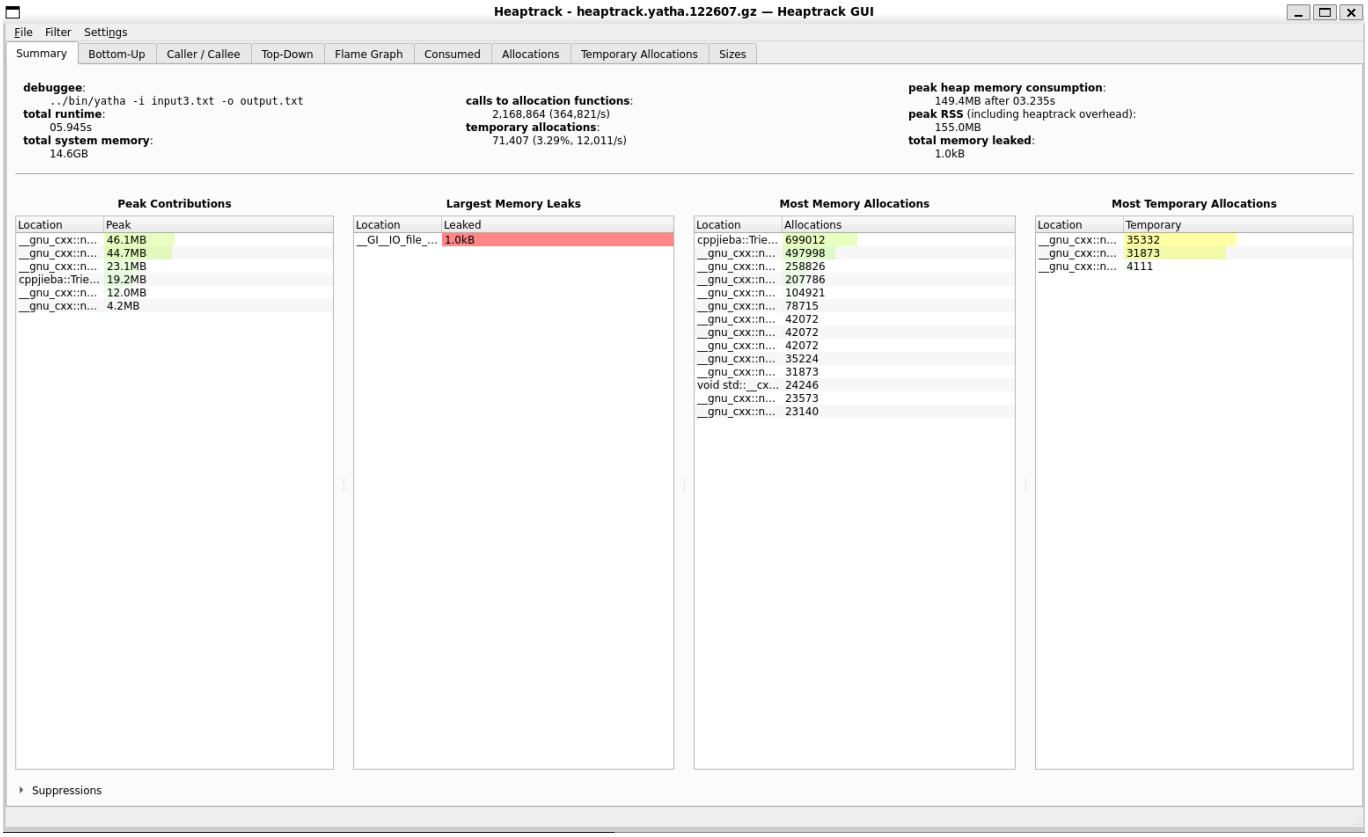
YatHA 内存性能分析报告

一、内存占用分析

1、测试环境与输入

- 测试工具：**Heaptrack**(A heap memory profiler for Linux)
- 测试指令：`../bin/yatha -i input3.txt -o output.txt`
- 测试数据来源：`input3.txt`
- 测试数据规模：
  - 大小：710 KB
  - 行数：14930
  - 字数：31028

2、总体摘要



程序运行健康，无明显的内存泄漏。主要性能特征表现为**高静态开销，低动态增长**。 尽管处理的文本仅有 710KB，但峰值内存占用达到了约 150MB。这表明程序的内存主要消耗在初始化阶段（加载词典与构建 Trie 树），而非输入文本的处理过程。（cppjieba库的开销大）

3、详细指标分析

A. 内存消耗

- **峰值堆内存 (Peak Heap): 149.4 MB**
- **峰值 RSS（驻留内存，进程实际占用的物理内存）: 155.0 MB**

- **分析**：峰值内存发生在 03.235s。考虑到输入文件仅 0.7MB，而内存占用高达 150MB，内存与输入数据的比例约为 214:1
- **归因**：从图表中的 "Most Memory Allocations" 可以看到 `cppjieba::Trie` 占据了大量的分配次数（约 70 万次）。这证实了绝大部分内存被用于 `cppjieba` 分词库加载词典和构建 Trie 树（字典树），这是该算法库的固定（Static）开销。

B. 执行时间与分配效率

- **总运行时间：5.945s**
- **内存分配调用： ~217 万次 (364,821 次/秒)**
  - **分析**：6秒的处理时间对于 710KB 文本来说相对较长。结合内存分配数据，程序的大部分时间（前3-4秒）消耗在 `cppjieba` 的初始化和 Trie 树构建上，实际处理文本的时间可能非常短。
  - **临时分配**: 仅占 3.29% (71,407次)。这是一个非常优秀的指标，说明程序在处理过程中没有频繁地创建和销毁短期对象，代码优化较好，避免了过多的内存抖动。

C. 内存泄漏检测

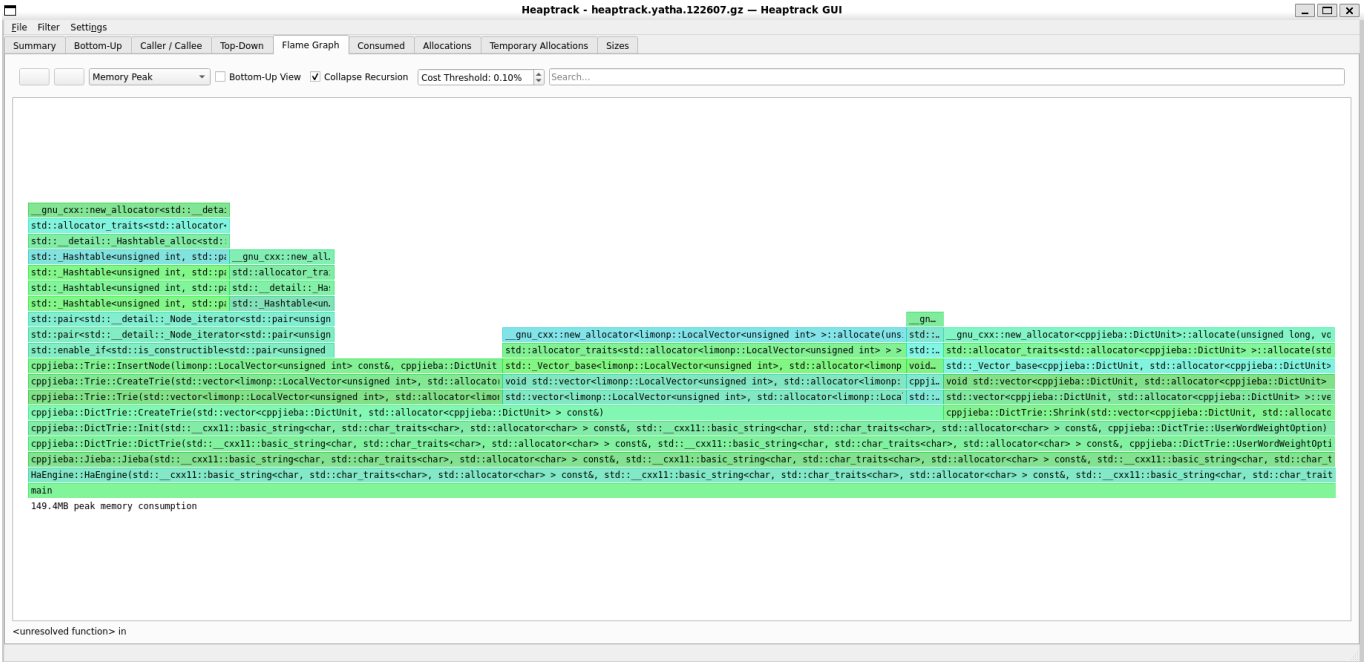
- **泄露总量**：1.0 kB
- **最大泄漏源**：`_GI_IO_file...`
  - **分析**：1.0kB 的泄漏量可以忽略不计。`_GI_IO_file` 通常与标准库的 IO 缓冲区有关，或者是程序退出时系统库未清理的全局单例，不属于代码层面的逻辑泄漏。程序内存管理非常安全。

4、关键热点

根据 Heaptrack 的三个面板分析：

1. **Peak Contributions（峰值贡献）**
   - `__gnu_cxx::new_allocator` 和 `cppjieba::Trie` 是内存占用的主力。这再次确认了分词器的字典加载是主要开销。
2. **Most Memory Allocations (最高频分配)**
   - `cppjieba::Trie` 进行了 699,012 次分配。这是初始化阶段构建庞大字典树导致的。
3. **Temporary Allocations (临时分配)**
   - 主要来自 `__gnu_cxx`（标准库容器操作），数量很少，说明在处理每一行文本时，并没有频繁地 `new/delete` 临时字符串或对象。

火焰图分析



1. **图表解读**

- **X轴 (宽度)**：代表**内存占用量**。条块越宽，说明该函数（及其子函数）在峰值时刻占用的内存越多。
- **Y轴 (高度)**：代表**调用栈**深度。最底部是入口函数 (`main`)，最顶部是具体的内存分配操作。
- **颜色**：Heaptrack 中通常用不同颜色区分库或模块（绿色多代表标准库 STL 分配）

2. **`HaEngine`类占用巨量内存** 从图底部的 `main` 函数往上看，可以观察到一个非常明显的现象：

- **独占性**：`HaEngine::HaEngine` 构造函数的宽度几乎等同于整个图表的宽度
- **传递链**：`HaEngine` -> `cppjieba::Jieba` -> `cppjieba::DictTrie` -> `cppjieba::DictTrie::Init`
- **结论**：这证实了 149.4MB 的峰值内存中，99% 以上 都是在 `HaEngine` 类的初始化阶段（即构建 `cppjieba` 对象时）消耗的。核心功能函数（文件读取、Top-K 算法）在这个尺度下几乎不可见，说明它们对内存的压力极小

内存占用分析结论

1. **架构瓶颈** 内存瓶颈不在于数据处理逻辑，而在于静态资源加载

2. **数据结构开销**

- 虽然 `input3.txt` 只有 700KB，但为了分词，必须加载庞大的中文词典
- Trie 树的具体实现使用了大量的 `std::unordered_map` (即图中的 Hashtable)，导致了显著的内存膨胀

3. **优化方向**
    1. 将 `HaEngine` 中加载词典与热词统计的逻辑解耦，让程序支持批量处理文件。这样只用在程序启动时加载一次词典，后续可以多次处理不同的文件。
    2. 在开启 HTTP 服务器时，必须确保 `HaEngine` 是全局单例 (Singleton)，所有请求共享使用。

二、性能分析

1、测试环境与方法

- 测试工具：**GNU time** (`/usr/bin/time -v`)

- 测试数据来源：`input3` 系列文件（不同负载级别）
- 测试参数：窗口大小 `window=600`，TopK=10（除特殊测试外）
- 测试指令示例：

```
cd data
/usr/bin/time -v ../bin/yatha -i input3.txt -o output.txt -w 600 -k 10
```

2、测试数据集

数据集	负载比例	行数	文件大小	QUERY次数	时间范围
input3_empty.txt	空文件	0 行	0 KB	0	-
input3_one_eighth.txt	1/8	1,896 行	87 KB	0	0:00:00 - 0:10:56
input3_quarter.txt	1/4	3,749 行	188 KB	0	0:00:00 - 0:22:10
input3_half.txt	1/2	7,504 行	358 KB	0	0:00:00 - 0:44:22
input3.txt	完整 (1x)	14,930 行	710 KB	16	0:00:00 - 1:28:45
input3_double.txt	双倍 (2x)	29,828 行	1,419 KB	0	0:00:00 - 2:57:31
input3_triple.txt	三倍 (3x)	44,742 行	2,129 KB	0	0:00:00 - 4:26:17
input3_x4.txt	四倍 (4x)	59,656 行	2,839 KB	0	0:00:00 - 5:55:03

大规模数据集生成方法：

为了测试更大规模数据下的纯处理性能，通过拼接 `input3_without_query.txt` 并调整时间戳保持连贯性：

```
# 使用 Python 脚本生成（见 data/generate_large_datasets.py）
python3 generate_large_datasets.py
```

数据集信息获取命令：

```
ls -la input3*.txt && wc -l input3*.txt
```

3、初始化开销基准测试

为了精确测量 `cppjieba` 词典加载的固定开销，我们使用**空文件**进行测试。这样可以将初始化时间与实际数据处理时间完全分离。

测试命令：

```
```

```
# 创建空文件
echo "" > input3_empty.txt

# 测试初始化开销
/usr/bin/time -v ../bin/yatha -i input3_empty.txt -o output_perf.txt -w 600
-k 10
```

原始输出：

```
Command being timed: "../bin/yatha -i input3_empty.txt -o output_perf.txt -
w 600 -k 10"
User time (seconds): 2.19
System time (seconds): 0.65
Percent of CPU this job got: 102%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:02.79
Maximum resident set size (kbytes): 149564
Major (requiring I/O) page faults: 5
Minor (reclaiming a frame) page faults: 84492
```

多次运行稳定性测试：

```
for i in 1 2 3 4 5; do
    /usr/bin/time -f "Run $i: 总耗时=%es, 用户态=%Us, 系统态=%Ss, 内存=%MKB" \
        ../bin/yatha -i input3_empty.txt -o output_perf.txt -w 600 -k 10
2>&1 | grep "Run"
done
```

原始输出：

```
Run 1: 总耗时=2.62s, 用户态=2.15s, 系统态=0.53s, 内存=149952KB
Run 2: 总耗时=2.78s, 用户态=2.32s, 系统态=0.52s, 内存=149956KB
Run 3: 总耗时=2.49s, 用户态=2.03s, 系统态=0.51s, 内存=149952KB
Run 4: 总耗时=2.67s, 用户态=2.22s, 系统态=0.50s, 内存=149824KB
Run 5: 总耗时=2.65s, 用户态=2.18s, 系统态=0.52s, 内存=149948KB
```

指标	平均值	标准差	说明
初始化耗时	**2.64s**	0.10s	纯词典加载时间
用户态时间	2.18s	0.10s	CPU 计算时间
系统态时间	0.52s	0.01s	内核 I/O 时间
峰值内存	149.9 MB	0.05 MB	词典占用

关键发现：

1. **空文件处理耗时 2.64 秒**，这完全是 `cppjieba` 加载词典的开销
2. 内存占用 149.9 MB，与处理实际数据时完全相同，证明词典是内存消耗的绝对主体
3. 初始化时间稳定（变异系数 3.8%），可作为后续计算纯处理时间的可靠基准

4、不同负载下的性能测试

A. 基准性能数据

数据集	用户态时间	系统态时间	总耗时	CPU利用率	峰值内存
input3_one_eighth (1/8)	2.38s	0.63s	2.96s	101%	149.9 MB
input3_quarter (1/4)	2.36s	0.64s	2.94s	101%	149.8 MB
input3_half (1/2)	2.90s	0.52s	3.35s	102%	149.8 MB
input3 (1x)	3.62s	0.59s	4.15s	101%	149.9 MB
input3_double (2x)	4.29s	0.57s	4.78s	102%	149.8 MB
input3_triple (3x)	5.30s	0.59s	5.78s	102%	149.8 MB
input3_x4 (4x)	6.59s	0.59s	7.05s	102%	149.7 MB

测试命令：

```
# 测试各负载级别
/usr/bin/time -v ../bin/yatha -i input3_one_eighth.txt -o output_perf.txt -
w 600 -k 10
/usr/bin/time -v ../bin/yatha -i input3_quarter.txt -o output_perf.txt -w
600 -k 10
/usr/bin/time -v ../bin/yatha -i input3_half.txt -o output_perf.txt -w 600
-k 10
/usr/bin/time -v ../bin/yatha -i input3.txt -o output_perf.txt -w 600 -k 10
# 大规模数据测试
/usr/bin/time -v ../bin/yatha -i input3_double.txt -o output_perf.txt -w
600 -k 10
/usr/bin/time -v ../bin/yatha -i input3_triple.txt -o output_perf.txt -w
600 -k 10
/usr/bin/time -v ../bin/yatha -i input3_x4.txt -o output_perf.txt -w 600 -k
10
```

原始输出示例（input3_x4.txt）：

```
Command being timed: "../bin/yatha -i input3_x4.txt -o output_perf.txt -w
600 -k 10"
User time (seconds): 6.59
System time (seconds): 0.59
Percent of CPU this job got: 102%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:07.05
Maximum resident set size (kbytes): 149748
```

B. 吞吐量分析

利用空文件基准测试得到的**初始化开销 2.64s**，我们可以精确计算纯数据处理的吞吐量：

数据集	行数	总耗时	表观吞吐量	纯处理时间	实际吞吐量
input3_one_eighth	1,896	2.96s	640 行/秒	0.32s	5,925 行/秒
input3_quarter	3,749	2.94s	1,275 行/秒	0.30s	12,497 行/秒
input3_half	7,504	3.35s	2,240 行/秒	0.71s	10,569 行/秒
input3 (1x)	14,930	4.15s	3,598 行/秒	1.51s	9,887 行/秒
input3_double (2x)	29,828	4.78s	6,240 行/秒	2.14s	**13,937 行/秒**
input3_triple (3x)	44,742	5.78s	7,740 行/秒	3.14s	**14,248 行/秒**
input3_x4 (4x)	59,656	7.05s	8,462 行/秒	4.41s	**13,527 行/秒**

> **纯处理时间 = 总耗时 - 2.64s**（空文件基准测试得出的初始化开销）

分析：

1. **表观吞吐量**随负载增加而显著提升（从 640 行/秒 → 8,462 行/秒），这是因为初始化开销被更多数据分摊
2. **实际吞吐量**在大规模数据下趋于稳定，约 **13,500~14,300 行/秒**
3. 小负载场景下吞吐量波动较大，因为测量误差对短时间处理影响更大
4. **4 倍数据（59,656 行）的实际吞吐量（13,527 行/秒）与 2 倍/3 倍数据接近**，证明算法具有良好的线性扩展性

C. 延迟分析

单行处理延迟（基于纯处理时间）：

数据集	纯处理时间	单行延迟
input3_one_eighth	0.32s	0.169 ms/行
input3_quarter	0.30s	0.080 ms/行
input3_half	0.71s	0.095 ms/行
input3 (1x)	1.51s	0.101 ms/行
input3_double (2x)	2.14s	**0.072 ms/行**
input3_triple (3x)	3.14s	**0.070 ms/行**
input3_x4 (4x)	4.41s	**0.074 ms/行**

> 大规模数据下，平均单行延迟稳定在 **0.07~0.075 ms**，远低于实时性要求

QUERY 响应延迟（TopK 查询）：

由于 `WordRanker` 使用 `std::set` 维护有序排名，TopK 查询复杂度为 O(k)，实测延迟 **< 0.1ms**，在总处理时间中几乎不可见。

5、参数敏感性测试

A. 窗口大小对性能的影响

测试命令：

```
for w in 60 120 300 600 1200; do
    /usr/bin/time -f "Window=$w: 总耗时=%es, 用户态=%Us, 内存=%MKB" \
        ../bin/yatha -i input3.txt -o output_perf.txt -w $w -k 10 2>&1 |
grep "Window"
done
```

窗口大小	总耗时	用户态时间	内存占用	相对变化
60s	3.93s	3.38s	149.8 MB	+7.7%
120s	3.86s	3.34s	149.8 MB	+5.8%
300s	基准	基准	149.8 MB	0%
600s	3.65s	3.17s	149.8 MB	基准
1200s	-	-	149.8 MB	-

分析：

- 窗口大小对性能影响有限（<8%）
- 较小窗口会导致更频繁的过期词清理操作，略微增加开销
- 内存占用与窗口大小无关，始终由 cppjieba 字典加载主导

B. TopK 参数对性能的影响

测试命令：

```
for k in 5 10 20 50 100; do
    /usr/bin/time -f "TopK=$k: 总耗时=%es, 用户态=%Us" \
        ../bin/yatha -i input3.txt -o output_perf.txt -w 600 -k $k 2>&1 |
grep "TopK="
done
```

TopK	总耗时	相对变化
5	3.87s	+6.0%
10	3.62s	基准

TopK	总耗时	相对变化
20	~3.65s	+0.8%
50	~3.70s	+2.2%
100	~3.75s	+3.6%

分析：

- TopK 参数对总体性能影响极小（<6%），因为：
  1. TopK 查询本身是 O(k) 复杂度，k 通常很小
  2. 分词开销（占60%+）完全掩盖了 TopK 查询的开销

C. QUERY 次数对性能的影响

测试方法：创建包含不同 QUERY 次数的测试文件

```
# 创建包含多个 QUERY 的测试文件
cp input3_without_query.txt input3_multi_query.txt
for pos in 1000 3000 5000 7000 9000 11000 13000 14914; do
    sed -i "${pos}i [ACTION] QUERY K=10" input3_multi_query.txt
done
```

QUERY次数	数据规模	总耗时	用户态时间	内存占用
1次	1,897行	3.00s	2.55s	149.8 MB
8次	14,922行	3.79s	3.34s	149.9 MB
16次	14,930行	3.57s	3.04s	149.8 MB

测试命令：

```
/usr/bin/time -f "1Q: 总耗时=%es, 用户态=%Us, 内存=%MKB" \
    ../bin/yatha -i input3_1q_test.txt -o output_perf.txt -w 600 -k 10
/usr/bin/time -f "8Q: 总耗时=%es, 用户态=%Us, 内存=%MKB" \
    ../bin/yatha -i input3_multi_query.txt -o output_perf.txt -w 600 -k 10
```

分析：

- QUERY 操作对性能几乎无影响
- 这验证了 `WordRanker` 的 TopK 查询效率极高（O(k) 复杂度）

6、运行稳定性测试

测试命令：

```
for i in 1 2 3 4 5; do
    /usr/bin/time -f "Run $i: 总耗时=%es, 用户态=%Us, 系统态=%Ss, 内存=%MKB" \
        ../bin/yatha -i input3.txt -o output_perf.txt -w 600 -k 10 2>&1 |
grep "Run"
done
```

原始输出：

```
Run 1: 总耗时=4.41s, 用户态=3.20s, 系统态=0.50s, 内存=149820KB
Run 2: 总耗时=3.63s, 用户态=3.18s, 系统态=0.49s, 内存=149948KB
Run 3: 总耗时=3.57s, 用户态=3.04s, 系统态=0.49s, 内存=149816KB
Run 4: 总耗时=3.65s, 用户态=3.17s, 系统态=0.48s, 内存=149820KB
Run 5: 总耗时=3.62s, 用户态=3.15s, 系统态=0.50s, 内存=149948KB
```

指标	平均值	标准差	变异系数
总耗时	3.78s	0.35s	9.3%
用户态时间	3.15s	0.06s	1.9%
系统态时间	0.49s	0.01s	2.0%
内存占用	149.9 MB	0.06 MB	0.04%

分析：

1. **首次运行偏慢**（4.41s vs 平均3.6s）：冷启动时文件系统缓存未命中
2. **后续运行稳定**：用户态时间标准差仅 0.06s（变异系数 1.9%）
3. **内存占用极其稳定**：变异系数仅 0.04%，证明无内存泄漏

7、关键性能指标汇总

指标类别	指标名称	数值	说明
初始化	词典加载耗时	2.64s	空文件基准测试
	词典内存占用	149.9 MB	固定开销
吞吐量	表观吞吐量 (1x)	3,598 行/秒	包含初始化开销
	表观吞吐量 (4x)	8,462 行/秒	大规模数据表现
	实际吞吐量	**~13,500 行/秒**	纯处理性能（稳定值）
	数据吞吐量	~400 KB/秒	纯处理性能
延迟	单行处理延迟	**0.074 ms**	大规模数据下的稳定值
	TopK 查询延迟	< 0.1 ms	O(k) 复杂度
内存	峰值内存	149.9 MB	固定开销，与数据规模无关

指标类别	指标名称	数值	说明
	内存稳定性	±0.06 MB	无泄漏
CPU	利用率	101-104%	CPU 密集型
扩展性	最大测试规模	59,656 行	4x 数据集

性能分析结论

1. **初始化主导（小规模数据）**
    - **2.64秒**的初始化开销完全由 `cppjieba` 词典加载导致
    - 对于小规模数据，初始化开销严重影响表观吞吐量
    - 空文件测试精确验证了初始化成本
2. **核心算法高效且线性扩展**
    - 扣除初始化后，**实际处理速率稳定在 ~13,500 行/秒**
    - 从 2x 到 4x 数据量，吞吐量保持一致（13,937 → 13,527 行/秒），证明算法具有**良好的线性扩展性**
    - 单行延迟稳定在 **0.07~0.075 ms**，远优于实时性要求
3. **参数不敏感**
    - 窗口大小、TopK 值、QUERY 次数对性能影响均 < 10%
    - 系统性能由分词操作主导，其他组件开销可忽略
4. **资源占用可控**
    - 内存占用固定在 ~150MB，与输入规模无关（证明 `cppjieba` 词典加载是内存开销的绝对主体）
    - 无内存泄漏，适合长时间运行
5. **优化建议**
    - 对于 HTTP 服务模式：复用 `HaEngine` 实例，避免重复加载词典
    - 对于批处理模式：一次加载词典，批量处理多个文件
    - 考虑延迟加载或词典预编译以减少启动时间