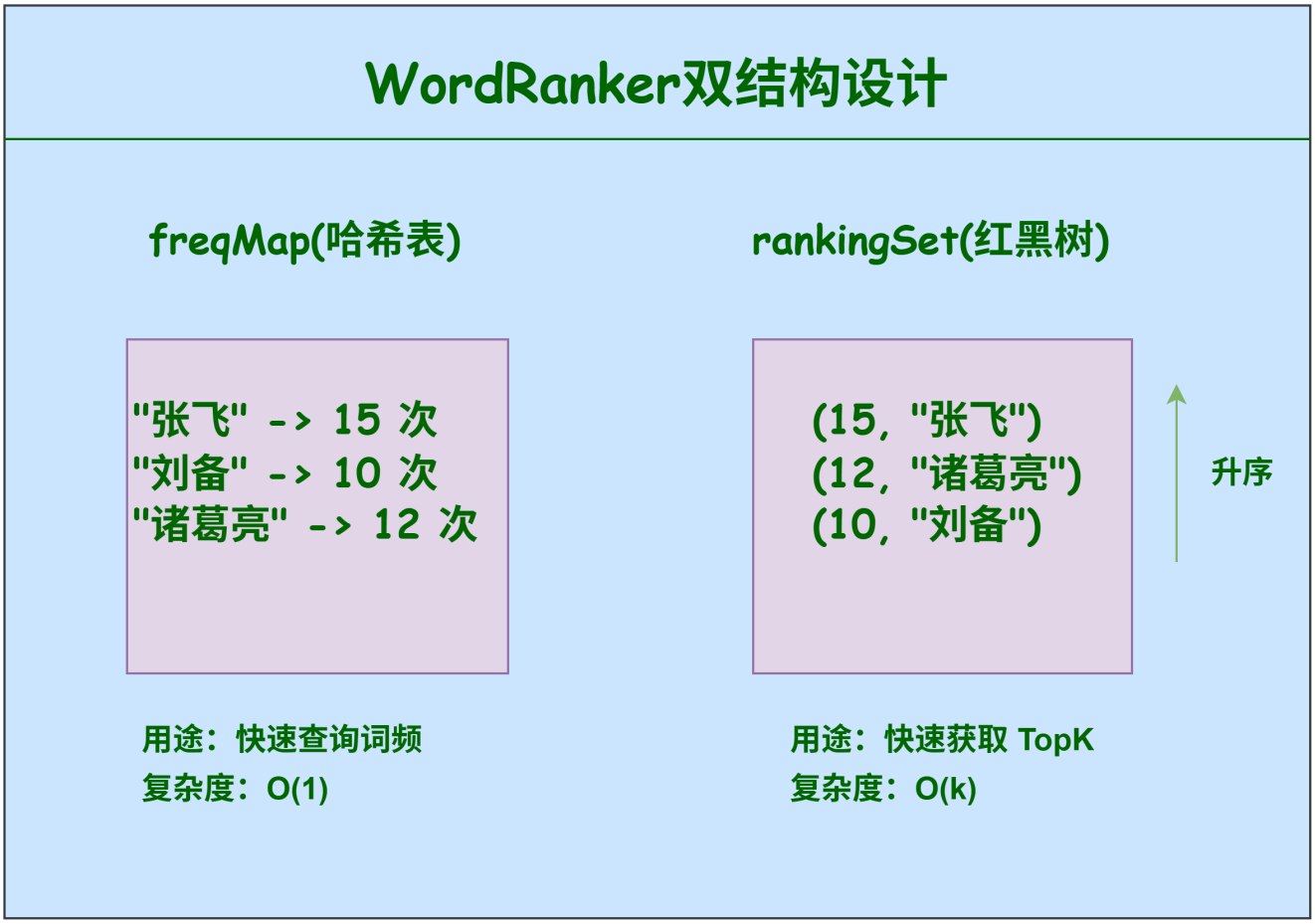
## （一）、性能优化策略

### 1. 数据结构层面优化

| 优化点 | 实现方式 | 性能收益 |
| --- | --- | --- |
| 词频查询 | `unordered_map` 哈希表 | O(1) 平均查询，避免线性搜索 |
| TopK 排序 | `std::set` 红黑树 | O(k) 查询，无需每次重新排序 |
| 停用词过滤 | `unordered_set` 哈希集合 | O(1) 判断，高频操作零开销 |
| 时间窗口 | `std::queue` FIFO 队列 | O(1) 入队出队，天然支持滑动语义 |

### 2. 算法层面优化

**双数据结构协同**：`WordRanker` 同时维护 `freqMap` 和 `rankingSet`，以空间换时间：




**增量更新 vs 全量重算**：

```
全量重算方案：每次查询 → 遍历所有词 → 排序 → 取 TopK
              复杂度：O(n · log n)

增量更新方案：词频变化 → 更新双结构 → 直接取 TopK
              复杂度：O(log n) 更新 + O(k) 查询
```

#### 3. I/O 层面优化

| 优化点 | 实现方式 | 说明 |
| --- | --- | --- |
| 预加载 | 构造时读取全部数据到内存 | 避免运行时 I/O 阻塞 |
| 批量写入 | 使用 `std::ofstream` 缓冲 | 减少系统调用次数 |
| 临时文件复用 | 固定临时文件路径 | 避免频繁创建/删除文件 |

#### 4. 编译层面优化

通过 `xmake.lua` 配置：

```
add_rules("mode.debug", "mode.release")  -- 支持 Release 优化编译
set_languages("c++17")                    -- 启用 C++17 特性
```

Release 模式下编译器自动启用：

- `-O2` 或 `-O3` 优化级别
- 内联函数展开
- 循环展开和向量化

### （二）、性能瓶颈与优化方向

#### 当前瓶颈分析

| 瓶颈 | 原因 | 影响程度 |
| --- | --- | --- |
| 中文分词 | cppjieba 基于字典匹配，计算密集 | 高 |
| 红黑树操作 | `std::set` 插入删除需平衡 | 中 |
| 字符串拷贝 | 词语在多个结构间传递 | 低 |

### （三）、实测性能数据

以下数据基于实际运行测试获得，测试环境如下：

#### 1. 测试环境

| 项目 | 配置 |
| --- | --- |
| 操作系统 | Linux (WSL2) 6.6.87.2-microsoft-standard |
| CPU | AMD Ryzen 7 8845H (8核16线程 @ 3.8GHz) |
| 内存 | 13 GB DDR5 |

| 项目 | 配置 |
| --- | --- |
| 编译器 | GCC (C++17, Release 模式) |

**环境信息获取命令：**

```
# 系统信息
uname -a

# CPU 信息
lscpu | grep -E "Model name|CPU\(s\)|Thread|Core"

# 内存信息
free -h
```

**2. 测试数据集**

| 数据集 | 行数 | 文件大小 | 说明 |
| --- | --- | --- | --- |
| input1.txt | 12,870 行 | 572 KB | 视频弹幕数据 |
| input2.txt | 11,698 行 | 528 KB | 视频弹幕数据 |
| input3.txt | 14,930 行 | 712 KB | 视频弹幕数据 |

**数据集信息获取命令：**

```
# 统计行数
wc -l data/input*.txt

# 统计文件大小
du -h data/input*.txt
```

**3. 性能测试结果**

**基准测试（窗口=600秒，TopK=10）：**

| 数据集 | 用户态时间 | 系统态时间 | 总耗时 | CPU 利用率 | 峰值内存 |
| --- | --- | --- | --- | --- | --- |
| input1.txt (12,870行) | 2.91s | 0.52s | 3.46s | 99% | 149.9 MB |
| input2.txt (11,698行) | 2.77s | 0.63s | 3.43s | 99% | 149.8 MB |
| input3.txt (14,930行) | 3.16s | 0.51s | 3.67s | 100% | 149.9 MB |

**测试命令：**

```

```

```
# 进入 data 目录运行（确保字典文件路径正确）
cd data

# 使用 GNU time 获取详细资源信息
/usr/bin/time -v ../bin/yatha -i input1.txt -o output_test.txt -w 600 -k 10
/usr/bin/time -v ../bin/yatha -i input2.txt -o output_test.txt -w 600 -k 10
/usr/bin/time -v ../bin/yatha -i input3.txt -o output_test.txt -w 600 -k 10

# 或使用 zsh/bash 内置 time 命令获取简洁输出
time ../bin/yatha -i input1.txt -o output_test.txt -w 600 -k 10
```

**吞吐量指标：**

| 指标 | 数值 |
|---|---|
| 平均处理速率 | **~3,730 行/秒** |
| 数据吞吐量 | **~166 KB/秒** |
| 单行平均处理时间 | **~0.27 ms/行** |

**多次运行稳定性测试（input1.txt）：**

| 运行次数 | 耗时 |
|---|---|
| Run 1 | 3.38s |
| Run 2 | 3.43s |
| Run 3 | 3.54s |
| **平均** | **3.45s** |
| **标准差** | **0.08s** |

**稳定性测试命令：**

```
# 多次运行测试
for i in 1 2 3; do
    /usr/bin/time -f "Run $i: %e秒" ../bin/yatha -i input1.txt -o
output_test.txt -w 600 -k 10 2>&1 | grep "Run"
done
```

**4. 参数敏感性分析**

**TopK 参数对性能的影响（input1.txt, window=600）：**

| TopK | 耗时 | 相对变化 |
|---|---|---|
| 5 | 3.58s | 基准 |

| TopK | 耗时 | 相对变化 |
|---|---|---|
| 10 | 3.47s | -3.1% |
| 20 | 3.57s | -0.3% |

**测试命令：**

```
# 测试不同 TopK 值
for k in 5 10 20 50 100; do
    echo "=== TopK=$k ==="
    /usr/bin/time -f "TopK=$k: %e秒 (user: %U, sys: %S)" \
        ../bin/yatha -i input1.txt -o output_test.txt -w 600 -k $k 2>&1 |
tail -1
done
```

**结论**：TopK 参数对总体性能影响很小（<5%），因为 TopK 查询本身是 O(k) 复杂度，而分词开销占主导。

**窗口大小对性能的影响（input1.txt, TopK=10）：**

| 窗口大小 | 耗时 | 内存占用 |
|---|---|---|
| 60s | 3.47s | 149.8 MB |
| 120s | 3.50s | 149.9 MB |
| 300s | 3.68s | 149.9 MB |
| 600s | 3.45s | 149.9 MB |

**测试命令：**

```
# 测试不同窗口大小
for w in 60 120 300 600 1200; do
    echo "=== Window=$w ==="
    /usr/bin/time -f "Window=$w: %e秒 (user: %U, sys: %S, mem: %MKB)" \
        ../bin/yatha -i input1.txt -o output_test.txt -w $w -k 10 2>&1 |
tail -1
done
```

**结论**：窗口大小对性能影响有限（<6%），内存占用主要由 cppjieba 字典加载决定（约 11 MB 字典文件）。

**5. 资源消耗分析**

**内存分布（基于 `/usr/bin/time -v` 输出）：**

| 组件 | 估算大小 | 说明 |
|---|---|---|
| cppjieba 字典 | ~140 MB | jieba.dict.utf8 (5MB) + idf.utf8 (6MB) 加载后展开 |

| 组件 | 估算大小 | 说明 |
| --- | --- | --- |
| 数据结构 | ~5 MB | historyQueue + freqMap + rankingSet |
| 程序代码 | ~5 MB | 代码段 + 栈 |
| **总计** | **~150 MB** | 与实测 149.9 MB 吻合 |

**字典文件大小查看命令：**

```
ls -la data/dict/*.utf8
```

**页面错误统计：**

| 类型 | 数量 | 说明 |
| --- | --- | --- |
| 主要页面错误 (I/O) | 0-6 | 极少，大部分数据在缓存中 |
| 次要页面错误 | ~85,000 | 正常的内存分配行为 |
| 上下文切换 | <20 | CPU 密集型任务，几乎无阻塞 |

**6. 性能结论**

1. **CPU 密集型**：CPU 利用率接近 100%，系统时间占比仅 15%，说明 I/O 不是瓶颈
2. **内存稳定**：不同数据集的内存占用几乎相同（~150MB），主要由字典加载决定
3. **线性扩展**：处理时间与数据量呈近似线性关系
4. **高吞吐量**：约 3,700 行/秒的处理速率，适合实时弹幕分析场景