热词统计与分析系统 - 系统设计文档

1. 背景假设与外部依赖

1.1 背景假设

1. **数据格式**：输入数据为带有时间戳的中文文本流，格式为 `[HH:MM:SS] 文本内容`。
2. **时间语义**：时间戳表示文本产生的时刻，系统基于此进行滑动窗口统计。
3. **数据规模**：单次分析的文本量在数万至数十万条之间，单条文本长度不超过 1000 字符。
4. **实时性要求**：系统需在毫秒级响应用户的 Top-K 查询请求。

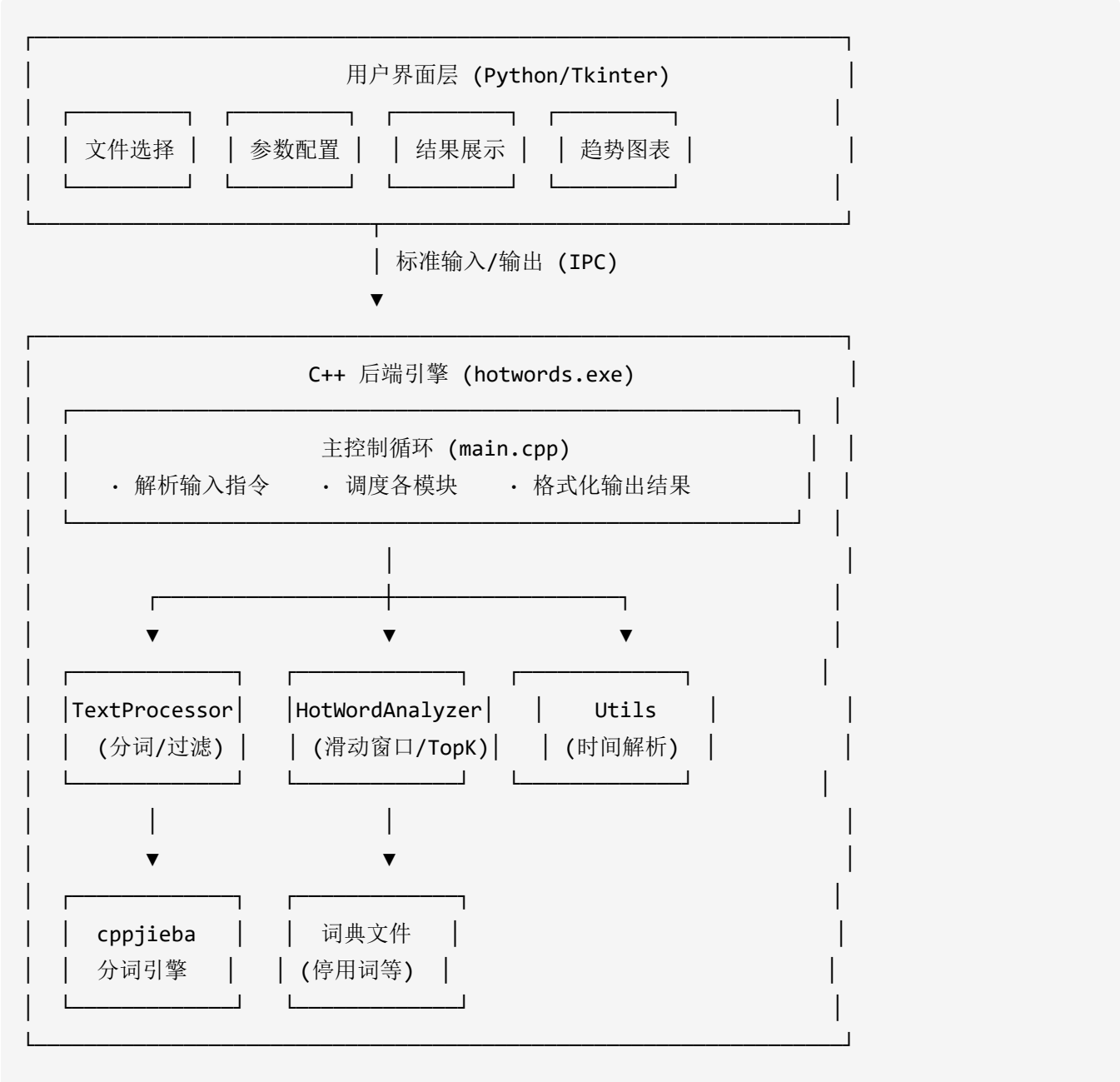
1.2 外部依赖

依赖项	版本	用途	许可证
cppjieba	5.0.3	高性能中文分词	MIT License
matplotlib	3.x	Python 数据可视化	PSF License
psutil	5.x	系统资源监控	BSD-3-Clause
MinGW g++	8.0+	C++ 编译器	GPL
Python	3.8+	前端 GUI 运行环境	PSF License

---

2. 系统架构设计

2.1 整体架构图



2.2 项目文件结构

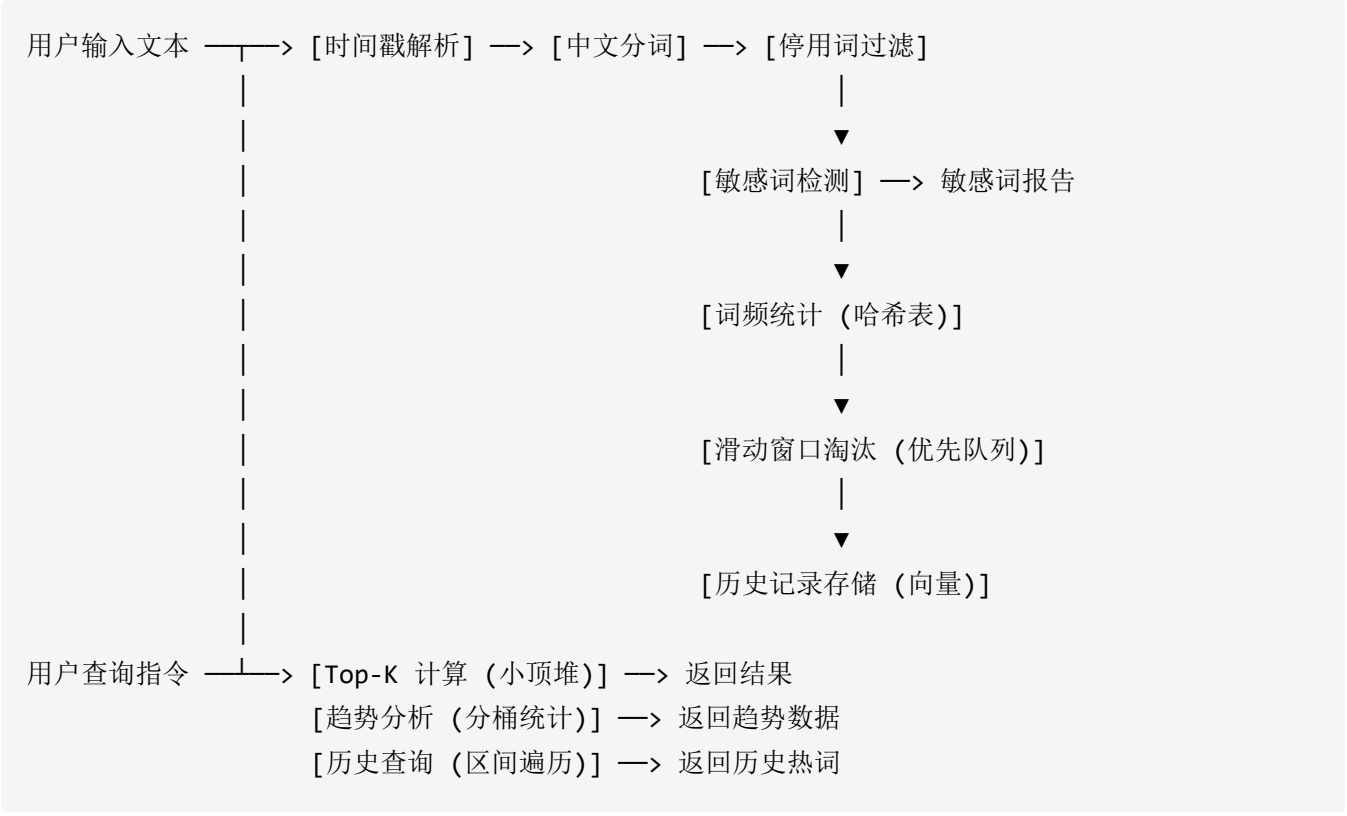
```
Project/
├── StartSystem.bat          # 一键启动脚本
├── README.md                # 项目说明文档
├── cppjieba/                # 中文分词库（第三方）
├── dict/                    # 词典文件
│   ├── jieba.dict.utf8      # 主词典
│   ├── hmm_model.utf8       # HMM 模型
│   ├── user.dict.utf8       # 用户自定义词典（保留词）
│   ├── stop_words.utf8      # 停用词表
│   ├── sensitive_words.utf8 # 敏感词表
│   └── idf.utf8             # IDF 词频文件
├── data/                    # 测试数据
│   ├── input1.txt           # 测试数据集 1（6,378 条）
│   ├── input2.txt           # 测试数据集 2（5,783 条）
│   ├── input3.txt           # 测试数据集 3（7,316 条）
│   ├── test_simple.txt      # 简单功能测试
│   └── test_wave.txt        # 特殊符号过滤测试
├── docs/                    # 文档目录
│   ├── 系统设计文档.md
│   ├── 性能测试报告.md
│   └── images/              # 文档图片
└── HotWordSystem/           # 系统主目录
    ├── Makefile             # 总控编译脚本
    ├── backend/             # C++ 后端
    │   ├── Makefile         # 后端编译脚本
    │   ├── src/
    │   │   ├── main.cpp             # 主程序入口
    │   │   ├── HotWordAnalyzer.hpp  # 核心统计引擎
    │   │   ├── TextProcessor.hpp    # 分词与过滤
    │   │   └── Utils.hpp            # 工具函数
    │   ├── bin/                     # 编译输出
    │   └── tests/                   # 单元测试
    │       ├── test_analyzer.cpp
    │       ├── test_processor.cpp
    │       └── test_utils.cpp
    └── frontend/            # Python 前端
        ├── main.py          # GUI 主程序
        ├── engine.py        # 后端通信模块
        ├── monitor.py       # 资源监控模块
```

```
├── charts.py          # 图表绘制模块
├── trend_manager.py   # 趋势分析管理
├── query_manager.py   # 查询管理
├── word_manager.py    # 词典管理（敏感词/保留词）
├── file_handler.py    # 文件操作
└── widgets.py         # 通用 UI 组件
```

2.3 模块职责说明

模块	文件	职责
主控制器	`main.cpp`	解析输入、调度模块、格式化输出
文本处理器	`TextProcessor.hpp`	中文分词、停用词过滤、敏感词检测、保留词处理
热词分析器	`HotWordAnalyzer.hpp`	滑动窗口管理、词频统计、Top-K 计算、趋势分析
工具类	`Utils.hpp`	时间戳解析与格式化
GUI 主程序	`main.py`	界面布局、事件分发、用户交互
引擎客户端	`engine.py`	管理后端进程、收发消息
资源监控	`monitor.py`	监控 CPU/内存占用
图表绘制	`charts.py`	绘制趋势曲线图
词典管理	`word_manager.py`	敏感词和保留词的增删改查
趋势管理	`trend_manager.py`	趋势数据收集与图表展示

2.4 数据流图



---

3. 核心数据结构设计

3.1 实时计数器：哈希表 (std::unordered_map)

选型理由:

- 词频更新是最高频的操作，需要 $O(1)$ 的增删改查。
- 哈希表的平均时间复杂度为 $O(1)$，在词汇量较大时依然高效。

代码实现:

```
std::unordered_map<std::string, int> wordCounts;

// 增加词频
wordCounts[word]++;

// 减少词频（窗口淘汰时）
wordCounts[word]--;
if (wordCounts[word] <= 0) {
    wordCounts.erase(word);
}
```

复杂度分析:

操作	平均复杂度	最坏复杂度
插入/更新	$O(1)$	$O(n)$ (哈希冲突)
查询	$O(1)$	$O(n)$
删除	$O(1)$	$O(n)$

设计取舍:

- 选择 `unordered_map` 而非 `map`，牺牲有序性换取更快的访问速度。
- 未使用 Trie 树，因为中文词汇的平均长度较短，哈希表更简洁高效。

3.2 时间窗口管理器：优先队列 (std::priority_queue)

选型理由:

- 滑动窗口需要高效地找出"最早过期"的记录。
- 小顶堆可以在 $O(\log n)$ 时间内完成插入和删除最小元素。

数据结构定义:

```
struct ExpirationEvent {
    long long timestamp;  // 记录的时间戳
    std::string word;     // 对应的词语

    // 小顶堆：时间早的排在堆顶
    bool operator>(const ExpirationEvent& other) const {
        return timestamp > other.timestamp;
    }
};

std::priority_queue<ExpirationEvent,
                    std::vector<ExpirationEvent>,
                    std::greater<ExpirationEvent>> eventQueue;
```

窗口淘汰算法:

```
void expireData() {
    // 持续检查堆顶，直到堆顶记录在窗口内
    while (!eventQueue.empty() &&
           eventQueue.top().timestamp <= currentTime - windowSize) {
        std::string expiredWord = eventQueue.top().word;
        // 从哈希表中扣减词频
        if (wordCounts.count(expiredWord)) {
            wordCounts[expiredWord]--;
            if (wordCounts[expiredWord] <= 0) {
                wordCounts.erase(expiredWord);
            }
        }
        eventQueue.pop();
    }
}
```

复杂度分析:

操作	复杂度
插入新记录	$O(\log n)$
淘汰过期记录	均摊 $O(\log n)$

操作	复杂度
查询堆顶	$O(1)$

滑动窗口定义：

- **窗口类型**：基于时间的滑动窗口（Time-based Sliding Window）
- **窗口大小**：默认 600 秒（10 分钟），支持动态修改
- **淘汰策略**：当系统时间推进时，自动淘汰时间戳早于 `currentTime - windowSize` 的所有记录
- **时间同步方式**：以输入数据中的时间戳为准，系统时间单调递增

3.3 Top-K 维护结构：小顶堆

选型理由：

- 从 N 个词中选出频率最高的 K 个，使用小顶堆可以达到 $O(N \log K)$。
- 相比排序的 $O(N \log N)$，当 $K \ll N$ 时效率更高。

算法实现：

```cpp
std::vector<std::pair<std::string, int>> getTopK(int k) {
    typedef std::pair<int, std::string> ScorePair;
    // 小顶堆：频率最低的在堆顶
    std::priority_queue<ScorePair,
                        std::vector<ScorePair>,
                        std::greater<ScorePair>> minHeap;

    // 遍历所有词，维护大小为 K 的堆
    for (const auto& it : wordCounts) {
        minHeap.push({it.second, it.first});
        if (minHeap.size() > k) {
            minHeap.pop();  // 弹出最小的，保留最大的 K 个
        }
    }

    // 提取结果并逆序（从大到小）
    std::vector<std::pair<std::string, int>> result;
    while (!minHeap.empty()) {
        result.push_back({minHeap.top().second, minHeap.top().first});
        minHeap.pop();
    }
    std::reverse(result.begin(), result.end());
    return result;
}
```

复杂度分析：

- 时间复杂度：$O(N \log K)$，其中 N 为当前窗口内的不同词数
- 空间复杂度：$O(K)$，堆的大小固定为 K

正确性保证：

- 每次查询时从哈希表重新构建 Top-K，确保结果与当前窗口状态一致。
- 未采用增量维护策略（如 Lossy Counting），避免引入近似误差。

3.4 历史记录存储：向量 (std::vector)

用途：存储所有历史数据，支持趋势分析和历史查询。

```
struct HistoryRecord {
    long long timestamp;
    std::string word;
};
std::vector<HistoryRecord> fullHistory;
```

设计取舍：

- 选择 `vector` 而非数据库，因为本系统为单次运行的分析工具，无需持久化。
- 内存占用与数据量成正比，适用于中等规模数据。

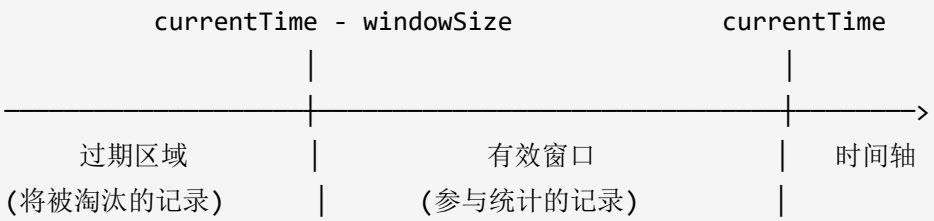
---

4. 滑动窗口详细设计

4.1 窗口定义

属性	值
窗口类型	基于时间 (Time-based)
默认大小	600 秒 (10 分钟)
最小粒度	1 秒
支持动态调整	是

4.2 窗口淘汰策略



淘汰触发时机：

1. 每次接收新数据时，先更新 `currentTime`，再执行淘汰。
2. 用户动态缩小窗口大小时，立即触发淘汰。

4.3 实时性保证

- **增量更新**：新数据到来时，仅更新涉及的词条，不重新扫描全表。
- **惰性淘汰**：过期数据在下次操作时统一处理，避免定时器开销。
- **查询响应**：Top-K 查询的时间复杂度为 $O(N \log K)$，N 为窗口内词数，通常在毫秒级完成。

---

5. 文本预处理设计

5.1 分词策略

- **分词引擎**：cppjieba（精确模式）
- **自定义词典**：支持用户添加保留词，确保专业术语不被切分

5.2 过滤规则

过滤类型	处理方式	配置文件
停用词	直接丢弃	`stop_words.utf8`
敏感词	丢弃并记录出现次数	`sensitive_words.utf8`

过滤类型	处理方式	配置文件
ASCII 单字符	丢弃（如英文标点、数字）	-
全角符号	丢弃（如 `～。，！？、；：“”【】《》—…（）` 等）	-

5.3 敏感词处理流程

```
bool isSensitive(const std::string& word) {
    if (sensitiveWords.count(word)) {
        sensitiveCounts[word]++;  // 记录出现次数
        return true;
    }
    return false;
}
```

---

6. 进程间通信设计

6.1 通信协议

前端与后端通过**标准输入/输出 (stdin/stdout)** 进行通信，使用自定义文本协议。

前端 → 后端（输入）：

```
[HH:MM:SS] 文本内容           # 数据输入
[ACTION] QUERY K=10           # 查询指令
[ACTION] SET_WINDOW SIZE=300 # 配置指令
```

后端 → 前端（输出）：

```
[TOPK] 1. 人工智能 (15次)     # 热词结果
[SENSITIVE] 敏感词X (3次)     # 敏感词报告
[TREND_DATA] WORD=AI DATA=...# 趋势数据
```

6.2 异步处理

- 前端使用 Python 的 `threading` 模块异步读取后端输出。
- 后端使用同步 I/O，逐行处理输入。

---

7. 性能优化策略

7.1 时间优化

优化点	方法	效果
词频统计	哈希表 $O(1)$ 更新	避免线性查找
Top-K 查询	小顶堆 $O(N \log K)$	优于排序 $O(N \log N)$
窗口淘汰	优先队列 $O(\log n)$	避免遍历全部记录

7.2 空间优化

优化点	方法
词频表	仅存储窗口内的词，过期词自动删除
历史记录	使用结构体减少内存碎片

7.3 实测性能参考

根据性能测试报告的实测结果：

指标	实测值
平均吞吐量	约 8,200 条/秒
平均处理延迟	约 0.12 ms/条
内存占用 (C++ 引擎)	约 125 MB

指标	实测值
内存占用 (Python 界面)	约 150 MB
系统总内存	约 275 MB

详细测试数据请参考《性能测试报告》。

---

8. 高级功能实现详解

8.1 敏感词过滤功能

功能描述：对输入文本中的敏感词进行实时检测和屏蔽，同时统计敏感词出现次数。

实现原理：

1. 在系统启动时，从 `sensitive_words.utf8` 文件加载敏感词到哈希集合。
2. 分词后，逐个检查词语是否在敏感词集合中。
3. 若匹配，则从结果中剔除，并累加该敏感词的出现计数。

核心代码：

```
// TextProcessor.hpp
std::unordered_set<std::string> sensitiveWords;      // 敏感词集合
std::unordered_map<std::string, int> sensitiveCounts; // 敏感词计数

bool isSensitive(const std::string& word) {
    if (sensitiveWords.count(word)) {
        sensitiveCounts[word]++;
        return true;
    }
    return false;
}
```

复杂度分析：

- 敏感词查询：$O(1)$（哈希集合）

- 空间复杂度：$O(S)$，S 为敏感词数量

GUI 支持：

- 用户可通过"词典管理 → 敏感词管理"界面动态添加/删除敏感词。
- 修改后需重新启动后端引擎生效。

8.2 保留词/自定义词典功能

功能描述：允许用户添加专业术语或特定词汇，确保这些词不会被分词器错误切分。

实现原理：

1. cppjieba 支持用户词典（`user.dict.utf8`）。
2. 用户词典中的词会被优先识别，不会被进一步切分。

词典格式：

cppjieba 支持两种格式：

1. 简单格式（仅词语）：

```
cppjieba
C++
Python
```

2. 完整格式（词语 + 词频 + 词性）：

```
人工智能 1000 n
机器学习 500 n
深度学习 500 n
```

默认使用简单格式，系统会自动分配默认词频。

GUI 支持：

- 用户可通过"词典管理 → 保留词管理"界面动态添加/删除保留词。

- 系统会自动为新词分配默认词频和词性。

8.3 趋势分析功能

功能描述：分析词语频率随时间的变化趋势，绘制可视化曲线图。

实现原理：

1. **历史记录存储**：所有词语及其时间戳存入 `fullHistory` 向量。
2. **分桶统计**：将时间轴按用户指定的间隔（Interval）划分为若干桶。
3. **频率计算**：遍历历史记录，统计每个桶内目标词的出现次数。
4. **可视化**：使用 matplotlib 绘制折线图。

核心代码：

```
// HotWordAnalyzer.hpp
std::pair<std::vector<long long>, std::vector<int>> getTrendData(
    const std::string& word, int intervalSeconds) {

    int numBuckets = (currentTime - startTime) / intervalSeconds + 1;
    std::vector<long long> times(numBuckets);
    std::vector<int> counts(numBuckets, 0);

    for (int i = 0; i < numBuckets; ++i) {
        times[i] = startTime + (i + 1) * intervalSeconds;  // 时间段结束点
    }

    for (const auto& record : fullHistory) {
        if (record.word == word) {
            int bucket = (record.timestamp - startTime) / intervalSeconds;
            if (bucket >= 0 && bucket < numBuckets) {
                counts[bucket]++;
            }
        }
    }
    return {times, counts};
}
```

复杂度分析：

- 时间复杂度：$O(H)$，H 为历史记录总数
- 空间复杂度：$O(B)$，B 为桶的数量

8.4 历史查询功能

功能描述：查询任意时间段内的热词排名。

实现原理：

1. 接收用户指定的起止时间和 K 值。
2. 遍历历史记录，筛选出时间范围内的词语。
3. 使用小顶堆计算该范围内的 Top-K。

指令格式：

```
[ACTION] HISTORY START=0 END=600 K=10
```

核心代码：

```
std::vector<std::pair<std::string, int>> getHistoryTopK(
    long long startTime, long long endTime, int k) {

    std::unordered_map<std::string, int> historyCounts;

    for (const auto& record : fullHistory) {
        if (record.timestamp >= startTime && record.timestamp <= endTime) {
            historyCounts[record.word]++;
        }
    }

    // 使用小顶堆计算 Top-K（代码同 getTopK）
    ...
}
```

8.5 动态窗口大小功能

功能描述：运行时动态调整滑动窗口的时间范围。

实现原理：

1. 接收新的窗口大小参数。
2. 更新 `windowSize` 变量。
3. 立即触发过期数据淘汰（窗口缩小时）。

指令格式：

```
[ACTION] SET_WINDOW SIZE=300
```

核心代码：

```
void setWindowSize(int newSizeSeconds) {
    windowSize = newSizeSeconds;
    expireData();  // 立即淘汰超出新窗口的数据
}
```

8.6 交互式可视化功能

功能描述：提供图形用户界面，实时展示热词列表和趋势曲线。

技术栈：

组件	技术	用途
GUI 框架	Python Tkinter	窗口、按钮、文本框
图表绘制	Matplotlib	趋势曲线图
资源监控	psutil	内存实时显示

界面功能：

- **热词列表**：实时显示当前窗口内的 Top-K 热词。
- **趋势图表**：支持单词趋势和多词对比趋势。
- **词典管理**：动态编辑敏感词和保留词。
- **资源监控**：显示后端进程和前端的内存占用。

8.7 资源监控功能

功能描述：实时监控系统的内存占用情况。

实现原理：

1. 前端通过 `psutil` 库获取后端进程的 PID。
2. 定时（每秒）读取前端和后端进程的内存使用量。
3. 在界面上以数值和曲线图形式展示。

核心代码：

```
# monitor.py
import psutil

class ResourceMonitor:
    def get_usage_text(self, process_pid=None):
        py_proc = psutil.Process(os.getpid())
        py_mem = py_proc.memory_info().rss / 1024 / 1024  # MB

        cpp_mem = 0
        if process_pid:
            cpp_proc = psutil.Process(process_pid)
            cpp_mem = cpp_proc.memory_info().rss / 1024 / 1024

        total_mem = py_mem + cpp_mem
        return f"界面内存: {py_mem:.1f} MB\n引擎内存: {cpp_mem:.1f} MB\n总计: {total_mem:.1f} MB"
```

---

9. 错误处理与健壮性

场景	处理方式
词典文件缺失	打印警告，使用空词表继续运行
时间戳格式错误	跳过该行，打印错误日志
非法指令	忽略并返回提示信息
后端进程崩溃	前端检测并提示用户重启

---

10. 可扩展性设计

10.1 当前支持的扩展

- **动态词典**：修改词典文件后重启后端即可生效。
- **动态窗口**：运行时可通过 `[ACTION] SET_WINDOW SIZE=n` 调整窗口大小。

10.2 未来可扩展方向

- **多线程分词**：将分词与统计分离到不同线程。
- **持久化存储**：将历史数据写入数据库，支持跨会话查询。
- **Web 界面**：将 GUI 替换为 Web 前端，支持远程访问。

---

11. 总结

本系统通过合理的数据结构选型和算法设计，实现了高效的实时热词统计功能：

1. **哈希表**保证了词频更新的 $O(1)$ 效率。
2. **优先队列**实现了高效的滑动窗口淘汰机制。
3. **小顶堆**优化了 Top-K 查询的时间复杂度。
4. **前后端分离**架构兼顾了性能与用户体验。

系统完整实现了题目要求的核心功能，并扩展了敏感词过滤、趋势分析、历史查询等高级特性，是一个功能完整、结构清晰的数据结构综合应用案例。