

前言

MINI-LSM

Build a simple key-value storage engine in a week.
And extend your LSM engine in the second + third week.

本课程将教你如何使用 Rust 构建 LSM 树存储引擎。

什么是 LSM，以及为什么使用 LSM？

日志结构合并树是一种维护键值对的数据结构。这种数据结构在 TiDB 和 CockroachDB 等分布式数据库系统中被广泛用作底层存储引擎。基于 LevelDB 的 RocksDB 是 LSM 树存储引擎的一种实现。它提供了许多键值访问功能，并应用于许多生产系统。

一般来说，LSM 树是一种适合追加操作的数据结构。将 LSM 与其他键值数据结构（如 RB 树和 B 树）进行比较更为直观。对于 RB 树和 B 树，所有数据操作都是原地进行的。也就是说，当你想要更新键对应的值时，引擎会用新值覆盖其原始内存或磁盘空间。但在 LSM 树中，所有写入操作，即插入、更新、删除，都是延迟应用于存储的。引擎将这些操作批处理成 SST（排序字符串表）文件，并写入磁盘。一旦写入磁盘，引擎不会直接修改它们。在特定的后台任务——压缩中，引擎会合并这些文件以应用更新和删除。

这种架构设计使 LSM 树易于使用。

1. 数据在持久存储上是不可变的。并发控制更为直接。将后台任务（压缩）卸载到远程服务器是

可能的。直接从 S3 等云原生存储系统存储和提供数据也是可行的。

2. 更改合并算法允许存储引擎在读取、写入和空间放大之间进行平衡。数据结构是通用的，通过调整合并参数，我们可以针对不同的工作负载优化LSM结构。

本课程将教你如何使用Rust编程语言构建基于LSM树的存储引擎。

先决条件

- 你应该了解Rust编程语言的基础知识。阅读Rust书籍就足够了。
- 你应该了解键值存储引擎的基本概念，即为什么我们需要复杂的架构来实现持久性。如果你之前没有数据库系统和存储系统的经验，你可以在PingCAP人才计划中实现Bitcask。
- 了解LSM树的基础知识不是必需的，但我们建议你阅读一些相关内容，例如LevelDB的整体思想。提前了解这些知识将帮助你熟悉可变内存表和不可变内存表、SST、合并、WAL等概念。

你从这个课程中应该期待什么

完成这个课程后，你应该深入理解基于LSM的存储系统的工作原理，获得设计此类系统的实践经验，并将所学知识应用于你的学习和职业生涯。你将理解此类存储系统的设计权衡，并找到最佳方法来设计基于LSM的存储系统以满足你的工作负载需求/目标。这门非常深入的课程涵盖了现代存储系统（即RocksDB）的所有关键实现细节和设计选择，基于作者在几个类似LSM的存储系统中的经验，你将能够直接将所学知识应用于工业界和学术界。

结构

该课程是一门综合课程，包含几个部分（周）。每周有七个章节；你可以在2到3小时内完成每个章节。每个部分的头六个章节将指导你构建一个可运行的系统，而每周的最后一个章节将是一个休闲时间章节，它

实现了一些比你在过去六天内构建的更简单的事情。每个章节都将有必需的任务、检查你的理解问题和奖励任务。

测试

我们提供一套完整的测试套件和一些CLI工具，供您验证您的解决方案是否正确。请注意，测试套件并非详尽无遗，即使通过所有测试用例，您的解决方案也可能并非100%正确。在实现系统后续部分时，您可能需要修复早期的错误。我们建议您仔细思考您的实现，尤其是在存在多线程操作和竞态条件的情况下。

解决方案

我们在mini-lsm主仓库中实现了一个解决方案，该解决方案满足了课程中要求的所有功能。同时，我们还提供了一个mini-lsm解决方案检查点仓库，其中每个提交对应课程中的一个章节。

将这样的检查点仓库与mini-lsm课程保持同步具有挑战性，因为每个错误修复或新功能都必须经过所有提交（或检查点）。因此，这个仓库可能不使用最新的启动代码，也可能不包含mini-lsm课程的最新功能。

TL;DR: 我们不保证解决方案检查点仓库包含正确的解决方案、通过所有测试或具有正确的文档注释。对于正确的实现以及实现所有内容后的解决方案，请查看主仓库中的解决方案。

<https://github.com/skyzh/mini-lsm/tree/main/mini-lsm>。

如果你在课程的某个部分卡住，或者需要帮助确定功能实现的位置，你可以参考这个仓库寻求帮助。你可以比较提交之间的差异来了解发生了什么变化。在整个章节中，你可能需要多次修改mini-lsm课程中的某些函数，并且你可以在这个仓库中了解每个章节具体需要实现的内容。

你可以访问解决方案检查点仓库：<https://github.com/skyzh/mini-lsm-solution-checkpoint>。

反馈

您的反馈非常感谢。我们于2024年根据学生的反馈从头开始重写了整个课程。请分享您的学习体验，帮助我们持续改进课程。欢迎加入Discord社区并分享您的经验。

我们为什么要重写的长故事：这门课程最初计划为学生提供一个通用指南，让他们从一个空目录开始，根据我们提供的规范实现他们想做的事情。我们只有很少的测试来检查行为是否正确。然而，原来的课程过于开放，给学习体验带来了巨大的障碍。由于学生在事先没有整个系统的概览，而且指示很模糊，有时他们很难知道为什么做出某个设计决策，以及他们需要实现什么目标。课程的一些部分非常紧凑，以至于在一个章节内无法传递预期的内容。因此，我们完全重新设计了课程，以更轻松的学习曲线和更清晰的学习目标。原来的为期一周的课程现在分为两周（第一周关于存储格式，第二周关于深入压缩），并增加了一个关于MVCC的部分。希望您觉得这门课程有趣且对您的研究和职业有帮助。我们想感谢在第一天编码后和Hello中评论的所有人，课程下一次更新计划是什么时候？——您的反馈极大地帮助我们改进了课程。

许可

本课程的源代码根据 Apache 2.0 许可，而书籍根据 CC BY-NC-SA 4.0 许可。

这门课程会永远免费吗？

是的！现在公开的所有内容将永远免费，并接收终身更新和错误修复。同时，我们可能会提供付费代码审查和办公时间服务。至于 DLC 部分（剩余的章节），截至 2024 年我们还没有计划完成它们，并且尚未决定是否公开。

社区

您可以加入 skyzh 的 Discord 服务器，与 mini-lsm 社区一起学习。



开始使用

现在，你可以在 Mini-LSM 课程概述中了解 LSM 结构的概览。

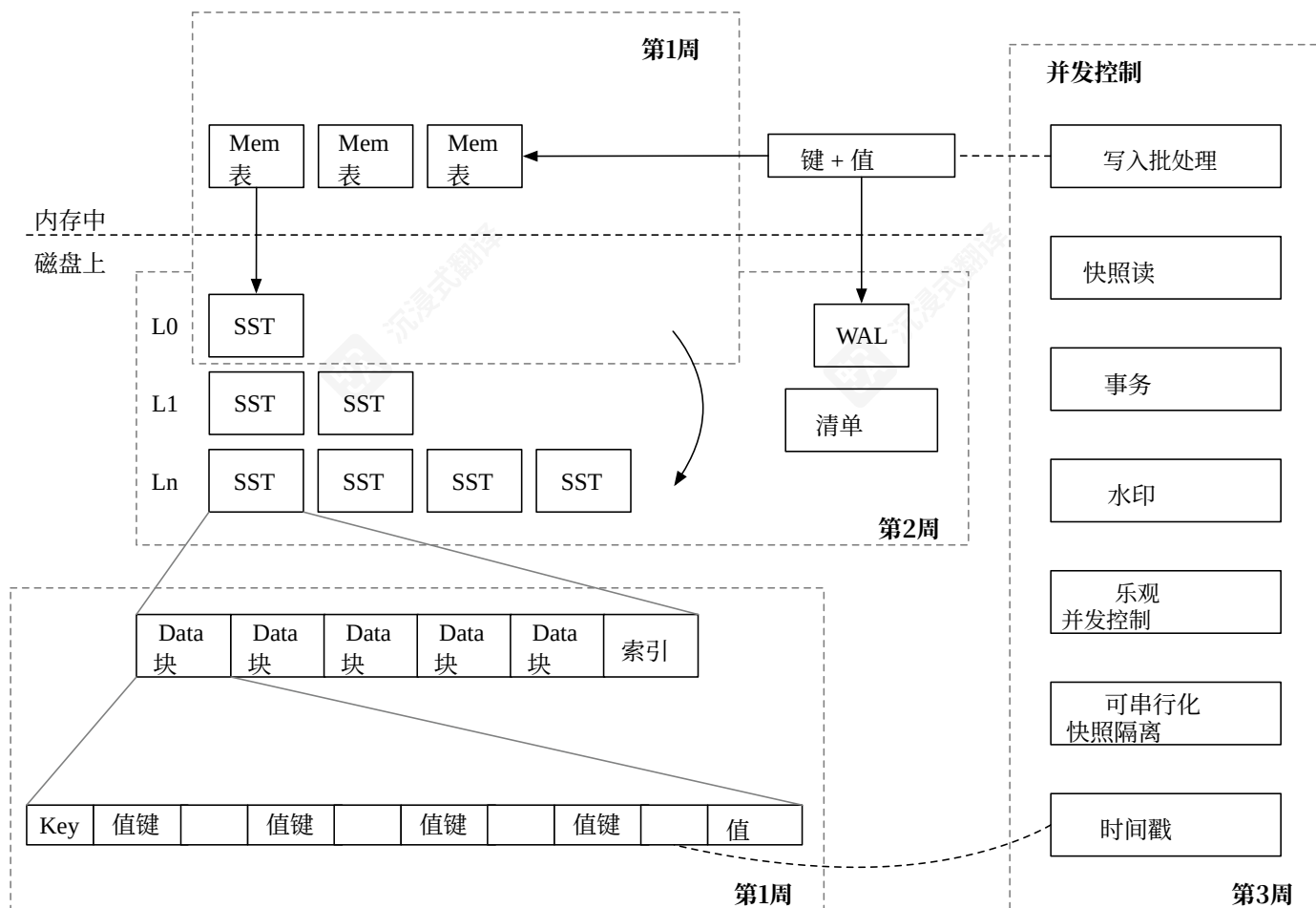
关于作者

截至撰写时（2024年初），Chi 从卡内基梅隆大学获得了计算机科学硕士学位，并从上海交通大学获得了学士学位。他一直在研究各种数据库系统，包括 TiKV、AgateDB、TerarkDB、RisingWave 和 Neon。自 2022 年以来，他作为助教在 BusTub 教育系统上为 CMU 数据库系统课程工作了三个学期，为课程添加了许多新功能和更多挑战（可以查看重新设计的查询执行项目和极具挑战性的多版本并发控制项目）。除了在 BusTub 教育系统上工作外，他还维护着 RisingLight 教育数据库系统。Chi 对探索 Rust 编程语言如何融入数据库世界感兴趣。如果你也对这个主题感兴趣，可以查看他的上一门课程：构建矢量化表达式框架 `type-exercise-in-rust` 和构建矢量数据库 `write-you-a-vector-db`。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。

Mini-LSM 课程概述

课程结构



我们三个部分（周）来完成这门课程。在第一周，我们将专注于存储结构以及LSM存储引擎的存储格式。在第二周，我们将深入探讨压缩操作并实现存储引擎的持久性支持。在第三周，我们将实现多版本并发控制。

- 第一周：Mini-LSM
- 第二周：压缩操作和持久化
- 第三周：多版本并发控制

请查看环境设置来配置环境。

LSM概述

一个LSM存储引擎通常包含三个部分：

1. 预写日志用于持久化临时数据以进行恢复。
2. 磁盘上的SSTs用于维护LSM树结构。
3. 内存中的内存表用于批处理小写入。

存储引擎通常提供以下接口：

- `Put(key, value)`：将键值对存储在LSM树中。
- `Delete(key)`：删除一个键及其对应的值。
- `Get(key)`：获取键对应的值。
- `Scan(range)`：获取一个键值对范围。

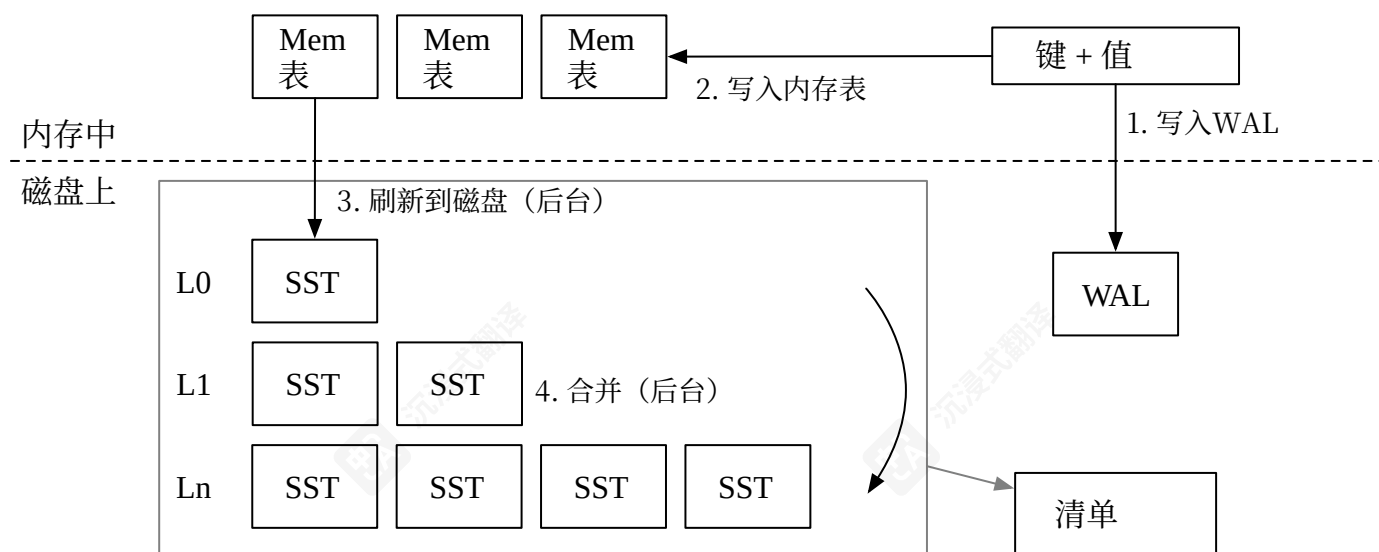
为确保持久性，

- `Sync()`：确保所有在 之前的操作都持久化到磁盘。

一些引擎选择将 和 合并为一个称为 `WriteBatch` 的单一操作，该操作接受一个键值对批次。

在本课程中，我们假设 LSM 树使用的是分层压缩算法，该算法在实际系统中常用。

写入路径

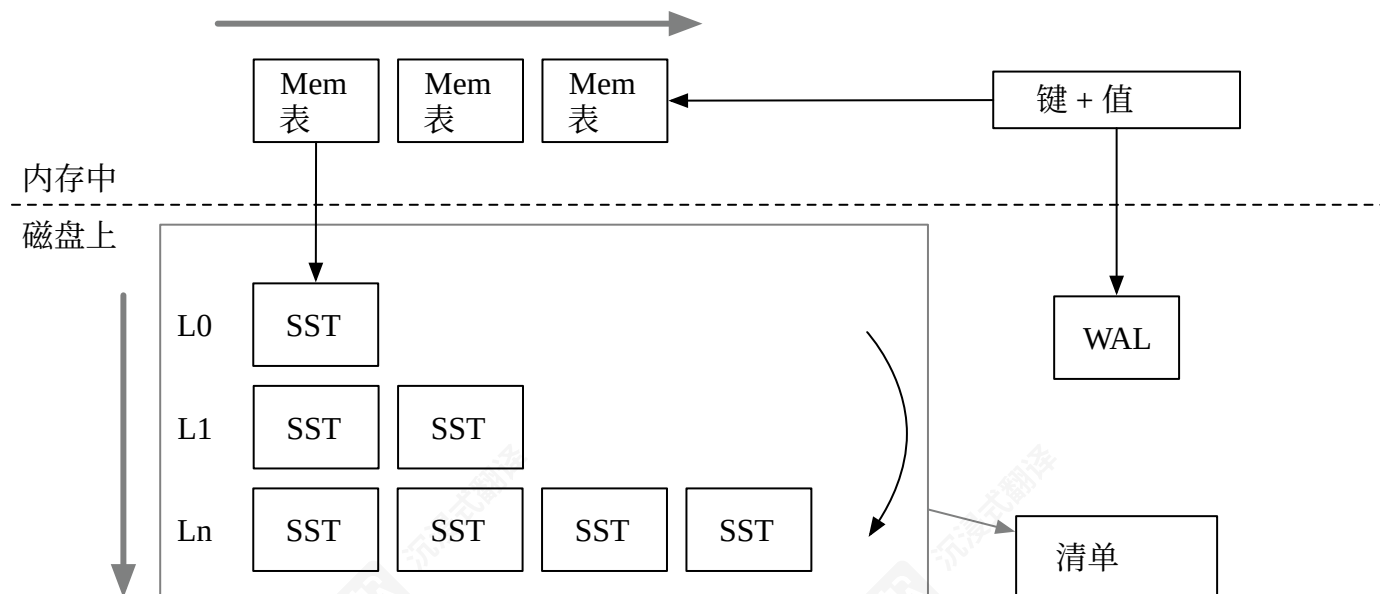


LSM的写入路径包含四个步骤：

1. 将键值对写入预写日志，以便在存储引擎崩溃后可以恢复。
2. 将键值对写入内存表。完成 (1) 和 (2) 后，我们可以通知用户写入操作已完成。
3. (后台) 当内存表满时，我们将它们冻结为不可变内存表，并在后台将它们刷新为SST文件到磁盘。
4. (在后台) 引擎会将一些层级中的某些文件压缩到较低层级，以保持LSM树的良好形状，从而降低读取放大。

读取路径

1. 在所有内存表中查找键值对（新到旧）



2. 在SSTs中查找键值对（顶层到底层）

当我们想要读取一个键时，

1. 我们将首先从最新到最旧探测所有内存表。
2. 如果键未找到，我们将搜索包含SSTs的整个LSM树以查找数据。

读取有两种类型：查找和扫描。查找在LSM树中查找一个键，而扫描在存储引擎中迭代范围内的所有键。我们将在整个课程中涵盖这两种类型。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？请在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z提供，根据CC BY-NC-SA 4.0协议授权。

环境设置

启动代码和参考解决方案可在 <https://github.com/skyzh/mini-lsm> 获取。

安装 Rust

See <https://rustup.rs> for more information.

克隆仓库

```
git clone https://github.com/skyzh/mini-lsm
```

启动代码

```
cd mini-lsm/mini-lsm-starter  
code .
```

安装工具

您需要最新的稳定 Rust 来编译此项目。最低要求是 1.74。

```
cargo x install-tools
```

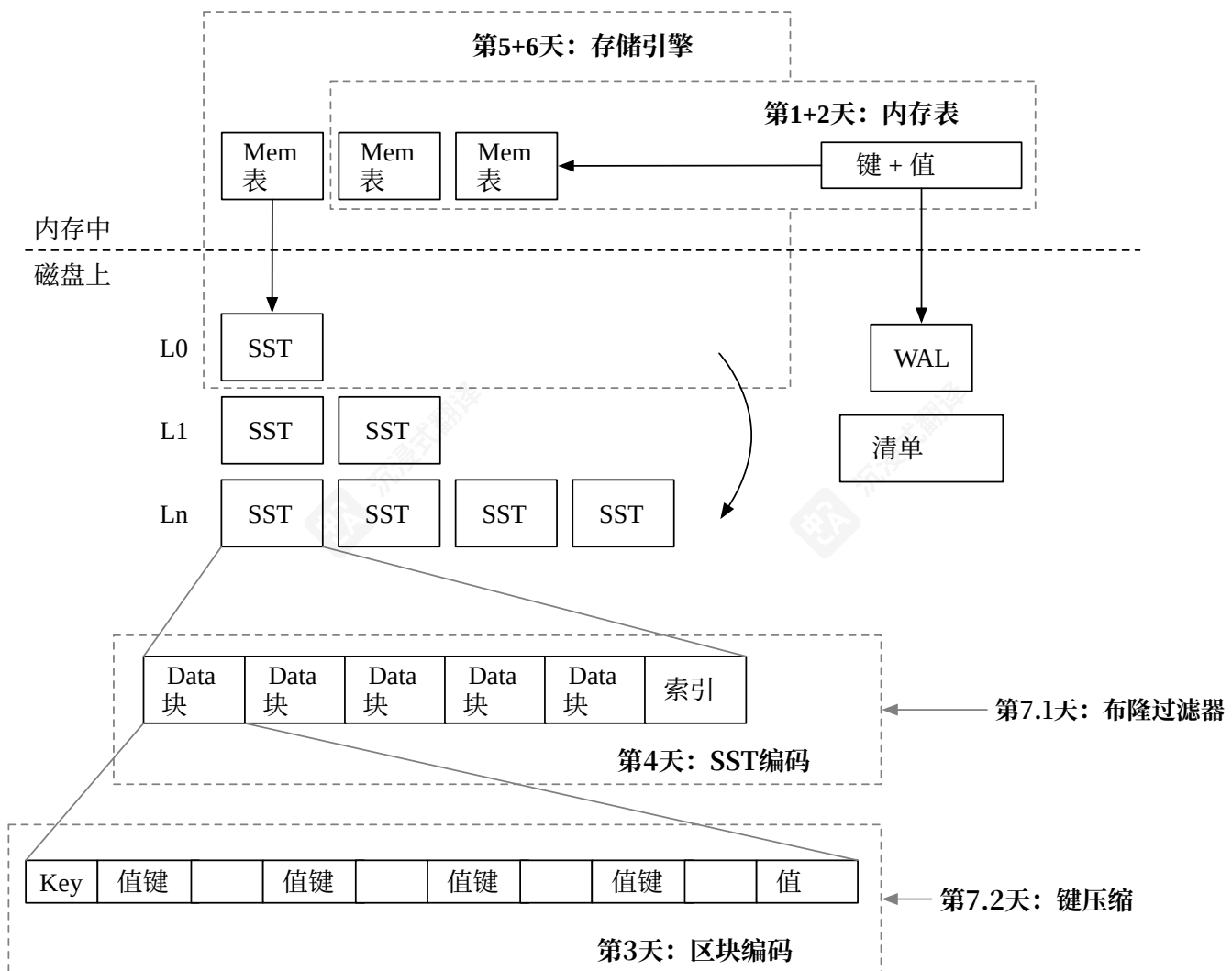
运行测试

```
cargo x copy-test --week 1 --day 1  
cargo x scheck
```

现在，你可以开始第1周：Mini-LSM。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权使用。

第1周概览：Mini-LSM



在课程的第一个星期，你将构建存储引擎所需的存储格式、系统的读取路径和写入路径，并实现一个基于LSM的键值存储的工作版本。这一部分有7个章节（天）。

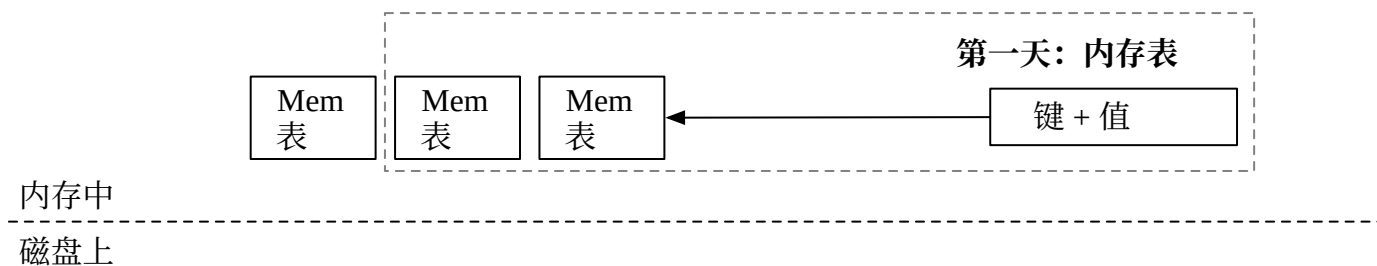
- 第1天：内存表。你将实现系统的内存读取和写入路径。
- 第2天：合并迭代器。你将扩展你在第一天所构建的内容，并实现一个`scan` 接口用于你。
- 第3天：区块编码。现在我们开始磁盘结构的第一个步骤，并构建区块的编码/解码。
- 第4天：SST编码。SSTs由区块组成，在一天结束时，你将拥有LSM磁盘结构的基本构建模块。
- 第5天：读取路径。现在我们已经有了内存和磁盘结构，可以将它们结合起来，为存储引擎提供一个完整的读取路径。
- 第6天：写入路径。在第5天，测试框架生成结构，在第6天，您将自行控制SST刷新。您将实现刷新到级别-0 SST，存储引擎是完整的。

- 第7天：SST优化。我们将实现几种SST格式优化，并提高系统的性能。

在一周结束时，您的存储引擎应该能够处理所有 get/scan/put 请求。唯一缺失的部分是将 LSM 状态持久化到磁盘以及一种更高效的方式来组织磁盘上的 SST。您将拥有一个可工作的 Mini-LSM 存储引擎。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z拥有，根据CC BY-NC-SA 4.0授权。

Memtables



在本章节中，你将：

- 基于跳表实现内存表。
- 实现冻结内存表逻辑。
- 实现 LSM 读路径 `get` 用于 `MemTable`。

要复制测试用例到入门代码并运行它们，

```
cargo x copy-test --week 1 --day 1
cargo x scheck
```

任务 1: SkipList Memtable

在这个任务中，你需要修改：

`src/mem_table.rs`

首先，让我们实现 LSM 存储引擎的内存结构——内存表。我们选择 `crossbeam` 的 `skiplist` 实现作为内存表的数据结构，因为它支持无锁并发读取和写入。我们将不深入讲解跳表的工作原理，简而言之，它是一个有序的键值映射，可以轻松地支持并发读取和写入。

`crossbeam-skiplist` 提供了与 Rust std 的 `BTreeMap` 类似的接口：插入、获取和迭代。唯一的区别是，修改接口（即 `insert`）只需要一个不可变引用到跳表，而不是可变引用。因此，在实现内存表结构时，你的实现中不应该获取任何互斥锁。

你还会注意到，`MemTable` 结构没有 `delete` 接口。在 `mini-lsm` 实现中，删除表示为一个对应空值的键。

在这个任务中，您需要实现 `MemTable::get` 和 `MemTable::put` 以便修改内存表。请注意，如果 `put` 已经存在，它应该始终覆盖一个键。

存在。你不会在单个内存表中拥有相同键的多个条目。

我们使用 `bytes crate` 来在内存表中存储数据。`bytes::Byte` 与 `Arc<[u8]>` 相似。当你克隆 `Bytes`，或者获取 `Bytes` 的切片时，底层数据不会被复制，因此克隆它是廉价的。相反，它只是创建了一个新的对存储区域的引用，当该区域没有引用时，存储区域将被释放。

任务 2：引擎中的单个内存表

在这个任务中，你需要修改：

`src/lsm_storage.rs`

现在，我们将第一个数据结构——内存表，添加到 LSM 状态中。在

`LsmStorageState::create`，你会发现当创建 LSM 结构时，我们会初始化一个 id 为 0 的内存表。这是初始状态中的可变内存表。在任何时刻，引擎将只有一个可变的内存表。内存表通常有一个大小限制（即 256MB），当它达到大小限制时，将被冻结为一个不可变的内存表。

查看 `lsm_storage` 与两个结构代表
存储引擎： `MiniLsmStorage` 轻量级的封装。
`LsmStorageInner`。您将在 `LsmStorageInner` 中实现大部分功能，直到第2周压缩。

`LsmStorageState` stores the current structure of the LSM存储引擎. For now, we will only use the `current` field, which stores the current mutable 内存表. In this task, you 将需要实现 `LsmStorageInner::delete`。所有这些都应该直接将请求派发给当前内存表。



内存中

磁盘上

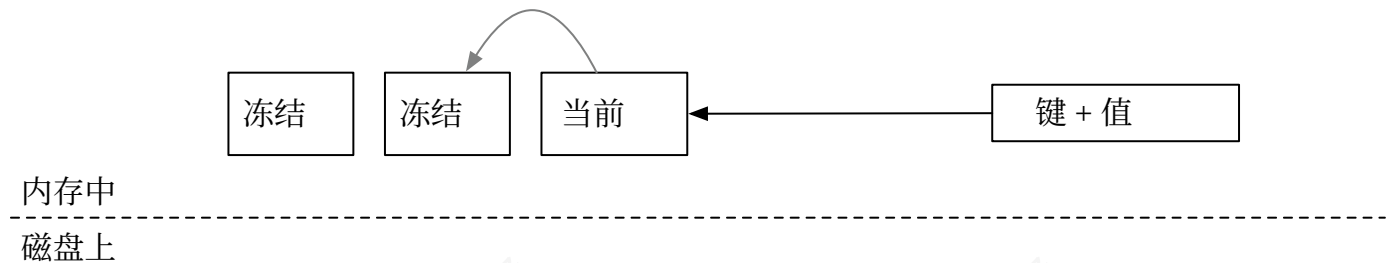
您的 `delete` 实现应该简单地为一个键放置一个空的切片，我们称之为删除墓碑。您的 `get` 实现应该相应地处理此情况。

要访问内存表，您需要获取 `state` 锁。由于我们的内存表实现只需要 `put` 的不可变引用，您只需要获取 `state` 的读锁即可修改内存表。这允许多个线程并发访问内存表。

任务 3：写入路径 - 冻结内存表

在这个任务中，您需要修改：

```
src/lsm_storage.rs  
src/mem_table.rs
```



内存表不能持续增长，当达到大小限制时，我们需要冻结它们（稍后刷新到磁盘）。您可以在 `LsmStorageOptions` 中找到内存表大小限制，该限制等于 SST 大小限制（不是 `num_memtables_limit`）。这不是一个硬限制，您应该尽可能冻结内存表。

在此任务中，当在内存表中 `put/delete` 一个键时，您需要计算近似内存表大小。这可以通过在 `put` 调用时简单地添加键和值的总字节数来计算。尽管键被 `put` 两次，尽管跳表只包含最新值，但在近似内存表大小中，您可以将其计算两次。一旦内存表达达到限制，您应该调用 `force_freeze_memtable` 冻结内存表并创建一个新的。

`LsmStorageInner` 中的 `state: Arc<RwLock<Arc<LsmStorageState>>>` 字段这样结构化，以并发和安全地管理 LSM 树的整体状态，主要使用写时复制（CoW）策略：

1. 内部 `Arc<LsmStorageState>`：这保存了实际 `LsmStorageState`（其中包含内存表列表、SST 引用等）的不可变快照。克隆这个 `Arc` 非常便宜（只是一个原子引用计数增加），并且为任何读者提供了一致且不变的视图，以供他们在操作期间使用。

2. `RwLock<Arc<LsmStorageState>>`：这个读写锁保护当前 `Arc<LsmStorageState>`（活动快照）{

- 读者获取一个读锁，克隆 `Arc<LsmStorageState>`（获取他们自己的当前快照引用），然后快速释放读锁。然后他们可以处理他们的快照，而无需进一步锁定。
- 写者（在修改状态时，例如冻结一个内存表）将：
 - 创建一个新的 `LsmStorageState` 实例，通常通过从当前快照克隆数据，然后应用修改来创建。
 - 将这个新状态包裹在一个新的 `Ar`
 - 在 `RwLock` 上获取写锁。

- 用新的替换旧的
- 释放写锁。

。

3. 外部 `Arc<RwLock<...>>`：这允许 `RwLock` 本身（以及因此访问和更新状态的机制）在多个线程或可能需要与 `LsmStorageInner` 交互的应用程序部分之间安全共享。

这种 CoW 方法确保读取器始终看到有效的、一致的状态快照，并经历最小的阻塞。写入器通过交换整个状态快照来原子性地更新状态，减少了持有关键锁的时间，从而提高了并发性。

因为可能有多个线程将数据写入存储引擎，

`force_freeze_memtable` 可能会被多个线程并发调用。您需要考虑在这种情况下如何避免竞态条件。

您可能希望在多个位置修改 LSM 状态：冻结可变内存表、将内存表刷新到 SST，以及 GC/压缩。在这些所有修改期间，可能会有 I/O 操作。构建锁定策略的一种直观方法是：

```
fn freeze_memtable(&self) {  
    let state = self.state.write();  
    state.immutable_memtable.push(/* something */);  
    state.memtable = MemTable::create();  
}
```

...你修改LSM状态写锁中的所有内容。

目前这很好用。但是，考虑一下你想要为每个创建的内存表创建预写式日志文件的情况。

```
fn freeze_memtable(&self) {  
    let state = self.state.write();  
    state.immutable_memtable.push(/* something */);  
    state.memtable = MemTable::create_with_wal()?; // <- could take several  
    milliseconds  
}
```

现在当我们冻结内存表时，其他线程在几毫秒内无法访问LSM状态，这会导致延迟激增。

为了解决这个问题，我们可以将I/O操作放在锁区域之外。


```
fn freeze_memtable(&self) {
    let memtable = MemTable::create_with_wal()?; // <- could take several
    milliseconds
    {
        let state = self.state.write();
        state.immutable_memtable.push(/* something */);
        state.memtable = memtable;
    }
}
```

然后，我们在状态写锁区域内没有昂贵的操作。现在，考虑这样一种情况：内存表即将达到容量限制，并且两个线程成功地将两个键放入内存表，并且它们在放入这两个键后都发现内存表达到了容量限制。它们都会对内存表进行大小检查，并决定将其冻结。在这种情况下，我们可能会创建一个空的内存表，然后立即将其冻结。

要解决这个问题，所有状态修改都应该通过状态锁进行同步。

```
fn put(&self, key: &[u8], value: &[u8]) {
    // put things into the memtable, checks capacity, and drop the read lock on
    LSM state
    if memtable_reaches_capacity_on_put {
        let state_lock = self.state_lock.lock();
        if /* check again current memtable reaches capacity */ {
            self.freeze_memtable(&state_lock)?;
        }
    }
}
```

你会在未来的章节中经常注意到这种模式。例如，对于L0刷新，

```
fn force_flush_next_imm_memtable(&self) {
    let state_lock = self.state_lock.lock();
    // get the oldest memtable and drop the read lock on LSM state
    // write the contents to the disk
    // get the write lock on LSM state and update the state
}
```

这确保了只有一个线程能够修改LSM状态，同时仍然允许对LSM存储的并发访问。

在这个任务中，你需要修改 `put` 和 `delete` 以尊重内存表的软容量限制。当它达到限制时，调用 `force_freeze_memtable` 冻结内存表。请注意，我们没有关于这个并发场景的测试用例，你需要自己考虑所有可能的竞态条件。此外，请记住检查锁区域，以确保临界区是最小必需的。

你可以简单地分配下一个内存表ID为 `self.next_sst_id()`。请注意，`imm_memtables` 按从最新诸内存表。也就是说，

`imm_memtables.first()` 应该是最后一个冻结的内存表。

任务 4：读取路径 - 获取

在这个任务中，您需要修改：

`src/lsm_storage.rs`

现在您有多个内存表，您可以修改您的读取路径 `get` 函数以获取键的最新版本。确保您从最新的内存表开始探测到最早的内存表。

测试您的理解

- 为什么内存表不提供 `delete` API？
- 内存表存储所有写入操作而不是仅存储键的最新版本是否有意义？例如，用户将 `a->1`、`a->2` 和 `a->3` 存入同一个内存表。
- 是否可以使用其他数据结构作为LSM的内存表？使用跳表的优缺点是什么？
- 为什么我们需要 `state` 和 `state_lock` 的组合？我们只能使用 `state.read()` 吗？
- 为什么存储和探测内存表的顺序很重要？如果一个键出现在多个内存表中，应该返回哪个版本给用户？
- 内存表 `memtable` 的布局是否高效 / 是否具有良好的数据局部性？（想想 `Byte` 在跳表 `skiplist` 中的实现和存储方式……）有哪些可能的优化可以使内存表更高效？
- 所以我们在本课程中使用 `parking_lot` 锁。它的读写锁是公平锁吗？如果有一个写者在等待现有读者停止，尝试获取锁的读者可能会发生什么？
- 在冻结内存表后，是否有可能某些线程仍然持有旧的LSM状态并写入这些不可变的内存表？您的解决方案如何防止这种情况发生？
- 有多个地方，你可能首先对状态获取读锁，然后删除它并获取写锁（这两个操作可能位于不同的函数中，但由于一个函数调用了另一个函数，它们是按顺序发生的）。这与直接将读锁升级为写锁有何不同？是否需要升级而不是获取和删除，以及执行升级的成本是什么？

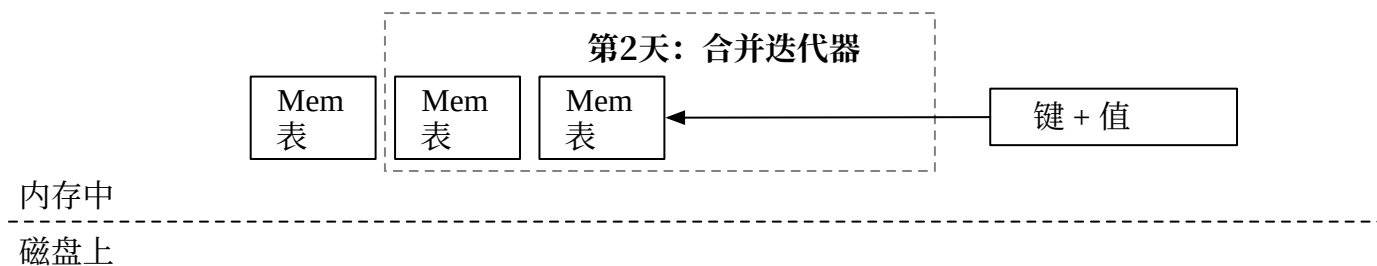
我们不会提供这些问题的参考答案，并且欢迎在Discord社区中讨论它们。

奖励任务

- 更多内存表格式。您可以实现其他内存表格式。例如，B树内存表、向量内存表和ART内存表。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z拥有，根据CC BY-NC-SA 4.0许可协议授权。

合并迭代器



在本章节中，你将：

- 实现内存表迭代器。
- 实现合并迭代器。
- 实现 LSM 读路径 `scan` 用于 表。

要复制测试用例到入门代码并运行它们，

```
cargo x copy-test --week 1 --day 2
cargo x scheck
```

任务 1：内存表迭代器

在本章节中，我们将实现 LSM `scan` 接口。 `scan` 使用迭代器 API 按顺序返回键值对范围。在上一章节中，你已经实现了 `get` API 以及创建不可变内存表的逻辑，现在你的 LSM 状态应该包含多个内存表。你需要首先在单个内存表上创建迭代器，然后在所有内存表上创建合并迭代器，最后实现迭代器的范围限制。

在本任务中，你需要修改：

`src/mem_table.rs`

所有 LSM 迭代器实现 `StorageIterator` 特征。它有 4 个函数：`key`、`value`、`next` 和 `is_valid`。如果你熟悉 Rust 的标准库 `Iterator` 特征，可能会觉得 `StorageIterator` 有些不同。相反，`StorageIterator` 采用基于游标的 API，这是一种在数据库系统中常见的设计模式，并显著受 RocksDB 迭代器（参见 [RocksDB 迭代器](#) 和 [RocksDB 迭代器](#) 以供参考）的启发。

当迭代器被创建时，它的游标将停在某个元素上，并且 `key / value` 将返回满足起始条件（即起始键）的内存表/块/SST 中的第一个键。这两个接口将返回一个 `&[u8]` 以避免复制。

从调用者的角度来看，典型的使用模式是：

```
let mut iter: impl StorageIterator = ...; while iter.is_valid() {
    let key = iter.key(); let value = iter.value(); // Process key and value
    iter.next()?; // Advance to the next item, handling potential errors}
```

`StorageIterator`

法有所不同：

- `next()`：此方法仅负责尝试将游标移动到下一个元素。它返回一个 `Result` 来报告在此推进过程中遇到的任何错误（例如，I/O问题）。它本身并不保证新位置有效，仅保证移动尝试已做出。
- `is_valid()`：此方法指示迭代器的当前游标是否指向一个有效的数据元素。它不会推进迭代器。

因此，作为 `StorageIterator` 的实现者，在每次调用 `next()`（即使它在不从 `next()` 操作本身返回错误的情况下成功）后，您有责任更新内部状态，以便 `is_valid()` 正确反映新的游标位置是否实际指向一个有效项。

总之，`next` 将游标移动到下一个位置。`is_valid` 返回迭代器是否已到达末尾或出错。您可以假设 `next` 仅在 `is_valid` 返回 `true` 时被调用。对于迭代器，将有一个 `FusedIterator` 包装器，在迭代器无效时阻止对 `next` 的调用，以避免用户误用迭代器。

回到内存表迭代器。你应该已经发现迭代器没有关联任何生命周期。想象一下你创建一个 `Vec<u64>` 并调用 `vec.iter()`，迭代器类型将类似于 `VecIterator<'a>`，其中 `'a` 是 `vec` 对象的生命周期。这同样适用于 `SkipMap`，其 `iter` API 返回具有生命周期的迭代器。然而，在我们的情况下，我们不想在我们的迭代器上具有这样的生命周期，以避免使系统过于复杂（并且难以编译...）。

如果迭代器没有生命周期泛型参数，我们应该确保每次使用迭代器时，底层的跳表对象没有被释放。实现这一点的唯一方法是 将 `Arc<SkipMap>` 对象放入迭代器本身。为了定义这样的结构，

```
pub struct MemtableIterator {
    map: Arc<SkipMap<Bytes, Bytes>>,
    iter: SkipMapRangeIter<'???'>,}
```

好的，这里有问题：我们想要表达迭代器的生命周期与结构中的 `map` 相同。我们该如何做到呢？

这是本课程中你将遇到的第一件也是最棘手的 Rust 语言相关的事情——自引用结构。如果可能的话，可以编写如下内容：

```
pub struct MemtableIterator { // <- with lifetime 'this
    map: Arc<SkipMap<Bytes, Bytes>>, iter: SkipMapRangeIter<'this>,
}
```

然后问题就解决了！你可以借助一些第三方库如ouroboros 来实现。它提供了一种简单的方式来定义自引用结构。也可以使用unsafe Rust来完成（实际上，ouroboros 本身也使用unsafe Rust内部...）

我们已经利用 ouroboros 为你定义了自引用的 MemtableIterator 字段。你需要基于这个提供的结构来实现 MemtableIterator 逻辑和 Memtable::scan API。

任务 2：合并迭代器

在这个任务中，你需要修改：

src/iterators/merge_iterator.rs

现在你已经有了多个内存表（memtable），并且会创建多个内存表迭代器。你需要合并内存表的结果，并将每个键的最新版本返回给用户。

MergeIterator 内部维护了一个二叉堆。考虑将 n 排序序列（我们的迭代器）合并为一个排序输出的挑战；二叉堆在这里是自然的选择，因为它高效地帮助识别当前哪个序列持有整体最小的元素。你会看到二叉堆的顺序是，具有最低头部键值的迭代器排在最前面。当多个迭代器具有相同的头部键值时，最新的一个排在最前面。注意你需要处理错误（即，当迭代器无效时），并确保键值对的最新版本输出。

例如，如果我们有以下数据：

```
iter1: b->del, c->4, d->5
iter2: a->1, b->2, c->3
iter3: e->4
```

合并迭代器输出的序列应该是：

```
a->1, b->del, c->4, d->5, e->4
```

合并迭代器的构造函数接受一个迭代器向量。我们假设索引较低（即第一个）的那个拥有最新数据。

在使用 Rust 二叉堆时，您可能会发现 `peek_mut` 函数

```
let Some(mut inner) = heap.peek_mut() {
    *inner += 1; // <- do some modifications to the inner item
}
// When the PeekMut reference gets dropped, the binary heap gets reordered
// automatically.

let Some(mut inner) = heap.peek_mut() {
    PeekMut::pop(inner) // <- pop it out from the heap
}
```

一个常见的陷阱在于错误处理。例如，

```
let Some(mut inner_iter) = self.iters.peek_mut() {
    inner_iter.next()?; // <- will cause problem
}
```

如果 `next` 返回错误（即由于磁盘故障、网络故障、校验和错误等），它就不再有效。然而，当我们退出 `if` 条件并将错误返回给调用者时，`PeekMut` 的删除操作会尝试在堆中移动元素，这会导致访问无效迭代器。因此，您需要自己处理所有错误，而不是在 `PeekMut` 的范围内使用 `?`。

我们希望尽可能避免动态分发，因此我们不在系统中使用 `Box<dyn StorageIterator>`。相反，我们更倾向于使用泛型进行静态分发。此外请注意，`StorageIterator` 使用泛型关联类型（GAT），因此它可以将 `KeySlice` 和 `&[u8]` 作为键类型。我们将更改 `KeySlice` 以包含第 3 周的时间戳，现在使用单独的类型可以使过渡更加平滑。

从这一部分开始，我们将使用 `Key<T>` 来表示 LSM 键类型，并将它们与类型系统中的值区分开来。你应该使用 `Key<T>` 提供的 API，而不是直接访问内部值。在第三部分中，我们将向这种键类型添加时间戳，使用键抽象将使过渡更加平滑。目前，`KeySlice` 等同于 `&[u8]`，`KeyVec` 等同于 `Vec<u8>`，`KeyBytes` 等同于 `Bytes`。

任务 3：LSM 迭代器 + 融合迭代器

在这个任务中，你需要修改：

`src/lsm_iterator.rs`

我们使用 `LsmIterator` 结构来表示内部的 LSM 迭代器。在整个课程中，当你向系统中添加更多迭代器时，你需要多次修改这个结构。目前，因为我们只有多个内存表，它应该定义为：

```
type LsmIteratorInner = MergeIterator<MemTableIterator>;
```

您可以继续实现 `LsmIterator` 结构，该结构会调用相应的内部迭代器，并且也会跳过已删除的键。

我们在这个任务 `task4_iterator` 。任务4中会有一个集成测试。

然后，我们希望在迭代器上提供额外的安全性，以避免用户误用它们。当迭代器无效时，用户不应调用 `key`、`value` 或 `next` 。同时，如果 `next` 返回错误，他们应停止使用迭代器。

`FusedIterator` 是一个围绕迭代器的包装器，用于跨所有迭代器规范化行为。您可以自己继续实现它。

任务4：读取路径 - 扫描

在这个任务中，您需要修改：

```
src/lsm_storage.rs
```

我们终于到了——凭借您已经实现的全部迭代器，您终于可以实现LSM引擎的`scan` 接口。您只需用内存表迭代器（记住将最新的内存表放在合并迭代器的前面）构造一个LSM迭代器，您的存储引擎就能够处理扫描请求。

测试您的理解

- 使用您的合并迭代器的时间/空间复杂度是多少？
- 为什么我们需要为内存表迭代器使用自引用结构？
- 如果一个键被删除（存在删除墓碑），您需要将其返回给用户吗？您在哪里处理这个逻辑？
- 如果一个键有多个版本，用户会看到所有版本吗？您在哪里处理这个逻辑？
- 如果我们想消除自引用结构，并且让`memtable`迭代器（即 `MemtableIterator<'a>` ，其中 `'a = memtable`或 `LsmStorageInner`生命周期）具有生命周期，是否仍然可以实现 `scan` 功能？

- 如果 (1) 我们在跳表内存表上创建一个迭代器 (2) 有人向内存表插入新键 (3) 迭代器会看到新键吗?
- 如果你的键比较器无法为二叉堆实现提供稳定顺序, 会发生什么?
- 我们为什么需要确保合并迭代器按迭代器构造顺序返回数据?
- 是否可以针对LSM迭代器实现一个Rust风格迭代器 (即 `next(&self) -> (Key, Value)`) ? 优缺点是什么?
- 扫描接口类似于 `fn scan(&self, lower: Bound<&[u8]>, upper: Bound<&[u8]>)`。如何与Rust风格范围 (即 `key_a..key_b`) 兼容? 如果你实现这个, 尝试将完整范围..传递给接口, 看看会发生什么。
- 入门代码提供了合并迭代器接口来存储 `Box<I>` 而不是 `I`。这背后的原因可能是什么?

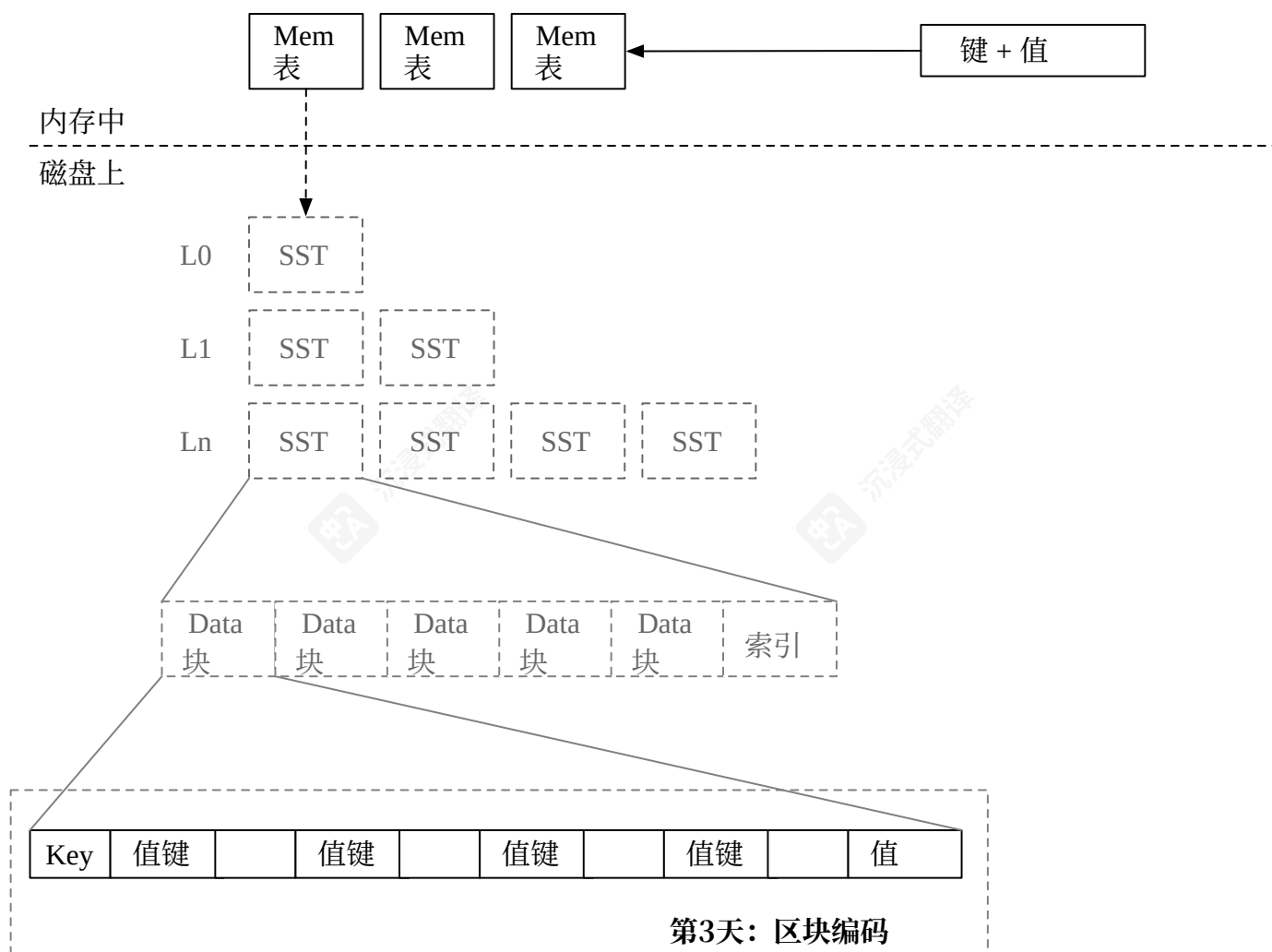
我们不提供问题的参考答案, 欢迎在Discord社区讨论。

奖励任务

- 前台迭代器。在本课程中, 我们假设所有操作都是短期的, 因此我们可以在迭代器中持有对内存表的引用。如果一个迭代器被用户长时间持有, 整个内存表 (可能为256MB) 即使已经刷新到磁盘, 也会一直留在内存中。为了解决这个问题, 我们可以向用户提供一个 `ForegroundIterator / LongIterator`。迭代器将定期创建新的底层存储迭代器, 以允许资源进行垃圾回收。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时? 在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作, 根据CC BY-NC-SA 4.0协议授权。

块



在本章节中，您将：

- 实现SST区块编码。
- 实现SST区块解码和区块迭代器。

要复制测试用例到启动代码并运行它们，

```
cargo x copy-test --week 1 --day 3
cargo x scheck
```

任务 1：区块构建器

你已经在前两章中实现了LSM存储引擎的所有内存结构。现在该构建磁盘结构了。磁盘结构的基本单元是块。块通常为4KB大小（大小可能因存储介质而异），这相当于操作系统和SSD上的页大小。一个块存储有序的键值对。一个SST由多个

在这个任务中，你需要修改：

我们课程中的块编码格式如下:

每个条目是一个键值对。

键长度和值长度都是2个字节，这意味着它们的最大长度是65535。(内部存储为 `u16`)

我们假设键永远不会为空，而值可以为空。空值意味着在其他系统部分的视图中，相应的键已被删除。对于 `BlockBuilder` 和 `BlockIterator`，我们只是将空值原样处理。

在每个块结束时，我们将存储每个条目的偏移量和条目总数。例如，如果第一个条目位于块的0位置，第二个条目位于块的12位置。

块的页脚将如上所示。每个数字都存储为 `u16`。

块有一个大小限制，它是 `target_size`。除非第一个键值对超过目标块大小，否则您应确保编码块的大小始终小于或等于

到 `target_size`。(在提供的代码中, 这里的 `target_size` 基本上是 `block_size`)

当 `build` 被调用时, `BlockBuilder` 将生成数据部分和未编码的条目偏移量。这些信息将存储在 `Block` 结构中。由于键值条目以原始格式存储, 偏移量存储在一个单独的向量中, 因此在解码数据时可以减少不必要的内存分配和处理开销——你只需要将原始块数据复制到 `data` 向量中, 并每隔 2 字节解码条目偏移量, 而不是创建类似 `Vec<(Vec<u8>, Vec<u8>)>` 的东西来将所有键值对存储在一个内存块中。这种紧凑的内存布局非常高效。

在 `Block::encode` 和 `Block::decode` 中, 你需要按照上述指示格式编码/解码块。

任务 2: 块迭代器

在这个任务中, 你需要修改:

```
src/block/iterator.rs
```

现在我们已经有一个编码的块, 我们需要实现 `BlockIterator` 接口, 以使用户可以在块中查找/扫描键。

`BlockIterator` 可以使用 `Arc<Block>` 创建。如果 `create_and_seek_to_first` 被调用, 它将定位到块中的第一个键。如果 `create_and_seek_to_key` 被调用, 迭代器将定位到第一个满足 `>=` 提供的键的键。例如, 如果 1、3、5 在一个块中。

```
let mut iter = BlockIterator::create_and_seek_to_key(block, b"2");
assert_eq!(iter.key(), b"3");
```

上述 `seek 2` 将使迭代器定位到下一个可用的键 2, 在本例中是 3。

迭代器应该从块中复制 `key` 并将它们存储在迭代器内部 (未来我们将有键压缩, 你将不得不这样做)。对于值, 你只需要在迭代器中存储开始/结束偏移量, 而不需要复制它们。

当 `next` 被调用时, 迭代器将移动到下一个位置。如果我们到达块的末尾, 我们可以将 `key` 设置为空, 并从 `is_valid` 返回 `false`, 以便调用者可以在可能的情况下切换到另一个块。

测试您的理解

- 在块中查找键的时间复杂度是多少？
- 当您在您的实现中查找不存在的键时，游标会停在何处？
- 所以 `Block` 仅仅是一个原始数据向量和一个偏移量向量。我们能将它们改为 `Byte` 和 `Arc<[u16]>` 吗，并将所有迭代器接口改为返回 `Byte` 而不是 `&[u8]`？（假设我们使用 `Byte::slice` 来返回一个块切片而不复制。）优缺点是什么？
- 您的实现中写入块中的数字的字节序是什么？
- 您的实现容易受到恶意构建的块的影响吗？如果用户故意构建一个无效的块，会发生无效内存访问或 OOM 吗？
- 一个块可以包含重复的键吗？
- 如果用户添加的键大于目标块大小，会发生什么？
- 考虑一下LSM引擎是基于对象存储服务（S3）的情况。你会如何优化/更改块格式和参数，使其适用于此类服务？
- 你喜欢珍珠奶茶吗？为什么喜欢或不喜欢？

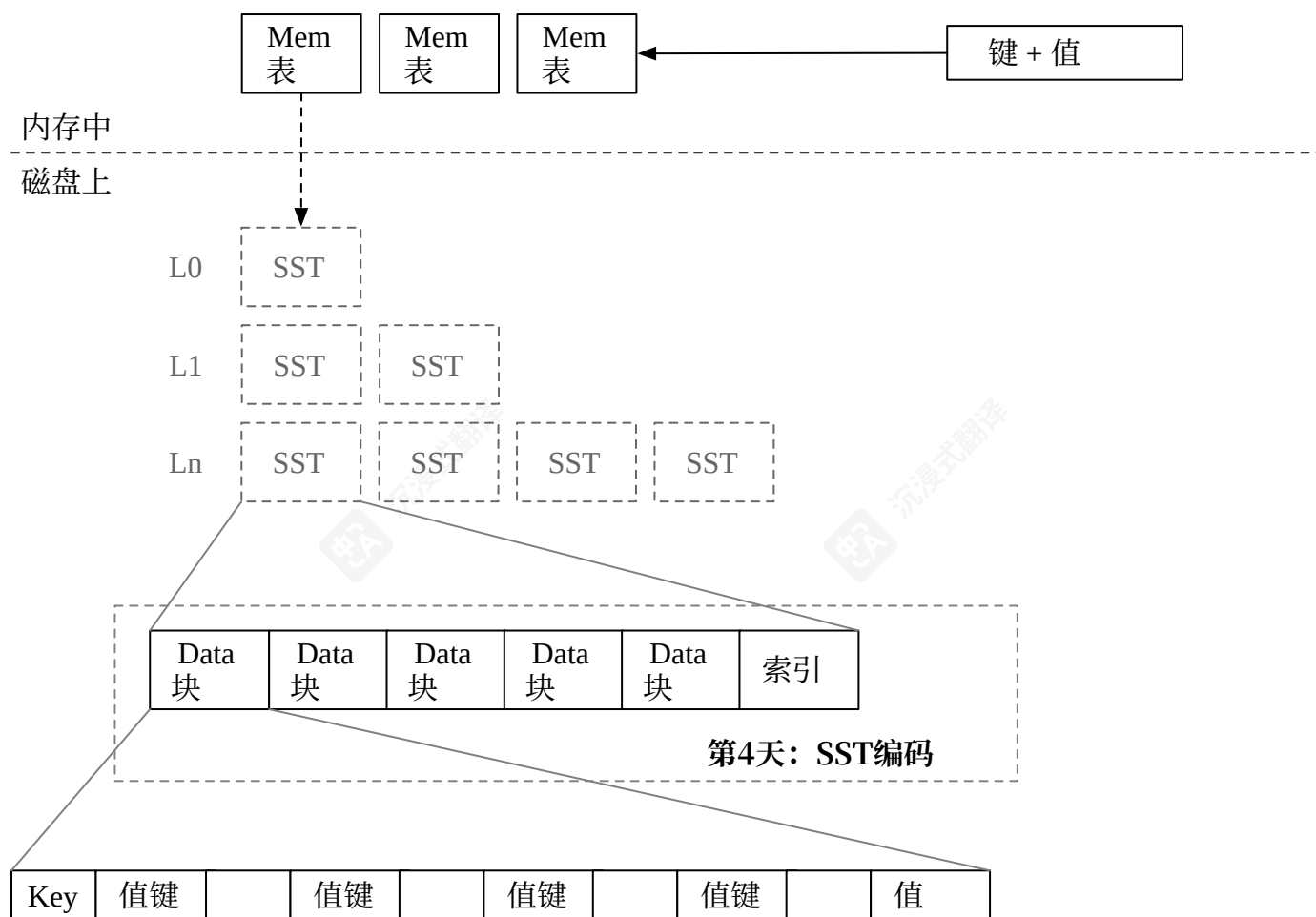
我们不会提供问题的参考答案，欢迎在Discord社区讨论。

奖励任务

- 反向迭代器。你可以为你的 `BlockIterator` 实现 `prev`，以便你能够反向迭代键值对。你也可以有一个反向合并迭代器和反向SST迭代器（在下一章节）的变体，这样你的存储引擎就可以进行反向扫描。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。

排序字符串表 (SST)



在本章节中，您将：

- 实现SST编码和元数据编码。
- 实现SST解码和迭代器。

要复制测试用例到启动代码并运行它们，

```
cargo x copy-test --week 1 --day 4
cargo x scheck
```

任务 1: SST构建器

在这个任务中，你需要修改：

```
src/table/builder.rs
src/table.rs
```

SSTs 由存储在磁盘上的数据块和索引块组成。通常，数据块是惰性加载的——它们不会加载到内存中，直到用户请求它。索引

块也可以按需加载，但在本课程中，我们做简单的假设，即所有SST索引块（元块）都可以放入内存（实际上我们没有专门的索引块实现。）通常，一个SST文件的大小为256MB。

SST构建器与块构建器类似 -- 用户将在构建器上调用 `add` 。您应该在SST构建器内部维护一个 `BlockBuilder` ，并在必要时拆分块。此外，您还需要维护块元数据 `BlockMeta` ，它包括每个块中的第一个/最后一个键以及每个块的偏移量。 `build` 函数将编码SST，使用 `FileObject::create` 将所有内容写入磁盘，并返回一个 `SsTable` 对象。

SST的编码方式如下：

Block Section		Meta Section	
Extra			
data block ... data block		metadata	meta block
offset (u32)			

您还需要实现 `estimated_size` 函数的 `SsTableBuilder` ，以便调用者知道何时可以开始写入新的SST来写入数据。该函数不需要非常精确。假设数据块包含比元块更多的数据，我们可以简单地返回数据块的大小给 `estimated_size` 。

除了SST构建器，您还需要完成块元数据的编码/解码，以便 `SsTableBuilder::build` 可以生成一个有效的SST。

任务 2：SST迭代器

在这个任务中，您需要修改：

```
src/table/iterator.rs
src/table.rs
```

像 `BlockIterator` 一样，你需要实现一个对 SST 的迭代器。请注意，你应该按需加载数据。例如，如果你的迭代器位于块 1，它不应该在到达下一个块之前将任何其他块内容保存在内存中。

`SsTableIterator` 应该实现 `StorageIterator` 特性，以便它将来可以与其他迭代器组合。

需要注意的一点是 `seek_to_key` 函数。基本上，您需要通过二分搜索块元数据来查找可能包含键的块。键可能不存在于 LSM 树中，因此块迭代器在查找后会立即会无效。例如，

```
-----  
| block 1 | block 2 |   block meta   |  
-----  
| a, b, c | e, f, g | 1: a/c, 2: e/g |  
-----
```

我们建议仅使用每个块的第一个键进行二分搜索，以降低您的实现复杂度。如果我们在这个 SST 中执行 `seek(b)`，那就相当简单——使用二分搜索，我们可以知道块1包含键 $a \leq \text{keys} < e$ 。因此，我们加载块1并将块迭代器定位到相应位置。

但如果我们执行 `seek(d)`，我们会定位到块1，如果我们仅将第一个键作为二分搜索标准，但在块1中查找 `d` 会到达块末尾。因此，我们应该在查找后检查迭代器是否无效，并在必要时切换到下一个块。或者你可以利用最后一个键元数据直接定位到一个正确的块，这取决于你。

任务3：块缓存

在这个任务中，你需要修改：

```
src/table/iterator.rs  
src/table.rs
```

你可以在 `SsTable` 上实现一个 `cached` 函数。

我们使用 `moka-rs` 作为我们的块缓存实现。块通过 `(sst_id, block_id)` 作为缓存键。在 `cached` 函数中，你可以使用 `try_get_with` 从缓存中获取块；如果缓存未命中，可以填充缓存。如果有多个请求读取同一个块且缓存未命中，`try_get_with` 将仅向磁盘发出一个读取请求并将结果广播给所有请求。

此时，您可以将表迭代器更改为使用 `read_block_cached` 而不是 `read_block` 以利用块

测试您的理解

- 在 SST 中查找键的时间复杂度是多少？

- 当你在你的实现中查找一个不存在的键时，游标会停在何处？
- 是否可以（或有必要）对SST文件进行原地更新？
- SST通常很大（即256MB）。在这种情况下，复制/扩展Vec 的成本会很高。你的实现是否为你的SST构建器预先分配了足够的空间？你是如何实现的？
- 查看 moka 块缓存，为什么它返回 Arc<Error> 而不是原始的 Error？
- 使用块缓存是否保证内存中最多有固定数量的块？例如，如果你有一个 moka 4GB的块缓存和4KB的块大小，内存中同时会有超过4GB/4KB数量的块吗？
- 是否可以将列式数据（即一个包含100个整数列的表）存储在LSM引擎中？当前的SST格式仍然是一个好的选择吗？
- 考虑LSM引擎基于对象存储服务（即S3）构建的情况。你会如何优化/更改SST格式/参数和块缓存，使其适用于此类服务？
- 目前，我们将所有SST的索引加载到内存中。假设您为索引预留了16GB的内存，您能估计您的LSM系统可以支持的最大数据库大小吗？（这就是为什么您需要索引缓存！）

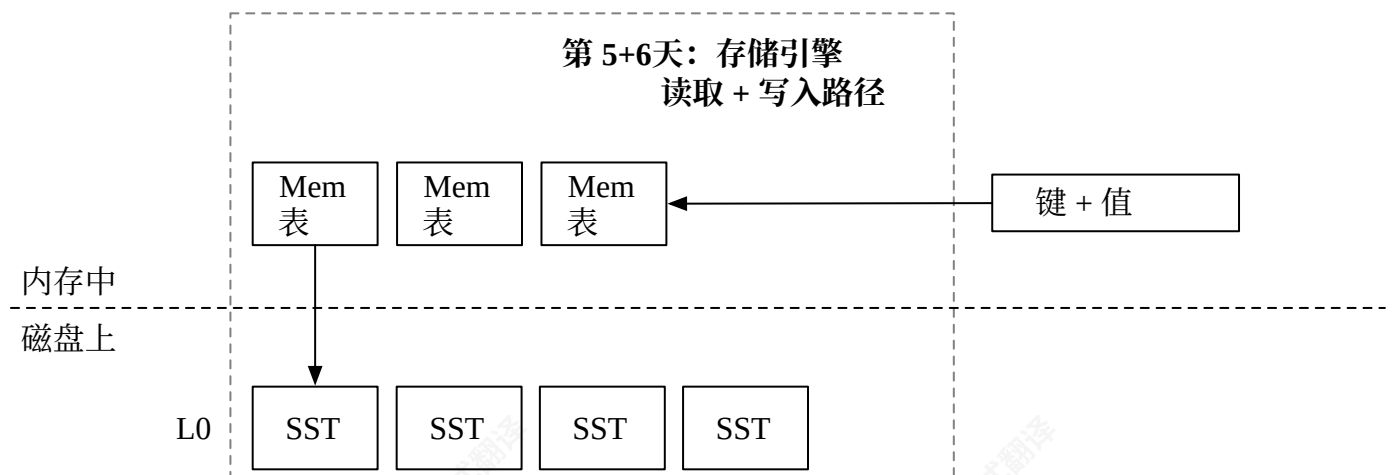
我们不会提供问题的参考答案，并且欢迎在Discord社区中讨论它们。

奖励任务

- 探索不同的SST编码和布局。例如，在《Lethe: Enabling Efficient Deletes in LSMs》论文中，作者为SST添加了辅助键支持。或者您可以使用B+ Tree作为SST格式，而不是排序块。
- 索引块。将块索引和块元数据拆分为索引块，并按需加载它们。
- 索引缓存。除了数据块缓存外，使用单独的缓存来缓存索引。
- I/O优化。将块对齐到4KB边界，并使用直接I/O绕过系统页缓存。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z发布，根据CC BY-NC-SA 4.0协议授权。

读取路径



在本章节中，您将：

- 将SST集成到LSM读路径。
- 实现LSM读路径 `get` 使用SST。
- 实现LSM读路径 `scan` 使用SST。

将测试用例复制到入门代码中并运行它们，

```
cargo x copy-test --week 1 --day 5
cargo x scheck
```

任务 1：双合并迭代器

在这个任务中，您需要修改：

```
src/iterators/two_merge_iterator.rs
```

您已经实现了一个合并迭代器，该迭代器合并相同类型的迭代器（即内存表迭代器）。现在我们已经实现了SST格式，我们既有磁盘SST结构，也有内存中的内存表。当我们从存储引擎扫描时，我们需要将来自内存表迭代器和SST迭代器的数据合并成一个。在这种情况下，我们需要一个 `TwoMergeIterator<X, Y>`，它合并两种不同类型的迭代器。

您可以在 `two_merge_iterator.rs` 中实现 `TwoMergeIterator`。由于我们这里只有两个迭代器，因此不需要维护一个二叉堆。相反，我们可以简单地使用一个标志来指示要读取哪个迭代器。类似于 `MergeIterator`，如果两个迭代器中找到相同的键，则第一个迭代器具有优先权。

任务 2：读取路径 - 扫描

在这个任务中，您需要修改：

```
src/lsm_iterator.rs  
src/lsm_storage.rs
```

在实现 `TwoMergeIterator` 后，我们可以更改 `LsmIteratorInner` 为以下类型：

```
type LsmIteratorInner =  
TwoMergeIterator<MergeIterator<MemTableIterator>,  
MergeIterator<SsTableIterator>>>;
```

这样，我们的 LSM 存储引擎的内部迭代器将是一个结合 memtables 和 SSTs 数据的迭代器。

目前，我们的 SST 迭代器不支持扫描的结束边界。为了解决这个问题，您需要在 `LsmIterator` 本身中实现这个边界检查。这涉及更新 `LsmIterator::new` 构造函数以接受一个 `end_bound` 参数：

```
pub(crate) fn new(iter: LsmIteratorInner, end_bound: Bound<Bytes>) ->  
Result<Self> {}
```

然后，您需要修改 `LsmIterator` 的迭代逻辑，以确保当内部迭代器的键达到或超过指定的 `end_bound` 时停止。

我们的测试用例将在 `l0_sstables` 中生成一些 memtables 和 SSTs，您需要在这个任务中正确地扫描所有这些数据。您不需要刷新 SSTs，直到下一章节。因此，您可以继续修改您的 `LsmStorageInner::scan` 接口，以创建一个覆盖所有 memtables 和 SSTs 的合并迭代器，从而完成您的存储引擎的读取路径。

因为 `SsTableIterator::create` 涉及 I/O 操作且可能较慢，我们不想在 `state` 临界区中执行此操作。因此，你应该首先读取 `state` 并克隆 LSM 状态快照的 `Arc`。然后，你应该释放锁。之后，你可以遍历所有 L0 SSTs 并为每个创建迭代器，然后创建一个合并迭代器来检索数据。

```
fn scan(&self) {  
    let snapshot = {  
        let guard = self.state.read();  
        Arc::clone(&guard)  
    };  
    // create iterators and seek them  
}
```

在LSM存储状态下，我们仅将SST ID存储在 `l0_sstables` 向量中。您需要从 `sstables` 哈希表中检索实际的SST对象。

任务 3：读取路径 - 获取

在这个任务中，您需要修改：

`src/lsm_storage.rs`

对于 `get` 请求，它将被作为在内存表中的查找来处理，然后对 SST 进行扫描。您可以在探测所有内存表后，在所有 SST 上创建一个合并迭代器。您可以查找用户想要查找的键。查找有两种可能性：键与用户探测的键相同，键不同 / 不存在。当键存在且与探测的键相同时，您才应该将值返回给用户。您还应该像上一节中那样减少状态锁的临界区。还要记得处理已删除的键。

测试您的理解

- 考虑一个用户有一个迭代器，它迭代整个存储引擎，并且存储引擎有 1TB 大小，因此扫描所有数据需要 ~1 小时。如果用户这样做，会出现什么问题？（这是一个好问题，我们在课程的不同的点会多次问到它...）
- 另一种由一些LSM树存储引擎提供的流行接口是multi-get（或向量get）。用户可以传递他们想要检索的键列表。该接口返回每个键的值。例如，`multi_get(vec!["a", "b", "c", "d"])` -> `a=1,b=2,c=3,d=4`。显然，一个简单的实现：`for key in keys { get(key) }`，以及你能做哪些优化来使其更高效？（提示：`get`过程中的一些操作只需要对所有键执行一次，除此之外，你可以考虑改进磁盘I/O接口以更好地支持这个multi-get接口）。

我们不会提供这些问题的参考答案，欢迎在Discord社区中讨论。

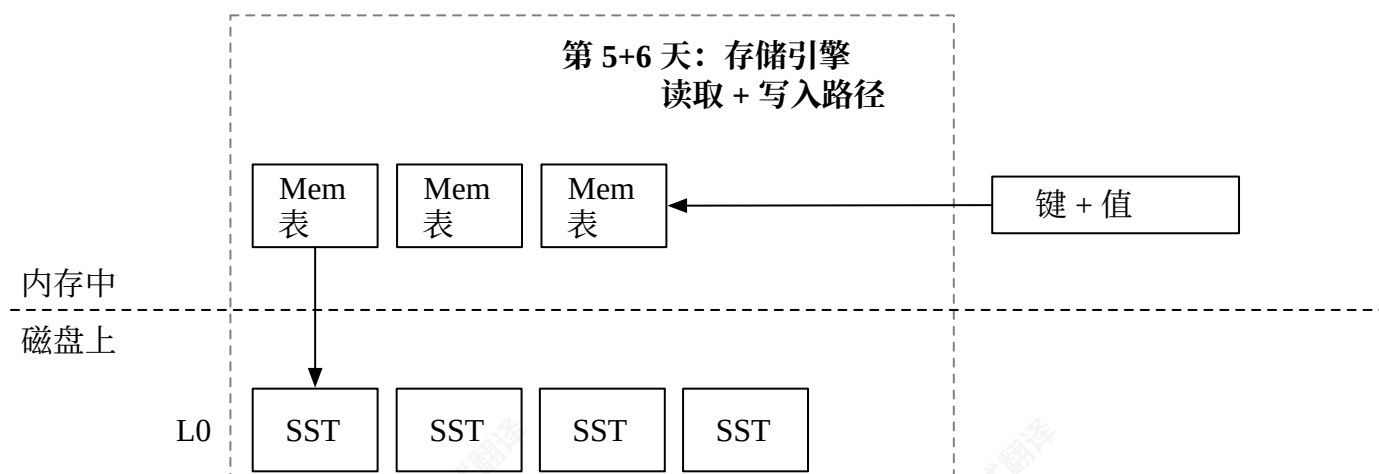
奖励任务

- 动态分发的成本。实现一个 `Box<dyn StorageIterator>` 版本的合并迭代器，并进行基准测试以查看性能差异。

- 并行查找。创建合并迭代器需要加载所有底层 SST 的第一个块（当你创建 `SSTIterator` 时）。你可以并行化创建迭代器的过程。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。

写入路径



在本章节中，您将：

- 使用 L0 刷新实现 LSM 写入路径。
- 实现正确更新 LSM 状态的逻辑。

将测试用例复制到入门代码中并运行它们，

```
cargo x copy-test --week 1 --day 6
cargo x scheck
```

任务 1：将内存表刷新到 SST

此时，我们已经准备好了所有内存中的数据和磁盘文件，存储引擎能够读取并合并来自这些结构的数据。现在，我们将实现将数据从内存移动到磁盘的逻辑（即所谓的刷新），并完成 Mini-LSM 第1周课程。

在这个任务中，你需要修改：

```
src/lsm_storage.rs
src/mem_table.rs
```

你需要修改 `LSMStorageInner::force_flush_next_imm_memtable` 和 `MemTable::flush`。在 `LSMStorageInner::open` 中，如果 LSM 数据库目录不存在，你需要创建它。要将内存表刷新到磁盘，我们需要做三件事：

- 选择一个要刷新的内存表。
- 为内存表创建一个对应的 SST 文件。
- 从不可变memtable列表中删除内存表，并将SST文件添加到L0 SSTs中。

目前我们还没有解释L0（0级）SSTs是什么。一般来说，它们是作为内存表刷新（memtable flush）的直接结果直接创建的SSTs文件集。在本课程的第一周，我们只在磁盘上会有L0 SSTs。在第二周，我们将深入研究如何在磁盘上使用分层（leveled）或分层（tiered）结构高效地组织它们。

注意，创建SST文件是一个计算密集且耗时的操作。同样，我们不想长时间持有 `state` 读/写锁，因为它可能会阻塞其他操作并在LSM操作中产生巨大的延迟峰值。此外，我们使用 `state_lock` 互斥锁来序列化LSM树中的状态修改操作。在这个任务中，你需要仔细思考如何使用这些锁，以使LSM状态修改无竞态条件，同时最小化临界区。

我们没有并发测试用例，你需要仔细思考你的实现。此外，请记住，不可变memtable列表中的最后一个memtable是最早的，也是你应该刷新的那个。

► 剧透：刷新L0伪代码

任务 2：刷新触发器

在这个任务中，你需要修改：

```
src/lsm_storage.rs
src/compact.rs
```

当内存中的memtable（不可变+ 可变）数量超过`num_memtable_limit` LSM存储选项中的数量时，你应该将最早的memtable刷新到磁盘。这是由后台的刷新线程完成的。刷新线程将使用 `MiniLSM` 结构启动。我们已经实现了启动线程和正确停止线程的必要代码。

在这个任务中，你需要实现 `LsmStorageInner::trigger_flush` 在 `compact.rs` 中，以及 `MiniLsm::close` 在 `lsm_storage.rs` 中。`trigger_flush` 将每 50 毫秒执行一次。如果内存表数量超过限制，你应该调用 `force_flush_next_imm_memtable` 来刷新一个内存表。当用户调用 `close` 函数时，你应该等待刷新线程（以及第 2 周的压缩线程）完成。

任务 3：过滤 SSTs

现在你已经有一个完全可用的存储引擎，并且可以使用 `mini-lsm-cli` 来与你的存储引擎交互。

```
cargo run --bin mini-lsm-cli -- --compaction none
```

然后，

```
fill 1000 3000
get 2333
flush
fill 1000 3000
get 2333
flush
get 2333
scan 2000 2333
```

如果你填充更多数据，你可以看到你的刷新线程在工作，并自动刷新 L0 SSTs 而不使用 `flush` 命令。

最后，在我们结束这周之前，让我们在过滤 SSTs 之前实现一个简单的优化。根据用户提供的键范围，我们可以轻松地过滤掉一些不包含键范围的 SSTs，这样我们就不需要在合并迭代器中读取它们。

在这个任务中，你需要修改：

```
src/lsm_storage.rs
src/iterators/*
src/lsm_iterator.rs
```

你需要更改你的读取路径函数以跳过不可能包含键/键范围的 SSTs。你需要为你的迭代器实现 `num_active_iterators`，以便测试用例可以检查你的实现是否正确。对于 `MergeIterator` 和 `TwoMergeIterator`，它是所有子迭代器 `num_active_iterators` 的总和。请注意，如果你没有修改 `MergeIterator` 启动代码中的字段，请记得也要考虑 `MergeIterator::current`。对于 `LsmIterator` 和 `FusedIterator`，只需从内部迭代器返回活动迭代器的数量。

您可以实现如 `range_overlap` 和 `key_within` 的辅助函数来简化您的代码。

测试您的理解

- 如果用户两次请求删除一个键，会发生什么？
- 迭代器初始化时，同时会加载多少内存（或块的数量）？
- 一些疯狂的用户想要分叉他们的 LSM 树。他们想要启动引擎来摄取一些数据，然后分叉它，这样他们就能得到两个相同的数据集然后进行操作

将它们分开处理。一种简单但效率不高的实现方式是简单地将所有SSTs和内存结构复制到新目录中并启动引擎。但是请注意，我们从不修改磁盘文件，实际上我们可以重用父引擎的SST文件。你认为如何才能高效地实现这种分支功能而不复制数据？（查看Neon分支功能）。

- 想象你正在构建一个多租户LSM系统，在单个128GB内存的机器上托管10k个数据库。Memtable大小限制设置为256MB。对于这个配置，你需要多少内存用于Memtable？
 - 显然，你并没有足够的内存来容纳所有这些内存表。假设每个用户仍然有自己的内存表，你如何设计内存表刷新策略才能使其工作？是否合理让所有这些用户共享同一个内存表（即，通过将租户ID编码为键前缀）？

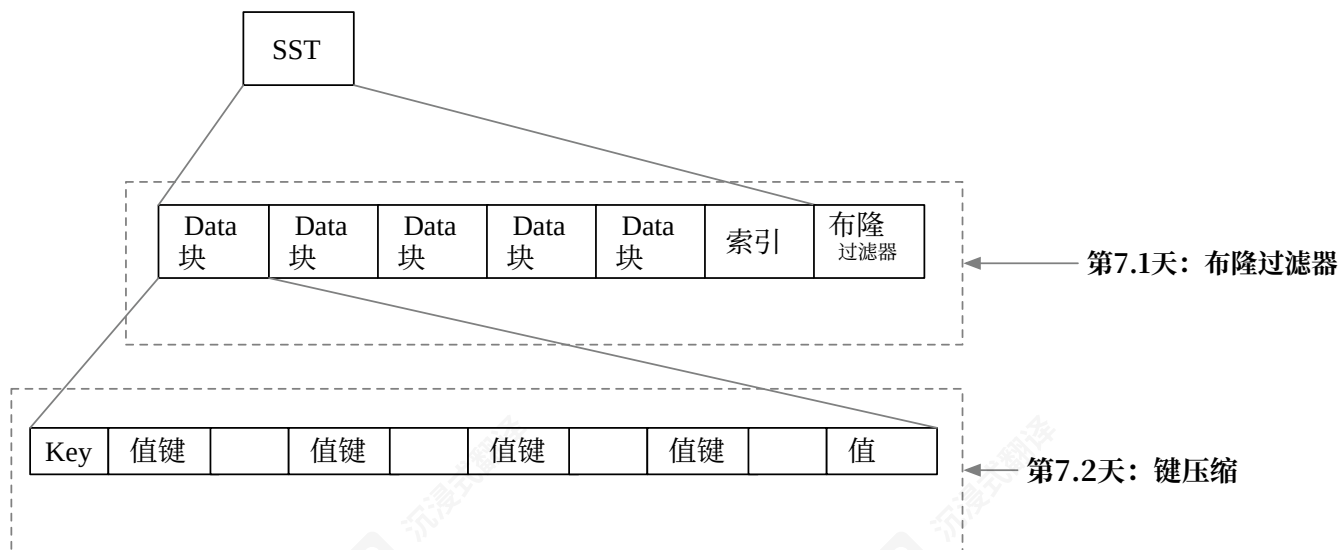
我们不会提供这些问题的参考答案，欢迎在Discord社区中讨论。

奖励任务

- 实现Write/L0停滞。当内存表数量超过最大数量太多时，你可以阻止用户向存储引擎写入。你也可以在实现压缩后，在第2周为L0表实现写入停滞。
- 前缀扫描。你可以通过实现前缀扫描接口并使用前缀信息来过滤更多的SST。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由 Alex Chi Z 著作，根据 CC BY-NC-SA 4.0 授权许可。

零食时间：SST优化



在上一章节中，您已经构建了一个支持 get/scan/put 的存储引擎。在本周结束时，我们将实现一些简单但重要的SST格式优化。欢迎参加 Mini-LSM 的第1周零食时间！

在本章节中，你将：

- 在SSTs上实现布隆过滤器，并将其集成到LSM读路径 `get`。
- 在SST块格式中实现键压缩。

要复制测试用例到入门代码并运行它们，

```
cargo x copy-test --week 1 --day 7
cargo x scheck
```

任务 1：布隆过滤器

布隆过滤器是一种概率性数据结构，用于维护一组键。你可以向布隆过滤器添加键，并且可以知道哪些键可能存在于/必须不存在于添加到布隆过滤器的键集中。

通常需要一个哈希函数来构建布隆过滤器，并且一个键可以有多个哈希值。让我们看一下下面的示例。假设我们已经有一些键的哈希值，并且布隆过滤器有7位。

[注意：如果你想更好地理解布隆过滤器，请查看]

```

hash1 = ((character - a) * 13) % 7
hash2 = ((character - a) * 11) % 7
b -> 6 4
c -> 5 1
d -> 4 5
e -> 3 2
g -> 1 3
h -> 0 0

```

如果我们向7位布隆过滤器插入b、c、d，我们将获取：

```

bit 0123456
insert b    1 1
insert c  1  1
insert d    11
result  0101111

```

当探测布隆过滤器时，我们为键生成哈希值，并查看相应的位是否已被设置。如果所有位都设置为 true，则该键可能存在于布隆过滤器中。否则，该键必定不存在于布隆过滤器中。

对于 `e -> 3 2`，由于位 2 没有被设置，它不应该在原始集合中。对于 `g -> 1 3`，因为两个位都被设置，它可能存在于集合中，也可能不存在。对于 `h -> 0 0`，两个位（实际上是一个位）都没有被设置，因此它不应该在原始集合中。

```

b -> maybe (actual: yes)
c -> maybe (actual: yes)
d -> maybe (actual: yes)
e -> MUST not (actual: no)
g -> maybe (actual: no)
h -> MUST not (actual: no)

```

请记住，在上一章节的结尾，我们基于键范围实现了 SST 过滤。现在，在 `get` 读取路径上，我们也可以使用布隆过滤器来忽略不包含用户想要查找的键的 SST，从而减少从磁盘读取的文件数量。

在这个任务中，你需要修改：

```
src/table/bloom.rs
```

在实现中，您将根据键哈希（键哈希是u32数字）构建一个布隆过滤器。对于每个哈希值，您需要设置 `k` 位。这些位是通过：

```

let delta = (h >> 17) | (h << 15); // h is the key hash
for _ in 0..k {
    // TODO: use the hash to set the corresponding bit
    h = h.wrapping_add(delta);
}

```

我们提供所有用于进行魔法数学的骨架代码。您只需要实现构建布隆过滤器和探测布隆过滤器的过程。

任务 2：在读取路径上集成布隆过滤器

在这个任务中，你需要修改：

```
src/table/builder.rs
src/table.rs
src/lsm_storage.rs
```

对于布隆过滤器编码，你可以将布隆过滤器附加到你的 SST 文件的末尾。你需要将布隆过滤器偏移量存储在文件的末尾，并相应地计算元偏移量。

Block Section				Meta Section			
data block ... data block metadata meta block offset bloom filter							
bloom filter offset							
				varlen		u32	
u32						varlen	

我们使用 `farmhash crate` 来计算键的哈希值。在构建 SST 时，你也需要通过使用 `farmhash` 来构建布隆过滤器。

`farmhash::fingerprint32`。你需要使用块元数据对布隆过滤器进行编码/解码。你可以选择布隆过滤器的假阳性率为 0.01。除了 `starter code` 中提供的那些结构之外，根据需要，你可能还需要在结构中添加新字段。

之后，你可以修改 `get` 读取路径 根据布隆过滤器过滤 SSTs。

我们对此部分没有集成测试，你需要确保你的实现仍然通过所有之前的章节测试。

任务 3：键前缀编码 + 解码

在这个任务中，你需要修改：

```
src/block/builder.rs
src/block/iterator.rs
```

由于SST文件按顺序存储键，用户可能会存储具有相同前缀的键，因此我们可以在SST编码中压缩前缀以节省空间。

我们将当前键与块中的第一个键进行比较。我们按以下方式存储键：

```
key_overlap_len (u16) | rest_key_len (u16) | key (rest_key_len)
```

The `key_overlap_len` 指示有多少字节与块中的第一个键相同。例如，如果我们看到一个记录：`5|3|LSM`，其中块中的第一个键是 `mini-something`，我们可以恢复当前键为 `m` `SM`。

编码完成后，您还需要在块迭代器中实现解码。根据需要，您可能需要在 `starter code` 提供的结构之外向结构中添加新字段。

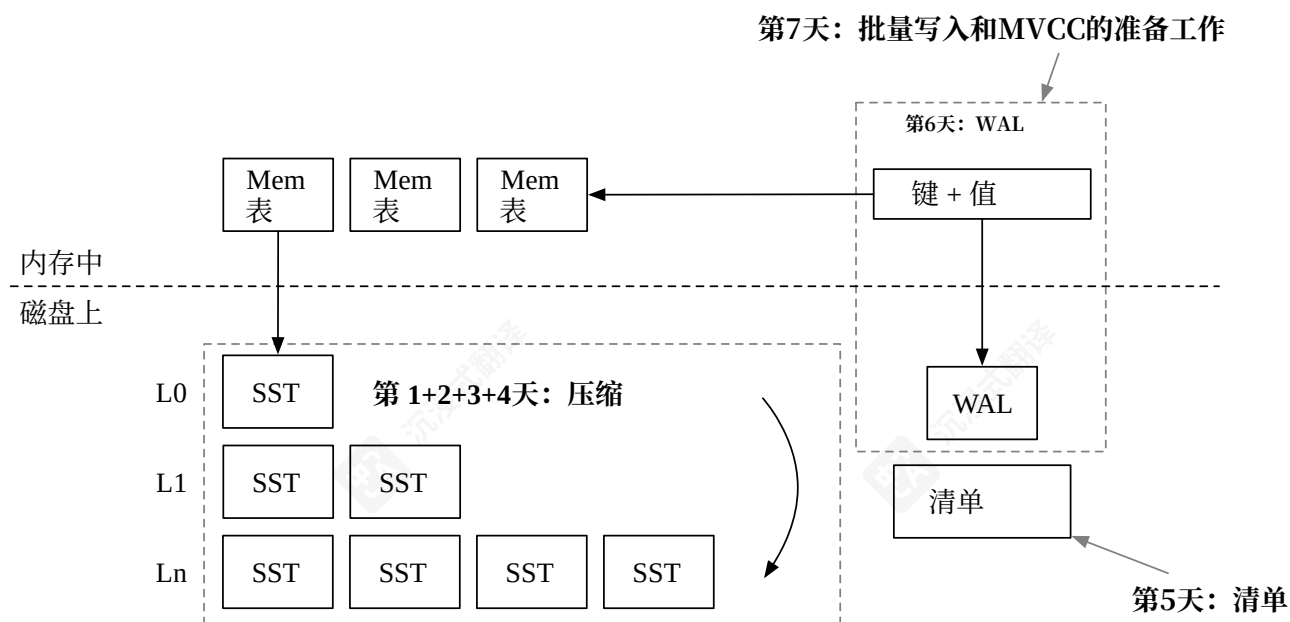
测试您的理解

- 布隆过滤器如何帮助 SST 过滤过程？它关于键能告诉您什么信息？（可能不存在/可能存在/必须存在/必须不存在）
- 考虑我们需要反向迭代器的情况。我们的键压缩是否会影响反向迭代器？
- 你能使用布隆过滤器进行扫描吗？
- 使用键前缀编码而不是使用块中的第一个键来处理相邻键的优缺点可能是什么？

我们不提供问题的参考答案，并且欢迎在Discord社区中讨论它们。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。

第二周概述：合并操作和持久化



在上周，您已经实现了LSM存储引擎的所有必要结构，并且您的存储引擎已经支持读写接口。在本周，我们将深入探讨SST文件的磁盘组织，并研究在系统中实现性能和成本效率的最佳方法。我们将用4天时间学习不同的压缩策略，从最简单的到最复杂的，然后实现存储引擎持久性的剩余部分。在本周结束时，您将拥有一个功能完整且高效的LSM存储引擎。

这部分有7个章节（天）：

- 第一天：压缩实现。您将合并所有L0 SSTs到一个有序运行。
- 第2天：简单分层压缩。您将实现一个经典的分层压缩算法，并使用压缩模拟器来查看它的工作效果。
- 第3天：分层/通用压缩。您将实现RocksDB通用压缩算法，并了解其优缺点。
- 第4天：分层压缩。您将实现RocksDB分层压缩算法。这种压缩算法也支持部分压缩，以减少峰值空间使用。
- 第5天：清单。您将把LSM状态存储在磁盘上并从中恢复。
- 第6天：预写日志 (WAL)。用户请求将被路由到内存表和预写日志 (WAL)，以便所有操作都将被持久化。
- 第7天：写入批处理和校验和。您将实现写入批处理API（为第3周MVCC做准备）并为所有存储格式添加校验和。

压缩和读取放大

让我们先谈谈压缩。在上一部分，您只需将内存表刷新到L0 SST。想象一下，您已经写入了数GB的数据，现在有100个SST。每个读取请求（无过滤）都需要从这些SST中读取100个块。这

放大是读取放大——即你需要向磁盘发送的I/O请求的数量对于一次获取操作。

为了减少读取放大，我们可以将所有L0 SSTs合并到一个更大的结构中，这样就可以只读取一个SST和一个块来检索请求的数据。假设我们仍然有这些100个SSTs，现在，我们对这100个SSTs进行归并排序，以生成另外100个SSTs，每个SST包含不重叠的键范围。这个过程是压缩，而这100个不重叠的SSTs是一个有序运行。

为了使这个过程更清晰，让我们来看一个具体的例子：

```
SST 1: key range 00000 - key 10000, 1000 keys  
SST 2: key range 00005 - key 10005, 1000 keys  
SST 3: key range 00010 - key 10010, 1000 keys
```

我们在LSM结构中有3个SSTs。如果我们需要访问键02333，我们将需要探测所有这3个SSTs。如果我们能进行压缩，我们可能会得到以下3个新的SSTs：

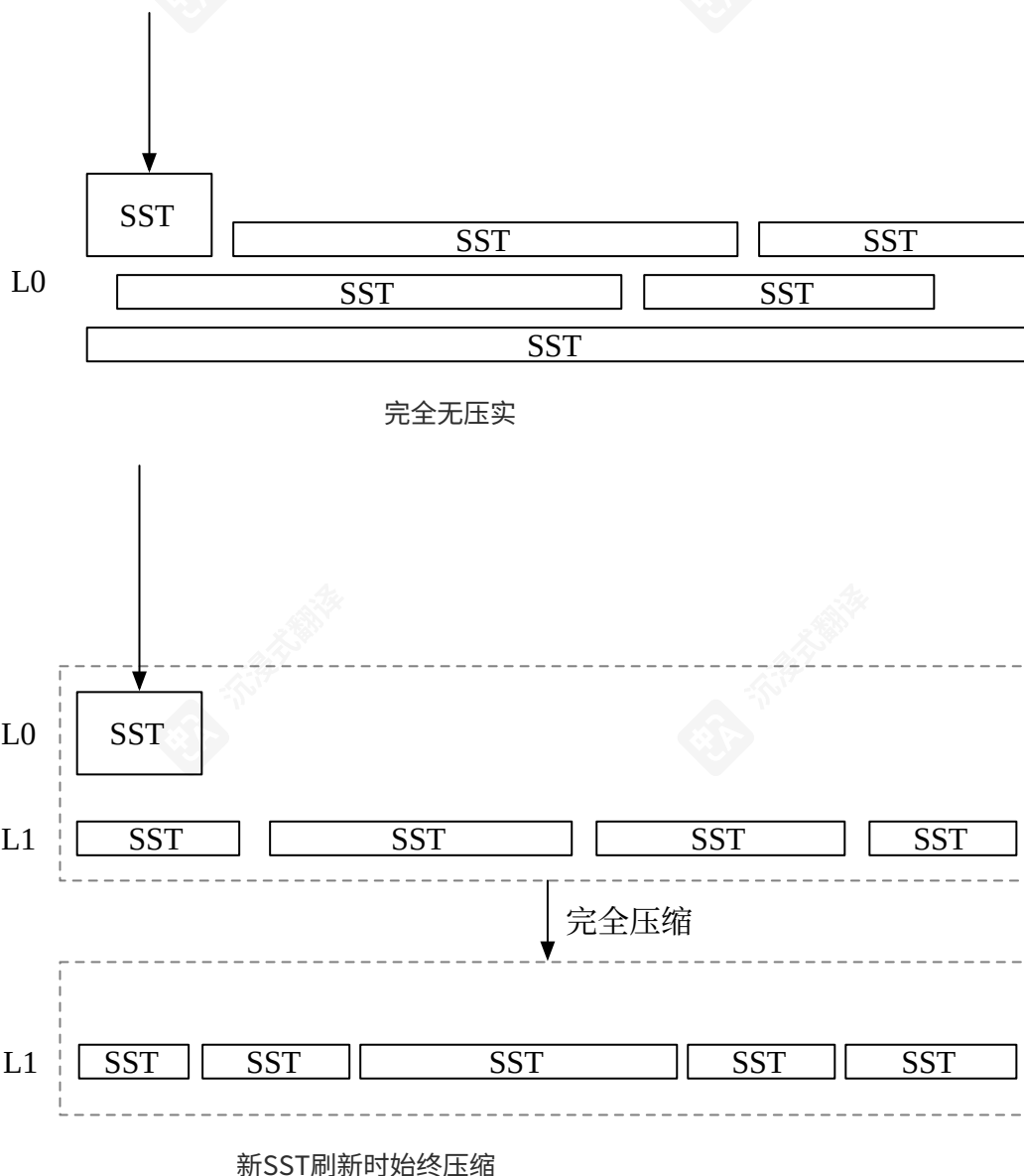
```
SST 4: key range 00000 - key 03000, 1000 keys  
SST 5: key range 03001 - key 06000, 1000 keys  
SST 6: key range 06000 - key 10010, 1000 keys
```

3个新的SST是通过合并SST 1、2和3创建的。我们可以获取一个排序后的3000个键，然后将它们分成3个文件，以避免出现超级大的SST文件。现在我们的LSM状态有3个不重叠的SST，我们只需要访问SST 4来查找键02333。

压缩和写入放大的两个极端

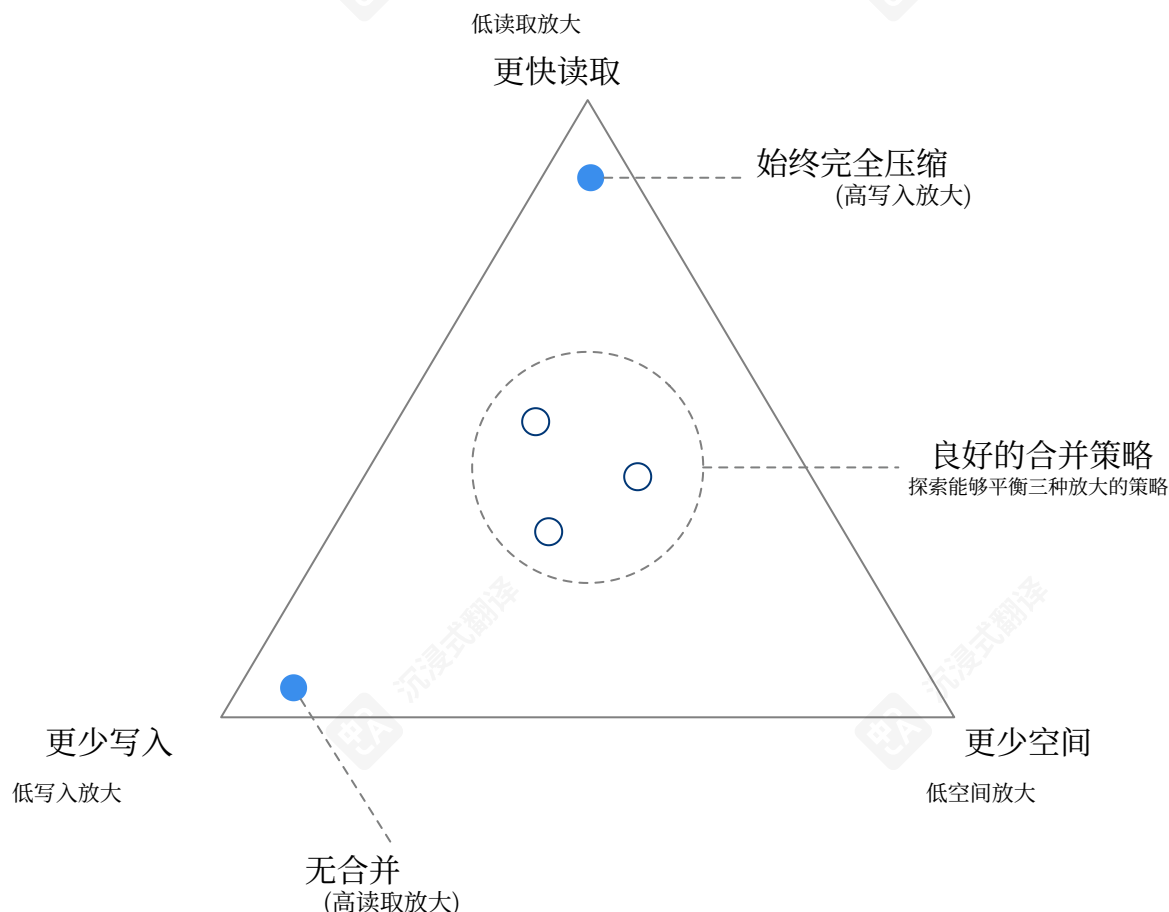
因此，从上面的例子来看，我们处理LSM结构有两种简单的方法——完全不进行压缩，以及在新SST刷新时始终进行完全压缩。

压缩是一个耗时的操作。它需要从一些文件中读取所有数据，并将相同数量的文件写入磁盘。此操作需要大量的CPU资源和I/O资源。完全不进行压缩会导致读取放大，但它不需要写入新文件。始终进行完全压缩可以减少读取放大，但它需要不断重写磁盘上的文件。



内存表刷新到磁盘与写入磁盘的总数据之比是写入放大。也就是说，无合并的写入放大比率为1x，因为一旦SST被刷新到磁盘，它们将保留在那里。始终执行合并的写入放大非常高。如果我们每次获取一个SST都执行完全压缩，写入磁盘的数据将是刷新的SST数量的平方。例如，如果我们向磁盘刷新了100个SST，我们将执行2个文件、3个文件、...、100个文件的合并，其中实际写入磁盘的数据量约为5000个SST。在这种情况下，写入100个SST后的写入放大将是50x。

一个好的压缩策略可以平衡读取放大、写入放大和空间放大（我们很快会谈及这一点）。在通用LSM存储引擎中，通常不可能找到一个能在这三个因素中实现最低放大的策略，除非引擎可以利用某些特定的数据模式。LSM的好处在于，我们可以理论上分析压缩策略的放大情况，而所有这些操作都在后台发生。我们可以选择压缩策略并动态改变它们的某些参数，以调整存储引擎到最佳状态。压缩策略都是关于权衡的，基于LSM的存储引擎使我们能够在运行时选择要权衡的内容。



业界一种典型的工作负载是：用户首先将数据批量导入存储引擎，通常每秒吉字节，当产品启动时。然后，系统上线，用户开始在系统上执行小事务。在第一阶段，引擎应该能够快速导入数据，因此我们可以使用最小化写入放大的压缩策略来加速此过程。然后，我们调整压缩算法的参数，使其优化读取放大，并进行完全压缩以重新排序现有数据，以便系统在上线时能够稳定运行。

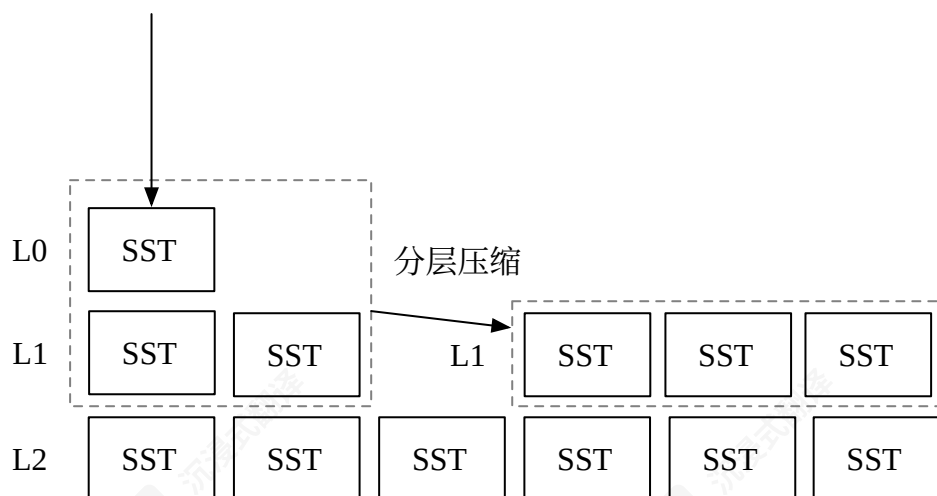
如果工作负载类似于时间序列数据库，那么用户可能总是按时间填充和截断数据。因此，即使没有压缩，这些仅追加的数据在磁盘上仍然可以有较低的放大。因此，在现实生活中，您应该关注用户的行为模式或特定需求，并使用这些信息来优化您的系统。

压缩策略概述

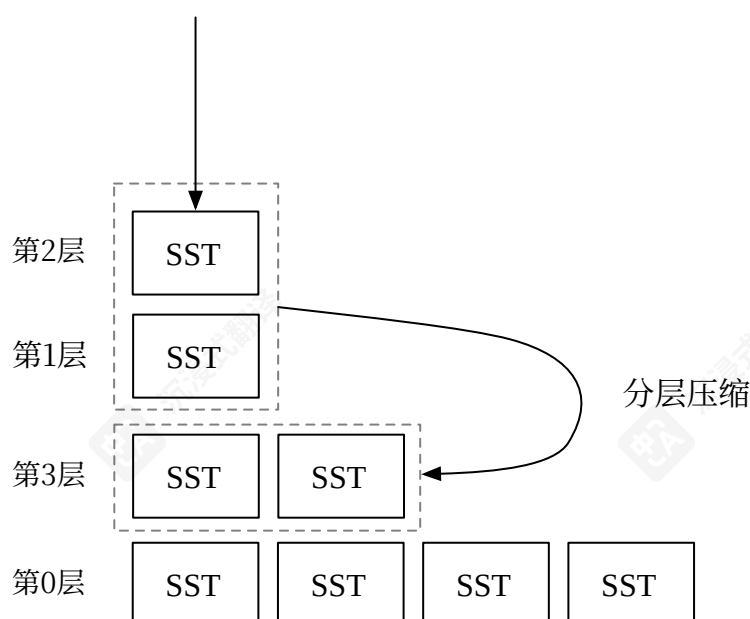
压缩策略通常旨在控制有序运行的数量，以保持读取放大在合理的数量范围内。压缩策略通常分为两种类型：分层和分层。

在分层压缩中，用户可以指定最大层数，这是系统中的排序运行数量（不包括L0）。例如，RocksDB通常在分层压缩模式下保持6层（排序运行）。在压缩过程中，来自两个相邻层级的SST将被合并，然后生成的SST将被放入这两个层级的较低层级。因此，您通常会在分层压缩中看到一个小排序运行与一个大排序运行合并。排序运行（层级）的大小呈指数级增长——较低层级的大小将是上层大小的 $\{v1\}$ 。

在分层压缩中，来自两个相邻层级的SST将被合并，然后生成的SST将被放入这两个层级的较低层级。因此，您通常会在分层压缩中看到一个小排序运行与一个大排序运行合并。排序运行（层级）的大小呈指数级增长——较低层级的大小将是上层大小的 `<some number>`。



在分层压缩中，引擎将通过合并它们或让新的SSTs刷新为新的有序运行（一层）来动态调整有序运行的数量，以最小化写入放大。在此策略中，您通常会看到引擎合并两个大小相等的有序运行。如果压缩策略选择不合并层，层数可能会很高，从而导致读取放大很高。在本课程中，我们将实现RocksDB的通用压缩，这是一种分层压缩策略。



空间放大

计算空间放大的最直观方法是除以LSM引擎实际使用的空间和用户空间使用量（即数据库大小、数据库中的行数等）。如果压缩不频繁发生，引擎将需要存储删除墓碑，并且有时同一键的多个版本，从而导致空间放大。

如果压缩不频繁发生，引擎将需要存储删除墓碑，并且有时同一键的多个版本，从而导致空间放大。

在引擎端，通常很难知道用户存储的确切数据量，除非我们扫描整个数据库并查看引擎中有多少过期版本。因此，估计空间放大的一个方法是除以完整存储文件大小和最后一层大小。这种估计方法背后的假设是，在用户填充初始数据后，工作负载的插入和删除率应该相同。我们假设用户端数据大小不会改变，因此最后一层包含某个时间点的用户数据快照，上层包含新的更改。当压缩将所有内容合并到最后一层时，我们可以使用这种估计方法得到1x的空间放大因子。

请注意，压缩也会占用空间——你不能在压缩完成之前删除正在压缩的文件。如果你对数据库进行完全压缩，你需要与当前引擎文件大小相同的可用存储空间。

在本部分，我们将提供一个压缩模拟器来帮助您可视化压缩过程和您的压缩算法的决策。我们提供最小的测试用例来检查您的压缩算法的特性，您应该密切关注统计信息和压缩模拟器的输出，以了解您的压缩算法的效果如何。

持久性

在实现合并算法后，我们将实现系统中的两个关键组件：清单，它是一个存储LSM状态的文件，以及WAL，它将内存表数据持久化到磁盘上，然后再将其刷新为SST。完成这两个组件后，存储引擎将具有完整的持久性支持，并可用于您的产品。

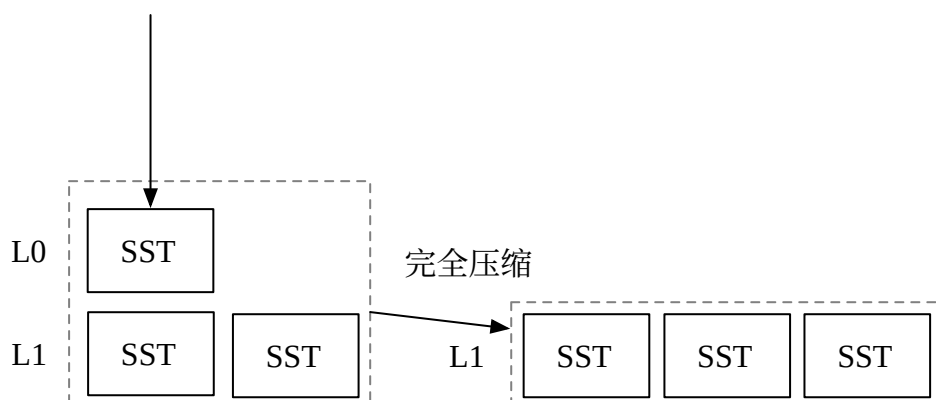
如果您不想深入合并，您也可以完成第2.1和2.2章节，实现一个非常简单的分层压缩算法，然后直接进入持久性部分。实现完整的分层压缩和通用压缩并非在第2周构建可工作的存储引擎所必需的。

零食时间

在实现合并操作和持久化之后，我们将有一个关于实现批量写入接口和校验和的短章节。

非常感谢您的反馈。欢迎加入我们的Discord社区。发现问题时？请在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z拥有，根据CC BY-NC-SA 4.0许可协议授权。

压缩实现




在本章节中，你将：

- 实现压缩逻辑，该逻辑组合一些文件并生成新文件。
- 实现更新LSM状态的逻辑，并在文件系统上管理SST文件。
- 更新LSM读取路径以包含LSM层级。

将测试用例复制到入门代码中并运行它们，

```
cargo x copy-test --week 2 --day 1
cargo x scheck
```

 在阅读本章之前，查看第2周概述可能会有所帮助，以便对压缩有一个总体了解。

任务 1：压缩实现

在这个任务中，您将实现执行压缩的核心逻辑——将一组SST文件归并排序成一个有序运行。您需要修改：

`src/compact.rs`

具体来说，是 `force_full_compaction` 和 `compact` 函数。`force_full_compaction` 是压缩触发器，它决定要压缩哪些文件并更新LSM状态。`compact` 执行实际的压缩工作，将一些SST文件合并并返回一组新的SST文件。

您的压缩实现应该获取存储引擎中的所有SST文件，使用 `MergeIterator` 对它们进行归并，然后使用SST构建器将结果写入新文件。如果文件太大，您需要将SST文件拆分。压缩完成后，您可以更新LSM状态，将所有新的有序运行添加到LSM树的第一级。此外，您还需要删除LSM树中未使用的文件。在您的

实现，你的SSTs应该只存储在两个地方：L0 SSTs和L1 SSTs。也就是说，LSM状态中的 `levels` 结构应该只有一个向量。在 `LsmStorageState` 中，我们已经初始化LSM，使其在 `levels` 字段中具有L1。

压缩不应阻塞L0刷新，因此您在合并文件时不应获取状态锁。您应该仅在压缩过程结束时更新LSM状态时获取状态锁，并在修改状态后立即释放锁。

您可以假设用户将确保只有一个压缩正在进行。

`force_full_compaction` 在任何时候都只会在一个线程中被调用。级别1中要放入的SSTs应按它们的第一个键排序，并且不应有重叠的键范围。

► 剧透：压缩伪代码

在您的压缩实现中，您目前只需要处理 `FullCompaction`，其中任务信息包含您需要压缩的SSTs。您还需要确保SSTs的顺序正确，以便键的最新版本被放入新的SST中。

因为我们总是压缩所有SSTs，如果我们发现一个键的多个版本，我们可以简单地保留最新版本。如果最新版本是删除标记，我们不需要将其保留在生成的SST文件中。这不适用于接下来几章节中的压缩策略。

有一些事情你可能需要考虑。

- 你的实现如何并行处理L0刷新与压缩？（在执行压缩时不获取状态锁，并且还需要考虑压缩过程中产生的新L0文件。）
- 如果你的实现完成压缩后立即删除原始SST文件，是否会在你的系统中引起问题？（通常在macOS/Linux上不会，因为操作系统实际上只有在没有文件句柄被持有时才会删除文件。）

任务 2: Concat 迭代器

在这个任务中，你需要修改，

```
src/iterators/concat_iterator.rs
```

现在你已经在你的系统中创建了有序运行，可以对读取路径进行一个简单的优化。你并不总是需要为你的SST创建合并迭代器。如果SST属于一个有序运行，你可以创建一个concat迭代器，该迭代器按顺序迭代每个SST中的键，因为一个有序运行中的SST不包含重叠的

键范围，并且它们按第一个键排序。我们不想预先创建所有SST迭代器（因为这会导致一个块读取），因此我们只在这个迭代器中存储SST对象。

任务 3：与读取路径集成

在这个任务中，您需要修改，

```
src/lsm_iterator.rs
src/lsm_storage.rs
src/compact.rs
```

现在我们已经有了您的 LSM 树的两级结构，并且您可以更改您的读取路径以使用新的 concat 迭代器来优化读取路径。

您需要更改 `LsmStorageIterator` 的内部迭代器类型。之后，您可以构建一个合并内存表和 L0 SST 的双合并迭代器，以及另一个合并该迭代器与 L1 concat 迭代器的合并迭代器。

您还可以更改您的压缩实现以利用 concat 迭代器。

您需要为 concat 迭代器实现 `num_active_iterators`，以便测试用例可以测试您的实现是否使用了 concat 迭代器，并且它应该始终为 1。

要交互式地测试您的实现，

```
cargo run --bin mini-lsm-cli-ref -- --compaction none # reference solution
cargo run --bin mini-lsm-cli -- --compaction none # your solution
```

然后，

```
fill 1000 3000
flush
fill 1000 3000
flush
full_compaction
fill 1000 3000
flush
full_compaction
get 2333
scan 2000 2333
```

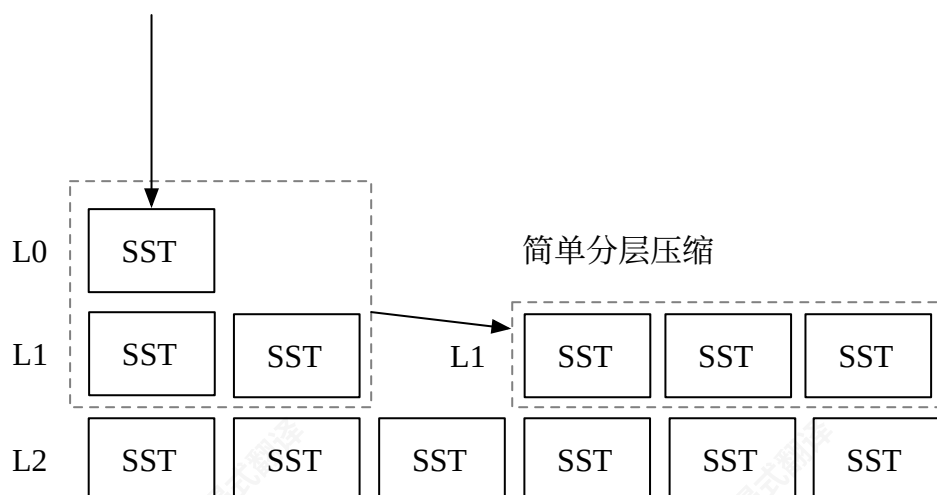
测试您的理解

- 读/写/空间放大的定义是什么？(这在前置章节中有所涵盖)
- 如何准确计算读/写/空间放大，以及如何估计它们？
- 删除键时，即使用户请求删除，键是否仍然会占用一些存储空间？
- 由于压缩会消耗大量的写入带宽和读取带宽，并可能干扰前台操作，因此当存在大量写入流量时，推迟压缩是一个好主意。在这种情况下，甚至停止/暂停现有的压缩任务也是有利的。您对此有何看法？（请阅读 [SILK：防止日志结构合并键值存储中的延迟峰值论文](#)！）
- 使用/填充块缓存进行压缩是一个好主意吗？或者压缩时完全绕过块缓存更好？
- 在系统中拥有一个 `struct ConcatIterator<I: StorageIterator>` 有意义吗？
- 一些研究人员/工程师建议将压缩任务卸载到远程服务器或无服务器lambda函数。这样做的好处是什么？以及远程压缩可能面临的潜在挑战和性能影响是什么？（想想压缩完成时的那个时刻，以及下一次读取请求时块缓存会发生什么...）

我们不会提供这些问题的参考答案，欢迎在Discord社区中讨论。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z发布，根据CC BY-NC-SA 4.0协议授权。

简单压缩策略



在本章节中，您将：

- 实现一个简单分层压缩策略，并在压缩模拟器上模拟它。
- 将压缩作为后台任务启动，并在系统中实现一个压缩触发器。

要复制测试用例到入门代码并运行它们，

```
cargo x copy-test --week 2 --day 2
cargo x scheck
```

i 在阅读本章之前，查看第2周概述可能会有所帮助，以便对压缩有一个总体了解。

任务 1：简单分层压缩

在本章节中，我们将实现第一个压缩策略——简单分层压缩。在这个任务中，你需要修改：

```
src/compact/simple_levelled.rs
```

简单分层压缩与原始LSM论文中的压缩策略类似。它为LSM树维护多个级别。当一个级别 (\geq L1) 过大时，它将合并该级别所有的SSTs与下一级别。压缩策略由

`SimpleLevelledCompactionOptions` 中定义的3个参数控制。

- `size_ratio_percent` : 较低级别文件数量 / 较高级别文件数量。实际上，我们应该计算文件的实际大小。但是，我们简化了

使用文件数量来简化模拟的公式。当比率过低（上层文件过多）时，我们应该触发合并。

- `level0_file_num_compaction_trigger`: 当 L0 中的 SST 数量大于或等于此数值时，触发 L0 和 L1 的合并。
- `max_levels`: LSM 树中（不包括 L0）的层级数量。

假设 `size_ratio_percent=200`（较低层级应具有上层 2 倍的文件数量），`max_levels=3`，`level0_file_num_compaction_trigger=2`，让我们看一下下面的示例。

假设引擎刷新了两个 L0 SST。这达到了

`level0_file_num_compaction_trigger`，并且你的控制器应该触发 L0->L1 合并。

```
--- After Flush ---
L0 (2): [1, 2]
L1 (0): []
L2 (0): []
L3 (0): []
--- After Compaction ---
L0 (0): []
L1 (2): [3, 4]
L2 (0): []
L3 (0): []
```

现在，L2 为空而 L1 有两个文件。L1 和 L2 的大小比例百分比为 $(L2/L1) \times$

。因此，我们将触发 L1+L2 合并，将数据推到 L2。L2 和这些 SST 的处理方式相同，经过 2 次合并后，它们将被放置在最底层。

```
--- After Compaction ---
L0 (0): []
L1 (0): []
L2 (2): [5, 6]
L3 (0): []
--- After Compaction ---
L0 (0): []
L1 (0): []
L2 (0): []
L3 (2): [7, 8]
```

继续刷新 SSTs，我们将发现：

```
L0 (0): []
L1 (0): []
L2 (2): [13, 14]
L3 (2): [7, 8]
```

此时， $L3/L2 = (1 / 1) \times 100 = 100 < \text{size_ratio_percent} (200)$ 。因此，我们需要在 L2 和 L3 之间触发合并。

```

--- After Compaction ---
L0 (0): []
L1 (0): []
L2 (0): []
L3 (4): [15, 16, 17, 18]

```

随着我们刷新更多的SSTs，我们可能会达到如下状态：

```

--- After Flush ---
L0 (2): [19, 20]
L1 (0): []
L2 (0): []
L3 (4): [15, 16, 17, 18]
--- After Compaction ---
L0 (0): []
L1 (0): []
L2 (2): [23, 24]
L3 (4): [15, 16, 17, 18]

```

因为 $L3/L2 = (4 / 2) * 100 = 200 \geq \text{size_ratio_percent} (200)$ ，我们不需要合并 L2 和 L3，最终会得到上述状态。简单分层压缩策略总是压缩一个完整级别，并保持层级之间的扇出大小，以便较低层级总是比上层大一些倍数。

我们已经初始化了 LSM 状态，使其具有 `max_level` 层级。你应该首先实现 `generate_compaction_task`，该功能根据上述 3 个标准生成合并任务。之后，实现 `apply_compaction_result`。我们建议你首先实现 L0 trigger，运行压缩模拟，然后实现大小比例触发器，再运行压缩模拟。要运行压缩模拟，

```

cargo run --bin compaction-simulator-ref simple # Reference solution
cargo run --bin compaction-simulator simple # Your solution

```

模拟器会将一个 L0 SST 刷新到 LSM 状态中，运行你的压缩控制器以生成合并任务，然后应用压缩结果。每次新的 SST 被刷新时，它都会重复调用控制器，直到不需要安排压缩，因此你应该确保你的合并任务生成器会收敛。

在你的压缩实现中，你应该尽可能减少活动迭代器的数量（即使用 `concat` 迭代器）。此外，请记住合并顺序很重要，并且你需要确保你创建的迭代器在多个版本的关键字出现时，能够按正确的顺序生成键值对。

此外，请注意实现中的一些参数是 0 基的，而另一些是 1 基的。在使用 `level` 作为向量索引时要小心。

注意：我们不为这部分提供细粒度单元测试。您可以运行压缩模拟器并与参考解决方案的输出进行比较，[以查看](https://skyzh.github.io/mini-lsm/print.html)

你的实现是正确的。

任务 2：压缩线程

在这个任务中，您需要修改：

`src/compact.rs`

现在您已经实现了您的压缩策略，您需要在后台线程中运行它，以便在后台压缩文件。在 `compact.rs` 中，`trigger_compaction` 将每 50 毫秒被调用一次，您需要：

1. 生成一个合并任务，如果没有任务需要调度，则返回 `ok`。
2. 运行压缩并获取新的 SSTs 列表。
3. 与您在前一章中实现的 `force_full_compaction` 类似，更新 LSM 状态。

任务 3：与读取路径集成

在这个任务中，您需要修改：

`src/lsm_storage.rs`

现在您有多个层级的 SST，您可以修改您的读取路径以包含新层级的 SST。您需要更新扫描/获取函数以包含 L1 以下的所有层级。此外，您可能需要再次更改 `LsmStorageIterator` 内部类型。

要交互式地测试您的实现，

```
cargo run --bin mini-lsm-cli-ref -- --compaction simple # reference solution
cargo run --bin mini-lsm-cli -- --compaction simple # your solution
```

然后，

```
fill 1000 3000
flush
fill 1000 3000
flush
fill 1000 3000
flush
get 2333
scan 2000 2333
```

当合并器触发压缩时，您可以打印一些内容，例如合并任务信息。

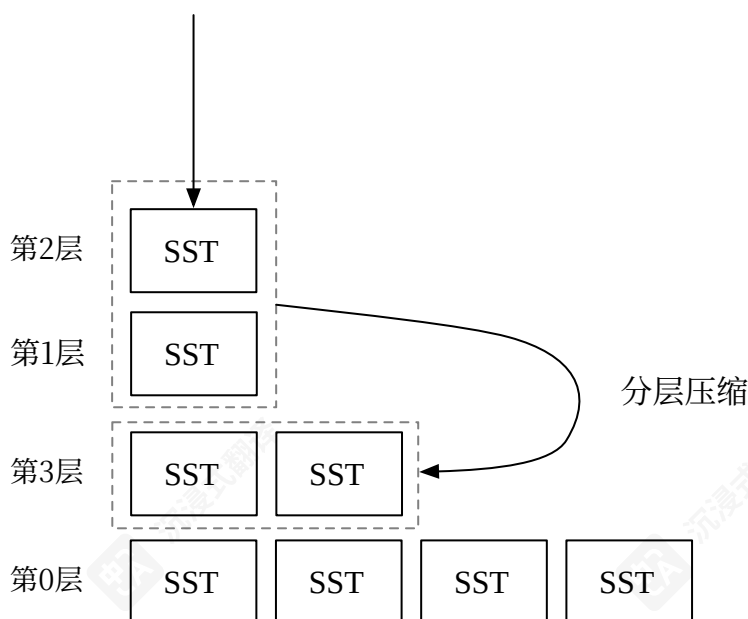
测试您的理解

- 分层压缩的写入放大估计是多少？
- 分层压缩的读取放大估计是多少？
- 键是否只有在用户请求删除并且已在最低级别压缩的情况下才会从LSM树中清除？
- 定期对LSM树进行完全压缩是一种好策略吗？为什么或为什么不？
- 即使它们不违反层级放大器，主动选择一些旧文件/级别进行压缩会是一个好选择，对吗？（看看Lethe论文！）
- 如果存储设备能够实现可持续的1GB/s写入吞吐量，并且LSM树的写入放大为10倍，用户可以从LSM键值接口获取多少吞吐量？
- 如果L2中有SST文件，能否直接合并L1和L3？这仍然会产生正确的结果吗？
- 到目前为止，我们一直假设我们的SST文件使用单调递增ID作为文件名。使用 `<level>_<begin_key>_<end_key>.sst` 作为SST文件名可以吗？那样可能会有什么潜在问题？（你可以在第3周问自己同样的问题……）
- 你所在城市的最喜欢的珍珠奶茶店是什么？（如果你在第1天第3周回答了“是”……）

我们不会提供问题的参考答案，欢迎在Discord社区讨论这些问题。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z发布，根据CC BY-NC-SA 4.0协议授权。

分层压缩策略



在本章节中，您将：

- 实现一个分层压缩策略，并在压缩模拟器上模拟它。将分层压缩策略集成到系统中。
-

本章中我们提到的分层压缩与RocksDB的通用压缩相同。我们将交替使用这两个术语。

要将测试用例复制到入门代码并运行它们，

```
cargo x copy-test --week 2 --day 3
cargo x scheck
```



在阅读本章之前，查看第2周概述可能会有所帮助，以便对压缩有一个总体了解。

任务 1：通用压缩

在本章中，您将实现RocksDB的通用压缩，它属于分层压缩策略家族。与简单分层压缩策略类似，我们仅使用文件数量作为此压缩策略的指标。当我们触发压缩作业时，我们始终在压缩作业中包含一个完整排序运行（层）。

任务 1.0：前提条件

在这个任务中，您需要修改：

```
src/compact/tiered.rs
```

在通用压缩中，我们不使用 LSM 状态下的 L0 SST。相反，我们直接将新的 SST 刷新到一个单一的有序运行（称为层）。在 LSM 状态下，`levels` 将现在包含所有层，其中最低的索引是最新的刷新的 SST。`levels` 向量中的每个元素存储一个元组：层 ID（用作层 ID）和该层中的 SST。每次您刷新 L0 SST 时，您应该将 SST 刷新到向量前端放置的层中。压缩模拟器根据第一个 SST ID 生成层 ID，您在实现中也应该这样做。

通用压缩仅在层数（有序运行）达到 `num_tiers` 时才会触发任务。否则，它不会触发任何压缩。

任务 1.1：由空间放大率触发

通用压缩的第一个触发器是由空间放大率。正如我们在概述章节中讨论的，空间放大可以通过 `engine_size / 估计`

`last_level_size`。在我们的实现中，我们通过 `all`

`levels except last level size / last level size` 计算空间放大率，以便该比率可以扩 `(+inf)` 而不 `if]`。这也与 RocksDB 的实现一致。

我们这样计算空间放大率的原因是因为我们以某种方式对引擎进行建模，使其存储固定数量的用户数据（即假设为 100GB），并且用户通过写入引擎不断更新值。因此，最终所有键都会被推到底层，底层的大小应该等同于数据量（100GB），上层包含尚未压缩到底层的更新数据。

当 `all`

`max_size_amplification_percent * 1%`，我们将需要触发一次完全压缩。例如，如果我们有一个 LSM 状态，如下所示：

```
Tier 3: 1
Tier 2: 1 ; all levels except last level size = 2
Tier 1: 1 ; last level size = 1, 2/1=2
```

假设 `max` 00，我们现在应该触发完全压缩。

在您实现此触发器后，您可以运行压缩模拟器。您将看到：

```
cargo run --bin compaction-simulator tiered --iterations 10
```

```
=== Iteration 2 ===  
--- After Flush ---  
L3 (1): [3]  
L2 (1): [2]  
L1 (1): [1]  
--- Compaction Task ---  
compaction triggered by space amplification ratio: 200  
L3 [3] L2 [2] L1 [1] -> [4, 5, 6]  
--- After Compaction ---  
L4 (3): [3, 2, 1]
```

通过这个触发器，当它达到空间放大率时，我们才会触发完全压缩。在模拟结束时，您将看到：

```
cargo run --bin compaction-simulator tiered
```



```
=== Iteration 7 ===
--- After Flush ---
L8 (1): [8]
L7 (1): [7]
L6 (1): [6]
L5 (1): [5]
L4 (1): [4]
L3 (1): [3]
L2 (1): [2]
L1 (1): [1]
--- Compaction Task ---
--- Compaction Task ---
compaction triggered by space amplification ratio: 700
L8 [8] L7 [7] L6 [6] L5 [5] L4 [4] L3 [3] L2 [2] L1 [1] -> [9, 10, 11, 12, 13,
14, 15, 16]
--- After Compaction ---
L9 (8): [8, 7, 6, 5, 4, 3, 2, 1]
--- Compaction Task ---
1 compaction triggered in this iteration
--- Statistics ---
Write Amplification: 16/8=2.000x
Maximum Space Usage: 16/8=2.000x
Read Amplification: 1x
```

```
=== Iteration 49 ===
--- After Flush ---
L82 (1): [82]
L81 (1): [81]
L80 (1): [80]
L79 (1): [79]
L78 (1): [78]
L77 (1): [77]
L76 (1): [76]
L75 (1): [75]
L74 (1): [74]
L73 (1): [73]
L72 (1): [72]
L71 (1): [71]
L70 (1): [70]
L69 (1): [69]
L68 (1): [68]
L67 (1): [67]
L66 (1): [66]
L65 (1): [65]
L64 (1): [64]
L63 (1): [63]
L62 (1): [62]
L61 (1): [61]
L60 (1): [60]
L59 (1): [59]
L58 (1): [58]
L57 (1): [57]
L33 (24): [32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 9,
10, 11, 12, 13, 14, 15, 16]
--- Compaction Task ---
--- Compaction Task ---
no compaction triggered
```

--- Statistics ---

Write Amplification: $82/50=1.640x$

Maximum Space Usage: $50/50=1.000x$

Read Amplification: $27x$

压缩模拟器中的 `num_tiers` 设置为8。然而，LSM状态中的层远多于8层，这会导致较大的读取放大。

当前的触发器仅减少空间放大。我们需要在合并算法中添加新的触发器来减少读取放大。

任务1.2：由大小比例触发

下一个触发器是大小比例触发器。该触发器维护层之间的大小比例。从第一层开始，我们计算 `this tier / sum of all previous tiers` 的大小。对于第一个遇到的大小值为 $> (100 + \text{size_ratio}) * 1\%$ 的层，我们将合并所有之前的层（不包括当前层）。我们仅在要合并的层多于 `min_merge_width` 层时才进行这种合并。

例如，给定以下LSM状态，并假设 `size_ratio = 1` 和 `min_merge_width = 2`。当，我们应该合并：

```
Tier 3: 1
Tier 2: 1 ; 1 / 1 = 1
Tier 1: 1 ; 1 / (1 + 1) = 0.5, no compaction triggered
```

示例2:

```
Tier 3: 1
Tier 2: 1 ; 1 / 1 = 1
Tier 1: 3 ; 3 / (1 + 1) = 1.5, compact tier 2+3
```

```
Tier 4: 2
Tier 1: 3
```

示例 3:

```
Tier 3: 1Tier 2: 2 ; 2 / 1 = 2, however, it does not make sense to compact only one
tier; also note that min_merge_width=2Tier 1: 4 ; 4 / 3 = 1.33, compact tier 2+3
```

```
Tier 4: 3
Tier 1: 4
```

使用此触发器，您将在压缩模拟器中观察到以下内容：

```
cargo run --bin compaction-simulator tiered
```

```
=== Iteration 49 ===
--- After Flush ---
L119 (1): [119]
L118 (1): [118]
L114 (4): [113, 112, 111, 110]
L105 (5): [104, 103, 102, 101, 100]
L94 (6): [93, 92, 91, 90, 89, 88]
L81 (7): [80, 79, 78, 77, 76, 75, 74]
L48 (26): [47, 46, 45, 44, 43, 37, 38, 39, 40, 41, 42, 24, 25, 26, 27, 28, 29,
30, 9, 10, 11, 12, 13, 14, 15, 16]
--- Compaction Task ---
--- Compaction Task ---
no compaction triggered
--- Statistics ---
Write Amplification: 119/50=2.380x
Maximum Space Usage: 52/50=1.040x
Read Amplification: 7x
```

```
cargo run --bin compaction-simulator tiered --iterations 200 --size-only
```

```
=== Iteration 199 ===
--- After Flush ---
Levels: 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 3 4 5 6 10
15 21 28 78
no compaction triggered
--- Statistics ---
Write Amplification: 537/200=2.685x
Maximum Space Usage: 200/200=1.000x
Read Amplification: 38x
```

1-SST层将更少，压缩算法将维护层的大小，从小到大按大小比例排列。但是，当LSM状态中的SSTs更多时，我们仍然会有超过 `num_tiers` 层的案例。为了限制层数，我们需要另一个触发器。

任务 1.3：减少排序运行

如果之前的触发器都没有产生压缩任务，我们将进行一次主要合并，将前 `max_merge_tiers` 层的SST文件合并成一个层，以减少层数。

启用此压缩触发器后，您将看到：

```
cargo run --bin compaction-simulator-ref tiered --iterations 200 --size-only
```

```
=== Iteration 199 ===
--- After Flush ---
Levels: 0 1 1 4 5 21 28 140
no compaction triggered
--- Statistics ---
Write Amplification: 742/200=3.710x
Maximum Space Usage: 280/200=1.400x
Read Amplification: 7x
```

您也可以尝试使用更多层级的分层压缩：

```
cargo run --bin compaction-simulator tiered --iterations 200 --size-only --num-tiers 16
```

```
=== Iteration 199 ===
--- After Flush ---
Levels: 0 1 1 1 1 1 1 1 1 1 15 175
no compaction triggered
--- Statistics ---
Write Amplification: 607/200=3.035x
Maximum Space Usage: 350/200=1.750x
Read Amplification: 12x
```

注意：我们不为这部分提供细粒度单元测试。您可以运行压缩模拟器，并与参考解决方案的输出进行比较，以查看您的实现是否正确。

任务 2：与读取路径集成

在这个任务中，您需要修改：

```
src/compact.rs
src/lsm_storage.rs
```

分层压缩不使用LSM状态的L0级别，您应该直接刷新

将您的内存表迁移到新层，而不是作为 L0 SST。您可以使用

`self.compaction_controller.flush_to_l0()` 判断是否刷新到L0。您可以使用第一个输出SST ID作为新有序运行的水平/层ID。您还需要修改压缩过程，为分层压缩作业构建合并迭代器。

相关阅读

[通用压缩 - RocksDB维基](#)

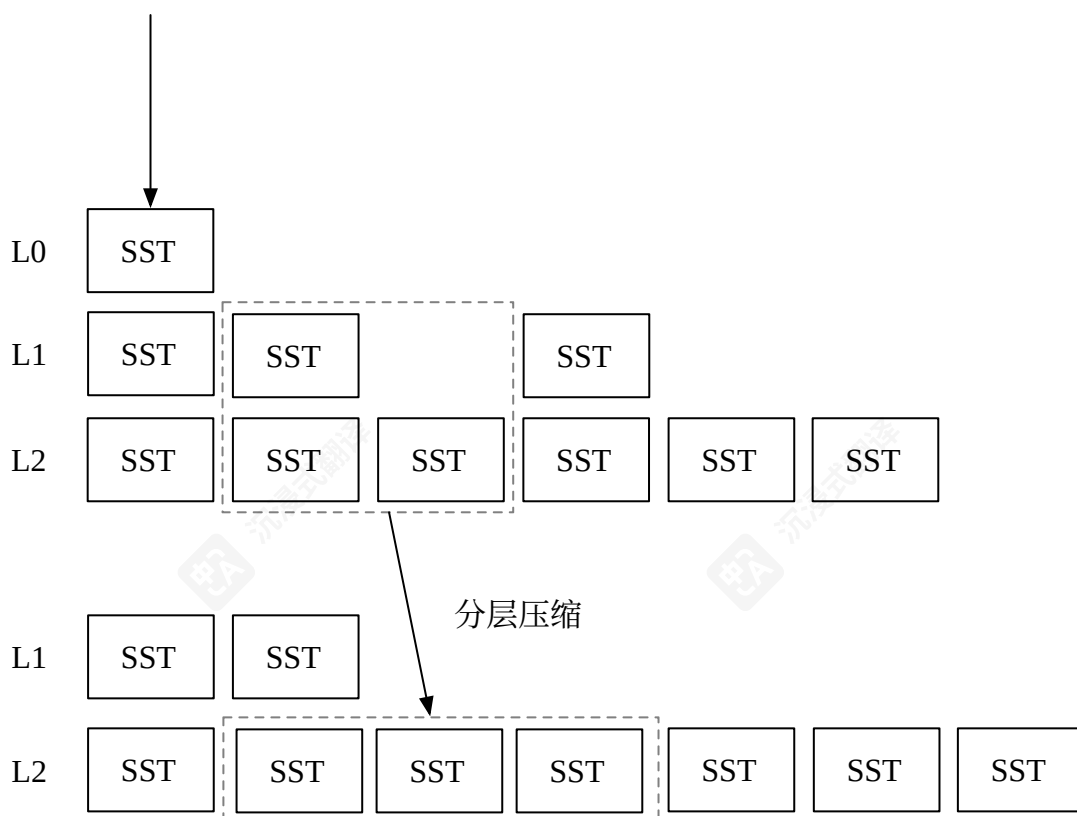
测试您的理解

- 分层压缩的估计写入放大是多少？(好吧，这很难估计...但如果没有最后的reduce有序运行触发器呢？)
- 分层压缩的估计读取放大是多少？
- 与简单的分层/分层压缩相比，通用压缩的优缺点是什么？
- 运行通用压缩需要多少存储空间（与用户数据大小相比）？
- 在LSM状态下，我们可以合并两个不相邻的层吗？
- 如果分层压缩的压缩速度无法跟上SST刷新，会发生什么？
- 如果系统并行调度多个压缩任务，需要考虑哪些因素？
- SSD也会写入自己的日志（基本上它是一种日志结构化存储）。如果SSD的写入放大为2倍，整个系统的端到端写入放大是多少？相关：ZNS：避免闪存固态硬盘的块接口税。
- 考虑用户选择为分层压缩保留大量排序运行（即300）的情况。为了使读取路径更快，保留一些数据结构以帮助减少查找每层中需要读取的SSTs的时间复杂度（即到 $O(\log n)$ ）是否是个好主意？请注意，通常您需要在每个排序运行中执行二分搜索以找到需要读取的键范围。（查看Neon的层映射实现！）

我们不提供问题的参考答案，欢迎在Discord社区中讨论。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z发布，根据CC BY-NC-SA 4.0协议授权。

分层压缩策略




在本章节中，您将：

- 实现分层压缩策略并在压缩模拟器上模拟它。
- 将分层压缩策略集成到系统中。

要复制测试用例到入门代码并运行它们，

```
cargo x copy-test --week 2 --day 4
cargo x scheck
```

 在阅读本章之前，查看第2周概述可能会有所帮助，以便对压缩有一个总体了解。

任务 1：分层压缩

在第2天的第2章节中，您已经实现了简单分层压缩策略。然而，实现中存在一些问题：

- 压缩总是包含一个完整级别。请注意，在完成压缩之前，您无法删除旧文件，因此，在压缩过程中，您的存储引擎可能会使用2倍的存储空间（如果是完全压缩）。分层压缩

存在相同的问题。在这一章节中，我们将实现部分压缩，我们选择从上层选择一个SST进行压缩，而不是完整级别。

- SSTs 可以跨空层级进行压缩。正如你在压缩模拟器中所见，当 LSM 状态为空时，引擎刷新一些 L0 SSTs，这些 SSTs 将首先被压缩到 L1，然后从 L1 到 L2，以此类推。一个最佳策略是直接将 L0 的 SST 放置到可能的最底层，以避免不必要的写入放大。

在本章节中，你将实现一个生产就绪的层级合并策略。该策略与 RocksDB 的层级合并相同。你需要修改：

```
src/compact/levelled.rs
```

要运行压缩模拟器，

```
cargo run --bin compaction-simulator levelled
```

任务 1.1：计算目标大小

在此压缩策略中，你需要知道每个 SST 的第一个/最后一个键以及 SST 的大小。压缩模拟器将为你设置一些模拟的 SSTs 以供访问。

你需要计算每个层级的目标大小。假设 `base_level_size_mb` 是 200MB，并且（除 L0 外）层数为 6。当 LSM 状态为空时，目标大小将是：

```
[0 0 0 0 0 200MB]
```

在底层超出 `base_level_size_mb` 之前，所有其他中间层级的目标大小将为 0。其思路是，当数据总量较小时，创建中间层级是浪费的。

当底层达到或超出 `base_level_size_mb` 时，我们将通过将 `level_size_multiplier` 除以大小来计算其他层级的目标大小。假设底层包含 300MB 的数据，以及 `level_size_multiplier=10`。

```
0 0 0 0 30MB 300MB
```

此外，最多只有一个级别可以具有一个目标大小低于 `base_level_size_mb`。假设我们现在在最后一个级别有 30GB 的文件，目标大小将是，

```
0 0 30MB 300MB 3GB 30GB
```

注意在这种情况下，L1 和 L2 的目标大小为 0，而 L3 是唯一一个目标大小为正且低于 `base_level_size_`

任务 1.2：确定基础层级

现在，让我们解决简单分层压缩策略中 SST 可能跨空层级压缩的问题。当我们用较低层级压缩 L0 SST 时，我们不会直接将其放入 L1。相反，我们会将其与第一个层级 `target size > 0` 进行压缩。例如，当目标层级大小为：

```
0 0 0 0 30MB 300MB
```

如果 L0 SST 的数量达到 `level0_file_num_compaction_trigger` 阈值，我们将用

现在，您可以生成 L0 压缩任务并运行压缩模拟器。

```
--- After Flush ---
```

```
L0 (1): [23]
```

```
L1 (0): []
```

```
L2 (0): []
```

```
L3 (2): [19, 20]
```

```
L4 (6): [11, 12, 7, 8, 9, 10]
```

```
...
```

```
--- After Flush ---
```

```
L0 (2): [102, 103]
```

```
L1 (0): []
```

```
L2 (0): []
```

```
L3 (18): [42, 65, 86, 87, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 61, 62, 52, 34]
```

```
L4 (6): [11, 12, 7, 8, 9, 10]
```

压缩模拟器中的层级数量为 4。因此，SST 应该直接刷新到 L3/L4。

任务 1.3：确定层级优先级

现在我们需要处理低于 L0 的压缩。L0 压缩始终具有最高优先级，因此如果它达到阈值，您应该先与其他层级一起压缩 L0。之后，我们可以通过 `current_size / target_size` 计算每个层级的压缩优先级。我们只压缩比率 `> 1.0` 的层级。比率最大的那个将 `current_size / target_size` 进行压缩。

们有：


```
L3: 200MB, target_size=20MB
L4: 202MB, target_size=200MB
L5: 1.9GB, target_size=2GB
L6: 20GB, target_size=20GB
```

压缩的优先级将是：

```
L3: 200MB/20MB = 10.0
L4: 202MB/200MB = 1.01
L5: 1.9GB/2GB = 0.95
```

L3 和 L4 需要分别与其较低层级进行压缩，而 L5 则不需要。并且 L3 的比率更大，因此我们将生成 L3 和 L4 的合并任务。压缩完成后，我们可能会安排 L4 和 L5 的压缩。

任务 1.4：选择要压缩的 SST

现在，让我们解决简单分层压缩策略中压缩总是包含完整级别的问题。当我们决定压缩两个层级时，我们总是从上层选择最旧的 SST。您可以通过比较 SST ID 来知道 SST 产生的时间。

选择要压缩的 SST 还有其他方法，例如，通过查看删除墓碑的数量。您可以将其作为奖励任务的一部分来实现。

在您选择上层 SST 后，您需要找到所有在较低层级中与上层 SST 键重叠的 SST。然后，您可以生成一个包含一个上层 SST 和重叠的较低层级 SST 的合并任务。

当压缩完成后，您需要从状态中删除 SST，并将新的 SST 插入到正确的位置。请注意，除了 L0 层之外，您应该在所有层级中按第一个键的顺序保持 SST ID 有序。

运行压缩模拟器，您应该看到：

```
--- After Compaction ---
L0 (0): []
L1 (4): [222, 223, 208, 209]
L2 (5): [206, 196, 207, 212, 165]
L3 (11): [166, 120, 143, 144, 179, 148, 167, 140, 189, 180, 190]
L4 (22): [113, 85, 86, 36, 46, 37, 146, 100, 147, 203, 102, 103, 65, 81, 105,
75, 82, 95, 96, 97, 152, 153]
```

各层的大小应保持在级别乘数比率之下。以及合并任务：

```
Upper L1 [224.sst 7cd080e..=33d79d04]
Lower L2 [210.sst 1c657df4..=31a00e1b, 211.sst 31a00e1c..=46da9e43] -> [228.sst
7cd080e..=1cd18f74, 229.sst 1cd18f75..=31d616db, 230.sst 31d616dc..=46da9e43]
```

...应仅从上层有一个SST。

注意：我们不为这一部分提供细粒度单元测试。您可以运行压缩模拟器，并与参考解决方案的输出进行比较，以查看您的实现是否正确。

任务 2：与读取路径集成

在这个任务中，您需要修改：

```
src/compact.rs
src/lsm_storage.rs
```

实现应与简单分层压缩类似。请记住更改获取/扫描读取路径和压缩迭代器。

相关阅读

[分层压缩 - RocksDB维基](#)

测试您的理解

- 分层压缩的写入放大估计是多少？
- 分层压缩的读取放大估计是多少？
- 为压缩找到一个好的键拆分点可能会潜在地减少写入放大，或者根本无关紧要？（考虑用户写入以某些前缀开头的键，00 和 01。这两个前缀下的键数量不同，它们的写入模式也不同。如果我们总能将 00 和 01 分成不同的 SSTs……）
- 想象一个用户之前在使用分层（通用）压缩，现在想迁移到分层压缩。这种迁移可能有哪些挑战？以及如何进行迁移？
- 反过来呢，如果用户想从分层压缩迁移到分层压缩，会怎样？
- 如果分层压缩的压缩速度无法跟上SST刷新，会发生什么？

- 如果系统并行调度多个压缩任务，需要考虑哪些因素？
- 分层压缩的峰值存储使用量是多少？与通用压缩相比呢？
- 是否确实如此：在较低的 `level_size_multiplier` 下，你总能获得较低的写入放大？
- 如果一个用户完全不用压缩，决定迁移到分层压缩，需要做什么？
- 有些人提议在将L0表推送到较低层级之前，先进行L0内部合并（压缩L0表并仍将它们放在L0中）。这样做可能有哪些好处？（可能相关：PebblesDB SOSP'17）
- 考虑这样一种情况：上层有两个 `[100, 200]`, `[201, 300]` 的表，下层有 `[50, 150]`, `[151, 250]`, `[251, 350]`。在这种情况下，你仍然想一次压缩上层的一个文件吗？为什么？

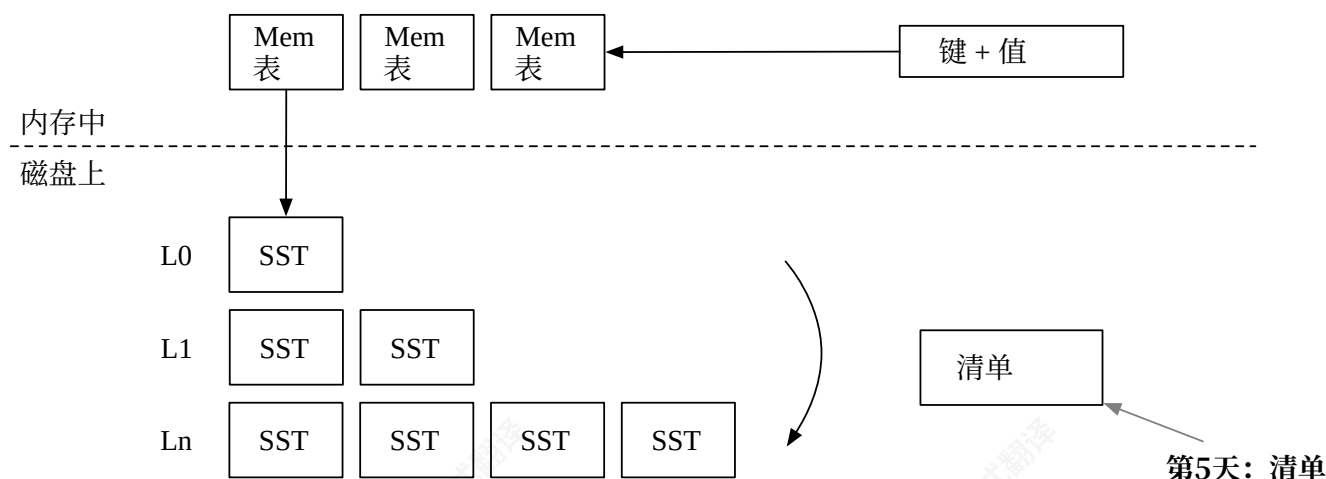
我们不会提供这些问题的参考答案，欢迎在Discord社区中讨论它们。

奖励任务

- SST摄取。在LSM树中的数据迁移/批次导入中，一个常见的优化是让上游生成其数据的SST文件，并将这些文件直接放置在LSM状态中，而无需经过写入路径。
- SST选择。除了选择最旧的SST，您还可以考虑其他启发式方法来选择要压缩的SST。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。

清单



在本章节中，您将：

- 实现清单文件的编码和解码。
- 系统重启时从清单中恢复。

要复制测试用例到入门代码并运行，

```
cargo x copy-test --week 2 --day 5
cargo x scheck
```

任务 1：清单编码

系统使用一个清单文件来记录引擎中发生的所有操作。目前，只有两种类型：压缩和SST刷新。当引擎重新启动时，它将读取清单文件，重建状态，并加载磁盘上的SST文件。

存储LSM状态的方法有很多。最简单的方法之一是将完整状态存储到一个JSON文件中。每次我们进行压缩或刷新新的SST时，都可以将整个LSM状态序列化到文件中。这种方法的缺点是，当数据库变得超级大（即10k SSTs）时，将清单写入磁盘会超级慢。因此，我们设计了清单为一个仅追加文件。

在这个任务中，你需要修改：

```
src/manifest.rs
```

我们使用JSON编码清单记录。你可以使用 `serde_json::to_vec` 将一个清单记录编码为json，写入清单文件，并进行fsync。当你从清单文件读取时，你可以使用

```
serde_json::Deserializer::fr
```

记录流。您不需要存储记录长度或类似信息，因为 `serde_json` 可以自动找到记录的分割点。

清单格式如下：

```
| JSON record | JSON record | JSON record | JSON record |
```

再次注意，我们不记录每条记录的字节数信息。

引擎运行数小时后，清单文件可能会变得非常大。此时，您可以定期压缩清单文件以存储当前快照并截断日志。这可以作为奖励任务之一来实现。

任务 2：写入清单

现在，您可以继续修改您的 LSM 引擎以在必要时写入清单。在此任务中，您需要修改：

```
src/lsm_storage.rs
src/compact.rs
```

目前，我们仅使用两种类型的清单记录：SST 刷新和压缩。SST 刷新记录存储被刷新到磁盘的 SST ID。压缩记录存储合并任务和生成的 SST ID。每次您将一些新文件写入磁盘时，首先同步文件和存储目录，然后写入清单并同步清单。清单文件应写入 `<path>/MANIFEST`。

要同步目录，您可以实现 `sync_dir` 功能，您可以使用 `File::open(dir).sync_all()? 来同步它。在Linux中，目录是一个包含目录中文件列表的文件。通过对目录执行fsync，您将确保如果断电，新写入（或删除）的文件对用户可见。`

请为后台压缩触发器（分层/简单/通用）和用户请求强制压缩时，都编写压缩清单记录。

任务3：关闭时刷新

在这个任务中，您需要修改：

```
src/lsm_storage.rs
```

您需要实现 `close` 函数。如果 `self.options.enable_wal = false`（我们将在下一章节中介绍WAL），您应该在停止存储引擎之前将所有内存表刷新到磁盘，以便所有用户更改都能持久化。

任务 4：从状态恢复

在此任务中，您需要修改：

`src/lsm_storage.rs`

现在，您可以修改 `open` 函数以从清单文件恢复引擎状态。要恢复它，您需要首先生成您需要加载的 SST 列表。您可以通过调用 `apply_compaction_result` 并在 LSM 状态中恢复 SST ID 来完成此操作。之后，您可以迭代状态并加载所有 SST（更新 `sstables` 哈希映射）。在此过程中，您需要计算最大 SST ID 并更新 `next_sst_id` 字段。之后，您可以使用该 ID 创建一个新的内存表，并将 ID 增加一。

如果您已实现分层压缩，每次应用压缩结果时，您可能已经对 SST 进行了排序。但是，使用清单恢复时，您的排序逻辑可能会中断，因为在恢复过程中，您无法知道每个 SST 的起始键和结束键。要解决此问题，您需要读取 `in_recovery` 函数的 `apply_compaction_result` 标志。在恢复过程中，您不应尝试检索 SST 的第一个键。在 LSM 状态恢复并且所有 SST 都打开后，您可以在恢复过程的末尾进行排序。

可选地，您可以在清单中包含每个 SST 的起始键和结束键。这种策略用于 RocksDB/BadgerDB，以便在压缩应用过程中，您不需要区分恢复模式和正常模式。

您可以使用 `mini-lsm-cli` 来测试您的实现。

```
cargo run --bin mini-lsm-cli
fill 1000 2000
close
cargo run --bin mini-lsm-cli
get 1500
```

测试您的理解

- 何时需要调用 `fsync`？为什么？ `sync` 目录？
- 您需要写入清单的地方有哪些？

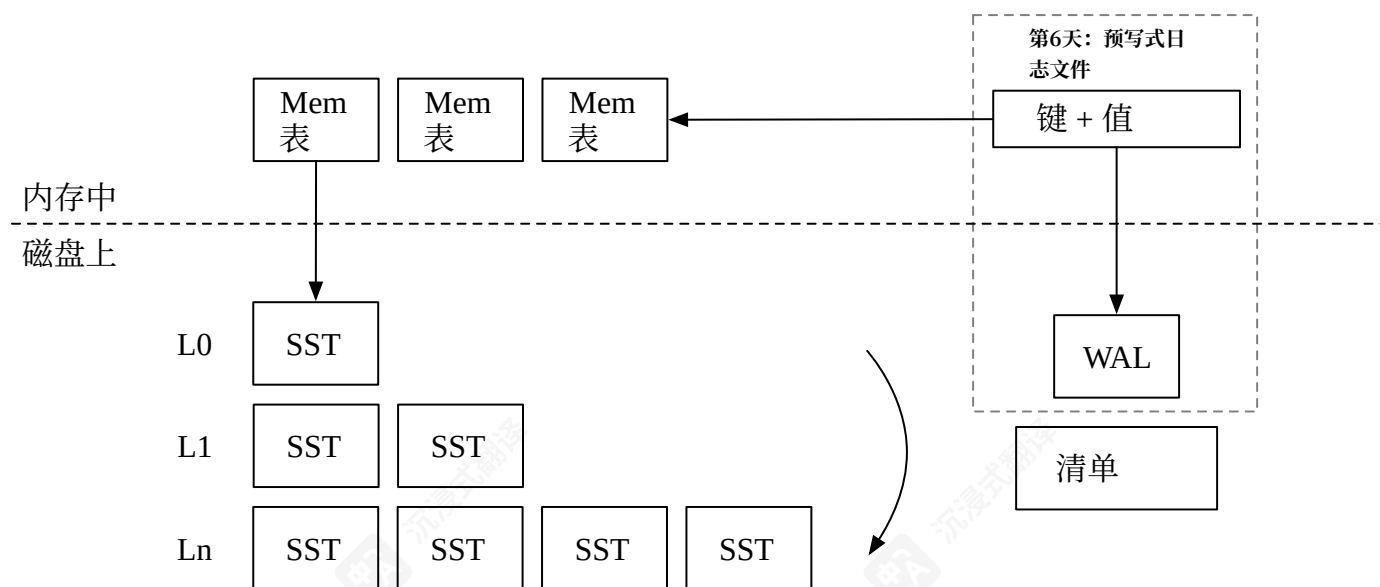
- 考虑一种不使用清单文件的LSM引擎的替代实现。相反，它在每个文件的头部记录级别/层级信息，每次启动时扫描存储目录，并且仅从目录中的文件恢复LSM状态。这种实现是否能够正确维护LSM状态，以及可能存在哪些问题/挑战？
- 当前，我们在创建合并迭代器之前创建所有 SST/concat 迭代器，这意味着在开始扫描过程之前，我们必须将所有层级中第一个 SST 的第一个块加载到内存中。我们在清单中有 start/end 键，是否可以利用这些信息来延迟数据块的加载，并使返回第一个键值对的时间更快？
- 是否可以不在清单中存储层级信息？即，我们仅将拥有的 SST 列表存储在清单中，而不包含层级信息，并使用键范围和时间戳信息（SST 元数据）重新构建层级。

奖励任务

- 清单压缩。当清单文件中的日志数量过大时，你可以重写清单文件，仅存储当前快照，并将新的日志追加到该文件中。
- 并行打开。在收集完要打开的 SST 列表后，你可以并行打开并解码它们，而不是逐一进行，从而加速恢复过程。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。

预写日志 (WAL)



在本章节中，您将：

- 实现预写式日志文件的编码和解码。
- 系统重启时从预写式日志文件中恢复内存表。

要复制测试用例到入门代码并运行它们，

```
cargo x copy-test --week 2 --day 6
cargo x scheck
```

任务 1：预写式日志编码

在这个任务中，你需要修改：

`src/wal.rs`

在上一章节中，我们已经实现了清单文件，以便LSM状态可以被持久化。我们还实现了 `close` 功能，在停止引擎之前将所有内存表刷新到SSTs。现在，如果系统崩溃（即断电），我们可以将内存表的修改记录到WAL（预写日志），并在重启数据库时恢复WALs。WAL仅在 `self.options.enable_wal = true` 时启用。

WAL 编码只是一个键值对的列表。

```
| key_len | key | value_len | value |
```


您还需要实现 `recover` 功能来读取 WAL 并恢复内存表的状态。

请注意，我们使用 `BufWriter` 来写入 WAL。使用 `BufWriter` 可以减少对操作系统的系统调用次数，从而降低写入路径的延迟。当用户修改键时，数据并不保证写入磁盘。相反，引擎仅保证当 `sync` 被调用时数据被持久化。要正确地将数据持久化到磁盘，您需要首先通过调用 `flush()` 将数据从缓冲区写入器刷新到文件对象，然后使用 `get_mut().sync_all()` 对文件进行 `fsync`。请注意，您只需要在引擎的 `sync` 被调用时进行 `fsync`。在写入数据时，您不需要每次都进行 `fsync`。

任务 2：集成 WALs

在这个任务中，您需要修改：

```
src/mem_table.rs
src/wal.rs
src/lsm_storage.rs
```

`MemTable` 有一个 WAL 字段。如果 `wal` 字段设置为 `Some(wal)`，您需要在更新内存表时追加到 WAL。在您的 LSM 引擎中，如果 `enable_wal = true`，您需要创建 WALs。当创建新的内存表时，您还需要使用 `ManifestRecord::NewMemtable` 记录更新清单。

您可以使用 `create_with_wal` 函数创建一个带有 WAL 的内存表。WAL 应该写入存储目录中的 `<memtable_id>.wal`。如果这个内存表被刷新为 L0 SST，内存表 ID 应该与 SST ID 相同。

任务 3：从 WALs 中恢复

在这个任务中，您需要修改：

```
src/lsm_storage.rs
```

如果启用了 WAL，在加载数据库时，您需要根据预写式日志文件恢复内存表。您还需要实现数据库的 `sync` 功能。`sync` 的基本保证是引擎确保数据持久化到磁盘（并在重启时恢复）。为此，您只需同步当前内存表对应的 WAL 即可。

```
cargo run --bin mini-lsm-cli -- --enable-wal
```

记住从状态中恢复正确的 `next_sst`.

`max{memtable id, sst id} + 1`。在你的 `close` 函数中，如果 `enable_wal` 设置为 `true`，你不应该将内存表刷新到 SST 中，因为 WAL 本身就提供了持久性。你应该等待所有压缩和刷新线程退出后才能关闭数据库。

Test Your Understanding

- When should you call `fsync` in your engine? What happens if you call `fsync` too often (i.e., on every put键请求)?
- How costly is the `flush` operation in general on an SSD (固态硬盘)?
- When can you tell the user that their modifications (put/delete) have been 持久化?
- How can you handle 损坏的数据 in WAL?
- 是否可以设计一个不使用WAL（即使用L0作为WAL）的LSM引擎？这种设计的后果是什么？

我们不提供问题的参考答案，欢迎在Discord社区中讨论。

非常感谢您的反馈。欢迎加入我们的Discord社区。发现问题时？请在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。

批量写入和校验和

在上一章节中，你已经构建了一个完整的基于LSM的存储引擎。在本周结束时，我们将实现一些简单但重要的存储引擎优化。欢迎来到Mini-LSM的第2周零食时间！

在本章节中，你将：

- 实现批量写入接口。
- 为块、SST元数据、清单和预写式日志文件添加校验和。

注意：本章没有单元测试。只要你通过所有之前的测试并确保校验和在你的文件格式中正确编码，就足够了。

任务 1：写入批处理接口

在这个任务中，我们将通过添加一个写入批处理API来为这门课程的第3周做准备。您需要修改：

`src/lsm_storage.rs`

用户向 `write_batch` 提供一批要写入数据库的记录。这些记录是

`WriteBatchRecord<T: AsRef<[u8]>>`，因此可以是 `Bytes`、`&[u8]` 或 `Vec<u8>`。记录 `T` 的类型：删除和写入。您可以像处理您的 `put` 和 `delete` 函数一样处理它们。

之后，您可以重构您的原始 `put` 和 `delete` 函数以调用 `write_batch`。

在实现此功能后，您应该在之前的章节中通过所有测试用例。

任务 2：块校验和

在这个任务中，您需要在编码SST时在每个块的末尾添加一个块校验和。您需要修改：

`src/table/builder.rs`
`src/table.rs`

SST的格式将更改如下：

Block Section							
Meta Section							
data block		checksum	...	data block	checksum	metadata	meta block
offset	bloom filter	bloom filter	offset				
varlen	u32			varlen	u32	varlen	u32
varlen	u32						

我们使用 `crc32` 作为我们的校验和算法。您可以使用 `crc32fast::hash` 在构建块后生成块的校验和。

通常情况下，当用户在存储选项中指定目标块大小时，该大小应包含块内容和校验和。例如，如果目标块大小为4096，而校验和占用4个字节，则实际块内容目标大小应为4092。然而，为了避免破坏之前的测试用例并简化操作，在我们的课程中，我们仍然将使用目标块大小作为目标内容大小，并在块的末尾简单追加校验和。

当您读取块时，您应在 `read_block` 正确生成块内容的切片。在实现此功能后，您应该通过之前章节的所有测试用例。

任务3：SST元数据校验和

在此任务中，您需要为布隆过滤器和块元数据添加块校验和：

```
src/table.rs
src/table/bloom.rs
src/table/builder.rs
```

	Meta Section				

no. of block metadata checksum meta block offset bloom filter					
checksum bloom filter offset					
u32 varlen u32 u32 varlen					
u32 u32					

您需要在 `Bloom::encode` 和 `Bloom::decode` 中的布隆过滤器末尾添加校验和。请注意，我们的大多数 API 都接受一个现有的缓冲区，实现将写入其中，例如 `Bloom::encode`。因此，您应该在写入编码内容之前记录布隆过滤器开头的偏移量，并且只对布隆过滤器本身进行校验和，而不是整个缓冲区。

之后，您可以在块元数据的末尾添加一个校验和。您可能会发现，在节的开头添加元数据长度也很有帮助，这样在解码块元数据时，将更容易知道在哪里停止。

任务 4：WAL 校验和

在这个任务中，您需要修改：

`src/wal.rs`

我们将在预写日志中执行按记录校验和。为此，您有两个选择：

- 生成键值记录的缓冲区，并使用 `crc32fast::hash` 一次性计算校验和。
- 一次写一个字段（例如，键长度、键切片），并使用 `crc32fast::Hasher` 对每个字段逐个增量计算校验和。

这是您的选择，您需要选择自己的冒险方式。两种方法应该产生完全相同的结果，只要您正确处理小端字节序 / 大端字节序即可。新的预写式日志编码应该是：

```
| key_len | key | value_len | value | checksum |
```

任务 5：清单校验和

最后，让我们在清单文件上添加一个校验和。清单类似于WAL，只是以前我们没有存储每条记录的长度。为了使实现更容易，我们现在在记录的开头添加记录长度的头部，并在记录的末尾添加校验和。

新的清单格式如下：

```
| len | JSON record | checksum | len | JSON record | checksum | len | JSON  
record | checksum |
```

实现所有功能后，你应该通过所有之前的测试用例。在本章节中，我们不提供新的测试用例。

测试你的理解

- 考虑以下情况：一个 LSM 存储引擎仅提供 `write_batch` 作为写入接口（而不是单个 `put` + 删除）。是否可以按以下方式实现：有一个单独的写入线程，并使用 `mpsc` 通道接收器来获取变更，所有线程将写入批处理发送到写入线程。写入线程是写入数据库的唯一写入点。这种实现的优缺点是什么？（如果你这样做，恭喜你，你将获得 BadgerDB！）
- 是否可以将所有块校验和一起放在 SST 文件的末尾，而不是与块一起存储？为什么？

我们不提供问题的参考答案，并且欢迎在Discord社区中讨论它们。

奖励任务

- 在损坏时恢复。如果存在校验和错误，请在安全模式下打开数据库，以便无法执行写入操作，并且仍然可以检索未损坏的数据。

非常感谢您的反馈。欢迎加入我们的Discord社区。发现问题？请在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由 Alex Chi Z 提供，根据 CC BY-NC-SA 4.0 许可证授权。

第3周概述：多版本并发控制

在本部分，你将实现你在前两周构建的LSM引擎上的MVCC。我们将向键中添加时间戳编码以维护键的多个版本，并更改引擎的部分内容，以确保旧数据根据是否有用户读取旧版本而被保留或进行垃圾回收。

本课程中MVCC部分的一般方法受BadgerDB启发，并部分基于BadgerDB。

MVCC的关键在于存储引擎中存储和访问键的多个版本。因此，我们需要将键格式更改为 `user_key + timestamp (u64)`。在用户界面方面，我们需要提供新的API来帮助用户访问历史版本。总之，我们将向键添加单调递增的时间戳。

在之前的部分中，我们假设较新的键位于LSM树的上层，较旧的键位于LSM树的较低层。在压缩过程中，如果多个层级中存在多个版本的键，我们仅保留最新版本，并且压缩过程将通过仅合并相邻层级/层来确保较新的键保留在上层。在MVCC实现中，时间戳较大的键是最新的键。在压缩过程中，我们只能在没有任何用户访问数据库较旧版本的情况下删除键。尽管不在上层保留键的最新版本可能仍然能为MVCC LSM实现产生正确结果，但在我们的课程中，我们选择保持不变式，如果有键的多个版本，较新的版本将始终出现在上层。

通常，有两种方式利用支持MVCC的存储引擎。如果用户将引擎作为独立组件使用，并且不希望手动分配键的时间戳，他们将使用事务API从存储引擎中存储和检索数据。时间戳对用户是透明的。另一种方式是将存储引擎集成到系统中，用户自行管理时间戳。要比较这两种方法，我们可以看看它们提供的API。我们使用BadgerDB的术语来描述这两种用法：隐藏时间戳的是非管理模式，而赋予用户完全控制权的是管理模式。

管理模式API

```
get(key, read_timestamp) -> (value, write_timestamp)
scan(key_range, read_timestamp) -> iterator<key, value, write_timestamp>
put/delete/write_batch(key, timestamp)
set_watermark(timestamp) # we will talk about watermarks soon!
```

非管理模式/普通模式API

```
get(key) -> valuescan(key_range) -> iterator<key,
value>start_transaction() -> txn
txn.put/delete/write_batch(key, timestamp)
```

如您所见，管理模式API要求用户在执行操作时提供时间戳。时间戳可能来自某些集中式时间戳系统，或来自其他系统的日志（即Postgres逻辑复制日志）。用户需要指定水印，即引擎可以删除的低于此版本的版本。

对于非管理模式API，其与之前实现的功能相同，只是用户需要通过创建事务来写入和读取数据。当用户创建事务时，他们可以获得数据库的一致状态（即快照）。即使其他线程/事务向数据库写入数据，这些数据对进行中的事务也是不可见的。存储引擎在内部管理时间戳，并不向用户暴露它们。

在本周，我们将首先花费3天时间对表格式和内存表进行重构。我们将键格式改为键切片和时间戳。之后，我们将实现必要的API，以提供一致性快照和事务。

本部分有7个章节（天）：

- 第一天：时间戳键重构。您将把 `key` 模块改为MVCC模块，并重构您的系统以使用带时间戳的键。
- 第2天：快照读 - 内存表和时间戳。您将重构内存表和写入路径，以支持多版本读写。
- 第3天：快照读 - 事务API。您将实现事务API并完成读取/写入路径的其余部分，以支持快照读。
- 第4天：水印和垃圾回收。您将实现水位线计算算法，并在压缩时实现垃圾回收以删除旧版本。
- 第5天：事务和乐观并发控制。您将把所有事务创建一个私有工作区，并批量提交它们，以便一个事务的修改不会对其他事务可见。
- 第6天：可串行化快照隔离。您将实现OCC可序列化检查，以确保对数据库的修改是可串行化的，并中止违反串行性的事务。
- 第7天：压缩过滤器。在本周结束时，我们将压缩时的垃圾回收逻辑泛化为一个压缩过滤器，该过滤器根据用户要求在压缩时删除数据。

您的反馈非常宝贵。欢迎加入我们的Discord社区。发现问题？请在
github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book ©
2022-2025 by Alex Chi Z根据CC BY-NC-SA 4.0协议授权。

时间戳键编码 + 重构

在本章节中，你将：

- 重构你的实现以使用键+表示。
- 使你的代码在新的键表示下能够编译。

要运行测试用例，

```
cargo x copy-test --week 3 --day 1
cargo x scheck
```

注意：MVCC子系统直到第3周第2天才完全实现。你只需要在本日结束时通过第3周第1天的测试和所有第1周的测试。第2周的测试将因压缩而无法运行。

任务0：使用MVCC键编码

您需要将键编码模块替换为MVCC模块。我们从原始键模块中删除了一些接口，并为键实现了新的比较器。如果您按照前几章的说明操作，并且没有在键上使用 `into_inner`，那么在所有重构完成后，您应该能在第3天通过所有测试用例。否则，您需要仔细检查那些只比较键而不查看时间戳的地方。

具体来说，键类型定义已经从以下内容更改：

```
pub struct Key<T: AsRef<[u8]>>(T);
```

...更改
为：

```
pub struct Key<T: AsRef<[u8]>>(T /* user key */, u64 /* timestamp */);
```

...其中我们与键关联了时间戳。我们仅在系统内部使用这种键表示。在用户界面方面，我们不要用户提供时间戳，因此某些结构仍然在引擎中使用 `&[u8]` 而不是 `KeySlice`。我们稍后将涵盖需要更改函数签名的位置。目前，您只需要运行，

```
cp mini-lsm-mvcc/src/key.rs mini-lsm-starter/src/
```

存储时间戳还有其它方法。例如，我们仍然可以使用 `pub struct Key<T: AsRef<[u8]> { key: T; ts: u64; }` 字节是时间戳。你也可以将其作为奖励任务的一部分来实现。

Alternative key representation: | user_key (varlen) | ts (8 bytes) | in a single slice

Our key representation: | user_key slice | ts (u64) |

在键+编码中，具有最小用户键和最大时间戳的键将首先排序。例如，

`("a", 233) < ("a", 0) < ("b", 233) < ("b", 0)`

任务 1：在块中编码时间戳

你会首先注意到，在替换键模块后，你的代码可能无法编译。在这一章节中，你唯一需要做的是让它能够编译。在这个任务中，你需要修改：

```
src/block.rs
src/block/builder.rs
src/block/iterator.rs
```

你会注意到 `raw_ref()` 和 `len()` 已经从键 API 中移除了。取而代之的是，我们需要 `key_ref` 来检索用户键的切片，以及 `key_len` 来检索用户键的长度。你需要重构你的块构建器和解码实现，以使用新的 API。此外，你还需要将你的块编码更改为编码时间戳。在 `BlockBuilder::add` 中，你应该这样做。新的块条目记录将如下：

```
key_overlap_len (u16) | remaining_key_len (u16) | key (remaining_key_len) |
timestamp (u64)
```

你可以使用 `raw_len` 来估计一个键所需的存储空间，并在用户键之后存储时间戳。

在更改块编码后，你需要在 `block.rs` 和 `iterator.rs` 中相应地更改解码。

任务 2：在 SST 中编码时间戳

然后，你可以继续修改表格式，

```
src/table.rs
src/table/builder.rs
src/table/iterator.rs
```

具体来说，您需要更改您的块元数据编码以包含键的时间戳。所有其他代码保持不变。由于我们在所有函数（即查找、添加）的签名中使用 `KeySlice`，新的键比较器应自动按用户键和时间戳对键进行排序。

在你的表构建器中，你可以直接使用 `key_ref()` 来构建布隆过滤器。这自然地为你的 SSTs 创建了一个前缀布隆过滤器。

任务 3：LSM 迭代器

由于我们使用关联泛型类型使大多数迭代器适用于不同的键类型（即 `&[u8]` 和 `KeySlice<'_>`），如果合并迭代器和连接迭代器实现正确，我们不需要修改它们。`LsmIterator` 是我们剥离内部键表示中的时间戳并将键的最新版本返回给用户的地方。在这个任务中，你需要修改：

```
src/lsm_iterator.rs
```

目前，我们不会修改 `LsmIterator` 的逻辑以仅保留键的最新版本。我们通过在将键传递给内部迭代器时向用户键附加时间戳，并在返回给用户时从键中剥离时间戳来使其能够编译。目前，你的 LSM 迭代器应该向用户返回同一键的多个版本。

任务 4：内存表

目前，我们保留内存表的逻辑。我们向用户返回一个键切片，并使用 `TS_DEFAULT` 刷新 SSTs。在下一章节中，我们将内存表改为 MVCC。在这个任务中，你需要修改：

```
src/mem_table.rs
```

任务 5：引擎读路径

在这个任务中，你需要修改，

```
src/lsm_storage.rs
```

现在键中有了时间戳，在创建迭代器时，我们需要查找带有时间戳的键，而不仅仅是用户键。您可以使用 `TS_RANGE_BEGIN` 创建一个键切片，这是最大的 `ts`。

当你检查一个用户键是否在表中时，你可以直接比较用户键，而无需比较时间戳。

此时，你应该构建你的实现并通过所有第1周的测试用例。系统中存储的所有键将使用 `TS_DEFAULT`（即时间戳0）。我们将使引擎完全支持多版本，并在接下来的两章中通过所有测试用例。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z发布，根据CC BY-NC-SA 4.0许可协议授权。

快照读 - 内存表和时间戳

在本章节中，您将：

- 重构你的内存表/WAL 以存储键的多个版本。
- 实现新的引擎写入路径以分配每个键一个时间戳。
- 让你的压缩过程了解多版本键。
- 实现新的引擎读取路径以返回键的最新版本。

在重构过程中，您可能需要根据需要将一些函数的签名从 `&self` 更改为 `self: &Arc<Self>`。

要运行测试用例，

```
cargo x copy-test --week 3 --day 2
cargo x scheck
```

注意：在本章节完成后，您还需要传递 `<= 2.4` 之后的所有内容。

任务 1：内存表、预写日志和读取路径

在这个任务中，您需要修改：

```
src/wal.rs
src/mem_table.rs
src/lsm_storage.rs
```

我们已经将引擎中的大多数键设置为 `KeySlice`，它包含一个字节键和一个时间戳。然而，我们系统中的某些部分仍然没有考虑时间戳。在我们的第一个任务中，您需要修改您的内存表和 WAL 实现以考虑时间戳。

您需要首先更改内存表中存储的 `SkipMap` 的

```
pub struct MemTable { // map: Arc<SkipMap<Bytes, Bytes>>,
  map: Arc<SkipMap<KeyBytes, Bytes>>, // Bytes -> KeyBytes
  // ...}
```

之后，您可以继续修复所有编译器错误，以完成此任务。

MemTable::get

我们保留获取接口，以便测试用例仍然可以探测内存表中的键的特定版本。在完成这项任务后，此接口不应在读取路径中使用。考虑到我们存储了 `KeyBytes`，它是 `(Bytes, u64)` 在跳表中的数据，而用户探测了 `KeySlice`，它是 `(&[u8], u64)`。我们必须找到一种方法将后者转换为前者的引用，以便我们可以检索跳表中的数据。

要实现这一点，您可以使用不安全代码来强制转换 `&[u8]` 为静态，并使用 `Bytes::from_static` 从静态切片创建一个 `bytes` 对象。这是安全的，因为 `Bytes` 不会尝试切片的内存，因为它被认为是静态的。

► 剧透：将 u8 切片转换为 Bytes

这不是一个问题，因为我们之前有 `Bytes` 和 `&[u8]`，其中 `Bytes` 实现 `Borrow<[u8]>`。

MemTable::put

签名应该更改为 `fn put(&self, key: KeySlice, value: &[u8])`，并且您需要在您的实现中将一个键切片转换为 `KeyBytes`。

MemTable::scan

签名应更改为 `fn scan(&self, lower`
。您需要将 `KeySlice` 转换为 `KeyBytes` 并使用这
些作为 `SkipMap::`

MemTable::flush

不要使用默认时间戳，现在在将内存表刷新到 SST 时应使用键时间戳。

MemTableIterator

它现在应存储 `(KeyB`]键类型应为 `KeySlice`。

Wal::恢复 和 Wal::放入

预写日志 现在应接受 键切片 而不是 用户键切片。在序列化和反序列化 WAL 记录 时，你应该将 时间戳 放入 WAL 文件 并对 时间戳 和你之前有的所有其他字段进行校验和。

WAL 格式 如下：

```
| key_len (exclude ts len) (u16) | key | ts (u64) | value_len (u16) | value |
checksum (u32) |
```

LsmStorageInner::获取

之前，我们实现 `get` 是先探测内存表，然后扫描 SST。现在我们将内存表改为使用新的键时间戳 API，我们需要重新实现 `get` 接口。最容易的方法是创建一个合并迭代器，覆盖我们已有的所有内容——内存表、不可变的内存表、L0 SST 和其他级别的 SST，就像你在 `scan` 中所做的那样，只是我们在 SST 上进行布隆过滤器过滤。

LsmStorageInner::scan

您需要整合新的内存表API，并将扫描范围设置为 `(user_key_begin, TS_RANGE_BEGIN)` 和 `(user_key_end, TS_RANGE_END)`。请注意，在处理排除边界时，您需要将迭代器正确定位到下一个键（而不是相同时间戳的当前键）。

任务 2：写入路径

在这个任务中，您需要修改：

`src/lsm_storage.rs`

我们在 `LsmStorageInner` 中有一个 `mvcc` 字段，其中包含我们本周需要用于多版本并发控制的所有数据结构。当您打开目录并初始化存储引擎时，您需要创建该结构。

在您的 `write_batch` 实现中，您需要为写入批处理中的所有键获取提交时间戳。您可以通过在逻辑开头使用 `self.mvcc().latest_commit_ts() + 1` 来获取时

`f.mvcc().update_commit_ts(ts)` 来递增下一个提交时间戳。为确保所有写入批处理具有不同的时间戳，并且新键位于旧键之上，您需要在函数开头持有写入锁

`self.mvcc().write_lock.lock()`，以便同一时间只有一个线程可以写入存储引擎。

任务 3：MVCC 压缩

在这个任务中，您需要修改：

`src/compact.rs`

我们在之前的章节中做的是仅保留键的最新版本，并在我们将键压缩到底层时删除键（如果键被删除）。有了MVCC，我们现在与键关联了时间戳，并且不能使用相同的压缩逻辑。

在本章节中，您可以简单地删除用于删除键的逻辑。您可以暂时忽略 `compact_to_bottom_level`，并且在压缩期间应保留键的所有版本。

此外，您需要以某种方式实现合并算法，使得具有不同时间戳的相同键被放入同一个SST文件中，即使它超过了SST大小限制。这确保了如果某个键在一个级别的SST中被找到，它将不会出现在该级别中的其他SST文件中，从而简化了系统许多部分的实现。

任务4：LSM迭代器

在这个任务中，您需要修改：

```
src/lsm_iterator.rs
```

在上一章节中，我们实现了LSM迭代器，以将具有不同时间戳的相同键视为不同的键。现在，我们需要重构LSM迭代器，以便如果从子迭代器检索到多个键版本，则仅返回键的最新版本。

您需要在迭代器中记录 `prev_key`。如果我们已经将键的最新版本返回给用户，我们可以跳过所有旧版本并继续到下一个键。

此时，您应该通过前几章的所有测试，除了持久性测试（2.5和2.6）。

测试你的理解

- MVCC 引擎和第2周你构建的引擎中，`get` 有什么差异？
- 在第2周，当 `get` 时，你会在第一个找到键的 `memtable`/级别停止。你能在 MVCC 版本中做到同样的事情吗？
- 你是如何将 `KeySlice` 与 `Key` 关联的？这是一个安全的操作吗？
- 为什么在写入路径中需要获取写锁？

我们不提供问题的参考答案，欢迎在 Discord 社区中讨论。

奖励任务

- 内存表获取的早期停止。我们不必为所有内存表和SST创建合并迭代器，而是可以按如下方式实现 `get`：如果我们发现内存表中存在某个键的版本，就可以停止搜索。SSTs也是如此。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。

快照读 - 引擎读路径和事务API

在本章节中，您将：

- 基于上一章节完成读取路径以支持快照读。
- 实现事务API以支持快照读。
- 实现引擎恢复过程以正确恢复提交时间戳。

最终，您的引擎将能够为用户提供一致的存储键空间视图。

在重构过程中，您可能需要根据需要将一些函数的签名从 `&self` 更改为 `self: &Arc<Self>`。

要运行测试用例，

```
cargo x copy-test --week 3 --day 3
cargo x scheck
```

注意：在完成本章后，您还需要为 2.5 和 2.6 通过测试用例。

任务 1：带读取时间戳的 LSM 迭代器

本章的目标是实现类似以下功能：

```
let snapshot1 = engine.new_txn();
// write something to the engine
let snapshot2 = engine.new_txn();
// write something to the engine
snapshot1.get(/* ... */); // we can retrieve a consistent snapshot of a
previous state of the engine
```

要实现这一点，我们可以在创建事务时记录读取时间戳（即最新提交的时间戳）。当我们对事务进行读取操作时，我们只会读取所有键版本中低于或等于读取时间戳的记录。

在此任务中，您需要修改：

`src/lsm_iterator.rs`

要这样做，您需要在 `LsmIterator` 中记录一个读取时间戳。

```
impl LsmIterator {
    pub(crate) fn new(
        iter: LsmIteratorInner,
        end_bound: Bound<Bytes>,
        read_ts: u64,
    ) -> Result<Self> {
        // ...
    }
}
```

并且你需要更改你的LSM迭代器 `next` 逻辑以找到 `key` 键。

任务 2：多版本扫描和获取

在这个任务中，您需要修改：

```
src/mvcc.rs
src/mvcc/txn.rs
src/lsm_storage.rs
```

现在我们在 LSM 迭代器中有了 `read_ts`，我们可以在事务结构上实现 `scan` 和 `get`，以便我们可以在存储引擎的某个时间点读取数据。

我们建议您在您的 `LsmStorageInner` 结构中创建辅助函数

`scan_with_ts` (接受 `original parameters*/、read_ts: u64`) 和 `get_with_ts` (如果需要)。存储引擎上的原始 `get/scan` 应该实现为创建一个事务（快照）并在该事务上执行 `get/scan`。调用路径将是：

```
LsmStorageInner::scan -> new_txn and Transaction::scan ->
LsmStorageInner::scan_with_ts
```

要在 `LsmStorageInner::scan` 中创建一个事务，我们需要向事务构造函数提供一个 `Arc<LsmStorageInner>`。因此，我们可以更改 `scan` 的签名以接受 `self: &Arc<Self>` 而不是简单地接受 `&self`，以便我们可以使用

```
let txn = self.mvcc().
```

您还需要更改您的 `scan` 函数以返回一个 `TxnIterator`。我们必须确保当用户迭代引擎时，快照是活跃的，因此，`TxnIterator` 存储快照对象。在 `TxnIterator` 中，我们可以暂时存储一个 `FusedIterator<LsmIterator>`。当我们实现 OCC 时，我们会将其更改为其他内容。

你现在不需要实现 `Transaction::put/delete`，所有修改仍然会通过引擎进行。

任务 3：在 SST 中存储最大时间戳

在这个任务中，您需要修改：

```
src/table.rs  
src/table/builder.rs
```

在你的 SST 编码中，你应该在块元数据之后存储最大时间戳，并在加载 SST 时恢复它。这将帮助系统在恢复系统时决定最新的提交时间戳。

任务 4：恢复提交时间戳

现在我们在 SSTs 中有最大时间戳信息，在 WAL 中有时间戳信息，我们可以获得引擎启动前提交的最新时间戳，并在创建 `mvcc` 对象时将其用作最新的提交时间戳。

如果 WAL 未启用，你可以通过在 SSTs 中找到最大时间戳来简单地计算最新的提交时间戳。如果 WAL 启用，你应该进一步迭代所有恢复的内存表并找到最大时间戳。

在这个任务中，你需要修改：

```
src/lsm_storage.rs
```

本节没有测试用例。完成本节后，您应该通过之前章节的所有持久性测试（包括2.5和2.6节）。

检验理解

- 到目前为止，我们假设我们的SST文件使用单调递增ID作为文件名。使用 `<level>_<begin_key>_<end_key>_<max_ts>.sst` 作为SST文件名可以吗？那样可能会有什么问题？
- 考虑一种事务/快照的替代实现。在我们的实现中，我们在迭代器和事务上下文中有 `read_ts`，以使用户始终可以访问基于时间戳的数据库版本的一致视图。是否可行将当前的 LSM 状态直接存储在事务上下文中，以获得一致的快照？（即，所有 SST ID、它们级别信息以及所有内存表 + 时间戳）这样做有什么优缺点？如果引擎没有内存表会怎样？如果引擎运行在像 S3 对象存储这样的分布式存储系统上会怎样？

- 考虑你正在实现一个MVCC Mini-LSM引擎的备份工具。仅仅复制所有SST文件出来而不备份LSM状态就足够了吗？为什么或为什么不？

我们不提供问题的参考答案，并且欢迎在Discord社区中讨论它们。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z提供，根据CC BY-NC-SA 4.0许可协议发布。

水印和垃圾回收

在本章节中，您将实现必要的结构来跟踪用户正在使用的最低读取时间戳，并在执行压缩时从SSTs中收集未使用的版本。

要运行测试用例，

```
cargo x copy-test --week 3 --day 4
cargo x scheck
```

任务 1：实现水印

在这个任务中，你需要修改：

```
src/mvcc/watermark.rs
```

水印是用于跟踪系统中最低 `read_ts` 的结构。当创建新事务时，它应该调用 `add_reader` 来添加其读取时间戳以进行跟踪。当事务中止或提交时，它应该从水印中移除自身。当调用 `watermark()` 时，水印结构返回系统中的最低 `read_ts`。如果没有进行中的事务，它将简单地返回 `None`。

你可以使用一个 `BTreeMap` 来实现水印。它维护一个计数器，记录每个 `read_ts` 有多少快照正在使用这个读取时间戳。你不应该在 B 树映射中包含有 0 个读者的条目。

任务 2：维护事务中的水印

在这个任务中，您需要修改：

```
src/mvcc/txn.rs
src/mvcc.rs
```

当事务开始时，您需要在水印中添加 `read_ts`，并在事务调用 `drop` 时删除它。

任务 3：压缩过程中的垃圾回收

在这个任务中，您需要修改：

`src/compact.rs`

现在我们已经有了系统的水印，我们可以在压缩过程中清理未使用的版本。

- 如果一个键的版本高于水印，保留它。
- 对于所有低于或等于水印的键版本，保留最新版本。

例如，如果有水印=3 和以下数据：

```
a@4=del <- above watermark
a@3=3    <- latest version below or equal to watermark
a@2=2    <- can be removed, no one will read it
a@1=1    <- can be removed, no one will read it
b@1=1    <- latest version below or equal to watermark
c@4=4    <- above watermark
d@3=del  <- can be removed if compacting to bottom-most level
d@2=2    <- can be removed
```

如果我们对这些键进行压缩，我们将得到：

```
a@4=del
a@3=3
b@1=1
c@4=4
d@3=del (can be removed if compacting to bottom-most level)
```

假设这些都是引擎中的键。如果我们以 $ts=3$ 进行扫描，我们将在压缩前/后获取 $a=3, b=1, c=4$ 。如果我们以 $ts=4$ 进行扫描，我们将在压缩前/后获取 $b=1, c=4$ 。压缩不会也不应该影响具有读取时间戳 \geq 水印的事务。

测试你的理解

- 在我们的实现中，我们通过`Transaction` 的生命周期自行管理水印（即所谓的非管理模式）。如果用户打算自行管理键时间戳和水印（即当他们有自己的时间戳生成器时），您需要在`write_batch/get/scan` API中做些什么来验证他们的请求？我们是否有任何架构假设在这种情况下可能难以维护？
- 为什么我们需要在事务迭代器中管理水印？

- 从SST文件中完全删除一个键的条件是什么？
- 目前，我们仅在压缩到底层时才删除键。还有其他优先时间可以删除键吗？（提示：你知道所有层级中每个SST的起始/结束键。）
- 考虑用户创建一个长时间运行的事务且我们无法进行垃圾回收的情况。用户持续更新单个键。最终，单个SST文件中可能会有一个拥有数千个版本的键。这会如何影响性能，以及你会如何处理它？

奖励任务

- O(1)水位线。你可以通过使用哈希表或循环队列来实现摊销O(1)水位线结构。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？请在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z授权，根据CC BY-NC-SA 4.0协议发布。

事务和乐观并发控制

在本章节中，您将实现 `Transaction` 的所有接口。您的实现将在事务内部维护一个私有工作区进行修改，并以批次方式提交，以便事务内的所有修改仅对自身可见，直到提交。我们仅在提交时检查冲突（即可串行化冲突），这是乐观并发控制。

要运行测试用例，

```
cargo x copy-test --week 3 --day 5
cargo x scheck
```

任务 1：本地工作区 + Put 和 删除

在这个任务中，你需要修改：

```
src/mvcc/txn.rs
```

你现在可以通过将相应的键/值插入到 `local_storage` 中来实现 `put` 和 `delete`，`local_storage` 是一个没有键时间戳的跳表内存表。请注意，对于删除操作，你仍然需要将其实现为插入一个空值，而不是从跳表中移除值。

任务 2：获取和扫描

在这个任务中，你需要修改：

```
src/mvcc/txn.rs
```

对于 `get`，你应该首先探测本地存储。如果找到值，根据是否是删除标记，返回值或 `None`。对于 `scan`，你需要为跳表实现一个 `TxnLocalIterator`，就像在 1.1 章节中为没有键时间戳的内存表实现迭代器时一样。你需要存储一个

`TwoMergeIterator<TxnLocalIterator, FusedIterator<LsmIterator>>` 在 `TxnIterator` 中。最后，鉴于 `TwoMergeI` 删除标记在子项中

迭代器，你需要修改你的 `TxnIterator` 实现以正确处理删除。

任务 3：提交

在这个任务中，你需要修改：

```
src/mvcc/txn.rs
```

我们假设一个事务只会在单个线程上使用。一旦你的事务进入提交阶段，你应该将 `self.committed` 设置为 `true`，以使用户不能对事务进行任何其他操作。你的 `put`、`delete`、`scan` 和 `get` 实现，如果事务已经被提交，应该报错。

你的提交实现应该简单地从本地存储收集所有键值对，并向存储引擎提交一个写入批处理。

任务 4：原子WAL

在这个任务中，你需要修改：

```
src/wal.rs  
src/mem_table.rs  
src/lsm_storage.rs
```

请注意，`commit` 涉及生成写入批处理，目前写入批处理不保证原子性。您需要更改WAL实现以生成写入批处理的头部和页脚。

新的WAL编码如下：

HEADER		BODY				
FOOTER						
u32	u16	var	u64	u16	var	...
u32						
batch_size	key_len	key	ts	value_len	value	more key-value pairs
...	checksum					

`batch_size` 是 BODY 部分的大小。 `checksum` 是 BODY部分的校验和。

没有测试用例来验证您的实现。只要您通过所有现有的测试用例并实现上述WAL格式，一切应该都很好。

你应该实现 `Wal::put_batch` 和 `MemTable::put_batch`。原始的 `put` 函数应该将单个键值对视为一批次。也就是说，此时，你的 `put` 函数应该调用 `put_batch`。

一个批次应该在同一个内存表和同一个WAL中处理，即使它超过了内存表的大小限制。

测试你的理解

- 到目前为止，我们已经实现了所有这些功能，系统是否满足快照隔离？如果不是，我们还需要做些什么来支持快照隔离？（注意：快照隔离与我们将在下一章中讨论的可串行化快照隔离不同）
- 如果用户想要批量导入数据（即1TB呢？），如果他们使用事务API来做这件事，你会给他们一些建议吗？这个情况下有任何优化的机会吗？
- 什么是乐观并发控制？如果我们改在Mini-LSM中实现悲观并发控制，系统会是什么样子？
- 如果你的系统崩溃并在磁盘上留下损坏的WAL，会发生什么？你如何处理这种情况？
- 当你提交事务时，是否需要将所有内容批量放入内存表，或者你可以简单地逐个放入键？为什么？

奖励任务

- 溢写到磁盘。如果事务的私有工作区过大，你可以将一些数据刷新到磁盘。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z提供，根据CC BY-NC-SA 4.0许可协议发布。

(A Partial) 可串行化快照隔离

现在，我们将在事务提交时间添加一个冲突检测算法，以便使引擎具有一定的可串行化级别。

要运行测试用例，

```
cargo x copy-test --week 3 --day 6
cargo x scheck
```

让我们通过一个可串行化的例子。考虑我们在引擎中有两个事务，它们是：

```
txn1: put("key1", get("key2"))
txn2: put("key2", get("key1"))
```

数据库的初始状态是 $key1=1$, $key2=2$ 。可串行化意味着执行的结果与按某种顺序逐个串行执行事务的结果相同。如果我们先执行 $txn1$ 然后执行 $txn2$ ，我们会得到 $key1=2$, $key2=2$ 。如果我们先执行 $txn2$ 然后执行 $txn1$ ，我们会得到 $key1=1$, $key2=1$ 。

然而，在我们的当前实现中，如果这两个事务的执行发生重叠：

```
txn1: get key2 <- 2
txn2: get key1 <- 1
txn1: put key1=2, commit
txn2: put key2=1, commit
```

我们会得到 $key1=2$, $key2=1$ 。这无法通过这两个事务的串行执行来产生。这种现象称为写偏差。

通过可串行化验证，我们可以确保对数据库的修改对应于串行执行顺序，因此，用户可以在需要可串行化执行的系统上运行一些关键工作负载。例如，如果用户在 Mini-LSM 上运行银行转账工作负载，他们期望在任何时间点的资金总额都是相同的。如果没有可串行化检查，我们无法保证这种不变式。

可串行化验证的一种技术是在系统中记录每个事务的读集和写入集。我们在提交事务之前进行验证（乐观并发控制）。如果事务的读集与其读取时间戳之后提交的任何事务的读集重叠，则我们失败验证，并中止该事务。

回到上面的例子，如果我们有 $txn1$ 和 $txn2$ 都在时间戳 = 1 开始。

```
txn1: get key2 <- 2
txn2: get key1 <- 1
txn1: put key1=2, commit ts = 2
txn2: put key2=1,
start serializable verification
```

当我们验证 txn2 时，我们将遍历所有在其预期提交时间戳之前启动且在其读取时间戳之后（在这种情况下， $1 < ts < 3$ ）的事务。唯一满足条件的事务是 txn1。txn1 的写入集是 key1，而 txn2 的读集是 key1。由于它们重叠，我们应该中止 txn2。

任务 1：在 Get 和写入集中跟踪读集

在这个任务中，你需要修改：

```
src/mvcc/txn.rs
src/mvcc.rs
```

当 get 被调用时，你应该将键添加到事务的读集中。在我们的实现中，我们存储键的哈希值，以减少内存使用并使探测读集更快，尽管这可能会导致两个键具有相同的哈希值时产生误报。你可以使用 `farmhash::hash32` 来为键生成哈希。请注意，即使 get 返回未找到的键，这个键也应该仍然在读集中被跟踪。

在 `LsmMvccInner::new_txn` 中，如果你应该创建一个空的读写集给事务，如果

任务 2：在扫描中跟踪读集

在这个任务中，你需要修改：

```
src/mvcc/txn.rs
```

在本课程中，我们仅保证 get 请求的完全可序列化。您仍然需要跟踪扫描的读集，但在某些特定情况下，您可能仍然会得到非可序列化结果。

要理解为什么这很困难，让我们通过以下示例来了解。

```
txn1: put("key1", len(scan(..)))
txn2: put("key2", len(scan(..)))
```

如果数据库以初始状态 $a=1, b=2$ 启动，我们应

$a=1, b=2, key1=2, key2=3$ 或 $a=$

是，如果事务执行

如下：

```
txn1: len(scan(..)) = 2txn2: len(scan(..)) = 2txn1: put key1 = 2,  
commit, read set = {a, b}, write set = {key1}txn2: put key2 = 2,  
commit, read set = {a, b}, write set = {key2}
```

这通过了我们的可串行化验证，并且不对应任何序列化执行顺序！因此，一个完全有效的可串行化验证需要跟踪键范围，并且如果仅调用 `get`，使用键哈希可以加速可串行化检查。请参考奖励任务，了解如何正确实现可串行化检查。

任务 3：引擎接口和可串行化验证

在此任务中，您需要修改：

```
src/mvcc/txn.rs  
src/lsm_storage.rs
```

现在，我们可以继续实现提交阶段的验证。每次我们处理事务提交时，您应该获取 `commit_lock`。这确保只有一个事务进入事务验证和提交阶段。

您需要遍历所有提交时间戳在范围 `(read_ts, expected_commit_ts)`（两个边界值均包含）的事务的读集是否与满足条件的任何事务的写入集重叠。如果我们可以提交事务，提交一个写入批处理，并将此事务的写入集插入到 `self.inner.mvcc().committed_txns` 中，其中键是提交时间戳。

如果 `write_set` 为空，可以跳过检查。只读事务总是可以提交。

您还应该修改 `put`、`delete` 和 `write_batch` 接口。我们建议您定义一个辅助函数 `write_batch_inner` 来处理写入批处理。如果 `options.serializable = true`，`put`、`delete` 和面向用户的 `write_batch` 应该创建事务而不是直接创建写入批处理。您的写入批处理辅助函数还应该返回一个 `u64` 提交时间戳，以便 `Transaction::Commit` 可以将提交的事务数据正确存储到 MVCC 结构中。

任务 4：垃圾回收

在这个任务中，你需要修改：

`src/mvcc/txn.rs`

当你提交一个事务时，你也可以清理已提交的事务映射，以删除所有低于水印的事务，因为它们将不会参与任何未来的可串行化验证。

测试您的理解

- 如果您有一些构建关系数据库的经验，您可能会思考以下问题：假设我们基于 Mini-LSM 构建一个数据库，其中我们将关系表中的每一行存储为键值对（键：主键，值：序列化行），并启用可序列化验证，数据库系统是否直接获得了 ANSI 可序列化隔离级别的能力？为什么或为什么不？
- 我们在这里实现的是实际上写快照隔离（参见 A critique of snapshot isolation），它保证了可序列化。是否存在执行是可序列化的，但会被写快照隔离验证拒绝的情况？
- 有一些数据库声称它们通过仅跟踪获取和扫描中访问的键（而不是键范围）来实现可序列化快照隔离。它们真的能防止由幻像引起的写偏差吗？（好吧……实际上，我是在谈论 BadgerDB。）

我们不会提供这些问题的参考答案，欢迎在 Discord 社区中讨论它们。

奖励任务

- 只读事务。启用可串行化后，我们需要跟踪事务的读集。
- 精确/谓词锁定。读集可以使用范围而不是单个键来维护。当用户扫描整个键空间时，这将很有用。这将还启用扫描的可序列化验证。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z授权，根据CC BY-NC-SA 4.0协议发布。

零食时间：压缩过滤器

恭喜！你做到了！在前一章中，你让你的LSM引擎实现了多版本功能，用户可以使用事务API与你的存储引擎交互。在本周结束时，我们将实现存储引擎的一些简单但重要的功能。欢迎参加Mini-LSM第3周的零食时间！

在本章节中，我们将使我们的压缩垃圾回收逻辑泛化为压缩过滤器。

目前，我们的压缩将简单地保留水印以上的键和水印以下的键的最新版本。我们可以为压缩过程添加一些魔法，以帮助用户自动收集一些未使用的数据作为后台作业。

考虑一个用户使用 Mini-LSM 存储数据库表的案例。表中的每一行都以表名作为前缀。例如，

```
table1_key1 -> row
table1_key2 -> row
table1_key3 -> row
table2_key1 -> row
table2_key2 -> row
```

现在用户执行了 `DROP TABLE table1`。引擎需要清理所有以 `table1` 开头的的数据。

有很多方法可以实现这个目标。Mini-LSM 的用户可以扫描所有以 `table1` 开头的键，并请求引擎删除它。然而，扫描一个非常大的数据库可能会很慢，并且会生成与现有键相同数量的删除墓碑。因此，扫描和删除不会释放被删除表占用的空间——相反，它会给引擎添加更多数据，并且只有当墓碑到达引擎的底层时，空间才能被回收。

或者，它们可以创建列族（我们将在剩余的“你的生活章节”中讨论）。它们将每个表存储在一个列族中，列族是一个独立的LSM状态，当用户删除表时，会直接删除对应的列族的SST文件。

在本课程中，我们将实现第三种方法：合并过滤器。合并过滤器可以在运行时动态添加到引擎中。在合并过程中，如果找到一个与合并过滤器匹配的键，我们可以在后台静默地将其删除。因此，用户可以将一个 `prefix=table1` 合并过滤器附加到引擎上，所有这些键在合并过程中都会被删除。

任务 1: 压缩过滤器

在这个任务中，您需要修改：

```
src/compact.rs
```

您可以迭代 `LsmStorageInner::compaction_filters` 中的所有压缩过滤器。如果低于水印的键的第一个版本与压缩过滤器匹配，则直接删除它，而不是将其保留在 SST 文件中。

要运行测试用例，

```
cargo x copy-test --week 3 --day 7  
cargo x scheck
```

您可以假设用户不会获取前缀过滤器范围内的键。并且，他们不会扫描前缀范围内的键。因此，当用户请求前缀过滤器范围内的键时返回错误值是可以的（即未定义行为）。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。

你的人生剩余部分（待定）

这是一个深入探讨LSM存储引擎优化和应用的高级部分，将使您的实现更生产就绪。我们仍在规划内容，这部分在不久的将来不会公开提供。

周 + 章节	主题	解决方案	入门 Code	文档
4.1	基准测试			
4.2	块压缩			
4.3	简单移动和并行压缩			
4.4	替代块编码			
4.5	速率限制器和I/O优化			
4.6	构建自己的块缓存			
4.7	构建自己的跳表			
4.8	异步引擎			
4.9	IO-uring基础I/O引擎			
4.10	预取			
4.11	键值分离			
4.12	列族			
4.13	分片			
4.14	合并优化			
4.15	Mini-LSM上的SQL			

Mini-LSM v1

这是一个遗留版本的 Mini-LSM 课程，我们将不再维护它。我们现在有这个课程的新版本。我们在这个书中保留遗留版本，以便搜索引擎可以将页面保留在索引中，用户可以点击链接到课程的新版本。

V1 课程概述

课程概述

在这个课程中，我们将用 7 天时间构建 LSM 树结构：

- 第一天：区块编码。SSTs 由多个数据块组成。我们将实现区块编码。
- 第2天：SST编码。
- 第3天：内存表和合并迭代器。
- 第4天：块缓存和引擎。为了减少磁盘I/O并最大化性能，我们将使用moka-rs构建LSM树的块缓存。在这一天，我们将获得一个功能性的（但非持久的）键值引擎，具有 `get`，`put`，`scan`，`delete` API。
- 第5天：压缩。现在，是时候为SST维护一个层级结构了。
- 第6天：恢复。我们将实现WAL和清单，以便引擎在重启后能够恢复。
- 第7天：布隆过滤器和键压缩。它们是LSM树结构中广泛使用的优化。

开发指南

我们为您提供入门代码（参见 `mini-lsm-starter crate`），其中我们简单地将所有函数体替换为 `unimplemented!()`。您可以根据此入门代码开始您的项目。我们提供了测试用例，但它们非常简单。我们建议您仔细思考您的实现并自行编写测试用例。

- 您可以使用 `cargo test` 运行所有测试用例并在您的代码库中进行风格检查。
- 您可以使用 `cargo run` 将测试用例复制到入门代码中。

您的反馈非常宝贵。欢迎加入我们的 Discord 社区。

发现问题？请在 github.com/skyzh/mini-lsm 上创建问题 / 拉取请求。

mini-lsm-book © 2022-2025 由 Alex Chi Z 创作，根据 CC BY-NC-SA 4.0 授权。

块构建器和块迭代器

i 这是一个 Mini-LSM 课程的遗留版本，我们不再维护它。我们现在有一个更好的这个课程的版本，并且这一章节现在是 Mini-LSM 第1周第3天：块。

在这一部分，你需要修改：

- `src/block/builder.rs`
- `src/block/iterator.rs`
- `src/block.rs`

你可以使用 `cargo x copy-test day1` 将我们提供的测试用例复制到入门代码目录。完成这一部分后，使用 `cargo x scheck` 检查样式并运行所有测试用例。如果你想要编写自己的测试用例，在 `block.rs` 中编写一个新的模块 `#[cfg(test)]`

。记住要删除你修改的模块顶部的 `#!`

，以便 `cargo clippy` 能够检查样式。

任务 1 - 块构建器

块是 LSM 中的最小读取单元。通常为 4KB 大小，类似于数据库页面。在每个块中，我们将存储一系列排序的键值对。

你需要修改 `BlockBuilder` 中的 `src/block/builder.rs` 来构建编码数据和偏移量数组。块包含两部分：数据和偏移量。

```
-----
| data | offsets | meta | | | | | |
| ---- | - - - - | - - - |
| entry | entry  | entry | entry | offset | offset | offset | offset |
num_of_elements |
-----
```

当用户向块（条目）添加键值对时，我们需要将其序列化为以下格式：

```
-----
|                                     Entry #1                                     | ... |
-----
| key_len (2B) | key (keylen) | value_len (2B) | value (varlen) | ... |
-----
```

键长度和值长度均为2字节，这意味着它们的最大长度为65535。(内部存储为 `u16`)

我们假设键永远不会为空，而值可以为空。空值意味着在系统的其他部分的视图中，相应的键已被删除。对于 `BlockBuilder` 和 `BlockIterator`，我们只是将空值原样处理。

在每个块的末尾，我们将存储每个条目的偏移量和条目总数。例如，如果第一个条目位于块的 0 位置，第二个条目位于块的 12 位置。

```
-----
|offset|offset|num_of_elements|
-----
|  0  |  12  |          2          |
-----
```

块的页脚如上所示。每个数字都存储为 `u16`。

块有一个大小限制，即 `target_size`。除非第一个键值对超过目标块大小，否则您应确保编码块的大小始终小于或等于 `target_size`。（在提供的代码中，这里的 `target_size` 实际上是 `block_size`）

当 `build` 被调用时，`BlockBuilder` 将生成数据部分和未编码条目偏移量。这些信息将存储在 `Block` 结构中。由于键值条目以原始格式存储，偏移量存储在单独的向量中，因此在解码数据时可以减少不必要的内存分配和处理开销——您需要做的是只需将原始块数据复制到 `data` 向量中，并每 2 字节解码条目偏移量，而不是创建类似 `Vec<(Vec<u8>, Vec<u8>)>` 的东西来在内存中存储一个块中的所有键值对。这种紧凑的内存布局非常高效。

对于编码和解码部分，您需要修改 `Block` 中的 `src/block.rs`。具体来说，您需要实现 `Block::encode` 和 `Block::decode`，它们将编码到 / 解码来自上述图中所示的数据布局。

任务 2 - 块迭代器

给定一个 `Block` 对象，我们需要提取键值对。为此，我们创建一个块上的迭代器并找到我们想要的信息。

`BlockIterator` 可以通过一个 `Arc<Block>` 来创建。如果调用 `create_and_seek_to_first`，它将定位在块中的第一个键。如果调用 `create_and_seek_to_key`，迭代器将定位在提供的键 `>=` 的第一个键。例如，如果 1、3、5 在一个块中。

```
let mut iter = BlockIterator::create_and_seek_to_key(block, b"2");
assert_eq!(iter.key(), b"3");
```

上述 `seek 2` 将使迭代器定位到 2 的下一个可用键，在这种情况下是 3。

迭代器应该从块中复制 `key` 和 `value` 并将它们存储在迭代器内部，以便用户可以在没有任何额外复制的情况下通过 `fn key(&self) -> &[u8]` 访问键和值，`> &[u8]` 直接返回

。

当 `next` 被调用时，迭代器将移动到下一个位置。如果我们到达块的末尾，我们可以将 `key` 设置为空，并从 `is_valid` 返回 `false`，以便调用者可以在可能的情况下切换到另一个块。

实现这部分后，你应该能够通过 `block/tests.rs` 中的所有测试用例。

额外任务


这里有一份你可以做的额外任务列表，以使块编码更加健壮和高效。

注意：实现这部分后，一些测试用例可能无法通过。你可能需要编写自己的测试用例。

- 实现块校验和。在解码块时验证校验和。
- 压缩/解压缩块。在 `build` 压缩并在解码时解压。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z授权，根据CC BY-NC-SA 4.0许可协议发布。

SST构建器 和 SST迭代器

 这是 Mini-LSM 课程的遗留版本，我们将不再维护它。我们现在有这个课程的更好版本，本章现在是 Mini-LSM 第1周第4天：排序字符串表 (SST) 的一部分。

在这一部分，你需要修改：

- `src/table/builder.rs`
- `src/table/iterator.rs`
- `src/table.rs`

你可以使用 `cargo x copy-test day2` 将我们提供的测试用例复制到入门代码目录。完成这一部分后，使用 `cargo x scheck` 检查样式并运行所有测试用例。如果你想要编写自己的测试用例，在 `table.rs` 中编写一个新的模块 `#[cfg(test)]`

。记住要删除你修改的模块顶部的 `#!`

，以便 `cargo clippy` 能够检查样式。

任务 1 - SST构建器

SST 由存储在磁盘上的数据块和索引块组成。通常，数据块是惰性加载的——它们不会加载到内存中，直到用户请求它们。索引块也可以按需加载，但在本课程中，我们做了简单的假设，即所有 SST 索引块（元块）都可以适合内存。通常，一个 SST 文件的大小为 256MB。

SST构建器与块构建器类似 -- 用户将在构建器上调用 `add` 。您应该在SST构建器内部维护一个 `BlockBuilder` ，并在必要时拆分块。此外，您需要维护块元数据 `BlockMeta` ，其中包括每个块中的第一个键和每个块的偏移量。 `build` 函数将编码SST，使用 `FileObject::create`将所有内容写入磁盘，并返回一个 `SsTable` 对象。请注意，在第二部分中，您不需要实际将数据写入磁盘。只需将所有内容作为向量保存在内存中，直到我们实现块缓存（第4天，任务5）。

SST 的编码方式如下：

Block Section		Meta Section	
Extra			
data block ... data block meta block ... meta block meta block			
offset (u32)			

您还需要实现 `estimated_size` 功能的 `SsTableBuilder`，以便调用者知道何时可以开始一个新的 SST 来写入数据。该功能不需要非常精确。假设数据块包含的数据远多于元块，我们可以简单地返回数据块的大小给 `estimated_size`。

您也可以将块对齐到4KB边界，以便将来能够进行直接I/O。这是一个可选的优化。

完成任务 1 的推荐顺序如下：

- 在 `src/table` 实现
 - 在实现 `SsTableBuilder` 之前，您可能想要查看 `src/table.rs`，因为 `Block`
 - 对于 `FileObject`，您在第4天之前至少需要实现 `read`、`size` 和 `create`（无需磁盘I/O）。
 - 对于 `BlockMeta`，当编码/解码 `BlockMeta` 到/来自缓冲区时，您可能想要添加一些额外信息。
- 在 `src/table` 实现 `BlockCache`
 - 与上述相同，你无需担心 `BlockCache`，直到第4天

完成任务 1 后，你应该能够通过所有当前测试，除了两个迭代器测试。

任务 2 - SST 迭代器

与 `BlockIterator` 类似，你需要实现一个 SST 的迭代器。请注意，你应该按需加载数据。例如，如果你的迭代器位于块 1，它不应在到达下一个块之前在内存中保留任何其他块内容。

`SsTableIterator` 应实现 `StorageIterator` 特性，以便将来可以与其他迭代器组合。

需要注意的一点是 `seek_to_key` 函数。基本上，您需要通过二分搜索块元数据来查找可能包含键的块。键可能

不存在于 LSM 树中，因此块迭代器在查找后立即会无效。例如，

```
-----  
| block 1 | block 2 | block meta |  
-----  
| a, b, c | e, f, g | 1: a, 2: e |  
-----
```

如果我们在这个 SST 中执行 `seek(b)`，那就很简单——使用二分搜索，我们可以知道块 1 包含键 $a \leq \text{keys} < e$ 。因此，我们加载块 1 并将块迭代器定位到相应位置。

但如果我们执行 `seek(d)`，我们会定位到块 1，但在块 1 中查找 `d` 会到达块末尾。因此，我们应该在查找后检查迭代器是否无效，并在必要时切换到下一个块。

额外任务


这里是一系列你可以做的额外任务，以使块编码更加健壮和高效。

注意：在实现这部分后，某些测试用例可能无法通过。你可能需要编写自己的测试用例。

- 实现索引校验和。解码时验证校验和。
- 探索不同的 SST 编码和布局。例如，在 Lethe 论文中，作者为 SST 添加了辅助键支持。

您的反馈非常感谢。欢迎加入我们的 Discord 社区。发现问题时？请在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由 Alex Chi Z 提供，根据 CC BY-NC-SA 4.0 协议授权。

内存表和合并迭代器

 这是一个Mini-LSM课程的遗留版本，我们将不再维护它。我们现在有一个更好的课程版本，本章现在是Mini-LSM第1周第1天：内存表和Mini-LSM第1周第2天：合并迭代器的一部分。

在这一部分，你需要修改：

- `src/iterators/merge_iterator.rs`
- `src/iterators/two_merge_iterator.rs`
- `src/mem_table.rs`

你可以使用 `cargo x copy-test day3` 将我们提供的测试用例复制到启动代码目录。完成这一部分后，使用 `cargo x scheck` 检查样式并运行所有测试用例。如果你想要编写自己的测试用例，在 `table.rs` 中编写一个新的模块 `#[cfg(test)]`

。记得在修改的模块顶部删除 `#![all(...)]` ，

`py` 能够实际检查样式。

这是 LSM 树基本构建块的最后一部分。实现合并迭代器后，我们可以轻松地从数据结构的不同部分（内存表 + SST）合并数据，并获取所有数据的迭代器。在第四部分，我们将把这些东西组合在一起，制作一个真正的存储引擎。

任务 1 - 内存表

在本课程中，我们使用 `crossbeam-skiplist` 作为内存表的实现。跳表类似于链表，其中数据存储存储在列表节点中，并且不会在内存中移动。跳表中的节点不是使用单个指针来指向下一个元素，而是包含多个指针，允许用户“跳过一些元素”，因此我们可以实现 $O(\log n)$ 搜索、插入和删除。

在存储引擎中，用户将创建对数据结构的迭代器。通常，一旦用户修改了数据结构，迭代器就会变得无效（C++ STL和Rust容器的情况就是这样）。然而，跳表允许我们在同一时间访问和修改数据结构，因此当存在并发访问时，有可能提高性能。有些论文认为跳表不好，但数据在内存中保持原位这一良好特性可以使我们的实现更容易。

在 `mem_table.rs` 中，您需要基于 `crossbeam-skiplist` 实现 `mem-table`。请注意，内存表仅支持 `get`，`scan` 和 `put` 而不包含 `delete`。删

以墓碑 `key -> empty value` 表示，实际数据将在压缩过程（第5天）中删除。请注意，所有 `get`，`scan`，`put` 函数仅需 `&self`，这意味着我们可以并发调用这些操作。

任务 2 - 内存表迭代器

您现在可以实现一个迭代器 `MemTableIterator`。

`memtable.iter(start, end)` 将创建一个迭代器，该迭代器返回范围内的所有元素 `start, end`。在这里，`start` 是 `std::ops::Bound`，它包含 3 个变体：`Unbounded`，`Included(key)`，`Excluded(key)`。`std::ops::Bound` 的表达能力消除了记住API是否具有闭区间或开区间的需要。

请注意，`crossbeam-skiplist`的迭代器与跳表本身具有相同的生命周期，这意味着在使用迭代器时，我们始终需要提供生命周期。这非常难用。您可以使用 `ouroboros crate` 创建一个自引用结构体，该结构体会擦除生命周期。您会发现`ouroboros`示例很有帮助。

```
pub struct MemTableIterator {  
    /// hold the reference to the skiplist so that the iterator will be valid.  
    map: Arc<Skiplist>  
    /// then the lifetime of the iterator should be the same as the  
    `MemTableIterator` struct itself  
    iter: Skiplist::Iter<'this>  
}
```

您还需要将Rust风格的迭代器API转换为我们的存储迭代器。在Rust中，我们使用 `next() -> Data`。但在本课程中，`next` 没有返回值，数据应该通过 `key()` 和 `value()` 获取。您需要想一个方法来实现这一点。

► Spoiler: the MemTableIterator 结构体

在这个设计中，您可能已经注意到，只要我们拥有迭代器对象，内存表就无法从内存中释放。在本课程中，我们假设用户操作是短暂的，因此这不会引起大问题。请查看附加任务以了解可能的改进方案。

你也可以考虑使用 `AgateDB` 的跳表实现，这可以避免创建自引用结构体的问题。

任务 3 - 合并迭代器

现在你已经有很多内存表和 SST，你可能想将它们合并以获取键的最新出现。在 `merge_iterator.rs` 中，我们有一个

一个合并所有相同类型迭代器的迭代器。在 `new` 函数中，索引位置较低的迭代器具有更高的优先级，也就 `new`，如果我们有：

```
iter1: 1->a, 2->b, 3->c
iter2: 1->d
iter: MergeIterator::create(vec![iter1, iter2])
```

最终的迭代器将产生 `1->a`、`2->b`、`3->c`。`iter1` 中的数据将覆盖其他迭代器中的数据。

您可以使用一个 `BinaryHeap` 来实现此合并迭代器。请注意，您永远不应该将任何无效迭代器放入二叉堆中。一个常见的陷阱是错误处理。例如，

```
let Some(mut inner_iter) = self.iters.peek_mut() {
    inner_iter.next()?; // <- will cause problem
}
```

如果 `next` 返回错误（即由于磁盘故障、网络故障、校验和错误等），它就不再有效。然而，当我们退出 `if` 条件并将错误返回给调用者时，`PeekMut` 的删除将尝试移动堆中的元素，这会导致访问无效迭代器。因此，您需要自己处理所有错误，而不是在 `PeekMut` 的范围内使用 `?`。

您还需要为存储迭代器定义一个包装器，以便 `BinaryHeap` 可以跨所有迭代器进行比较。

任务 4 - 双合并迭代器

LSM 有两种结构用于存储数据：内存中的内存表和磁盘上的 SSTs。在我们分别构建了所有 SSTs 和所有内存表的迭代器后，我们需要一个新的迭代器来合并两种不同类型的迭代器。这就是 `TwoMergeIterator`。

您可以在 `two_merge_iter.rs` 中实现 `TwoMergeIterator`。类似于 `MergeIterator`，如果相同的键在两个迭代器中都存在，则第一个迭代器具有优先权。

在本课程中，我们明确不使用类似 `Box<dyn StorageIter>` 的东西来避免动态分发。这是 LSM 存储引擎中的一个常见优化。

附加任务


- 实现不同的内存表，并查看它与跳表有何不同。例如，BTree 内存表。您会注意到，在不持有 B+ 树的情况下，很难获取一个迭代器。

与迭代器相同时间段的锁。你可能需要想出聪明的方法来解决这个问题。

- 异步迭代器。一个有趣的问题是探索是否可以在存储引擎中异步化所有操作。你可能会发现一些与生命周期相关的问题，需要解决这些问题。
- 前台迭代器。在本课程中，我们假设所有操作都是短期的，因此我们可以在迭代器中持有对内存表的引用。如果一个迭代器被用户长时间持有，整个内存表（可能为256MB）将保持在内存中，即使它已经被刷新到磁盘。为了解决这个问题，我们可以为用户提供一个 `ForegroundIterator` / `LongIterator` 。迭代器将定期创建新的底层存储迭代器，以允许资源的垃圾回收。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？请在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z提供，根据CC BY-NC-SA 4.0协议授权。

存储引擎和块缓存

 这是一个Mini-LSM课程的遗留版本，我们将不再维护它。我们现在有一个更好的课程版本，本章现在是Mini-LSM第1周第5天：读取路径和Mini-LSM第1周第6天：写入路径的一部分。

在这一部分，你需要修改：

- `src/lsm_iterator.rs`
- `src/lsm_storage.rs`
- `src/table.rs`
- 其他使用 `SsTable::r`

你可以使用 `cargo x copy-test day4` 将我们提供的测试用例复制到启动代码目录。完成这一部分后，使用 `cargo x scheck` 检查样式并运行所有测试用例。如果你想要编写自己的测试用例，在 `table.rs` 中编写一个新的模块 `#[cfg(test)]`

。记住要删除你修改的模块顶部的 `#!`

，以便 `cargo clippy` 能够实际检查样式。

任务 1 - Put 和 Delete

在实现 `put` 和 `delete` 之前，让我们重新回顾一下 LSM 树的工作原理。LSM 的结构包括：

- 内存表：一个活动可变内存表和多个不可变内存表。
- 预写日志：每个内存表对应一个WAL。
- SSTs：内存表可以以SST格式刷新到磁盘。SSTs按多个层级组织。

在这一部分，我们只需要获取锁，将条目（或墓碑）写入活动内存表。你可以修改 `lsm_storage.rs`。

任务 2 - 获取

要从 LSM 获取值，我们可以简单地从活动内存表、不可变的内存表（从最新到最早）以及所有 SST 进行探测。为了减少临界区，我们可以持有读锁，将所有指向内存表和 SST 的指针从

`LsmStorageInner` 结构中复制出来，并在临界区之外创建迭代器。在创建迭代器和探测时，要小心顺序。

任务 3 - 扫描

要创建扫描迭代器 `LsmIterator`，您需要使用 `TwoMergeIterator` 在内存表上合并 `MergeIterator`，并在 SST 上合并 `MergeIterator`。您可以在 `lsm_iterator.rs` 中实现这一点。可选地，您可以实现 `FusedIterator`，以便如果用户在迭代器变得无效后意外调用 `next`，底层的迭代器不会恐慌。

`TwoMergeIterator` 生成的键值对序列可能包含空值，这意味着值已被删除。`LsmIterator` 应过滤掉这些空值。它还需要正确处理起始和结束边界。

任务 4 - 同步

在这一部分，我们将在 `lsm_storage.rs` 中实现内存表和刷新到 L0 SST。与任务 1 类似，写入操作直接进入活动可变内存表。一旦 `sync` 被调用，我们将分两步将 SST 刷新到磁盘：

- 首先，将当前可变内存表移动到不可变内存表列表中，以便未来的请求不会进入当前内存表。创建一个新的内存表。所有这些都应该在一个单一的临界区中完成，并阻止所有读取操作。
- 然后，我们可以将内存表刷新到磁盘作为 SST 文件，而无需持有任何锁。
- 最后，在一个临界区中，删除内存表并将 SST 放入 `l0_tables`。

一次只能有一个线程同步，因此您应该使用互斥锁来确保这一要求。

任务5 - 块缓存

现在我们已经实现了 LSM 结构，可以开始将一些内容写入磁盘！在之前的 `table.rs` 中，我们实现了一个 `FileObject` 结构，但没有写入任何内容到磁盘。在这个任务中，我们将更改实现，以便：

- `read` 将使用 `read_exact_at` `instd::os::unix::fs::FileExt` 从磁盘读取，
- 文件的大小应该存储在结构体中，`size` 函数直接返回它。

- `create` 应该将文件写入磁盘。通常你应该在那个文件上调用 `fsync`。但这会大大减慢单元测试的速度。因此，我们直到第6天恢复才进行`fsync`。
- `open` 将在第6天恢复之前未实现。

之后，我们可以在 `SsTable` 上实现一个新的 `read_block_cached` 功能，以便我们可以利用块缓存来服务读取请求。在初始化 `LsmStorage` 结构时，您应该使用 `moka-rs` 创建一个4GB大小的块缓存。块通过SST ID +和块ID进行缓存。使用 `try_get_with` 从缓存中获取块/如果缓存未命中则填充缓存。如果有多个请求读取相同的块且缓存未命中，`try_get_with` 将只向磁盘发出一个读取请求，并将结果广播给所有请求。


请记得将 `SsTableIter` 缓存。

附加任务

- 如您所见，每次我们执行获取、put或删除操作时，都需要获取一个保护LSM结构的读锁；如果我们想要刷新，则需要获取一个写锁。这可能会导致很多问题。某些锁实现是公平的，这意味着只要有一个写者在等待锁，就没有读者可以获取锁。因此，写者必须等待最慢的读者完成其操作后才能实际执行一些工作。一种可能的优化是实现 `WriteBatch`。我们不需要立即将用户的请求写入内存表+ WAL。我们可以允许用户进行一批写入。
- 将块对齐到4K并使用直接I/O。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z授权，根据CC BY-NC-SA 4.0协议发布。

分层压缩


 这是一个Mini-LSM课程的遗留版本，我们将不再维护它。我们现在有一个更好的课程版本，本章现在属于：

- [Mini-LSM 第2周 第1天：压缩实现](#)
- [Mini-LSM 第2周 第2天：简单压缩策略](#)
- [Mini-LSM 第2周 第3天：分层压缩策略](#)
- [Mini-LSM 第2周 第4天：分层压缩策略](#)

我们没有将本章作为 Mini-LSM v1 的一部分完成。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z授权，根据CC BY-NC-SA 4.0协议发布。

预写日志 (WAL) 用于恢复

 这是一个Mini-LSM课程的遗留版本，我们将不再维护它。我们现在有一个更好的课程版本，本章现在属于：

- [Mini-LSM 第2周 第5天：清单](#)
- [Mini-LSM 第2周 第6天：预写日志 \(WAL\)](#)

我们未将本章作为 Mini-LSM v1 的一部分完成。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。

布隆过滤器




这是一个Mini-LSM课程的遗留版本，我们将不再维护它。我们现在有一个更好的课程版本，本章现在是Mini LSM 第1周第7天：SST优化的一部分。

我们未将本章作为 Mini-LSM v1 的一部分完成。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。

键压缩

 这是一个Mini-LSM课程的遗留版本，我们将不再维护它。我们现在有一个更好的课程版本，本章现在是Mini LSM 第1周第7天：SST优化的一部分。

我们未将本章作为 Mini-LSM v1 的一部分完成。

您的反馈非常感谢。欢迎加入我们的Discord社区。发现问题时？在 github.com/skyzh/mini-lsm上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z发布，根据CC BY-NC-SA 4.0协议授权。

接下来是什么

我们未将本章作为Mini-LSM v1的一部分完成。

非常感谢您的反馈。欢迎加入我们的Discord社区。发现了一个问题？在 github.com/skyzh/mini-lsm 上创建问题/拉取请求。mini-lsm-book © 2022-2025 由Alex Chi Z创作，根据CC BY-NC-SA 4.0协议授权。