

海量企业新闻向量数据库系统：完整技术指南 v2.0

Table of Contents

- [执行摘要](#)
- [一、Milvus 数据组织的完整体系](#)
- [二、Flush 机制：Growing Segment 何时转换为 Sealed Segment](#)
- [在实际生产中，flush 本身很快](#)
- [耗时的是之后的索引构建（在后台异步进行）](#)
 - [三、Segment 与 Compaction 的关系](#)
 - [四、完整的数据生命周期（修正版）](#)
 - [五、核心参数参考表](#)
 - [六、企业级最佳实践](#)
 - [七、面试核心答题模板（完整版）](#)
 - [八、常见问题与解答](#)
 - [九、总结建议](#)
 - [附录：完整配置示例](#)
- [milvus.yaml 推荐配置（企业新闻场景）](#)
- [推荐的应用层代码逻辑](#)

执行摘要

本文档是针对**企业级大规模向量数据库系统**的完整技术指南，涵盖数据组织、索引策略、性能优化、增量管理等核心内容。相比 v1.0，此版本新增了 **Segment 生命周期**、**Flush 机制**、**Compaction 触发条件** 等底层工作原理，帮助你在面试中从架构设计到系统细节的完整阐述。

一、Milvus 数据组织的完整体系

1.1 从向量到查询：数据层级关系（自下而上）

1. 向量 (Vector)	
├ 一条新闻一个向量	
├ 维度：1024 (float32)	
├ 大小：4KB	
└ 元数据：id, date, company 等	

↓ 批量

2. Batch (批处理)

- └ 多个向量分批插入
- └ 典型大小：100K-1M 条
- └ 耗时：<1 分钟
- └ 目的：提高插入效率

↓ 自动进入

3. Growing Segment (增长段)

- └ 定义：正在接收新数据的段
- └ 状态：在线写入，无索引
- └ 大小：0-512MB (自动管理)
- └ 查询：支持 (线性扫描，较慢)
- └ 个数：Collection 中最多 1 个
- └ 特性：数据可变，支持继续写入

↓ Flush (达到条件自动或手动)

4. Sealed Segment (封闭段)

- └ 定义：已停止写入的段
- └ 状态：只读，正构建或已完整索引
- └ 大小：~512MB
- └ 查询：支持 (使用索引，快速)
- └ 个数：Collection 中有多个
- └ 特性：数据不可变，优化存储

↓ 多个 Sealed Segment 积累

5. Compaction (合并操作)

- └ 定义：将多个小 Segment 合并为 1 个
- └ 触发：自动 (达到条件) 或手动
- └ 耗时：5-15 分钟 (后台异步)
- └ 效果：减少查询聚合次数 30-50%
- └ 结果：1 个大 Segment 替代多个小

↓

6. Partition (分区)

- └ 定义：按时间/业务维度划分
- └ 包含：多个 Segment (已/未合并)
- └ 支持：独立索引、独立管理

↓

7. Collection (集合)

- └ 定义：所有数据的总容器
- └ 包含：多个 Partition

1.2 不同概念的明确区分

易混淆的核心概念澄清：

概念	定义	数量	状态	特性
Query	用户搜索请求，与数据存储无关	动态	一次性	不影响数据组织
Document	集合中的一条数据（新闻向量）	多个	永久	组成 Segment
Growing Seg	正在接收新数据的段	1个	可写	无索引，后台转换
Sealed Seg	已停止写入的段	多个	只读	有索引，可查询
Compaction	合并小 Segment 的操作	不定	一次性	不改变数据，优化结构
Flush	将 Growing → Sealed 的操作	不定	一次性	触发索引构建

二、Flush 机制：Growing Segment 何时转换为 Sealed Segment

2.1 Flush 的定义与作用

Flush 是什么？

- 将 Growing Segment 中的数据"落盘"，转换为只读的 Sealed Segment
- 转换后会立即或后台开始构建索引
- 新建一个空的 Growing Segment 继续接收新数据

Flush 的作用：

- 保证数据可检索（只有 Sealed 才能被索引）
- 防止"永久 Growing"（数据一直不落盘）
- 实现分段存储和管理

2.2 Flush 的触发条件

Flush 有三种触发方式：

方式 1：自动触发 - 数据量达到上限

参数名：datanode.segment.max.size

默认值：512 MB（也可能是 256MB，取决于版本）

触发条件：Growing Segment 的数据量 \geq 512MB

实例计算：

- └─ 单条向量大小：1024 dim \times 4 bytes = 4KB
- └─ 512MB \div 4KB = 128 万条向量
- └─ 日新增 1M 向量：512MB/400MB \approx 1.28 天触发一次
- └─ 结果：约每 1-2 天自动 flush 一次

流程：

1. 后台 DataNode 定时 (每秒) 检查 Growing Segment 大小
2. 若达到阈值立即触发 Flush
3. Growing Seg → Sealed Seg (转换瞬间, <1秒)
4. 新建空 Growing Segment, 继续接收数据
5. 后台 IndexNode 开始为 Sealed Seg 构建索引

方式 2：自动触发 - 时间达到上限

参数名：common.retentionDuration

默认值：86400 秒 (24 小时)

触发条件：Growing Segment 存在时间 \geq 24 小时

场景：数据流量很小，即使 24 小时也未达到 512MB

实例：

- └ 日新增只有 10 万条向量 (400MB)
- └ 永远不会达到 512MB 上限
- └ 但 24 小时后系统仍会自动 flush
- └ 防止数据"被永久卡在 Growing Segment"

流程：

1. Growing Segment 创建时记录时间戳
2. 后台定期检查 (每 10-60 秒一次)
3. 若存在时间超过 86400 秒，立即 flush
4. 无论实际数据量多少，都会转为 Sealed

方式 3：手动触发

API：collection.flush()

使用场景：

- └ 批量导入完毕，希望立刻可查询
- └ 定期对账/清账时刻
- └ 定时分析任务前的数据一致性保证
- └ 性能测试或压力测试

代码示例：

```
from pymilvus import Collection

collection = Collection("news_embeddings")

# 插入大批量数据
for batch in batches:
    collection.insert(batch)

# 手动 flush, 立刻转为 Sealed Segment
collection.flush()
print("✓ Flush 完成, 数据已可查询")
print("└ Growing Segment → Sealed Segment")
print("└ 新建 Growing Segment 继续接收数据")
print("└ 后台开始构建索引 (30-90分钟)")
```

```
# 用户现在可以查询（即使索引还在构建中）
results = collection.search(...)
```

阻塞 vs 非阻塞：

```
collection.flush() # 默认阻塞，等到 flush 完成才返回
                  # 通常 <1 秒（只是数据从内存转移）
```

```
# 在实际生产中，flush 本身很快<a></a>
# 耗时的是之后的索引构建（在后台异步进行）<a></a>
```

2.3 推荐的 Flush 策略

对于企业新闻场景（日新增 1M 条）：

策略 1：全自动（简单，适合小规模）

- └ 依赖系统自动条件触发
- └ 无需手动干预
- └ 问题：可能数据积压 24 小时才 flush

策略 2：周期性手动 flush（推荐，企业级）

- └ 使用 Cron 任务或调度器
- └ 每 3-6 小时手动调用一次 collection.flush()
- └ 好处：数据更快进入索引，检索延迟低
- └ 代码示例：

```
import schedule
import time

def periodic_flush():
    collection.flush()
    print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] Flush 完成")

# 每 6 小时自动执行一次
schedule.every(6).hours.do(periodic_flush)

while True:
    schedule.run_pending()
    time.sleep(60)
```

策略 3：混合策略（最优，复杂但高效）

- └ 定期周期性手动 flush（每 6 小时）
- └ 大批量导入后立刻手动 flush
- └ 允许系统自动 flush（备选机制）
- └ 好处：性能最优，可控性强
- └ 适用：对检索实时性要求高的场景

三、Segment 与 Compaction 的关系

3.1 Segment 的生命周期

时间轴演示（日新增 1M 条新闻，采用周期性 flush 策略）：

Day 1 (00:00 开始)

- ├─ 00:00-06:00: Batch 1 插入 (100-200M 新闻)
 - ├─ Growing Segment 1 接收数据
 - ├─ 数据量: 从 0 → 400MB
 - └─ 状态: 无索引, 可查询 (线性扫描)
- ├─ 06:00: 手动 Flush
 - ├─ Growing Seg 1 → Sealed Seg 1 (转换 <1秒)
 - ├─ 新建 Growing Seg 2
 - └─ 后台 IndexNode 开始构建 Sealed Seg 1 的索引 (耗时 30-60 分钟, 用户无感知)
- ├─ 06:00-12:00: Batch 2 插入
 - ├─ Growing Seg 2 接收数据 (400MB)
 - └─ 同时 Sealed Seg 1 在后台构建索引
- ├─ 12:00: 手动 Flush
 - └─ Growing Seg 2 → Sealed Seg 2
- ├─ 18:00: 手动 Flush
 - └─ Growing Seg 3 → Sealed Seg 3
- └─ Day 1 结束:
 - ├─ 4 个 Sealed Segment (每 6h 一个)
 - ├─ 所有数据已可快速查询 (索引已完整)
 - ├─ 1 个 Growing Segment (准备接收 Day 2 数据)
 - └─ Segment 总数 = 5

Day 2 & 3: 重复过程, Segment 继续累积

- └─ Day 3 结束: Segment 总数 = 13

Day 4 (凌晨)

- ├─ 系统检测: Segment 数 = 13 > 阈值 (如 10)
- ├─ 触发 Compaction!
- ├─ 合并 Day 1 的 4 个 Segment → 1 个 Compacted Segment
 - ├─ 耗时: 15-30 分钟 (后台异步)
 - ├─ 用户无感知
 - └─ 完成后: Segment 总数 = 10 (从 13 → 10)

Day 5+: 系统进入稳定循环

- ├─ 持续新增 Segment (每天 4 个)
- ├─ 定期触发 Compaction (保持总数 < 阈值)
- ├─ Segment 数始终在 10-20 之间波动
- └─ 系统自我调节, 达到动态平衡

3.2 Compaction 的触发条件

自动触发 Compaction 的条件（任一满足）：

条件 1：Segment 数量过多

参数名：dataCoord 内部配置（不同版本可能不同）

推荐阈值：Segment 数 $>$ 20-30 个

监控和触发流程：

1. 系统每 10 秒检查一次 Segment 数量
2. 若超过阈值（如 30 个）
3. 触发 Compaction
4. 合并策略：选择 10-15 个小 Segment 合并为 1 个
5. 继续循环，直到 Segment 数 \leq 阈值

实际效果：

- └ 查询前：需在 30 个 Segment 上并行查询 \rightarrow 30 次聚合
- └ Compaction 后：只需在 15-20 个 Segment 上查询 \rightarrow 15-20 次聚合
- └ 性能提升：约 50%（查询速度快 2 倍）

条件 2：删除数据占比过高

参数名：datanode.binlog.garbage_collection_ratio

默认值：0.1（10%）

触发条件：已删除或标记删除的数据 $>$ 10% 的总数据量

场景和原因：

- └ 超过 TTL 的旧新闻被标记删除
- └ 每条 Segment 中都有一些“空洞”（已删数据）
- └ 空洞比例 $>$ 10% 时，触发 Compaction 清理
- └ 好处：回收存储空间，降低内存占用

实例：

- └ Sealed Seg 1: 1000MB 数据，删除 150MB（15% $>$ 10%）
- └ 触发 Compaction
- └ Compaction 后：850MB（物理删除，空间回收）

条件 3：定时自动检查

参数名：dataCoord.compactionRoundInterval

默认值：86400 秒（每天）

执行时间：通常在系统低负载时（如凌晨 2-4 点）

逻辑：

1. 每 24 小时系统自动检查
2. 评估是否满足条件 1 或 2
3. 满足则触发 Compaction
4. 即使条件未满足，定期检查也能发现潜在问题

条件 4：手动触发

API : `collection.compact()`

使用场景：

- └ 大量数据删除后，想立刻回收空间
- └ 性能测试前，想达到最优 Segment 状态
- └ 管理员手动干预，应对特殊情况
- └ 定期维护任务

代码示例：

```
compaction_id = collection.compact()

# 异步等待 Compaction 完成
import time
while True:
    state = collection.get_compaction_state(compaction_id)
    if state == "Completed":
        print("✓ Compaction 完成")
        break
    elif state == "Failed":
        print("✗ Compaction 失败")
        break
    else:
        print(f"Compaction 进行中... 状态: {state}")
        time.sleep(30)
```

3.3 完整的 Flush 与 Compaction 配置建议

企业新闻库推荐配置（日新增 1M 条）：

参数配置（在 `milvus.yaml` 中设置）：

Flush 相关配置	
datanode:	
segment:	
max_size: 512	
min_size: 128	
common:	
retentionDuration: 86400 # 24 小时	

Compaction 相关配置	
dataCoord:	
enableAutoCompaction: true	
compactionRoundInterval: 43200 # 12 小时	
dataNode:	
binlog:	
garbage_collection_ratio: 0.15 # 15%	

实际效果：

- └─ Flush 策略：
 - └─ 自动：数据量 \geq 512MB 或时间 \geq 24h
 - └─ 手动：每 6 小时周期性调用一次
 - └─ 结果：每天产生 4-8 个 Sealed Segment
- └─ Compaction 策略：
 - └─ 自动：每 12 小时检查一次
 - └─ 条件：Segment 数 $>$ 30 或删除占比 $>$ 15%
 - └─ 结果：Segment 数保持在 15-30 之间
- └─ 整体效果：
 - └─ 热层 (0-7天)：15-30 个 Segment
 - └─ 查询聚合：15-30 次 (vs 理想状况的 1 次)
 - └─ 性能对标：相对基线 (单 Segment) 损失 5-10%
 - └─ 权衡：时间 $<$ 200ms (可接受)，稳定性优先

四、完整的数据生命周期 (修正版)

Timeline：从新闻产生到冷层归档

Day 1 (00:00)

新闻产生

↓

向量化 → 多维嵌入 (3 个向量)

↓

批量插入 Collection

↓

自动进入 Growing Segment 1

- └─ 状态：在线写入，无索引
- └─ 查询：支持 (线性扫描)
- └─ 数据：持续累积中

Day 1 06:00：周期性手动 Flush

- └─ Growing Seg 1 → Sealed Seg 1
- └─ 新建 Growing Seg 2
- └─ IndexNode 后台构建 Sealed Seg 1 索引 (30-60min)

Day 1 06:35：Sealed Seg 1 索引完成

- └─ ✓ 该 Segment 可高速查询

重复 Flush 操作 (每 6 小时)

- └─ 结果：Day 1 产生 4 个 Sealed Segment

Day 1-7 (热层，HNSW 索引)

- └─ 每天产生 4-8 个 Sealed Segment
- └─ 自动 Compaction (维持 Segment 数 $<$ 30)
- └─ 查询延迟：P99 $<$ 100ms
- └─ 存储位置：内存 + SSD
- └─ 用户主要访问范围

Day 8 (转移到温层)

- └ Day 1 的数据满足 TTL (>7天)
- └ 自动标记为转移候选
- └ 后台任务启动：
 - └ 从热层读取数据
 - └ 重新索引 (HNSW → IVF)
 - └ 转移到温层分区
- └ 查询延迟：P99 100-500ms
- └ 存储位置：标准 SSD
- └ 偶尔访问 (对标、历史分析)

Day 31 (转移到冷层)

- └ 超过 30 天的数据
- └ 自动转移到冷层：
 - └ 从温层读取数据
 - └ 索引压缩 (IVF → IVF_PQ, 压缩 90%)
 - └ 转移到对象存储 (MinIO/S3)
- └ 查询延迟：1-5 秒
- └ 存储成本：1/10 (相对热层)
- └ 罕见访问 (合规审计、长期分析)

Day 365+ (永久归档)

- └ 数据永久保留
- └ 支持历史查询
- └ 成本极低
- └ 只在特殊需求时访问

五、核心参数参考表

参数名	默认值	说明
Segment 大小上限 datanode.segment.max_size	512 MB	Growing → Sealed 的数据量阈值
Segment 保留时间 common.retentionDuration	86400 秒 (24小时)	Growing 无数据也会 Flush 的时间
Segment 最小合并大小 datanode.segment.min_size	128 MB	Compaction 时不会处理 <128MB 的
删除数据清理比率 datanode.binlog.garbage... _collection_ratio	0.1 (10%)	删除占比 > 10% 时触发 Compaction
Compaction 检查间隔 dataCoord.compaction... CheckInterval	10 秒	每 10 秒检查一次是否需要 Compact
Compaction 执行间隔 dataCoord.compaction...	86400 秒	每 24 小时执行一

RoundInterval	(每天)	次定时检查	
---------------	------	-------	--

六、企业级最佳实践

6.1 推荐的三层架构配置

热层 (0-7天) :

- └ 策略：主动周期性 Flush (每 3-6 小时)
- └ 索引：HNSW
- └ 存储：内存 + SSD
- └ Segment 数：维持 15-30 个
- └ Compaction：自动触发 (Segment 数 $>$ 30)
- └ 查询延迟目标：P99 \leq 100ms

温层 (8-30天) :

- └ 策略：自动分层转移
- └ 索引：IVF (空间对标 50%)
- └ 存储：标准 SSD
- └ Segment 数：5-10 个 (已充分合并)
- └ Compaction：极少触发
- └ 查询延迟目标：P99 100-500ms

冷层 (31+天) :

- └ 策略：永久保留，按需查询
- └ 索引：IVF_PQ (压缩 90%)
- └ 存储：对象存储 (MinIO/S3)
- └ Segment 数：1-3 个 (充分压缩)
- └ Compaction：不需要
- └ 查询延迟：1-5 秒 (可接受)

6.2 监控和告警检查清单

☐ Segment 数监控

- └ 热层：目标 \leq 30 个，告警 $>$ 50 个
- └ 温层：目标 \leq 10 个，告警 $>$ 20 个
- └ 冷层：目标 \leq 5 个，告警 $>$ 10 个

☐ Flush 监控

- └ 定期检查手动 Flush 任务是否正常执行
- └ 监控 Growing Segment 大小 (不应无限增长)
- └ 告警：Growing Segment $>$ 600MB (接近上限)

☐ Compaction 监控

- └ 定期检查自动 Compaction 是否被触发
- └ 监控 Compaction 耗时 (正常 5-20min)
- └ 告警：Compaction 失败、超时 $>$ 60min

☐ 索引构建监控

- └ Sealed Segment 索引构建进度
- └ 构建耗时 (正常 30-90min)

- └ 告警：索引构建失败、>120min 未完成

- 存储空间监控

- └ 热层内存占用 (不应超过总内存 60%)
- └ SSD 占用 (不应超过 80%)
- └ 冷层对象存储 (成本跟踪)

七、面试核心答题模板（完整版）

Question 1：Milvus 中 Segment、Growing、Sealed、Compaction、Flush 的关系？

标准答案框架：

Milvus 的数据组织分为 7 层（从下到上）：

1. 向量层：单条新闻一个 1024 维向量 (4KB)
2. Batch 层：批量插入多个向量 (100K-1M 条)
3. Growing Segment：接收新数据的阶段，无索引
 - └ Flush 触发条件：数据 ≥ 512MB 或时间 ≥ 24h
4. Sealed Segment：Flush 后变成只读，后台构建索引
5. Compaction：多个小 Segment 合并为 1 个
 - └ 触发条件：Segment 数 > 30 或删除占比 > 10%
6. Partition：按时间/业务维度划分
7. Collection：整个向量库

核心流程：

新闻 → 向量化 → Growing Seg(无索引) → Flush → Sealed Seg(有索引)

- └ 后台异步 → 30-60 分钟完成
 - └ 期间用户已可查询 (虽然初期性能差)
 - └ 索引完成后性能显著提升
- └ 多个 Sealed Seg 积累 → Compaction 合并 → 性能优化

推荐策略：

- └ Flush：主动周期性 (每 6 小时) + 自动条件触发
- └ Compaction：自动 (系统自我调节)

Question 2：多少个 Document 变成一个 Segment？多少个 Segment 变成 Compaction？

标准答案框架：

这两个都是 "自动机制"，不是硬性设置：

Document → Segment (Flush 机制)：

- └ 自动触发：
 - └ 数据量 ≥ 512MB (通常 128-130 万条向量)
 - └ 或时间 ≥ 24 小时
- └ 手动触发：collection.flush()
- └ 企业新闻日新增 1M 条 → 每天产生 4-8 个 Segment

Segment → Compaction (合并机制)：

- └─ 自动触发条件（任一）：
 - └─ Segment 数 > 20-30 个
 - └─ 删除数据占比 > 10%
 - └─ 定时检查（每 12-24 小时）
- └─ 手动触发：collection.compact()
- └─ 合并效果：N 个小 Seg → 1 个大 Seg，查询快 50%

推荐配置（企业新闻场景）：

- └─ Segment 预期数量：热层 15-30，温层 5-10，冷层 1-3
- └─ Compaction 频率：自动（系统维护）+ 定期检查
- └─ 成本：日均 90 分钟（新索引构建）+ 后台自动优化

Question 3：Flush 和 Compaction 的性能影响？

标准答案框架：

Flush 的影响：

- └─ 时间成本：<1 秒（只是数据转移）
- └─ 索引成本：30-60 分钟（后台异步，用户无感知）
- └─ 用户查询：可立即开始（虽然初期无索引）
- └─ 最终性能：索引完成后恢复正常
- └─ 总体评价：无负面影响

Compaction 的影响：

- └─ 执行时间：5-20 分钟（后台异步）
- └─ 用户查询：完全无中断
- └─ 性能提升：查询速度快 30-50%（减少 Segment 聚合）
- └─ 存储回收：删除数据清理，空间节省 10-30%
- └─ 准确度：零影响（纯物理重组，语义不变）
- └─ 总体评价：无害化优化，全面收益

推荐策略结果：

- └─ 热层查询延迟：P99 < 100ms
- └─ 温层查询延迟：P99 100-500ms
- └─ 冷层查询延迟：1-5 秒
- └─ 存储成本：相比单层降低 60-70%
- └─ 维护复杂度：完全自动化，无需人工干预

八、常见问题与解答

Q1：Flush 和 Compaction 会影响查询吗？

A：不会。Flush 是后台操作，<1 秒完成。索引构建在后台异步进行，用户可立即查询。Compaction 也是后台异步，查询无中断。

Q2：Segment 数量越多越好还是越少越好？

A：越少越好（在合理范围内）。Segment 数多导致查询聚合次数多。推荐热层 15-30 个，通过定期 Compaction 维持。

Q3：手动 Flush 会阻塞数据插入吗？

A：不会。Flush 只是将 Growing → Sealed，新建一个空 Growing 继续接收数据。大概 <1 秒完成。

Q4：Compaction 是否会改变向量或精度？

A：不会。Compaction 是纯物理重组，向量数据完全相同，相似度计算完全相同，查询结果完全相同。

Q5：什么时候应该手动 Flush vs 自动 Flush？

A：生产环境推荐"定期手动 + 自动条件"组合。定期手动（如每 6 小时）确保数据及时进入索引；自动条件作为备选机制防止极端情况。

九、总结建议

核心三要点

1. **分层架构**：热→温→冷，不删除，按温度分配资源
2. **自动化管理**：Flush + Compaction 系统自我调节，无需过度人工干预
3. **性能平衡**：用分层交换成本，用并发换延迟，用自动化换复杂度

落地四步

- 第 1 个月：部署三层架构，配置 Segment 和 Compaction 参数
- 第 2 个月：实施周期性 Flush 策略，验证性能基线
- 第 3 个月：集成多向量、Cross-Encoder 精排
- 第 4 个月：完整监控体系上线，自动化告警

预期收益

性能提升：50-80%（查询延迟降低）
成本优化：60-70%（存储成本下降）
精度提升：15-25%（多维度融合）
可维护性：+95%（完全自动化）

附录：完整配置示例

```
# milvus.yaml 推荐配置（企业新闻场景）<a></a>

datanode:
  segment:
    max_size: 512          # Segment 大小上限 (MB)
    min_size: 128         # Segment 最小合并大小 (MB)

  binlog:
    garbage_collection_ratio: 0.15 # 删除占比触发 Compaction

common:
  retentionDuration: 86400      # Growing Segment 保留时间 (秒)

dataCoord:
  enableAutoCompaction: true   # 启用自动 Compaction
```

```
compactionCheckInterval: 10 # 检查间隔 (秒)
compactionRoundInterval: 43200 # 执行间隔 (12 小时)
```

推荐的应用层代码逻辑

```
def setup_optimal_workflow():
    """设置最优的 Flush + Compaction 工作流"""

    # 定期 Flush 任务 (每 6 小时)
    def periodic_flush():
        collection.flush()
        logging.info("定期 Flush 完成")

    # 监控 Segment 状态
    def monitor_segments():
        segment_count = get_segment_count()
        if segment_count > 50:
            logging.warning(f"Segment 过多: {segment_count}, 可手动触发 Compaction")

    # 启动定时任务
    schedule.every(6).hours.do(periodic_flush)
    schedule.every(1).hours.do(monitor_segments)
```