**电 子 科 技 大 学**

**作 业 报 告**

**学生姓名：解雨欣 学号：202452080506 指导教师：段翰聪**

1. **作业名称**

Linux环境高级编程大作业：小对象存储引擎

1. **作业要求**

**形式要求:**

基于C/C++，提供上述接口，并封装为SO库。

**实现约束:**

1. 所有小对象需统一存储在一个大文件中，设计内存缓存对小对象进行CACHE，Key为整型，Value支持任意数据结构;
2. 参考文件系统模型设计大文件的物理对象存储模型，元数据与对象数据寻址方法;
3. 需支持多线程环境，采用多生产者多消费者模型;完成持续读写压力测试，运行时间不低于12小时，插入对象不低于1千万个，观察CPU，内存，磁盘开销，并将压测分析输出到报告中。

**提交内容:**

可编译的源码(存储引擎和测试程序源代码)，可执行文件和SO库文件;设计文档(包括模块框架设计，重要流程图(主要是GET/PUT/DEL接口流程)，核心数据结构，运行时截图(操作相关结果)。

1. **系统设计**
   1. **模块框架设计**
      1. **本系统采用模块化设计，主要分为四个模块：**
2. **存储引擎模块（StorageEngine）：**提供对外的GET、PUT、DEL接口，使用线程池（ThreadPool）来处理并发任务。使用LRU缓存（LRUCache）进行数据缓存，以提高性能。使用文件存储（FileStore）作为持久化存储，缓存未命中的数据。
3. **缓存模块（LRUCache）**：负责在内存中缓存热点对象，采用分段的LRU（最近最少使用）策略，提高对象的读取速度。分为多个缓存段（Segment），每个段独立管理一定数量的缓存对象，支持并发访问。
4. **文件存储模块（FileStore）：**负责文件存储的读写操作。维护一个对象索引（ObjectMeta）来映射数据的存储位置。提供垃圾回收机制来清理无效数据，并支持文件压缩（compactFile）。在文件中存储数据，并通过索引文件进行管理。
5. **线程池模块（ThreadPool）**：管理多线程环境下的任务调度，采用多生产者多消费者模型，提高系统的并发处理能力。通过任务队列和工作线程池实现高效的任务分发和执行。  
   * 1. **以PUT操作为例，详细说明各模块之间的交互流程：**
6. **接收PUT 请求：**客户端通过外部接口调用StorageEngine::asyncPut(key, value, callback)发起PUT请求。
7. **任务提交：**StorageEngine将PUT任务封装为一个函数对象，并提交给ThreadPool的任务队列。
8. **任务调度与执行：**ThreadPool中的一个工作线程从任务队列中取出任务，执行该函数对象。
9. **文件存储操作：**工作线程调用StorageEngine::put(key, value)。StorageEngine::put首先调用FileStore::put(key, value)，将对象写入大文件，并更新内存中的索引表。
10. **缓存更新：**StorageEngine::put随后调用LRUCache::put(key, value)，将新对象添加到缓存中。如果缓存已满，根据LRU策略移除最久未使用的对象。
11. **任务完成与回调：**PUT操作完成后，若提供了回调函数，则调用回调函数通知操作结果。
    * 1. **模块间的同步机制：**
12. **互斥锁**
    1. **FileStore：**使用std::mutex file\_mtx\_保护文件读写操作，防止多个线程同时访问文件导致数据损坏、使用std::shared\_mutex index\_mtx\_保护索引表的读写，允许多个线程并发读取索引，提高读取性能。
    2. **LRUCacheSegment:** 每个缓存段使用独立的std::mutex mtx\_，确保对缓存段的并发访问安全，同时减少不同线程访问不同缓存段时的锁竞争。
    3. **ThreadPool：**使用std::mutex mutex\_和std::condition\_variable cv\_保护任务队列的访问和线程同步。
13. **条件变量**

std::condition\_variable tasks\_done\_cv\_用于通知线程池工作线程有新任务到来或线程池停止运行，实现高效的线程等待与唤醒机制。

1. **原子操作**

使用std::atomic变量（如stop\_、active\_tasks\_count\_等）实现无锁的状态管理，减少锁的使用，提高并发性能。

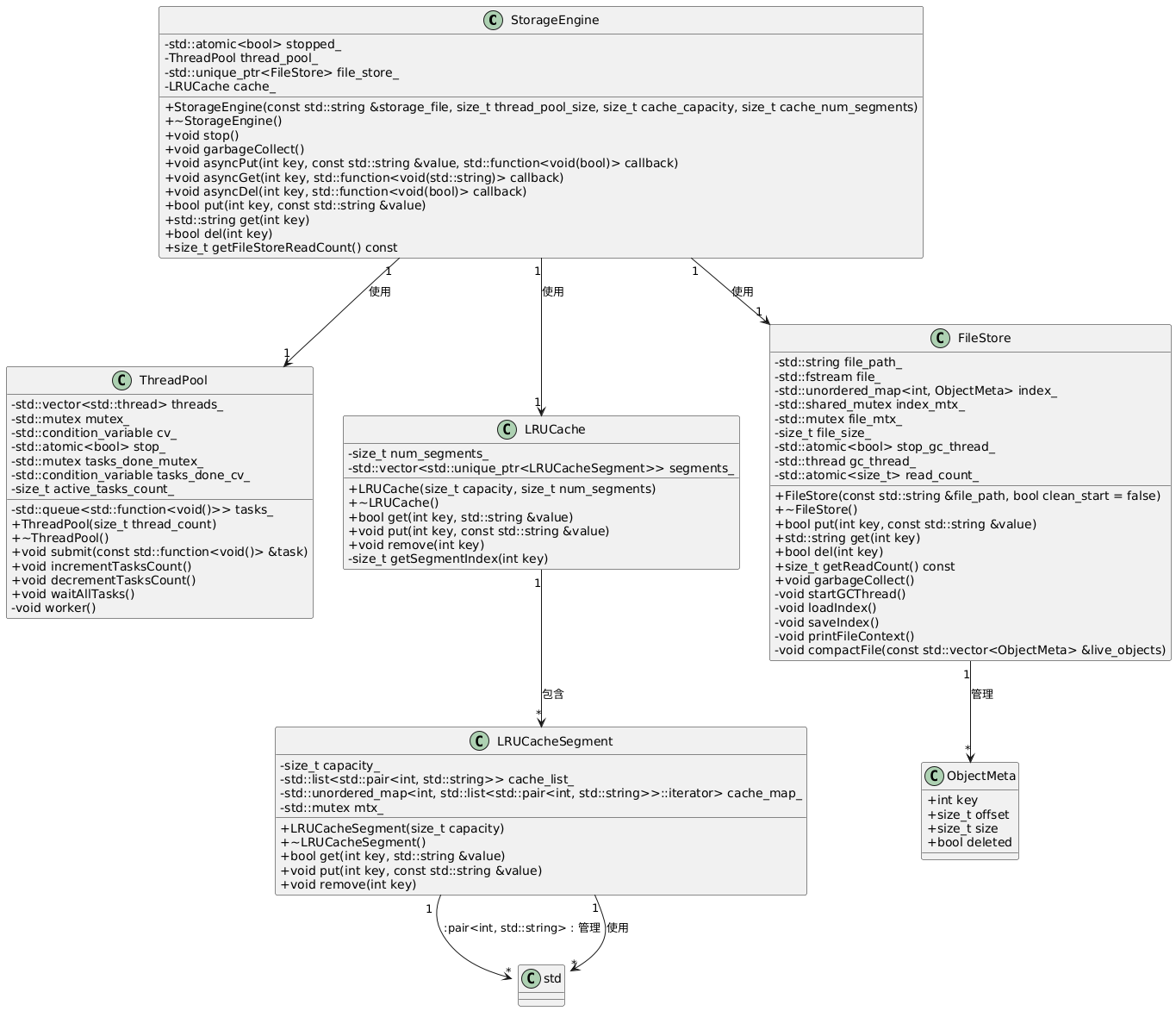
* + 1. **从索引文件查找数据的过程：**

**索引文件（.idx）**：存储键到数据在数据文件中偏移量和大小的映射。在系统启动时，FileStore会加载索引文件，恢复内存中的索引表。

**数据文件：**存储实际的值数据，采用追加写入的方式。每个数据项在文件中的位置由索引表中的offset和size决定。

**过程：**

1. **通过索引表查找键：**使用哈希表（std::unordered\_map）快速定位键对应的ObjectMeta。
2. **定位数据位置：**根据ObjectMeta中的offset和size，确定数据在文件中的具体位置和长度。
3. **读取数据：**通过文件流的seekg和read方法，获取实际的值数据。
   * 1. **整体框架图**

****

* 1. **核心数据结构**

1. **对象元数据（ObjectMeta）：**记录每个对象在大文件中的存储位置和大小，以及是否已被删除，便于快速检索和垃圾回收。

*// 对象元数据*

struct ObjectMeta

{

    int key; *// 对象的Key*

    size\_t offset; *// 数据在文件中的偏移量*

    size\_t size; *// 数据大小*

    bool deleted = false; *// 标记该对象是否已删除*

};

1. **缓存数据结构（LRUCacheSegment、LRUCache）：**将整个缓存划分为多个独立的缓存段，提高并发访问效率，减少锁竞争。每个缓存段管理一定数量的对象，采用LRU策略，通过双向链表和哈希表实现高效的插入、删除和查找。

*// 分段缓存*

class LRUCache

{

public:

    LRUCache(size\_t capacity, size\_t num\_segments);

    ~LRUCache() = default;

    bool get(int key, std::string &value);

    void put(int key, const std::string &value);

    void remove(int key);

private:

    size\_t num\_segments\_;

    std::vector<std::unique\_ptr<LRUCacheSegment>> segments\_;

    size\_t getSegmentIndex(int key);

};

*// 单个缓存段，使用LRU策略*

class LRUCacheSegment

{

public:

    LRUCacheSegment(size\_t capacity);

    ~LRUCacheSegment() = default;

    bool get(int key, std::string &value);

    void put(int key, const std::string &value);

    void remove(int key);

private:

    size\_t capacity\_;

    std::list<std::pair<int, std::string>> cache\_list\_;

    std::unordered\_map<int, std::list<std::pair<int, std::string>>::iterator> cache\_map\_;

    std::mutex mtx\_;

};

1. **索引表（unordered\_map）：**在内存中维护一个键到对象元数据的映射表，支持快速定位对象在大文件中的位置和大小。

    std::unordered\_map<int, ObjectMeta> index\_; *// 索引表 (Key -> ObjectMeta)*

1. **线程池数据结构（ThreadPool）：**管理多个工作线程，负责执行提交的任务，实现高效的多线程并发处理。

class ThreadPool

{

public:

    explicit ThreadPool(size\_t thread\_count);

    ~ThreadPool();

    void submit(const std::function<void()> &task);

    void incrementTasksCount();

    void decrementTasksCount();

    void waitAllTasks();

private:

    void worker(); *// 工作线程的主循环*

    std::vector<std::thread> threads\_; *// 工作线程*

    std::queue<std::function<void()>> tasks\_; *// 任务队列*

    std::mutex mutex\_; *// 保护任务队列的互斥锁*

    std::condition\_variable cv\_; *// 用于任务通知的条件变量*

    std::atomic<bool> stop\_; *// 控制线程池是否停止*

*// 等待所有任务完成*

    std::mutex tasks\_done\_mutex\_;

    std::condition\_variable tasks\_done\_cv\_;

    size\_t active\_tasks\_count\_ = 0;

};

* 1. **接口设计**
     1. **GET接口流程**

**接口功能：**GET接口用于根据给定的键（Key）检索对应的值（Value）。系统首先在缓存中查找，如果命中则直接返回，否则从文件存储中读取并更新缓存。

**工作流程：**

1. **接收GET请求：**

**StorageEngine** 接收到一个**GET(key)**请求。

1. **查询缓存：**

调用 **LRUCache::get(key, value)**。

如果缓存命中：返回**true**并将**value**设置为缓存中的值。

如果缓存未命中：继续执行文件存储查询。

1. **查询文件存储：**

调用 **FileStore::get(key)**。

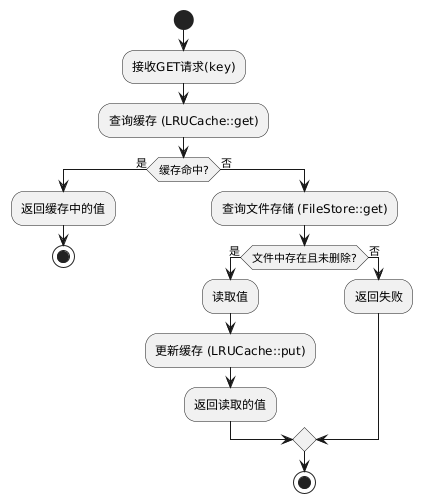
如果文件存储中存在该键且未删除：将读取到的值写入**value**。调用 **LRUCache::put(key, value)**更新缓存。最后返回**value**。

如果键不存在或已删除：返回空字符串或错误标志。

**4. 返回结果：**

**StorageEngine**返回GET操作的结果给调用方。

**流程图：**



**代码实现：**

bool FileStore::get(int key, std::string &value)

{

*// 查找索引*

    std::shared\_lock<std::shared\_mutex> index\_lock(index\_mtx\_);

    auto it = index\_.find(key);

    if (it == index\_.end() || it->second.deleted)

    {

        return false; *// Key 未找到或已被删除*

    }

    const ObjectMeta &meta = it->second;

*// 从文件读取数据*

    value.resize(meta.size);

    {

        std::lock\_guard<std::mutex> file\_lock(file\_mtx\_);

        file\_.seekg(meta.offset, std::ios::beg);

        file\_.read(&value[0], meta.size);

        if (!file\_)

        {

            std::cerr << "Failed to read from file." << std::endl;

            return false;

        }

        read\_count\_++;

    }

    return true;

}

bool LRUCacheSegment::get(int key, std::string &value)

{

    std::lock\_guard<std::mutex> lock(mtx\_);

    auto it = cache\_map\_.find(key);

    if (it == cache\_map\_.end())

    {

        return false;

    }

*// 移动到链表头部*

    cache\_list\_.splice(cache\_list\_.begin(), cache\_list\_, it->second);

    value = it->second->second;

    return true;

}

std::string StorageEngine::get(int key)

{

    std::string value;

    if (cache\_.get(key, value))

    {

        return value; *// 缓存命中*

    }

*// 缓存未命中，访问底层存储*

    value = file\_store\_->get(key);

    if (!value.empty())

    {

        cache\_.put(key, value); *// 更新缓存*

    }

    return value;

}

* + 1. **PUT接口流程**

**接口功能：**PUT接口用于根据给定的键（Key）存储或更新对应的值（Value）。系统将对象写入文件存储，并更新索引表和缓存。

**工作流程：**

1. **接收PUT请求：**

**StorageEngine** 接收到一个 **PUT(key, value)** 请求。

1. **写入文件存储：**

调用 **FileStore::put(key, value)**。

写入数据到大文件，并更新索引表中的对象元数据。

如果键已存在，则覆盖旧值，旧数据成为垃圾数据。

1. **更新缓存：**

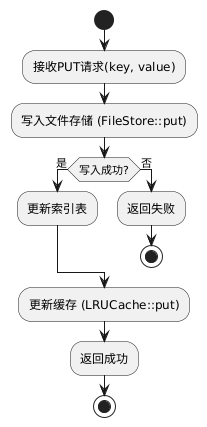
调用 **LRUCache::put(key, value)**，将新值添加到缓存中。

如果缓存已满，则根据LRU策略移除最久未使用的对象。

1. **返回结果：**

**StorageEngine** 返回 PUT 操作的结果给调用方。

**流程图：**



**代码实现：**

bool FileStore::put(int key, const std::string &value)

{

    std::unique\_lock<std::shared\_mutex> index\_lock(index\_mtx\_);

    size\_t offset;

    {

        std::lock\_guard<std::mutex> file\_lock(file\_mtx\_);

        offset = file\_size\_;

        file\_.seekp(file\_size\_, std::ios::beg);

        file\_.write(value.c\_str(), value.size());

        if (!file\_)

        {

            std::cerr << "Failed to write to file." << std::endl;

            return false;

        }

        file\_.flush();

        file\_size\_ += value.size();

    }

    index\_[key] = ObjectMeta{key, offset, value.size(), false};

    return true;

}

void LRUCacheSegment::put(int key, const std::string &value)

{

    std::lock\_guard<std::mutex> lock(mtx\_);

    auto it = cache\_map\_.find(key);

    if (it != cache\_map\_.end())

    {

*// 更新值并移动到链表头部*

        it->second->second = value;

        cache\_list\_.splice(cache\_list\_.begin(), cache\_list\_, it->second);

    }

    else

    {

*// 如果容量已满，移除最久未使用的项*

        if (cache\_list\_.size() >= capacity\_)

        {

            auto last = cache\_list\_.back();

            cache\_map\_.erase(last.first);

            cache\_list\_.pop\_back();

        }

*// 插入新项到链表头部*

        cache\_list\_.emplace\_front(key, value);

        cache\_map\_[key] = cache\_list\_.begin();

    }

}

bool StorageEngine::put(int key, const std::string &value)

{

*// 调用底层存储*

    if (!file\_store\_->put(key, value))

    {

        return false; *// 存储失败*

    }

*// 更新缓存*

    cache\_.put(key, value);

    return true;

}

* + 1. **DEL接口流程**

**接口功能：**DEL接口用于根据给定的键（Key）删除对应的对象。系统通过标记对象为删除状态，并从缓存中移除该对象。

**工作流程：**

1. **接收DEL请求：**

**StorageEngine** 接收到一个 **DEL(key)** 请求。

1. **标记对象为删除：**

调用 **FileStore::del(key)**。

在索引表中将对象的 **deleted** 标记设为 **true**，表示该对象已删除。

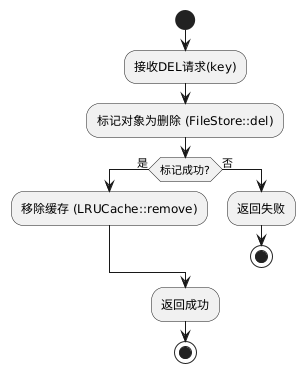
1. **移除缓存：**

调用 **LRUCache::remove(key)**，从缓存中移除该对象。

1. **返回结果：**

**StorageEngine** 返回 **DEL** 操作的结果给调用方。

**流程图：**



**代码实现：**

bool FileStore::del(int key)

{

    std::unique\_lock<std::shared\_mutex> index\_lock(index\_mtx\_);

    auto it = index\_.find(key);

    if (it == index\_.end() || it->second.deleted)

    {

        return false; *// Key 未找到或已被删除*

    }

*// 标记为已删除*

    it->second.deleted = true;

    return true;

}

void LRUCacheSegment::remove(int key)

{

    std::lock\_guard<std::mutex> lock(mtx\_);

    auto it = cache\_map\_.find(key);

    if (it != cache\_map\_.end())

    {

        cache\_list\_.erase(it->second);

        cache\_map\_.erase(it);

    }

}

bool StorageEngine::del(int key)

{

    cache\_.remove(key); *// 从缓存中删除*

    return file\_store\_->del(key);

}

1. **系统实现**
   1. **开发环境**
      1. **硬件配置**

**处理器（CPU）**：**13th Gen Intel® Core™ i9-13900H @ [2.60 GHz]×10核心**

**内存（RAM）：**约**16GB**

**主存储（NVMe SSD）：**约**1TB**（型号：**SAMSUNG MZVL21T0HCLR-00BL2**）

**网络：WSL2** 虚拟网络适配器

* + 1. **软件环境**

**操作系统**：**Ubuntu 22.04.5 LTS (WSL2)**

**内核版本**：**5.15.167.4-microsoft-standard-WSL2**

**编译器**：**GNU Compiler Collection (GCC) 11.4.0**

**构建工具**：**Make 4.3**

**性能分析工具：**

**gprof (GNU Binutils for Ubuntu) 2.38**：分析程序的性能

**vmstat、iostat、pidstat**：系统资源监控工具

**代码覆盖率分析工具：** **gcov** 和 **lcov**

**依赖库**：

**pthread：**随系统 C 库（libc6 2.35-0ubuntu3.8）提供，用于多线程支持

**gtest：**libgtest-dev 1.11.0-3 用于单元测试

**开发工具**：

**IDE：**Visual Studio Code 1.96.0

**版本控制**：Git 2.34.1

**脚本语言**：Bash、Python3.10.12

* + 1. **编译参数：**

g++ - std = c++ 20 - Wall - Wextra - O2 - g -pg - fPIC - lpthread - lgtest - lgtest\_main - shared - o libstorage\_engine.so \*.cpp

* 1. **实现细节**

本节介绍存储引擎各模块的关键实现细节，包括缓存管理、文件存储、线程池以及存储引擎的接口封装。详细的代码实现见工程文件，这里不再说明。

* + 1. **缓存管理（LRUCache）**

**设计**：为了提高读取性能，设计了分段的LRU缓存机制。将整个缓存划分为多个独立的缓存段（Segment），每个段独立管理一定数量的缓存对象，采用独立的锁，减少锁竞争，提升并发性能。

**LRUCache：**将缓存分为多个LRUCacheSegment，每个段管理部分键值对。通过哈希函数根据键值分配段，减少锁竞争，提升并发访问效率。

**LRUCacheSegment：**使用双向链表（std::list）维护缓存对象的访问顺序，实现LRU策略。使用哈希表（std::unordered\_map）实现键到链表节点的快速映射，支持O(1)的插入、删除和查找操作。每个缓存段使用独立的std::mutex保护，确保线程安全。

* + 1. **文件存储（FileStore）**

**设计：**所有小对象统一存储在一个大文件中，通过索引表快速定位对象的位置和大小。使用内存中的哈希表（std::unordered\_map）维护键到对象元数据（ObjectMeta）的映射，支持快速检索。定期执行垃圾回收，清理已删除或过期的对象，优化存储空间利用率。

**对象元数据（**ObjectMeta**）：**记录每个对象在文件中的存储位置（偏移量）、大小以及删除标记。

**索引表：使用**std::unordered\_map<int, ObjectMeta>在内存中维护键到对象元数据的映射，支持快速查找和更新。

**文件读写：**使用std::fstream以二进制模式打开大文件，支持随机读写操作。通过锁机制（file\_mtx\_）保护文件读写操作，防止多个线程同时访问文件导致数据损坏。

**垃圾回收：**实现一个后台garbageCollect线程，定期扫描索引表，识别已删除的对象，并通过文件压缩和重写优化存储空间。

* + 1. **线程池（ThreadPool）**

**设计：**为了支持高并发操作，系统设计了一个高效的线程池。线程池采用生产者-消费者模型，通过任务队列管理待执行的任务，并由工作线程从队列中取出任务执行。线程池设计注重任务调度的高效性和线程的复用，减少线程创建和销毁的开销。

**任务队列：**使用std::queue<std::function<void()>>存储待执行的任务。通过submit方法将任务添加到队列，并使用incrementTasksCount和decrementTasksCount跟踪活跃任务数量。

**工作线程：**创建固定数量的工作线程，在构造函数中启动。每个工作线程执行一个主循环，从任务队列中取出任务并执行。使用原子变量stop\_控制线程池的停止。通过std::mutex和std::condition\_variable实现线程安全的任务添加和取出。

**任务计数**：使用原子变量active\_tasks\_count\_跟踪当前活跃的任务数量，用waitAllTasks()方法等待所有任务完成。

void ThreadPool::submit(const std::function<void()> &task)

{

    {

        std::unique\_lock<std::mutex> lock(mutex\_);

        tasks\_.push(task);

        cv\_.notify\_one(); *// 通知一个线程处理任务*

    }

    incrementTasksCount();

}

void ThreadPool::worker()

{

    while (true)

    {

        std::function<void()> task;

        {

            std::unique\_lock<std::mutex> lock(mutex\_);  
 *// 等待新任务或停止信号*

            cv\_.wait(lock, [this]

                     { return stop\_ || !tasks\_.empty(); });

            if (stop\_ && tasks\_.empty())

            {

                return; *// 退出线程*

            }

            task = tasks\_.front();

            tasks\_.pop();

        }

        task(); *// 执行任务*

        decrementTasksCount();

    }

}

void ThreadPool::incrementTasksCount()

{

    std::lock\_guard<std::mutex> lock(tasks\_done\_mutex\_);

    active\_tasks\_count\_++;

}

void ThreadPool::decrementTasksCount()

{

    std::lock\_guard<std::mutex> lock(tasks\_done\_mutex\_);

    active\_tasks\_count\_--;

    if (active\_tasks\_count\_ == 0)

    {

        tasks\_done\_cv\_.notify\_all();

    }

}

void ThreadPool::waitAllTasks()

{

    std::unique\_lock<std::mutex> lock(tasks\_done\_mutex\_);

    tasks\_done\_cv\_.wait(lock, [this]

                        { return active\_tasks\_count\_ == 0; });

}

* + 1. **存储引擎（StorageEngine）**

**设计：**StorageEngine模块作为系统的对外接口，负责协调缓存模块（LRUCache）和文件存储模块（FileStore），处理GET、PUT、DEL等操作。为了提升系统的响应速度和并发处理能力，StorageEngine采用异步接口，通过线程池提交任务并使用回调机制通知操作结果。

异步接口：提供asyncGet、asyncPut、asyncDel方法，允许客户端异步发起操作。通过回调函数（std::function<void(bool)>等）通知操作结果，提升系统的并发处理能力。

同步接口：提供get、put、del方法，供内部任务执行使用，确保操作的正确性和一致性。在PUT操作中，先将数据写入文件存储，再更新缓存。在GET操作中，先查找缓存，若未命中则从文件存储读取并更新缓存。在DEL操作中，先标记文件存储中的对象为删除，再从缓存中移除对象。

线程池集成：使用ThreadPool管理任务的调度和执行，支持高并发的操作请求。

* 1. **封装为SO库**

为了实现模块化和灵活性，将存储引擎（StorageEngine）封装为动态链接库（Shared Object Library，SO库）。

1. **定义导出宏：**根据操作系统平台选择不同的导出符号。

#ifdef \_WIN32

#ifdef BUILD\_DLL

#define EXPORT \_\_declspec(dllexport)

#else

#define EXPORT \_\_declspec(dllimport)

#endif

#else

#define EXPORT

#endif

1. **生成动态库：**在**Makefile**中定义了动态库的生成规则，通过 **-shared** 选项生成共享库文件**libstorage\_engine.so**

*# 生成动态库*

LIB\_SO = $(LIB\_DIR)/libstorage\_engine.so

$(LIB\_SO): $(OBJ)

    mkdir -p $(LIB\_DIR)

    $(CXX) -shared -o $@ $^ $(LDFLAGS)

1. **动态库的使用**

在其他程序中使用该动态库时，用户首先在程序中包含动态库的头文件**engine.h**，再通过**-L**指定动态库的路径、**-l**选择要链接的动态库名称**libstorage\_engine.so**即可使用。

g++ - o test\_program test\_program.cpp - L /path/to/library - lstorage\_engine

1. **单元测试**
   1. **测试环境**

同4.1开发环境。本项目采用 **Google Test** 框架对存储引擎的各项功能进行单元测试，涵盖了基本操作、缓存机制、异步操作、垃圾回收以及并发处理等方面。

单元测试代码见tests/engine\_test.cpp和tests/test\_main.cpp。终端运行下面命令来进行单元测试。

make test

* 1. **测试用例**
     1. **基本同步PUT/GET测试**

**测试逻辑：**向存储引擎中插入一个键值对（PUT 操作）。通过 GET 操作检索该键，验证返回值是否正确。尝试更新已存在键的值（PUT 操作），验证是否允许更新。再次通过 GET 操作检索，验证更新后的值是否正确。  
**测试代码：**

*// 基本同步PUT/GET测试*

TEST\_F(EngineTest, BasicPutGet)

{

    StorageEngine engine(TEST\_DB\_FILE, 4, 100, 8);

    bool result = engine.put(1, "hello");

    EXPECT\_TRUE(result);

    std::string val = engine.get(1);

    EXPECT\_EQ(val, "hello");

*// 更新同一键的值*

    bool update\_result = engine.put(1, "world");

    EXPECT\_TRUE(update\_result);

*// GET验证更新后的值*

    std::string updated\_val = engine.get(1);

    EXPECT\_EQ(updated\_val, "world");

}

* + 1. **测试DELETE**

**测试逻辑：**向存储引擎中插入一个键值对（PUT 操作）。通过 GET 操作检索该键，验证返回值是否正确。执行 DEL 操作删除该键。再次通过 GET 操作检索该键，验证是否已被删除。

**测试代码：**

*// 测试DELETE*

TEST\_F(EngineTest, BasicDelete)

{

    StorageEngine engine(TEST\_DB\_FILE, 4, 100, 8);

    engine.put(2, "test");

    std::string val = engine.get(2);

    EXPECT\_EQ(val, "test");

    bool del\_res = engine.del(2);

    EXPECT\_TRUE(del\_res);

    val = engine.get(2);

    EXPECT\_TRUE(val.empty()); *// 已被删除，应无数据*

}

* + 1. **测试缓存命中**

**测试逻辑：**插入一个键值对并进行一次 GET 操作，确保数据被缓存。

记录文件存储的读取计数。再次进行 GET 操作，验证文件存储的读取计数未增加，表示缓存命中。

**测试代码：**

*// 测试缓存命中*

TEST\_F(EngineTest, CacheHit)

{

    StorageEngine engine(TEST\_DB\_FILE, 4, 16, 4);

*// 首次put并get，必然访问文件*

    engine.put(100, "cache\_value");

    std::string val1 = engine.get(100);

    EXPECT\_EQ(val1, "cache\_value");

*// 获取FileStore读计数*

    size\_t reads\_after\_first\_get = engine.getFileStoreReadCount();

*// 再次get同一key，如果缓存正常，FileStore读计数不会改变*

    std::string val2 = engine.get(100);

    EXPECT\_EQ(val2, "cache\_value");

    size\_t reads\_after\_second\_get = engine.getFileStoreReadCount();

    EXPECT\_EQ(reads\_after\_first\_get, reads\_after\_second\_get);

}

* + 1. **LRU缓存淘汰行为测试**

**测试逻辑：**插入多个键值对，填满缓存。访问部分键，改变其使用顺序。

插入新的键值对，触发缓存淘汰。验证被淘汰的键无法从缓存中获取，需要从文件存储中读取。

**测试代码：**

*// LRU缓存淘汰行为测试*

TEST\_F(EngineTest, LRUBehavior)

{

*// 线程池大小4，缓存容量3，缓存段数1*

    StorageEngine engine(TEST\_DB\_FILE, 4, 3, 1);

*// 初始读取计数*

    size\_t initial\_reads = engine.getFileStoreReadCount();

    EXPECT\_EQ(initial\_reads, 0);

*// 插入3个键值对*

    engine.put(1, "value1");

    engine.put(2, "value2");

    engine.put(3, "value3");

*// 这三个put操作应将数据写入缓存，不应触发FileStore::get*

    EXPECT\_EQ(engine.getFileStoreReadCount(), initial\_reads);

*// 获取所有3个键，应该命中缓存，不触发FileStore::get {3,2,1}*

    EXPECT\_EQ(engine.get(1), "value1");

    EXPECT\_EQ(engine.get(2), "value2");

    EXPECT\_EQ(engine.get(3), "value3");

*// 读取计数应保持不变*

    EXPECT\_EQ(engine.getFileStoreReadCount(), initial\_reads);

*// 访问键1，使其成为最近使用 {1,3,2}*

    EXPECT\_EQ(engine.get(1), "value1");

*// 读取计数仍应保持不变*

    EXPECT\_EQ(engine.getFileStoreReadCount(), initial\_reads);

*// 插入第4个键，缓存容量为3，应淘汰最久未使用的键2 {4,1,3}*

    engine.put(4, "value4");

*// 获取键2，应触发FileStore::get，因为键2被淘汰 {2,4,1}*

    EXPECT\_EQ(engine.get(2), "value2");

*// 读取计数应增加1*

    EXPECT\_EQ(engine.getFileStoreReadCount(), initial\_reads + 1);

*// 再次获取键2，应命中缓存，不触发FileStore::get*

    EXPECT\_EQ(engine.get(2), "value2");

*// 读取计数应保持不变*

    EXPECT\_EQ(engine.getFileStoreReadCount(), initial\_reads + 1);

*// 获取键3和键4，键4应命中缓存，键3应触发FileStore::get {3,2,4}*

    EXPECT\_EQ(engine.get(3), "value3");

    EXPECT\_EQ(engine.get(4), "value4");

*// 读取计数应增加1*

    EXPECT\_EQ(engine.getFileStoreReadCount(), initial\_reads + 2);

}

* + 1. **测试异步操作（asyncPut/asyncGet/asyncDel）**

**测试逻辑：**使用异步接口执行PUT操作，验证回调函数接收到成功信号。使用异步接口执行GET操作，验证回调函数接收到正确的值。使用异步接口执行DEL操作，验证回调函数接收到成功信号。最终通过同步GET操作验证删除结果。

**测试代码：**

*/ 测试异步操作（asyncPut/asyncGet/asyncDel）*

TEST\_F(EngineTest, AsyncOperations)

{

    StorageEngine engine(TEST\_DB\_FILE, 4, 100, 8);

    std::atomic<bool> put\_done{false};

    std::atomic<bool> get\_done{false};

    std::atomic<bool> del\_done{false};

    engine.asyncPut(10, "async\_val", [&put\_done](bool res)

                    {

        EXPECT\_TRUE(res);

        put\_done = true; });

*// 等待put完成*

    while (!put\_done.load())

    {

        std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

    }

    engine.asyncGet(10, [&get\_done](std::string val)

                    {

        EXPECT\_EQ(val, "async\_val");

        get\_done = true; });

    while (!get\_done.load())

    {

        std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

    }

    engine.asyncDel(10, [&del\_done](bool res)

                    {

        EXPECT\_TRUE(res);

        del\_done = true; });

    while (!del\_done.load())

    {

        std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

    }

*// 最终检查是否删除成功*

    auto final\_val = engine.get(10);

    EXPECT\_TRUE(final\_val.empty());

}

* + 1. **测试GC逻辑**

**测试逻辑：**插入多个键值对。删除部分键值对。触发垃圾回收。验证未删除的键值对仍可访问，删除的键值对无法访问。

**测试代码：**

*// 测试GC逻辑*

TEST\_F(EngineTest, GarbageCollectTest)

{

    StorageEngine engine(TEST\_DB\_FILE, 4, 100, 8);

*// 插入多条记录*

    for (int i = 0; i < 20; ++i)

    {

        engine.put(i, "value\_" + std::to\_string(i));

    }

*// 删除部分记录*

    for (int i = 0; i < 10; ++i)

    {

        engine.del(i);

    }

*// 调用GC*

    engine.garbageCollect();

*// 验证剩余记录*

    for (int i = 10; i < 20; ++i)

    {

        std::string val = engine.get(i);

        EXPECT\_EQ(val, "value\_" + std::to\_string(i));

    }

*// 删除的应为空*

    for (int i = 0; i < 10; ++i)

    {

        std::string val = engine.get(i);

        EXPECT\_TRUE(val.empty());

    }

}

* + 1. **简单并发测试**

**测试逻辑：**使用多个线程异步执行大量PUT操作，插入多个键值对。使用多个线程异步执行大量GET操作，检索插入的键值对。验证所有PUT和GET操作均成功且数据一致。

**测试代码：**

*// 简单并发测试*

TEST\_F(EngineTest, ConcurrentAccess)

{

    StorageEngine engine(TEST\_DB\_FILE, 8, 100, 16); *// 增加缓存段数*

    const int N = 1000;

    std::atomic<int> completed\_puts{0};

    std::atomic<int> completed\_gets{0};

*// 多线程写入*

    for (int i = 0; i < N; ++i)

    {

        engine.asyncPut(i, "val\_" + std::to\_string(i), [&completed\_puts](bool res){

            EXPECT\_TRUE(res);

            completed\_puts++; });

    }

*// 等待写入全部完成*

    while (completed\_puts.load() < N)

    {

        std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

    }

*// 多线程异步读*

    for (int i = 0; i < N; ++i)

    {

        engine.asyncGet(i, [&completed\_gets, i](std::string val)

                        {

            EXPECT\_EQ(val, "val\_" + std::to\_string(i));

            completed\_gets++; });

    }

    while (completed\_gets.load() < N)

    {

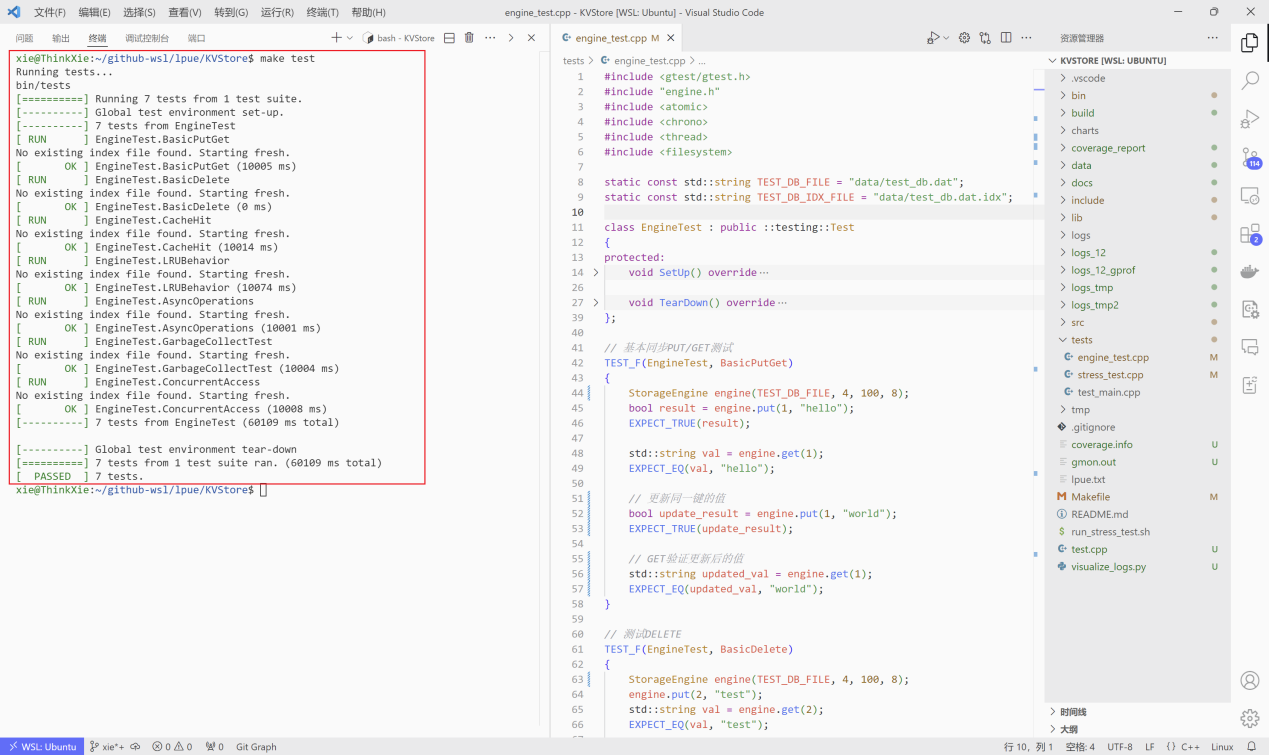
        std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

    }

}

* 1. **测试结果**

所有单元测试均已通过，程序运行结果如下。大部分单元测试耗时超过10秒，这是因为垃圾回收线程设置为10秒，程序在垃圾回收线程退出后才结束。



* 1. **代码覆盖率分析**

采用了代码覆盖率分析工具 **gcov** 和 **lcov**分析测试用例覆盖了代码中的哪些部分以评估单元测试的有效性。

1. **工具安装**

sudo apt-get update

sudo apt-get install lcov

1. **项目配置**

在Makefile中添加覆盖率标志 --coverage：

CXXFLAGS = -std=c++20 -Wall -Wextra -O2 -g --coverage -fPIC -pg -I$(INCLUDE\_DIR) -I/usr/local/include

LDFLAGS = -L/usr/local/lib -lpthread -lgtest -lgtest\_main -pg --coverage

重新编译并运行单元测试：

make clean

make

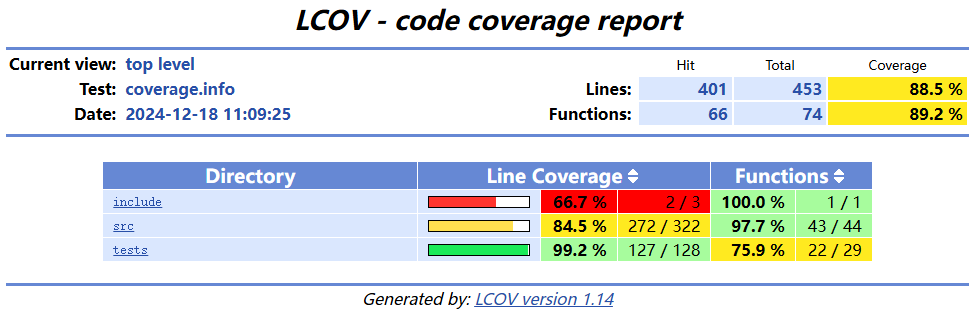
./bin/tests

1. **报告生成**

lcov --capture --directory . --output-file coverage.info --exclude '/usr/\*'

genhtml coverage.info --output-directory coverage\_report --ignore-errors source

在coverage\_report目录下的index.html里可以看到可视化的分析结果，总体覆盖率情况如下，可看出src目录里的**行覆盖率**为84.5%，**函数覆盖率**为97.7%，主要模块和功能均已被充分测试。



1. **压力测试**
   1. **测试环境**

同4.1开发环境。本项目采用**gprof**分析压力测试下程序的性能，用**vmstat**、**iostat**、**pidstat监控系统的资源，**记录CPU、内存、磁盘IO等性能指标。

* 1. **测试方法**

初始阶段插入至少1千万个键值对，确保存储引擎能处理大规模数据，再使用多个线程（16线程）同时执行大量的PUT、GET、DEL操作，并每隔2小时进行一次垃圾回收，测试运行时间为12小时，评估系统的长期稳定性。在测试过程中和结束后，进行抽样检查，确保存储引擎的数据一致性。同时脚本会记录运行时的系统资源使用情况和程序性能。

测试参数如下：

*// 配置参数*

static const int NUM\_THREADS = 16; *// 并发线程数*

static const size\_t KEY\_RANGE = 20000000; *// 键空间足够大，减少热点*

static const size\_t INITIAL\_PUT\_COUNT = 10000000; *// 至少插入1千万*

static const auto MAX\_DURATION = std::chrono::hours(12); *// 运行12小时*

static const std::string TEST\_DB\_FILE = "data/long\_stress\_test\_db.dat";

测试脚本见run\_stress\_test.sh，测试代码见tests/stress\_test.cpp，生成日志文件存储在logs目录中，用log\_analysis.py脚本分析日志生成的图表存储在charts目录中，用gprof分析gmont.out生成的分析报告见docs/gprof\_analysis\_report.txt。

终端运行下面的命令来进行压力测试：

make stress

终端运行下面的命令来进行压力测试并在logs目录下生成日志文件：

 ./run\_stress\_test.sh

终端运行下面的命令对生成的日志文件进行分析并生成可视化的图表：

python3 log\_analysis.py

终端运行下面的命令使用gprof生成性能报告：

gprof ./bin/stress\_test gmon.out > docs/gprof\_analysis\_report.txt

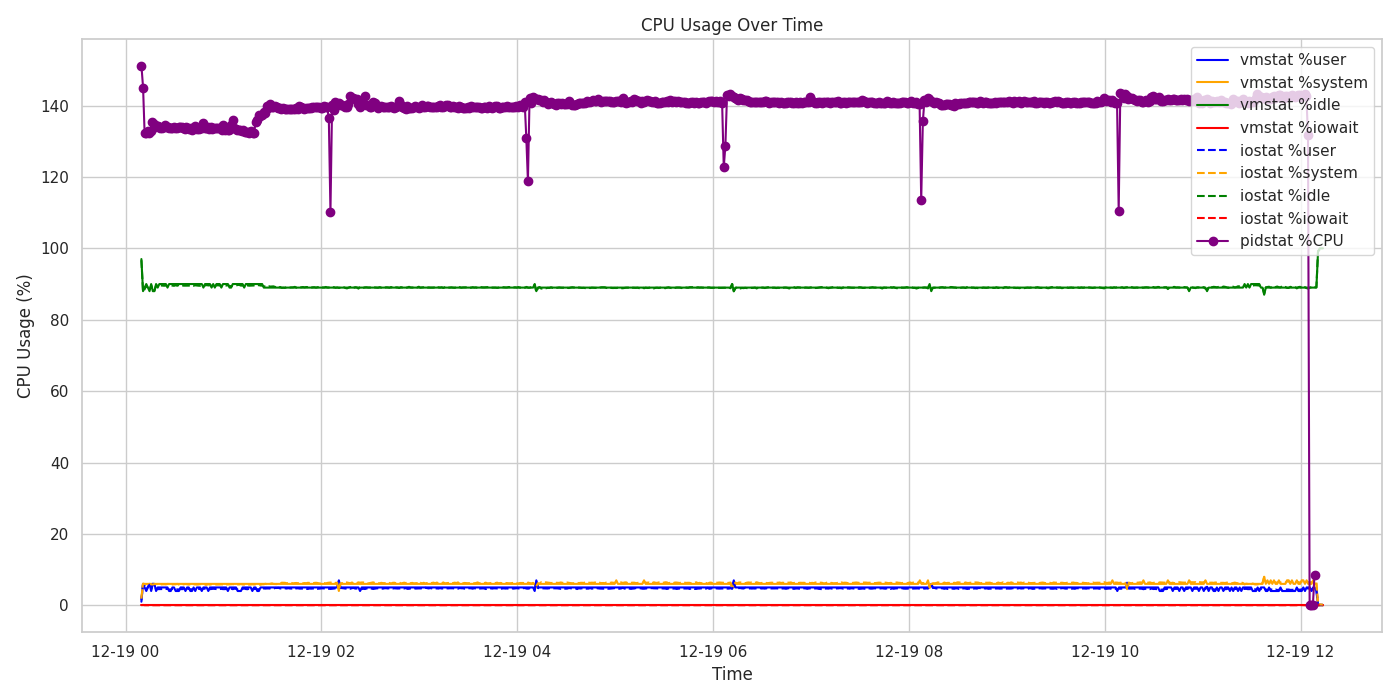
终端运行下面的命令将gprof输出转换为DOT文件后再转为图像：

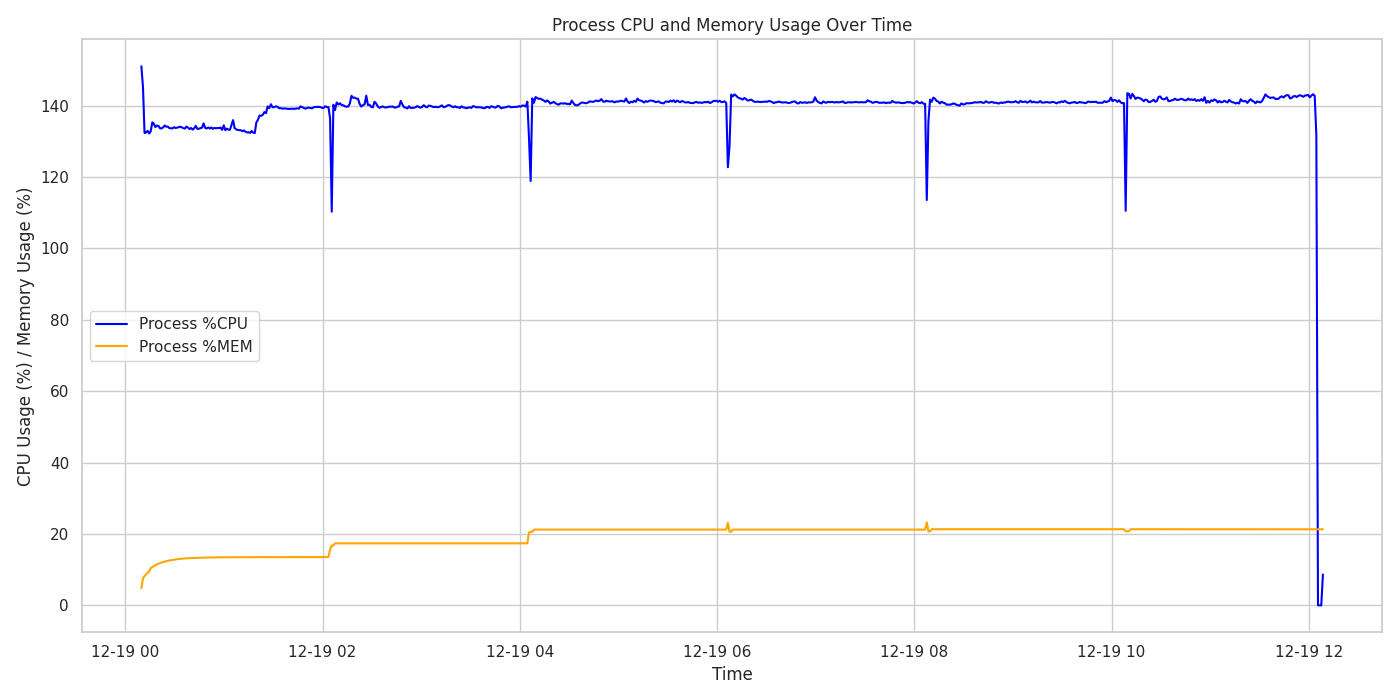
gprof2dot -f prof docs/gprof\_analysis\_report.txt -o docs/gprof\_call\_graph.dot

dot -Tpng docs/gprof\_call\_graph.dot -o charts/gprof\_call\_graph.png

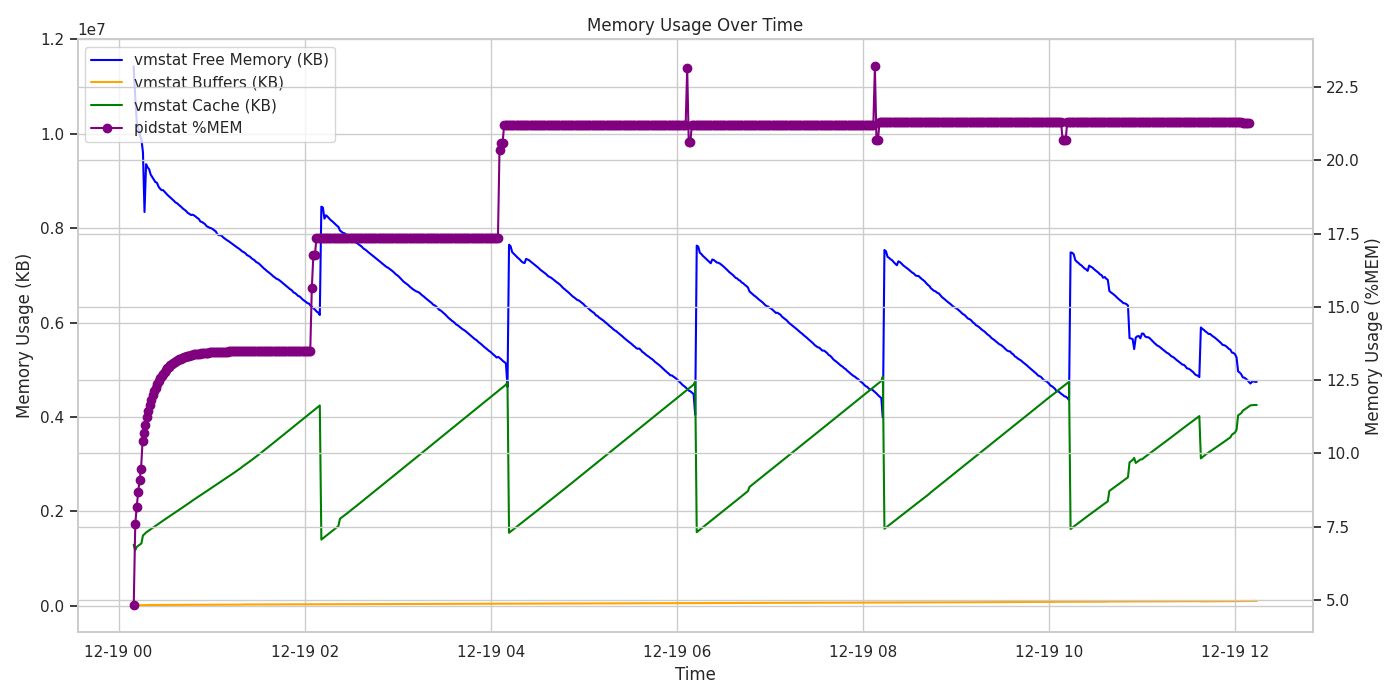
* 1. **测试结果分析**

1. **CPU分析：**从%idle数据来看CPU负载不大，存在大量空闲资源；%iowait较低，说明磁盘IO不是性能的瓶颈；pidstat %CPU数据表示stress\_test进程的CPU使用情况，大概在130%左右，因为该系统CPU总共有20个核心，所以负载并不大；同时注意到因为系统每隔两个小时就会触发垃圾回收线程，所以在对应的时间里CPU使用率会大幅降低。

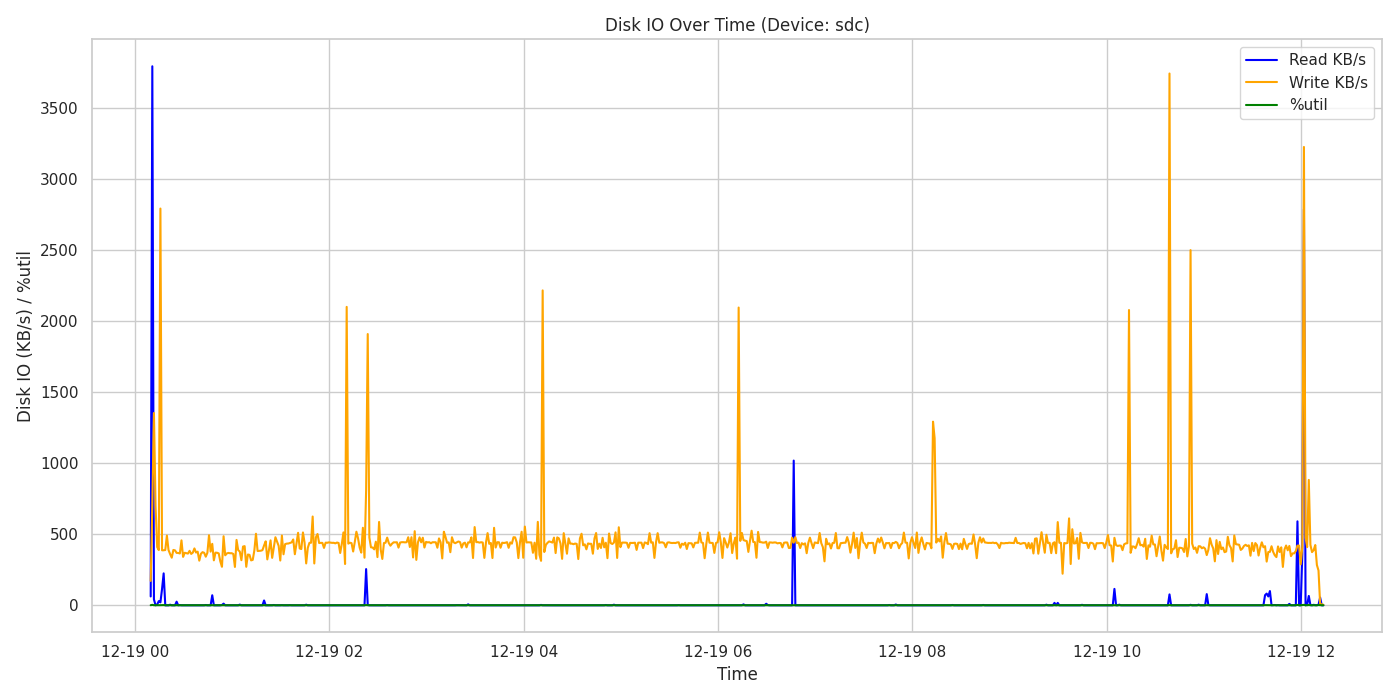
****

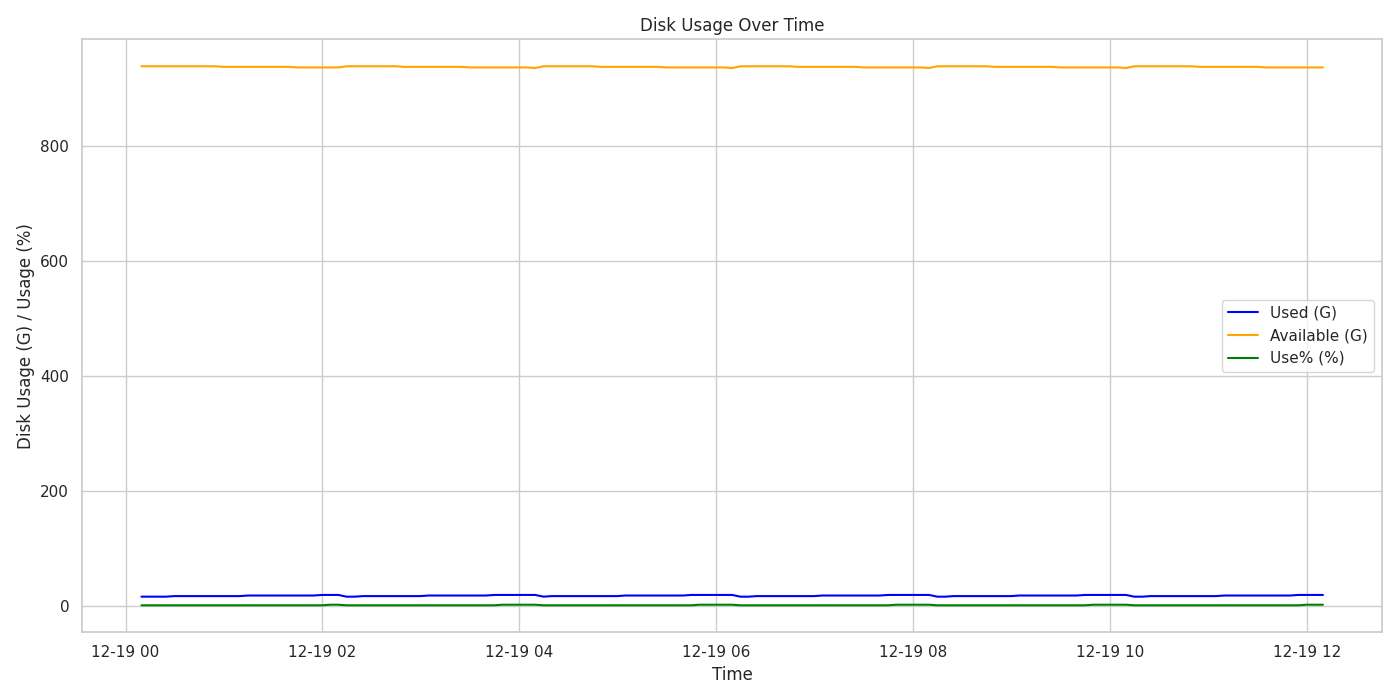
****

1. **内存分析：**在程序初始时先插入一千万数据，所以vmstat Free Memory数据逐渐降低，vmstat Cache数据逐渐升高；同时因为每两个小时会触发垃圾回收线程，两者的使用情况周期性的此消彼长；因为系统内存足够，所以vmstat Buffer数据稳定偏低；从pidstat%MEM数据总体来看随着时间推移，内存使用量在逐步增加至趋于稳定，中间的波动是受到了垃圾回收线程的影响；

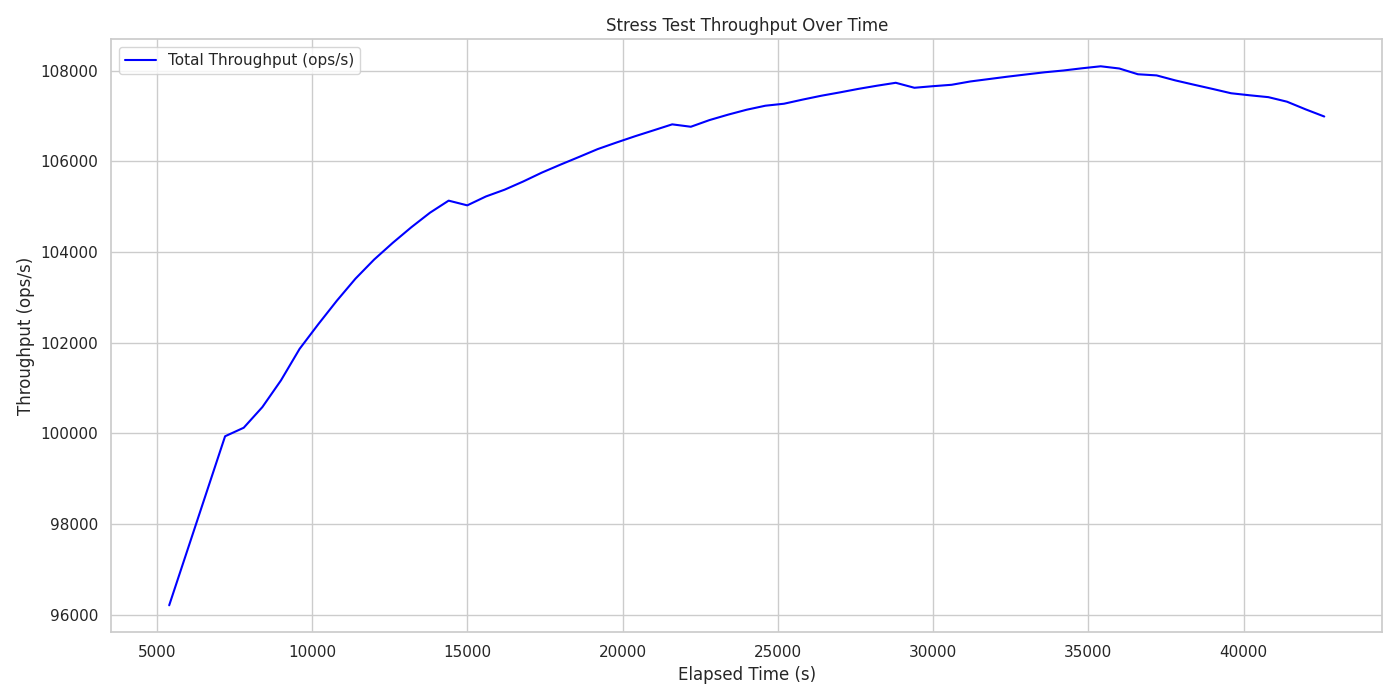
****

1. **磁盘分析：**通过程序运行时磁盘的使用情况表可以看出磁盘的剩余空间充足，读取和写入数据性能良好，同时在触发垃圾回收线程时因为要创建新的索引文件写入量会大幅增加；

****

****

1. **操作结果分析：**
   1. 吞吐量逐渐上升直到维持在约10万 ops/s左右，峰值108k ops/s左右，整体保持较为稳定，没有明显的持续下滑趋势。在整个12个小时测试结束时，总共完成了46.15亿次操作，平均吞吐量约为106830 ops/s。说明系统在持续高并发写读删操作的压力下性能已相当可观。
   2. Put操作无失败，每次插入和更新操作都成功写入了；Get操作有少许失败，但相对总的操作次数来说失败率可以忽略不计；Del操作失败率偏高，推测原因是在大范围的key空间中随机选择key进行操作，许多key可能不存在或已被删除；在最后对数据中的100个样本key进行一致性校验通过，说明缓存、存储和索引在压力测试下的数据是正确的一致的。

****

logs/stress\_test\_20241219\_000943.log：

No existing index file found. Starting fresh.

Stress Test Results:

Total operations: 4615045952

Put Success: 1545240944

Put Fail: 0

Get Success: 1534786645

Get Fail: 110915

Delete Success: 765748415

Delete Fail: 769159033

Elapsed Time: 43200 seconds

Throughput: 106830 ops/sec

Data consistency check (sample) passed.

1. **Gprof文件分析：**
   1. **平面图分析：**

**frame\_dummy (62.28%)**：通常由编译器生成，用于初始化全局和静态对象。在测试代码中，reference\_map 是一个全局 std::unordered\_map，导致大量初始化调用；访问全局的reference\_map 受互斥锁保护，多个线程同时访问时可能导致锁竞争，影响性能；

**operation\_thread(StorageEngine&) (22.56%)：**该函数负责执行PUT、GET、DEL操作，在高并发环境下频繁调用，导致其占用了较高的运行时间；

**std::mersenne\_twister\_engine<...>::\_M\_gen\_rand() (3.64%)：**该函数用于生成随机数，频繁调用导致其占用了显著的运行时间；

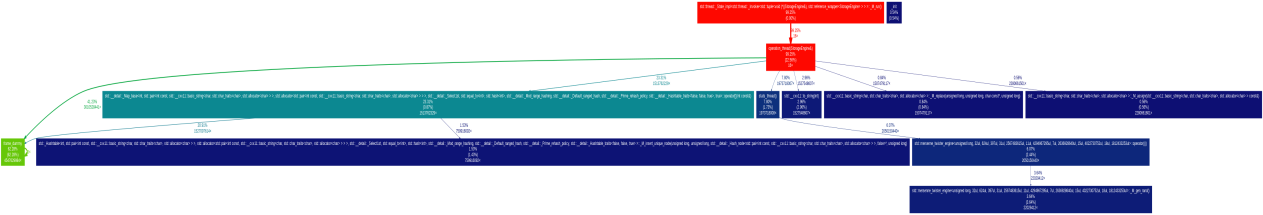
**std::\_\_cxx11::to\_string(int) (2.96%)：**将整数转换为字符串，用于生成操作值，频繁调用导致较高的运行时间。

**stats\_thread() (1.73%)：**负责定期输出操作统计信息，频繁访问和锁操作可能导致一定的运行时间。

**std::\_Hashtable<...>::\_M\_insert\_unique\_node(...) (1.43%)：**在 reference\_map 中插入键值对时调用，对哈希表进行频繁插入操作导致较高的运行时间。



* 1. **调用图**



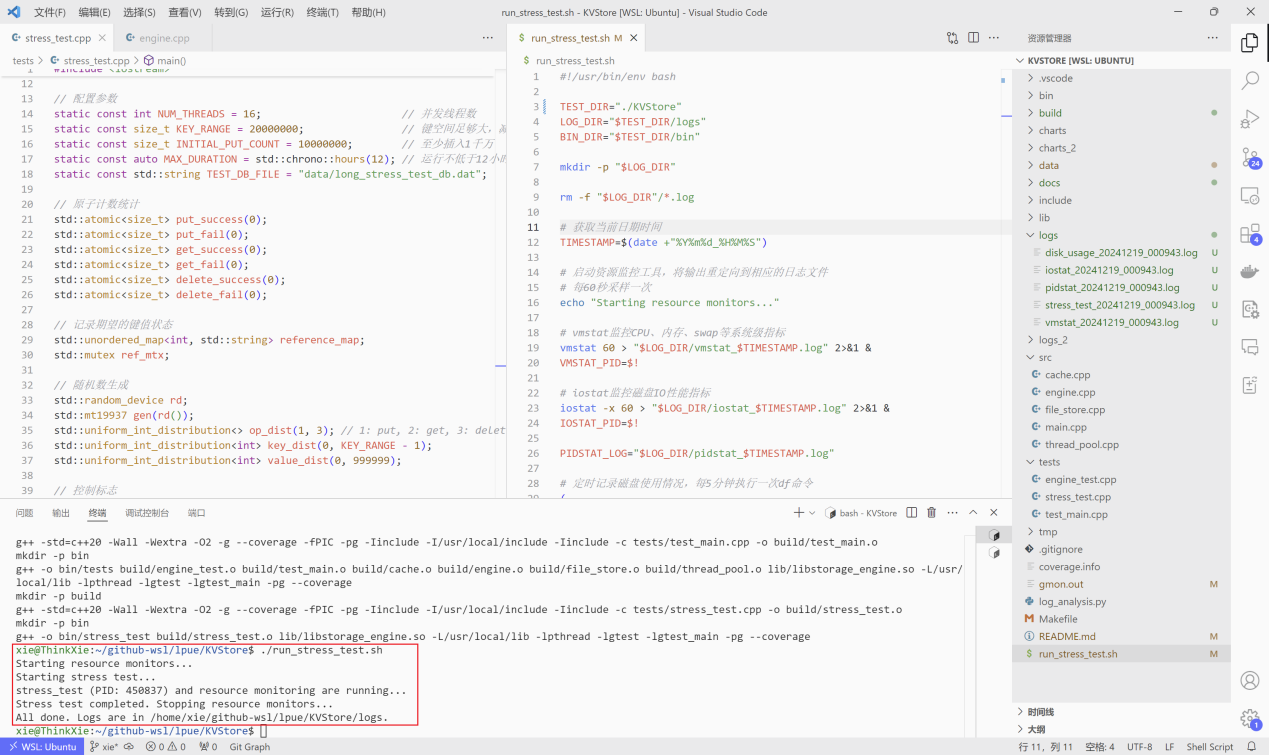
1. **总结**

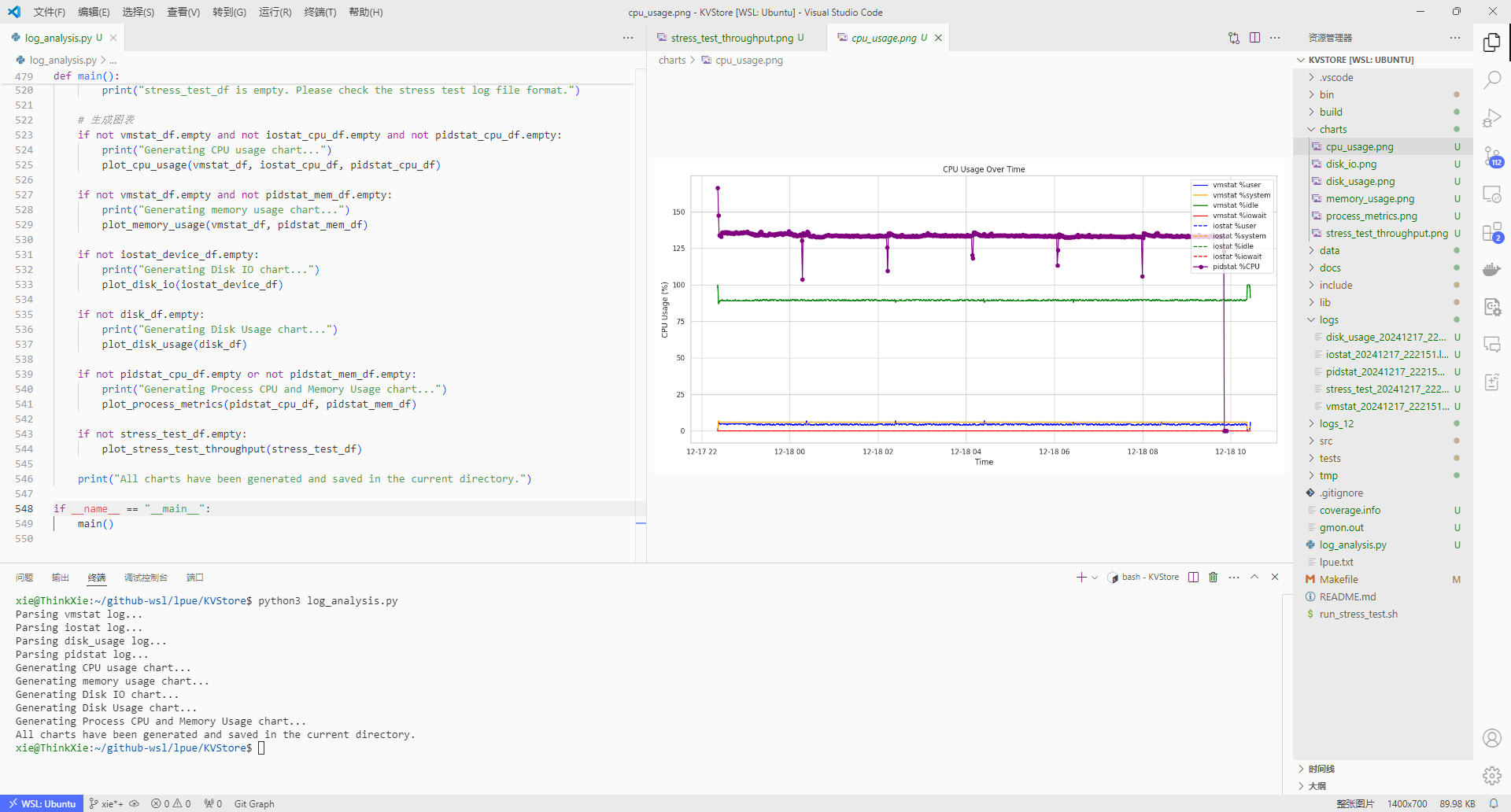
报告全面介绍了一个轻量级的KV存储引擎的架构、核心数据结构的设计、对它进行GET\PUT\DEL操作的内部逻辑、各模块之间的同步机制。并对该存储引擎的各部分功能进行了单元测试，也在高并发、大规模数据操作环境下进行了压力测试并用丰富的工具分析其性能和其运行时对系统资源的占用情况，对其的性能和稳定性进行了评估。

1. **附录**
   1. **源码链接**

Github仓库见 [rainy-xie/KVStorageEngine (github.com)](https://github.com/rainy-xie/KVStorageEngine)

* 1. **运行时截图**





* 1. **日志文件样例**

**logs\_2/disk\_usage\_20241216\_092654.log**

Filesystem      Size  Used Avail Use% Mounted on

/dev/sdc       1007G   17G  940G   2% /

Filesystem      Size  Used Avail Use% Mounted on

/dev/sdc       1007G   17G  940G   2% /

Filesystem      Size  Used Avail Use% Mounted on

/dev/sdc       1007G   17G  939G   2% /

Filesystem      Size  Used Avail Use% Mounted on

/dev/sdc       1007G   18G  939G   2% /

**logs\_2/iostat\_20241216\_092654.log**

Linux 5.15.167.4-microsoft-standard-WSL2 (ThinkXie)     12/16/24    \_x86\_64\_    (20 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle

           2.86    0.00    2.02    0.10    0.00   95.02

Device            r/s     rkB/s   rrqm/s  %rrqm r\_await rareq-sz     w/s     wkB/s   wrqm/s  %wrqm w\_await wareq-sz     d/s     dkB/s   drqm/s  %drqm d\_await dareq-sz     f/s f\_await  aqu-sz  %util

sda              1.25     84.78     0.43  25.56    0.29    67.57    0.00      0.00     0.00   0.00    0.00     0.00    0.00      0.00     0.00   0.00    0.00     0.00    0.00    0.00    0.00   0.08

sdb              0.11      2.40     0.00   0.00    0.12    21.84    0.00      0.00     0.00   0.00    6.00     2.00    0.00      0.00     0.00   0.00    0.00     0.00    0.00    8.00    0.00   0.01

sdc             30.27   1146.42    11.34  27.26    0.40    37.87    8.45    892.28    11.99  58.66    8.23   105.63    2.42  97050.82     1.60  39.71    0.17 40067.02    1.01    2.42    0.08   2.52

**logs\_2/pidstat\_20241216\_092654.log**

Linux 5.15.167.4-microsoft-standard-WSL2 (ThinkXie)     12/16/24    \_x86\_64\_    (20 CPU)

*09:26:59*      UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command

*09:27:59*     1000      6425   46.65  131.00    0.00    0.00  177.65     5  stress\_test

*09:26:59*      UID       PID  minflt/s  majflt/s     VSZ     RSS   %MEM  Command

*09:27:59*     1000      6425   2480.28      0.00 1426640  761712   4.69  stress\_test

**logs\_2/stress\_test\_20241216\_092654.log**

No existing index file found. Starting fresh.

[Stats] Elapsed: 600s | Total Ops: 54753637 | Put(S/F): 24917090/0 | Get(S/F): 14919030/25 | Del(S/F): 6624358/8293141 | Throughput: 91256.1 ops/s

[Stats] Elapsed: 1200s | Total Ops: 98702220 | Put(S/F): 39559221/0 | Get(S/F): 29570888/44 | Del(S/F): 13755313/15816755 | Throughput: 82251.9 ops/s

[Stats] Elapsed: 1800s | Total Ops: 142409265 | Put(S/F): 54130599/0 | Get(S/F): 44142058/65 | Del(S/F): 20987220/23149325 | Throughput: 79116.3 ops/s

[Stats] Elapsed: 41400s | Total Ops: 4440651373 | Put(S/F): 1486910141/0 | Get(S/F): 1476833698/2758 | Del(S/F): 736539760/740365016 | Throughput: 107262 ops/s

[Stats] Elapsed: 42000s | Total Ops: 4507142277 | Put(S/F): 1509074629/0 | Get(S/F): 1498995712/2805 | Del(S/F): 747610537/751458598 | Throughput: 107313 ops/s

[Stats] Elapsed: 42600s | Total Ops: 4576120278 | Put(S/F): 1532071921/0 | Get(S/F): 1521984716/2846 | Del(S/F): 759097661/762963135 | Throughput: 107421 ops/s

No existing index file found. Starting fresh.

Stress Test Results:

Total operations: 4635950938

Put Success: 1552016727

Put Fail: 0

Get Success: 1541930330

Get Fail: 2888

Delete Success: 769057372

Delete Fail: 772943621

Elapsed Time: 43200 seconds

Throughput: 107314 ops/sec

Data consistency check (sample) passed.

**logs\_2/vmstat\_20241216\_092654.log**

procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----

 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st

 6  0      0 11148032   9756 1063540    0    0    62    45  885 1847  3  2 95  0  0

 5  0      0 10241140   9904 1171868    0    0     1  1272 103874 218080 14 12 74  0  0

 4  0      0 9420168  10028 1265084    0    0     0  1630 98091 206151 13 12 75  0  0

 4  0      0 9327160  10136 1309480    0    0     0   969 102237 218751 11 12 77  0  0

 7  0      0 9138064  10232 1356436    0    0     0   868 109808 235738 11 12 77  0  0

 4  0      0 8977480  10328 1404424    0    0     0   718 115261 247466 11 12 77  0  0

 4  0      0 8818020  10428 1453116    0    0     0   736 118459 254629 11 12 77  0  0

 5  0      0 8574244  10524 1501676    0    0     0   798 118109 254384 11 12 77  0  0

 6  0      0 8433888  10752 1550308    0    0     2   811 118196 255173 11 12 77  0  0

 4  0      0 8306368  10848 1595652    0    0     0   770 111579 240888 11 12 77  0  0

 4  0      0 8196000  10944 1640248    0    0     0   724 110802 239219 11 12 77  0  0

 6  0      0 8096448  11044 1684840    0    0     0   865 112381 242524 11 12 77  0  0

 5  0      0 7994380  11140 1728844    0    0     0   688 110280 238245 11 12 77  0  0

 3  0      0 7908088  11244 1767652    0    0     0   629 98658 211036 10 13 78  0  0

 4  0      0 7854796  11344 1806500    0    0     1   654 98690 210832 10 13 77  0  0

 6  0      0 7775932  11440 1843680    0    0     0   608 94958 202606  9 13 78  0  0

 4  0      0 7703648  11536 1882620    0    0     0   650 100934 215350  9 13 78  0  0

 4  0      0 7636200  11636 1922664    0    0     0   655 102785 219424  9 13 78  0  0

 6  0      0 7570256  11740 1961044    0    0     0   639 99207 211717  9 13 78  0  0

 5  0      0 7512036  11844 1996880    0    0     0   593 91673 195442  9 13 77  0  0

 7  0      0 7446232  11940 2032408    0    0     0   586 91181 194482  9 13 78  0  0

* 1. **压力测试脚本文件**

*#!/usr/bin/env bash*

TEST\_DIR="./KVStore"

LOG\_DIR="$TEST\_DIR/logs"

BIN\_DIR="$TEST\_DIR/bin"

mkdir -p "$LOG\_DIR"

rm -f "$LOG\_DIR"/\*.log

*# 获取当前日期时间*

TIMESTAMP=$(date +"%Y%m%d\_%H%M%S")

*# 启动资源监控工具，将输出重定向到相应的日志文件*

*# 每60秒采样一次*

echo "Starting resource monitors..."

*# vmstat监控CPU、内存、swap等系统级指标*

vmstat 60 > "$LOG\_DIR/vmstat\_$TIMESTAMP.log" 2>&1 &

VMSTAT\_PID=$!

*# iostat监控磁盘IO性能指标*

iostat -x 60 > "$LOG\_DIR/iostat\_$TIMESTAMP.log" 2>&1 &

IOSTAT\_PID=$!

PIDSTAT\_LOG="$LOG\_DIR/pidstat\_$TIMESTAMP.log"

*# 定时记录磁盘使用情况，每5分钟执行一次df命令*

(

    while true; do

        df -h "$TEST\_DIR"

        sleep 300

    done

) > "$LOG\_DIR/disk\_usage\_$TIMESTAMP.log" 2>&1 &

DISK\_USAGE\_PID=$!

echo "Starting stress test..."

*# 启动压力测试程序并在后台运行*

$BIN\_DIR/stress\_test > "$LOG\_DIR/stress\_test\_$TIMESTAMP.log" 2>&1 &

STRESS\_PID=$!

*# 等待几秒钟以确保stress\_test启动成功，然后使用pidstat监控此进程*

sleep 5

pidstat -r -u 60 -p $STRESS\_PID > "$PIDSTAT\_LOG" 2>&1 &

PIDSTAT\_PID=$!

echo "stress\_test (PID: $STRESS\_PID) and resource monitoring are running..."

*# 等待压力测试结束*

wait $STRESS\_PID

echo "Stress test completed. Stopping resource monitors..."

*# 终止所有监控进程*

kill $VMSTAT\_PID

kill $IOSTAT\_PID

kill $PIDSTAT\_PID

kill $DISK\_USAGE\_PID

echo "All done. Logs are in $LOG\_DIR."

1. **致谢**

感谢老师布置的大作业让我收获很多。