



SHANDONGＵＮＩＶＥＲＳＩＴＹ　ＯＦ　ＴＥＣＨＮＯＬＯＧＹ

毕业设计说明书

基于LSM-Tree的键值存储系统的

设计与实现

学 院：计算机科学与技术学院

专 业：软件工程

学生姓名：陈现森

学 号：16110506014

指导教师：王海鹏

2020 年 6 月

# 摘 要

随着互联网的快速发展，大规模的数据量给传统关系型数据库带来了巨大考验。维护着大规模数据量的应用需要更高的读写性能。同时大数据时代的特点是存在着海量非结构化的数据。传统关系型数据库管理海量非结构化数据时性能表现并不强势。因此，NoSQL数据库应运而生，快速发展。键值存储便是NoSQL的一种经典类型。

存储引擎几乎决定着存储系统的性能。本文调研了如今流行的键值存储系统的存储引擎。使用Golang语言，实现了基于日志结构合并树的键值存储系统。有一个反直觉的事实是，顺序读写磁盘快于随机读写主存。日志结构合并树的核心思想便是放弃部分读性能以换取写性能，核心思路是顺序写入，将数据追加到日志后保存在内存中,内存数据达到阈值后有序写入硬盘中，按照一定策略合并硬盘中的数据文件，丢弃失效数据.将记录写入日志是直接追加，没有不必要的硬盘随机访问。在内存中有序地维护数据，也减少了将数据写入硬盘的随机访问次数。基于日志结构合并树的存储引擎有着优异的写入性能，以及不俗的读取性能。

关键词：键值存储、存储引擎、日志结构合并树

# Abstract

With the rapid development of the Internet, the large-scale data volume has brought a huge test to the traditional relational database. Applications that maintain large amounts of data require higher read and write performance. At the same time, the big data era is characterized by the existence of massive unstructured data. Traditional relational databases do not perform well when managing massive amounts of unstructured data. Therefore, NoSQL database came into being and developed rapidly. Key-value storage is a classic type of NoSQL.

The storage engine almost determines the performance of the storage system. This article investigates the storage engines of today's popular key-value storage systems. Using Golang language, a key value storage system based on log structure merge tree is realized. The counter-intuitive fact is that sequential reads and writes to disk are faster than random reads and writes to main memory. The core idea of ​​the log structure merge tree is to give up part of the read performance in exchange for write performance. The core idea is to write sequentially, append the data to the log and save it in memory, and write it to the hard disk in order after the memory data reaches the threshold. The strategy merges the data files in the hard disk and discards the invalid data. The record is directly appended to the log without unnecessary random hard disk access. Maintaining data in memory in an orderly manner also reduces the number of random accesses to write data to the hard disk. The storage engine based on the log structure merge tree has excellent write performance and good read performance.

**Key words:** Key-value Storage, Storage engine, Log-Structured Merge-Tree

# 目 录

[摘 要 I](#_Toc42251129)

[Abstract II](#_Toc42251130)

[目 录 III](#_Toc42251131)

[第一章 引言 1](#_Toc42251132)

[1.1 课题的背景和意义 1](#_Toc42251133)

[1.2 日志结构合并树的出现与发展 2](#_Toc42251134)

[1.3 研究内容与主要工作 2](#_Toc42251135)

[1.4 本文的结构安排 3](#_Toc42251136)

[第二章 相关技术与理论 4](#_Toc42251137)

[2.1 预写式日志 4](#_Toc42251138)

[2.2 跳表结构 4](#_Toc42251139)

[2.3 布隆过滤器 5](#_Toc42251140)

[2.4 日志结构合并树 6](#_Toc42251141)

[第三章 系统总体设计 8](#_Toc42251142)

[3.1 系统接口 8](#_Toc42251143)

[3.2 总体框架 8](#_Toc42251144)

[3.3 读写流程 10](#_Toc42251145)

[3.3.1 写入流程 10](#_Toc42251146)

[3.3.2 读取流程 11](#_Toc42251147)

[第四章 系统方案实现 12](#_Toc42251148)

[4.1 文件存储模块 12](#_Toc42251149)

[4.1.1 键值对类型 12](#_Toc42251150)

[4.2 数据日志模块 13](#_Toc42251151)

[4.3 内存表模块 14](#_Toc42251152)

[4.4 硬盘表模块 16](#_Toc42251153)

[4.4.1 SSTable 16](#_Toc42251154)

[4.4.2 SSTable的管理 18](#_Toc42251155)

[4.5 压缩合并模块 18](#_Toc42251156)

[4.6 布隆过滤器模块 20](#_Toc42251157)

[第五章 测试与分析 21](#_Toc42251158)

[5.1 测试环境 21](#_Toc42251159)

[5.2 功能测试 21](#_Toc42251160)

[5.2.1 增删改查 21](#_Toc42251161)

[5.2.2 系统恢复 21](#_Toc42251162)

[5.2.3 其他功能测试 22](#_Toc42251163)

[5.3 性能测试 22](#_Toc42251164)

[5.3.1 go的基准测试 22](#_Toc42251165)

[5.3.2 redis-benchmark基准测试 24](#_Toc42251166)

[结 论 27](#_Toc42251167)

[参考文献 28](#_Toc42251168)

[致 谢 29](#_Toc42251169)

[附录A 程序主要部分源代码 30](#_Toc42251170)

[附录B 程序使用说明书 31](#_Toc42251171)

# 第一章 引言

## 1.1 课题的背景和意义

在19世纪中期，人工成本远低于计算成本，当时的数据库工程师会耗费大量人工来重写数据库应用程序。图灵奖获得者Ted Codd于1970年提出了关系模型[1]，之后便出现了基于SQL的关系型数据库。而NoSQL则泛指非关系型的数据库。NoSQL的出现使关系型数据库的地位不再那么稳固。

NoSQL主要包括键值存储数据库、列存储数据库、文档型数据库及图数据库。键值存储数据库的有着NoSQL的优点，如易扩展、读写性能高。由于存取策略的不同，Key-value数据库相对于传统关系型数据库来说有着更快的读写速度。

传统关系型数据库由于数据容量与访问量的迅速增长，处理速度无法达到新的应用需求，因此键值存储系统更符合当前数据密集型应用的性能需求。Key-value数据库相对于传统关系型数据库有着更好的扩展能力。关系型数据库受到数据模型的限制，水平扩展（增加系统节点）会出现许多问题。键值存储系统的水平扩展能力更好，因此目前流行的分布式数据库的底层存储引擎大多都是Key-value存储引擎。

如今大数据的特点是，数据维度广，但每一个元组并不一定具备这些维度的信息。非结构化组织的数据由于没有固定模式和规则约束，关系型数据库在管理海量非结构化数据时不仅会加大运维人员的工作量，还会带来性能上的损失。对于许多传统企业来说，通常需要维护大量的非结构化的数据，由于Key-value数据库在管理非结构化数据上的优势，Key-value数据库自然成为了这类企业逐步信息化的有力工具。

作为Key-value数据库的核心，采用不同策略设计出的存储引擎对数据库在不同应用场景下的性能有着显著影响。目前主流的键值存储系统大部分都采用了LSM-Tree[2]（Log-structured Merge-tree），即日志结构合并树模型，如BigTable, LevelDB, Bitcask等。

## 1.2 日志结构合并树的出现与发展

对于不同的应用场景，数据库会采用不同索引方式及不同存储策略的存储引擎。目前最常见的存储引擎有三类：哈希存储引擎、B-tree存储引擎和LSM-Tree存储引擎。对于传统关系型数据库，B-tree在其存储引擎中应用广泛。B-tree索引的树的高度较低，索引时磁盘的读取次数少，因此也有较为优异的读取表现，但是在树上的修改，如增、删、改操作会带来大量的磁盘随机I/O，磁盘寻道时间会明显增多，性能也会因此受到较大影响。

对于键值存储系统，目前流行的存储引擎采用的是日志结构合并树的存储引擎和哈希索引的存储引擎。哈希存储引擎的读取性能优异，但是在数据量不断增大时，会产生哈希冲突的数据会显著增多，从而影响到建立索引的效率，因此较为普遍的解决方案是采用树形索引。

1969年，Patrick O’Neil提出了LSM-Tree架构。LSM-Tree架构的主要存储策略是通过在内存中处理写入请求，将数据批量写入到硬盘中，从而使对磁盘的随机写转换成顺序写，通过牺牲读取性能换取更高的写入性能。他在文中讨论了最简单的日志结构合并树模型，由两个组件构成，分别是常驻于内存的C0树与硬盘中的C1树。当C0树中的数据量超出阈值时，C0树中的数据会批量刷入磁盘上的C1树中。数据落盘后会在满足一定条件时进行数据合并操作，保证了磁盘上C1树的数据有序，同时也利于数据的读取。

2012年，R.Sears提出了bLSM-tree[3] 以提升日志结构合并树的读取速度。bLSM-tree采用了布隆过滤器(bloom filter)[4] 来过滤一些没有意义的查找，从而减少了大量的磁盘I/O。此外，Zhang引入了弹性布隆过滤器，Wang提出了pLSM-tree[5]来提高日志结构合并树的读取性能。为了减少日志结构合并树的写放大，LSM-Trie[6]通过使用字典树来存储LSM-Tree的数据信息，更改了数据合并方式。

在应用领域，谷歌公司基于日志结构合并树开发了键值存储系统BigTable。谷歌公司的两名开发人员开发了LevelDB并将其开源。

## 1.3 研究内容与主要工作

为实现一个可靠的基于LSM-tree的持久化的键值存储系统，本文调研了键值存储系统目前流行的架构以及日志结构合并树的出现与发展，研究了日志结构合并树在面对写多读少情况时表现优异的特点。一般意义上，涉及到磁盘操作的程序的性能瓶颈不在于处理器和内存，而是由磁盘带宽决定。由于持久化存储系统的读取操作往往要在文件中进行寻找，所以减少磁盘I/O是优化读取操作的重点。参考如今流行的谷歌开源单机键值存储系统LevelDB，本文基于日志结构合并树架构开发了一个键值存储系统，并在其基础上尝试了融入布隆过滤器与缓存，以优化系统读取速度。

## 1.4 本文的结构安排

本文对基于日志结构合并树的存储引擎进行调研，设计与实现了基于日志结构合并树的键值存储系统，章节内容安排如下：

第一章为引言。本章节首先介绍了SQL与NoSQL的出现、NoSQL中的键值存储系统、日志结构合并树的出现与发展，介绍了课题的研究背景和意义。

第二章为相关技术与理论。本章节主要介绍预写式日志、跳跃表结构、日志结构合并树及布隆过滤器。

第三章为系统总体设计，介绍系统的对外接口、总体框架、读写流程等。

第四章为系统方案实现，介绍文件存储、数据日志、内存表、硬盘表等多个模块的实现思路和方案。

第五章为性能测试与分析，对系统进行功能测试与性能测试，并与其他系统进行对比。

# 第二章 相关技术与理论

## 2.1 预写式日志

预写式日志(WAL, Write Ahead Log)，是数据库系统中常见的一种手段，用于保证数据操作的原子性和持久性，该文件只会追加，不会修改和覆盖。

在计算机工程领域中，问题大都可以通过添加一个中间层来解决。预写式日志的核心思想就是系统中所有的修改操作都要在提交前写入到日志文件中，通过日志记录数据的变化后，再写入缓冲区中，等缓冲区达到阈值时，再刷入磁盘进行持久化操作。在磁盘数据未损坏的情况下，预写式日志支持系统在崩溃重启后恢复到崩溃前的状态。在大多数情况下，预写式日志拥有更好的性能，并且预写式日志的读与写可以并发执行，不会相互阻塞。

在关系数据库中，预写式日志文件通常包括重做(Redo Log)和撤销(Undo Log)信息。预写式日志并不仅存在于传统数据库中，预写式日志在需要原子性和持久化的系统中也经常出现。在Linux文件系统中，预写式日志通常叫做Journaling。Journaling减轻了存储系统实现难度，使得文件系统支持原子写入。Journaling会在存储设备中申请一段空间来记录文件操作，修改操作会先写入日志，然后再进行文件修改操作，确定修改完成后再删除日志。系统挂载文件时，会先检查日志是否有未完成的操作。若有，则会进行重做。

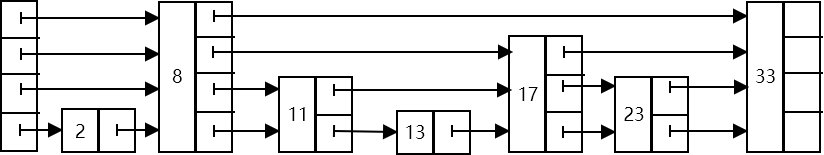
在使用复制状态机(Replicated State Machine)进行协作的系统中，如Raft[7]算法的实现就需要预写式日志配合来保证数据的安全性（在发生异常的情况）。

由此可见，预写式日志在存储系统中地位显著，从文件系统、数据库系统到分布式系统都有它的身影。

## 2.2 跳表结构

在日志结构合并树里，数据存在于内存与磁盘中。系统新写入的数据总会先插入到高速的内存缓冲区中，系统的读取操作也会首先在内存缓冲区中查找。由此可见，在内存缓冲区中维护一个高效的数据结构以支持数据的修改和查找操作对于日志结构合并树有重要意义。目前，日志结构合并树在内存中通常使用AVL树、红黑树或跳表来保存缓冲区的数据。跳表相对于另外两个数据结构来说，实现更为简单，也易于接受。本节研究内容基于对日志结构合并树的分析，在其中引入了基于内存的支持高效修改与查找的数据结构。

即使是有序链表，也无法通过二分查找来快速获取元素，因此查询时间复杂度只能达到O(n) 。参考了二分查找的思想，William Pugh提出了基于概率生成高度的跳表[8]，为每个有序链表中元素维护一个多层链表，如图2-1所示。跳跃表是在有序链表的基础上增加了向后的指针，通过这些指针，能够快速遍历到需要的位置，提高了查询的效率。跳表中每个元素的高度是由随机函数决定的，当然每个元素的高度要小于预设最大高度。

图2-1 跳表

如图2-1所示，假如我要查找23时，对于跳表来说只需要遍历3个节点即可找到23这个元素。跳表首先会从头节点的最高层找起，直到找到首个尾指针指向元素大于23的元素停止，即8这个元素；紧接着会在8这个元素的下一层指针开始寻找，同样找到首个尾指针指向元素大于23的元素停止，即17这个元素；然后会在17这个元素的下一层开始寻找，发现17的下一个元素即为23，查找结束。如果对于有序链表来说，查找过程则要逐个遍历到23。

跳表平均期望的查找、插入、删除时间复杂度都是O(logn)，可以在不少应用中代替树结构，如著名的内存键值数据库Redis和谷歌公司的知名项目BigTable。

## 2.3 布隆过滤器

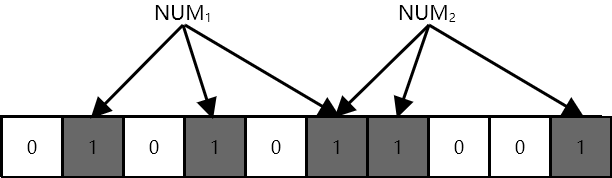


图2-2 插入两个元素后的布隆过滤器

由于在SSTable(Sorted String Table，即有序字符串表)中查找数据会涉及到磁盘I/O，所以LSM-Tree存储引擎的读取操作的性能会受到极大影响。研究者针对LSM-Tree读性能的缺陷提出了很多优化方案，布隆过滤器便是一种优化效果不错的方案，可以降低查询的平均磁盘I/O次数，提高存储系统的读取性能。

布隆过滤器由Burton Howard提出，是一种空间效率与时间效率都很高的随机数据结构，主要用于检测集合是否包含某一项元素。布隆过滤器实际由一串二进制向量与一系列哈希映射函数组成。因此布隆过滤器与通过哈希判断元素是否存在集合中相比可以节省相当大的空间。不过布隆过滤器存在一定概率的误报，即元素可能不在集合中，但布隆过滤器却认为该元素存在于集合中。

如果想判断集合中某个元素是否存在，最简单的思路就是将集合中的数据都保存起来，然后通过高效的比较来查找。时间复杂度更低的思路就是用哈希表来判断元素是否存在。布隆过滤器包含有m位的二进制向量数组，初始化每一位都为0。当插入一个元素时，通过n个相互独立的哈希函数计算出相应的哈希值，并将二进制数组对应位置置为1（如图2-2所示）。当查询一个元素时，通过n个哈希函数计算出哈希值，如果这n个哈希值在二进制数组中对应位置的值均为1，则该元素可能存在于集合中。若哈希值映射的值非均为1，则一定不存在。

可以看出，布隆过滤器的时间复杂度和空间复杂度都是常数。并且n个相互独立的哈希函数可以通过并行计算来加速计算哈希值。布隆过滤器误报元素存在于集合中便是因为不同元素的哈希值可能存在冲突。而n个相互独立的哈希函数则极大减小了完全冲突的可能性。

## 2.4 日志结构合并树

日志结构合并树的核心思想就是放弃部分读取性能，换取更快的写入性能，它通过预写日志保证数据的安全与完整，将用户写入的数据保存在内存中，并通过后台的归并操作将内存中的数据持久化到磁盘上的组件。由于在将数据持久化到磁盘组件的过程中，数据是有序的，可以顺序写入文件，减少了大量磁盘寻道时间。

在1969年Patrick O’Neil提出的LSM-Tree架构中，LSM-Tree是由两个或者更多的类树组件构成的。一个拥有两个组件的LSM-Tree由一个常驻内存的组件（图2-2中的C0 tree）与一个常驻硬盘的组件（图2-3中的C1 tree）组成。数据在系统中的写入流程是这样的，一条记录被追加到预写式日志中，随即会被插入到内存中的组件C0 tree上。数据插入到C0 tree上写入过程就结束了，所以数据的插入可以认为是没有磁盘寻道时间的开销的。但是当C0 tree中的数据达到预设的阈值后，C0 tree的数据就要在后台刷入磁盘中。读取流程都是先在C0 tree上查找，再到C1 tree上查找。可见，C0 tree的数据持久化到C1 tree上是有时间延迟的，当系统崩溃时就有可能产生数据丢失的情况，预写式日志正好弥补了这个缺点。在更普遍的情况下，硬盘中的组件可以是C1 tree - Cn tree。

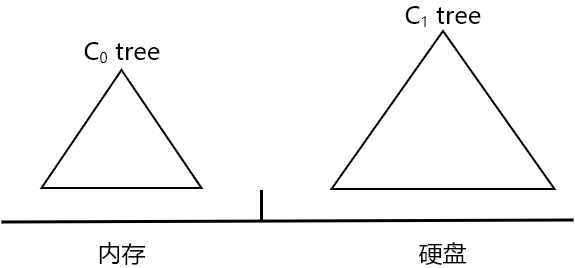


图2-3 LSM-Tree的简单模型

在LevelDB的实现中，内存中的类树组件叫做memtable，由跳表实现。LevelDB在内存中维护两个memtable，当可读可写的memtable的大小达到阈值时，会变成可读不可写的immutable memtable，同时生成一个新的memtable。硬盘中的类树组件由n层的SSTable文件组成，每一层有若干个大小为2MB的SSTable文件。

immutable memtable会在后台进程中压缩写入到位于0层的SSTable文件中。当第0层的SSTable文件个数或者大小之和达到预设阈值后，第0层部分范围重叠的SSTable文件会与第1层范围重叠的文件归并生成新的SSTable文件存储在第1层。第x（0<x<n）层的文件大小之和达到阈值时，会选取其中的一个文件与第x+1层的文件进行归并，归并后产生的新的SSTable文件位于第x+1层，参与归并的文件会被移除。LevelDB的写操作与上述两个组件的LSM-Tree写操作流程大致相同。LevelDB的读操作流程是先在memtable中查找，再去immutable memtable中查找，然后会在SSTable文件中逐层查找。

# 第三章 系统总体设计

## 3.1 系统接口

保持简洁易用的接口和优雅的设计是开发一个系统的重要原则。简洁易用的接口可以让用户更易于接受和使用，系统内部的实现细节对于用户是透明的。键值存储系统的最基本的接口包括Get、Put、Delete，修改操作可以直接用Put接口进行。针对系统的接口功能，本系统主要分为三类：

1）系统操作接口：键值存储系统的常规接口就是存储系统的创建与启动接口，还有系统的删除与关闭接口，同时提供了一个创建或打开系统接口。系统的创建接口提供了系统名称配置与内存数据大小限制。

2）数据操作接口：键值存储系统最常见的是增删改查接口，在本系统中，修改与增加记录操作都是由Put接口完成的。在使用系统时，需要注意写入和删除操作都需要配置一个bool类型的写入参数。该参数决定了操作数据的日志记录是否立即同步到磁盘，立即同步到磁盘可以保证数据的强一致，但是频繁的刷盘会严重影响系统写入性能，因此该参数默认为false。

## 3.2 总体框架

本键值存储系统基于日志结构合并树的存储模型开发，可以提供高效的读写操作，面对频繁的写入场景有着优秀表现。系统的总体架构如下：

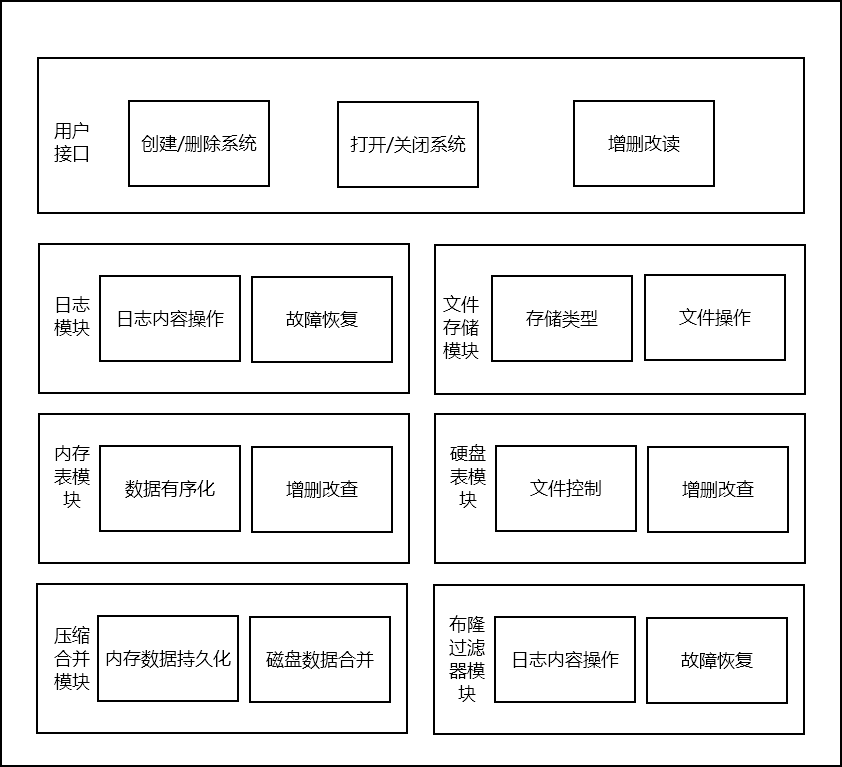


图3-1 系统总体架构

系统主要包括六个模块，分别是日志模块、文件存储模块、内存索引表模块、硬盘索引表模块、压缩合并模块和布隆过滤器模块。

日志模块主要功能是日志内容操作和读取日志进行故障恢复，需要调用文件存储模块进行存储日志记录，内存索引表中的数据记录未刷入磁盘前都是保存在日志文件中。

文件存储模块，提供数据在日志文件、硬盘数据文件、文件元信息文件的数据存储类型，提供在相应文件中数据的编解码功能与创建文件服务。

内存表模块，在内存维护可以高效读取写入的结构，用跳跃表实现。跳跃表中数据按键排序，且查询删除插入的平均时间复杂度能达到O(log N)。设计维护两个内存表，两个内存表本质是相同的，区别就是一个是只读内存表，另一个可读可写内存表。

硬盘表模块，根据数据冷热程度生成的有序数据文件是层次结构，每层包含多个有序数据文件。层数越小，数据越新。在每层数据量达到阈值时调用压缩合并模块，保障合并后数据的正确性。该模块的主要功能就是数据文件的记录操作，并维护数据文件的层次关系。

压缩合并模块，主要有两个功能，一是持久化内存表的数据到磁盘中的第零层数据文件；二是控制每层 磁盘表的数据文件大小，当某层数据文件大小超过阈值时，压缩合并模块会从该层中挑选文件合并到下一层中。

布隆过滤器模块的功能通过维护一个二进制数组，检查元素是否在集合中出现。

## 3.3 读写流程

### 3.3.1 写入流程

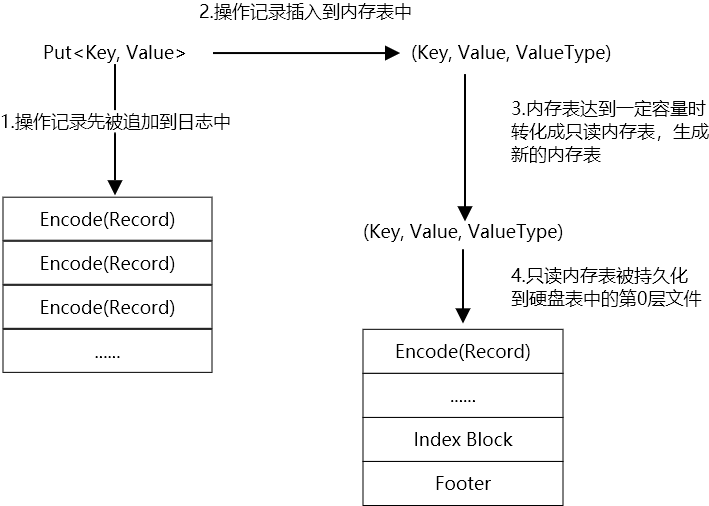


图3-2系统写入流程

当系统接收一条键值记录时，系统的会先将该条记录追加到日志记录中。在成功追加日志后，系统会将其插入到内存表中。内存空间是有限的，因此在内存表达到阈值时，系统证在操作的内存表会变成只读内存表，同时生成一个新的可读可写的内存表。压缩合并模块会适时的讲只读内存表刷入磁盘表的第0层中。

删除操作是种特殊的写入操作，删除操作相当于写入一个空值（在go语言中用nil表示）。修改操作可以直接写入一个更新的值，因为读数据会读最新插入的数据，所以写入过程不必考虑删除旧的值。

### 3.3.2 读取流程

当用户想要读取一条键值记录时，系统会先在可读可写的内存表中查找，然后再到只读内存表中查找。若内存中不存在目的数据，则需要到磁盘表中进行逐层检索。因为磁盘表的第0层文件是无序的，因此第0层的文件都需要进行查找。磁盘表模块维护了文件元信息（包括文件中键的范围），布隆过滤器模块也可以快速的判断目的数据是否在文件中，因此在磁盘数据文件中检索也得到了较大的优化。在内存表中查找的时间复杂度是O(log N)的，但是在磁盘中检索的时间复杂度是不确定的。

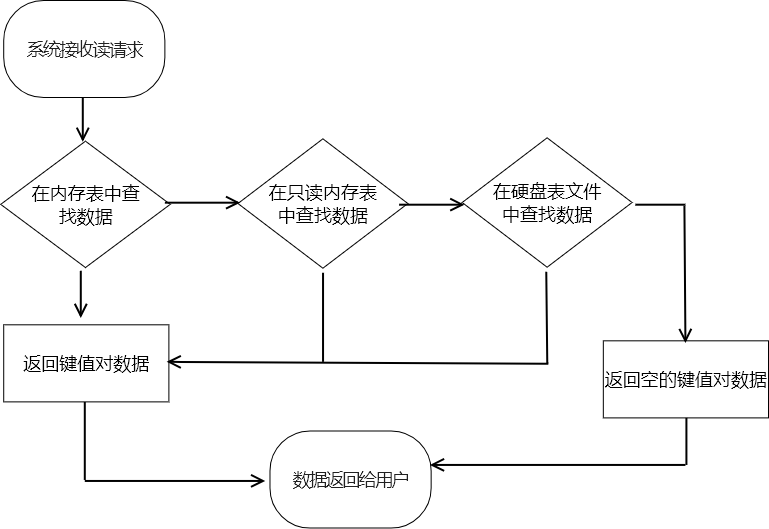


图3-3系统读取流程

# 第四章 系统方案实现

## 4.1 文件存储模块

文件存储模块提供了供其他模块使用的数据类型，提供支持对数据类型编解码，根据路径和文件名称创建文件。

本键值存储系统生成的文件共四类，分别是给系统加锁的DB\_LOCK文件、管理磁盘表文件元信息的manifest文件、数据日志文件wal与磁盘表数据文件sst。除了系统锁文件，其余三个文件名的构成均为类型\_数据库名\_时间戳，并且文件内容均为二进制。时间戳的精确程度决定能否保证文件id的唯一性，Golang语言time包中获取到的Linux系统的时间戳可以精确到纳秒，而在Windows系统下获取到的时间戳大概精确到毫秒。因此本系统适用于Linux开发环境。

### 4.1.1 键值对类型



图4-1 键值对格式

为了便于开发，系统只提供了一种类型的键值对，如图4-1所示。其中Key与Value的类型都是byte数组，Type是单字节整型数来表示数据记录的类型。数据记录的类型可以用0或者1表示，分别代表更新类型与删除类型。

写入文件时，该类型会被编码成二进制字节串（如图4-2）。字节串中包含该条数据总的长度（4字节整型数）、键的长度（4字节整型数）、键、数据操作类型（单字节整型数）、值的长度（4字节整型数）、值和这段数据的校验码。其中若该条数据操作类型是删除类型，则不会包含值的长度与值的内容。校验码的生成方法是循环冗余校验法，检查该条数据是否完整。

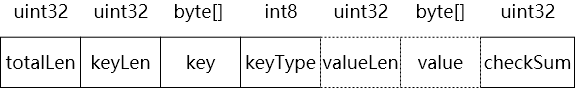


图4-2 键值对在文件中的格式

## 4.2 数据日志模块

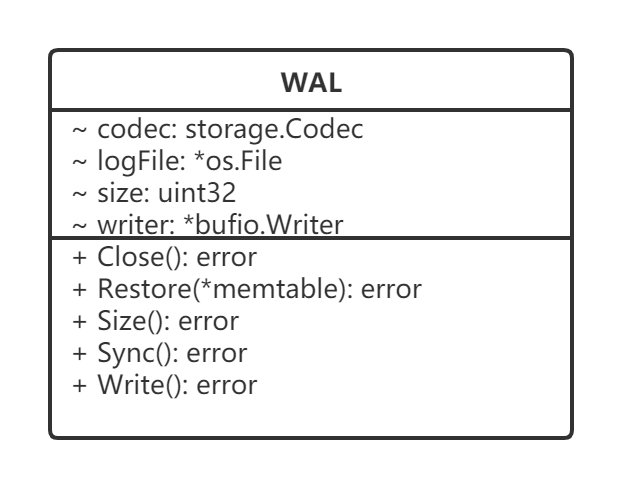


图4-3 WAL类

数据日志模块是系统中预写式日志的实现，主要由WAL类（如图4-3）实现其功能。WAL类由于需要进行文件编解码操作，因此调用了文件存储模块中的Codec类。系统写入日志时，可以强制立刻刷入磁盘，也可以正常使用带缓冲区的Writer写入文件。由于强制刷盘严重影响性能，因此默认使用后者写入文件。

该模块在本键值存储系统中起着极为重要的作用。存储数据操作记录，便于数据的恢复。系统提供高性能的写入，同时带来的是内存中未持久化的大量数据，数据日志模块的作用就是在内存中易失的数据保存下来，重新打开系统时恢复到系统关闭前的状态，恢复过程的伪代码如表4-1。

数据日志文件是由多条记录组成的。日志文件中保存的是按照大端序编码的二进制数据，二进制文件所占空间更小。系统正常情况下只会拥有一个与内存表相对应的日志文件，否则会出现一致性问题。因此系统在每次启动时都会寻找是否已存在日志文件，存在则会从日志中恢复出上次关闭时内存表的状态，否则新建日志文件。

表4-1 日志恢复伪代码

|  |
| --- |
| **Algorithm 1: Restore** |
| **Intput**: memtable mem  **Output**: error err   1. **for** 2. err :=log.Read(data) //data存储读出的字节 3. **if** err = io.EOF **then** **break** 4. **else** **if** err !=nil **then** **return** err 5. totalLen:= binary.BigEndian.Uint32(data) //解析出字节数组中的totalLen 6. err = log.Read(recBytes) //recBytes是大小为totalLen的切片 7. **if** err != nil **then return** err 8. record := Decode(recBytes) 9. **if** record.Type = RecordUpdate **then** 10. mem.Put(record.Key, record.Value) 11. **else** 12. mem.Delete(record.Key) |

## 4.3 内存表模块

记录被追加到日志文件后，会被传送到内存表模块进行处理。由于内存有限，内存表要有一个容量阈值。当正在操作的内存表大小超过阈值时，正在操作内存表会转化成只读内存表，同时生成一个新的可读可写的内存表供系统使用。系统中同时最多存在两个内存表，分别是可读可写即正在操作的内存表与只读内存表。在最坏情况下，可能出现只读内存表未刷盘，正在操作的内存表的大小也超过阈值，此时写操作会被阻塞，直到只读内存表刷盘后内存表也刷盘完毕，但该情况一般情况下应该不会发生。

内存表是由跳表实现的。跳表的排序规则是按照用户键将键值对从小到大进行排序，当用户键相同时，新插入的记录的值会覆盖旧的值。内存表按照这个规则维护大量的有序键值对，以O(log N)的复杂度实现高效的插入、查询、删除与修改。跳表的底层结构是链表，可以无锁实现，因此并发性能较好。

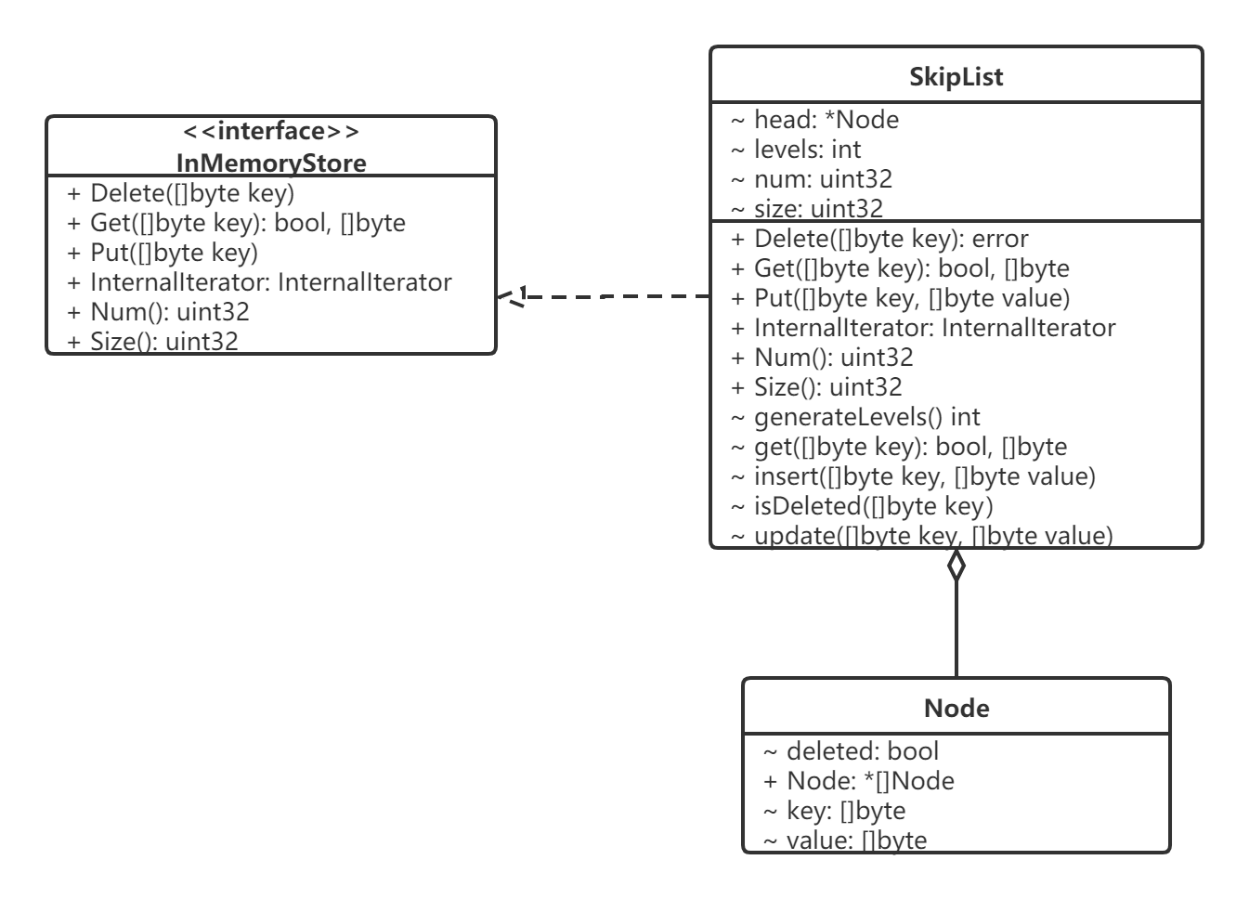


图4-4 skipList、Node类与InMemoryStore接口

在本系统中跳表实现了内存存储接口的方法（类图如图4-4所示）。内存表通过接口与迭代器接口来调用跳表的相关功能，屏蔽了底层实现细节。跳表的查找过程是增删改的基础，查找过程的伪代码如表4-2所示。

表4-2 跳表查找伪代码

|  |
| --- |
| **Algorithm 2: get** |
| **Intput**: key  **Output**: node   1. x := list.head 2. level := list.maxHeight - 1 3. **for** true { 4. next := x.getNext(level) 5. **if** key > nextkey **then** x = next 6. **else** 7. prev[level] = x 8. **if** level == 0 **then** **return** next 9. **else** 10. level-- 11. return nil |

## 4.4 硬盘表模块

硬盘表模块主要提供的功能是在数据文件的控制及对文件中数据的操作。在本系统中，数据文件采用SSTable，SSTable是存储、处理和交换数据的最流行的输出文件之一。SSTable可以高效存储大量有序的键值对，并且针对顺序读写操作做了优化。由于可能存在多个SSTable文件，所以需要对其进行管理。清单文件（Manifest files）保存了系统中所有的SSTable文件元信息，以便进行管理。

### 4.4.1 SSTable

SSTable的文件控制主要由Builder与Merger结构体（go语言中没有类的概念，但是可以通过结构体实现面向对象编程）实现，相关类图如图4-5所示。其中name是SSTable文件的名称，iter是安全访问内存表的迭代器，codec提供对数据编解码服务，writer是带缓冲区的文件写入，indexPerRecord是建立稀疏索引的区间长度，level是SSTable所在磁盘表中的层次。WriteTable是通过iter迭代器访问内存表，将内存表写入该builder所属的SSTable文件。

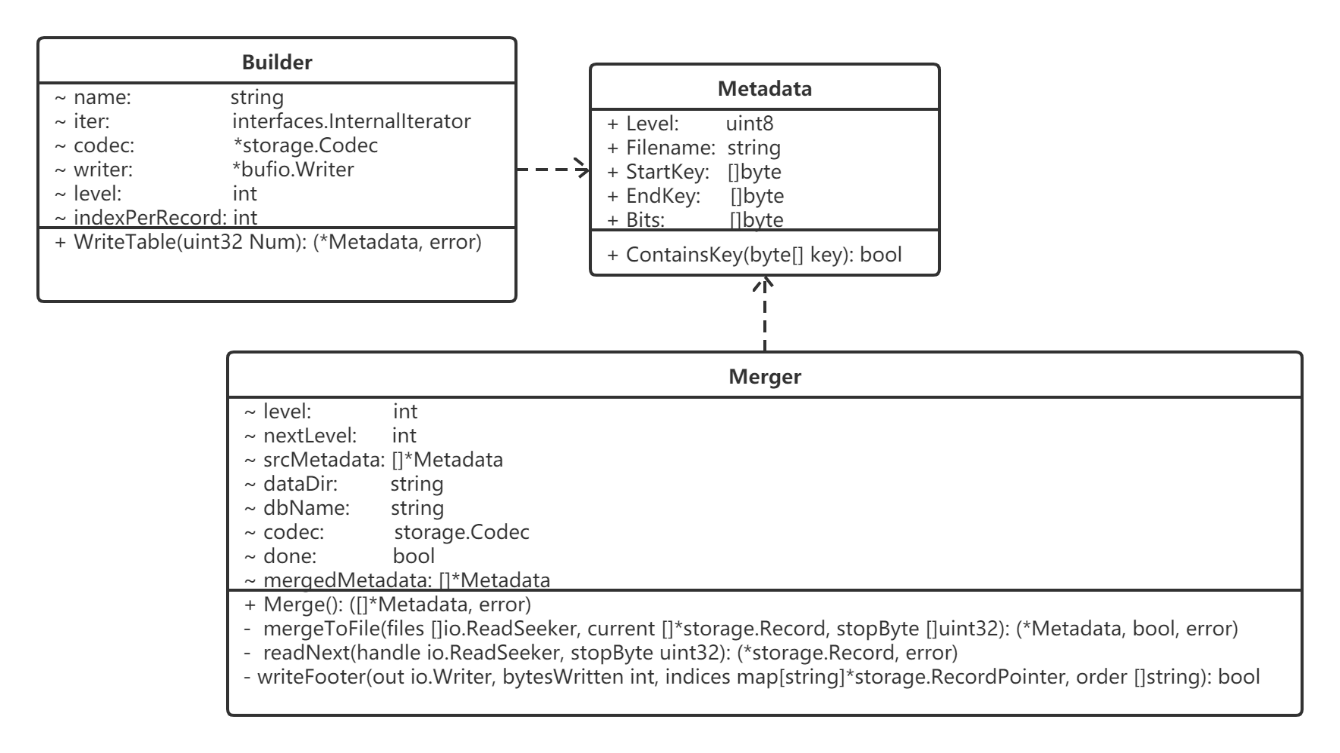


图4-5 SSTable相关类

Builder中的WriteTable()方法与Merger中的Merge()方法控制文件结构，使用固定格式将数据写入文件。SSTable文件具体格式如图4-6所示。文件首先由一系列编码后的record（如图4-2）组成，每隔indexPerRecord建立一个索引，图中的index block就是索引块，文件最后的footer保存index block的信息。索引块中的每个索引包括指向的record位置、record的长度与用户键。文件中的footer包括索引块中第一个索引的位置与长度，以及索引的数目。

Metadata就是SSTable文件元信息，包括文件所属层次、文件名称、最大键与最小键，以及在写文件时生成的布隆过滤器的数据。Merger为压缩合并模块提供了底层操作方法。

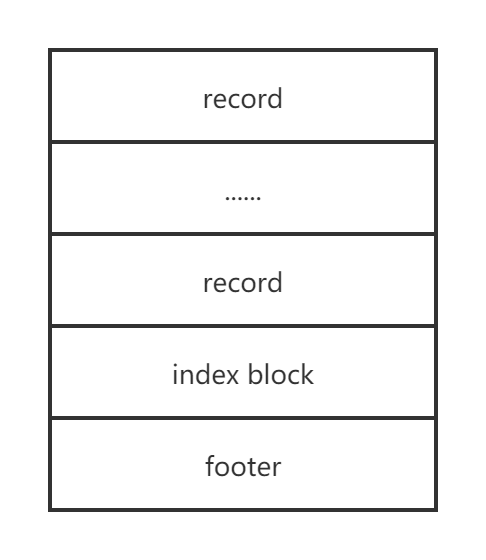


图4-6 SSTable文件结构

### 4.4.2 SSTable的管理

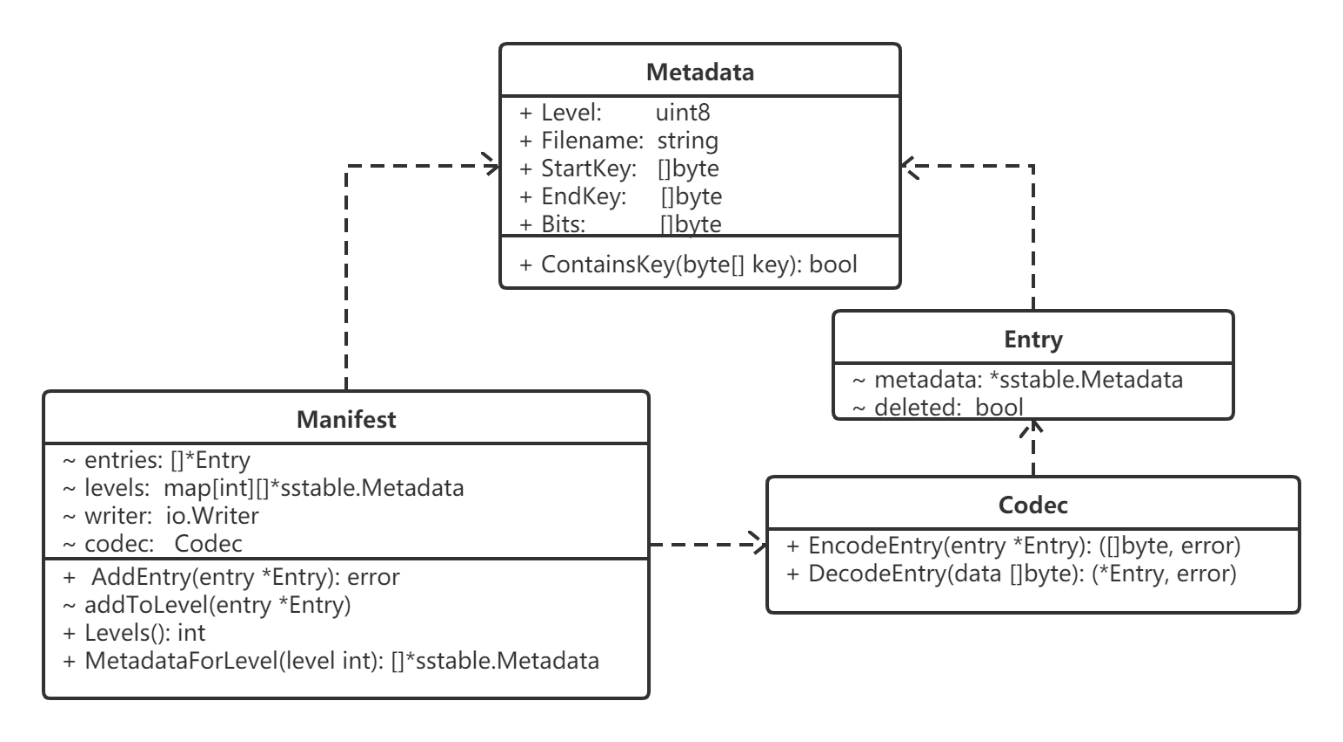


图4-7 Manifest相关类图

为了高效管理不同层次的SSTable文件，本系统使用清单文件保存所有文件元信息。清单文件由Manifest控制，Manifest的类图4-7所示。Manifest控制所有的SSTable文件元信息，包括获取某层所有的文件元信息、增加文件元信息、删除文件元信息、写入Manifest文件等。

## 4.5 压缩合并模块

因为系统的插入与删除操作都是在内存表中操作的，所以可能在文件中出现相同键值的键值对。SSTable文件在写入完成后除了合并操作之外是不会发生改变的。也就是说，系统中可能存在着已经失效的键值对。压缩合并模块的作用就是将已失效的键值对在合并文件的过程中移除，另外的作用就是将只读内存表写入文件并加入到硬盘表第0层。更少的文件意味着查询时可能的文件I/O次数更少。

系统的插入操作是在内存表中寻找合适位置插入记录，若存在则直接覆盖。删除操作是在内存表中寻找是否已经出现该键值，若出现过，则将该键值所在记录的删除标记置为true；否则插入一个删除标记为true的键值记录。因此可以看出在本系统中，删除是一个惰性操作，即延迟删除。所以删除操作在本系统中本质上就是一个插入操作。在只读内存表刷入硬盘后， SSTable文件中的删除记录成为无效记录。在内存表中插入与SSTable文件中某条记录相同键值的记录，SSTable文件中的记录也成为无效记录。

将只读内存表写入硬盘表中SSTable文件是通过按序访问只读内存表并按照一定格式写入第0层的新生成的SSTable文件中。系统利用多路归并通过多文件合并来清理无效键值对。假设待合并文件位于第x层，为了保证第x+1层文件的键的范围不重叠，系统将寻找第x+1层中与待合并文件键的范围重叠的文件来合并。合并流程图如图4-8所示。

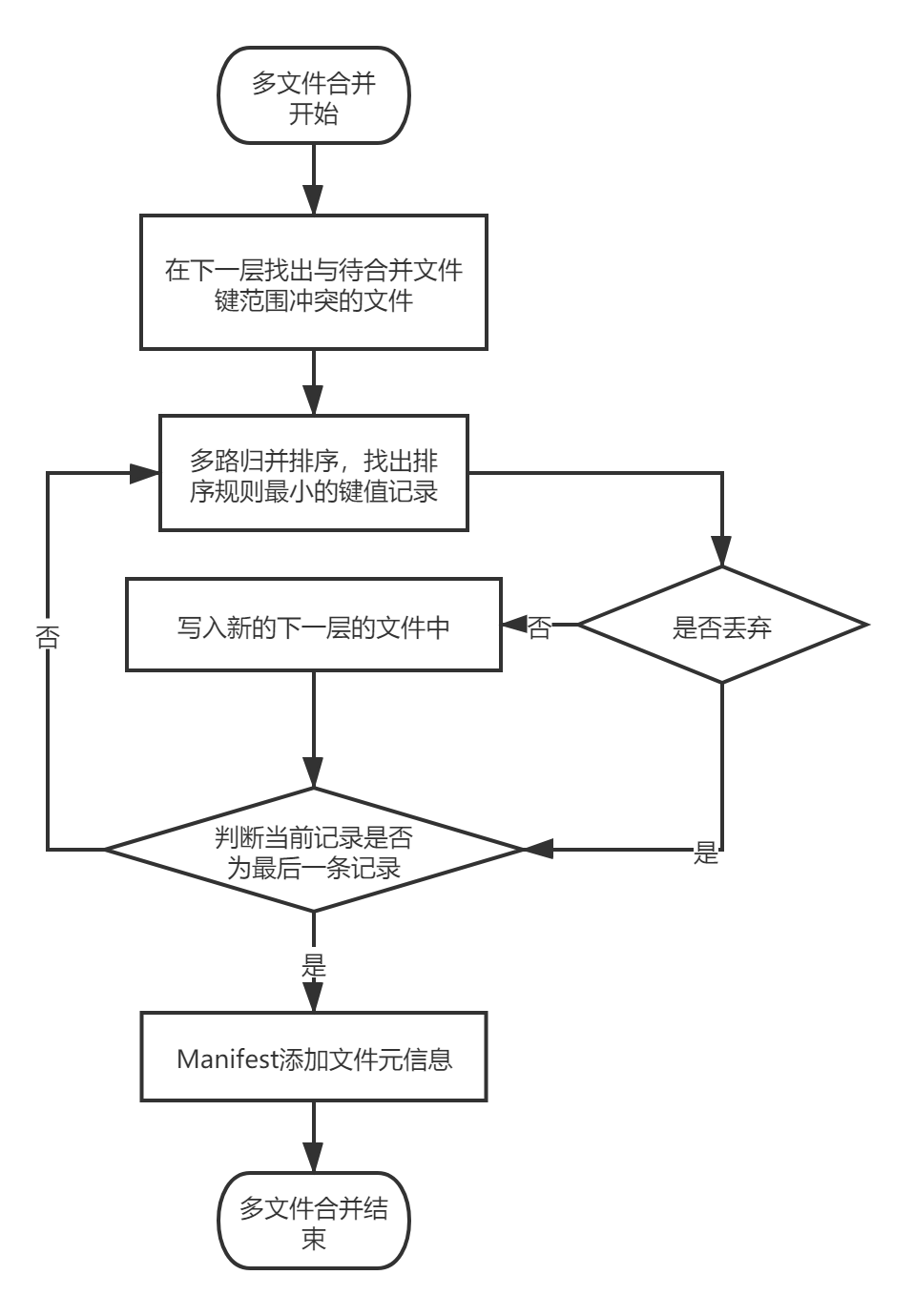


图4-8 多文件合并流程图

## 4.6 布隆过滤器模块

前面已经介绍过了布隆过滤器的理论，接下来介绍布隆过滤器在本系统中的实现。该模块主要由Bloom实现（类图如4-9所示），功能包括向集合中插入元素，检查元素是否存在于集合。另外还有新建Bloom以及通过字节数组恢复Bloom的功能。 假设我们要处理的数据总数是n，可以容忍的错误率是p，那么我们可以根据公式4-1确定出布隆过滤器的size。本系统中将布隆过滤器的错误率设为0.001。

前面提到过布隆过滤器需要k个相互独立的哈希函数。对于规律性较强的key，MurmurHash3算法的随机分布特征表现更良好。因此在这里我们采用k次MurmurHash3算法来代替k个相互独立的哈希函数生成哈希值。



图4-9 bloom类图

# 第五章 测试与分析

本章对系统的测试分为功能测试与性能测试.功能测试的目的是测试系统提供接口是否符合预期,在临界情况是否会发生异常.性能测试的目的是评测本键值存储系统在各种数据量下的性能指标。Golang语言自带了测试框架和工具，因此测试较为便捷。

## 5.1 测试环境

在开发过程中进行功能测试和性能测试的环境是Lenovo Xiaoxin310中的Ubuntu系统，该机器配置如表5-1所示。

表 5-1 环境配置

|  |  |
| --- | --- |
| 机器生产商 | Lenovo |
| 机器CPU | Inter(R) Core(TM) i7-6500U CPU @ 2.50GHz \* 4 |
| 机器内存 | 8.00GB |
| 机器磁盘空间 | 60.00GB |
| 机器操作系统 | Ubuntu16.04 LTS |

## 5.2 功能测试

在键值存储系统中，功能测试是对系统提供的用户接口进行测试，包括增删改查、重启系统状态的恢复等。

### 5.2.1 增删改查

为测试增删改查构造一组记录，调用系统Put接口，插入一组键值对<sdut,2019>，然后调用系统Get接口获取sdut对应的值，判断值是否为2019。再向系统中插入键值对<sdut，2020>，接着调用系统Get接口获取sdut对应的值，判断值是否是2020。调用Delete接口删除键sdut对应的键值对，接着调用系统Get接口获取sdut对应的值，判断值是否存在。

### 5.2.2 系统恢复

为测试系统恢复功能，向系统中分别插入键值对<sdut,2020>与<sdut,2021>，此时键值对存在于内存表中。关闭系统，内存表数据丢失。重启系统，查找键sdut对应的值是否为2021。

### 5.2.3 其他功能测试

Golang自带单元测试模块，单元测试是代码质量的保证。在开发过程中，编写了大量测试代码来测试各个模块的代码功能。图5-1所示为系统manifest模块的测试覆盖率。由于测试模块过多，不再详述。

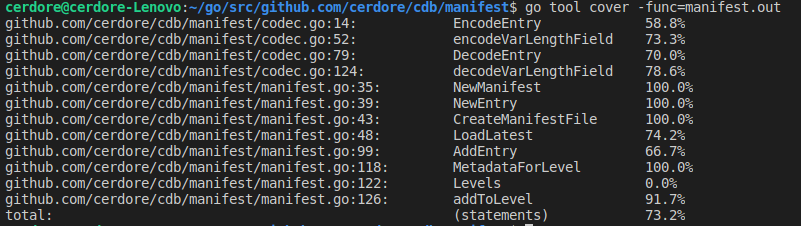


图5-1 manifest模块测试覆盖率

## 5.3 性能测试

### 5.3.1 go的基准测试

采用Golang的基准测试来测试系统的Put接口在1秒内的执行表现。插入记录的键的长度为12个字节，值的长度大于等于8个字节。如图5-2所示。BenchmarkPut-4指使用了4个CPU线程执行，1秒最多执行了270361次插入，平均每次操作花费9187纳秒。

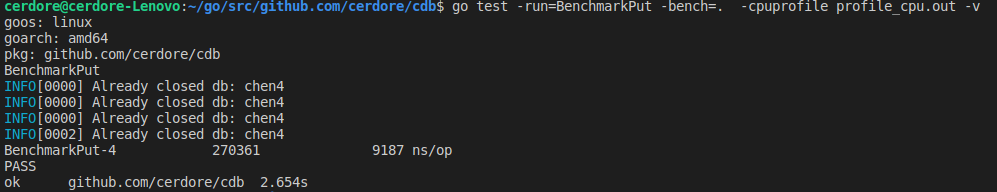


图5-2 基准测试-Put接口

接着通过Golang的pprof工具来分析性能，我们发现调用math/rand.seedrand消耗了大量时间。由于我们插入了大量数据，所以会多次生成内存表，每次新生成内存表时都需要调用随机函数来确定一个高度，消耗了较多时间。在图5-3中也可以明显看出随机数生成、文件的系统调用以及布隆过滤器的插入等操作消耗了大量时间。

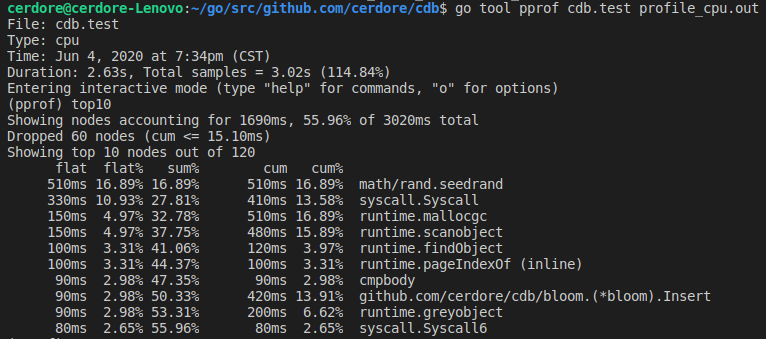


图5-3 运行时间分析

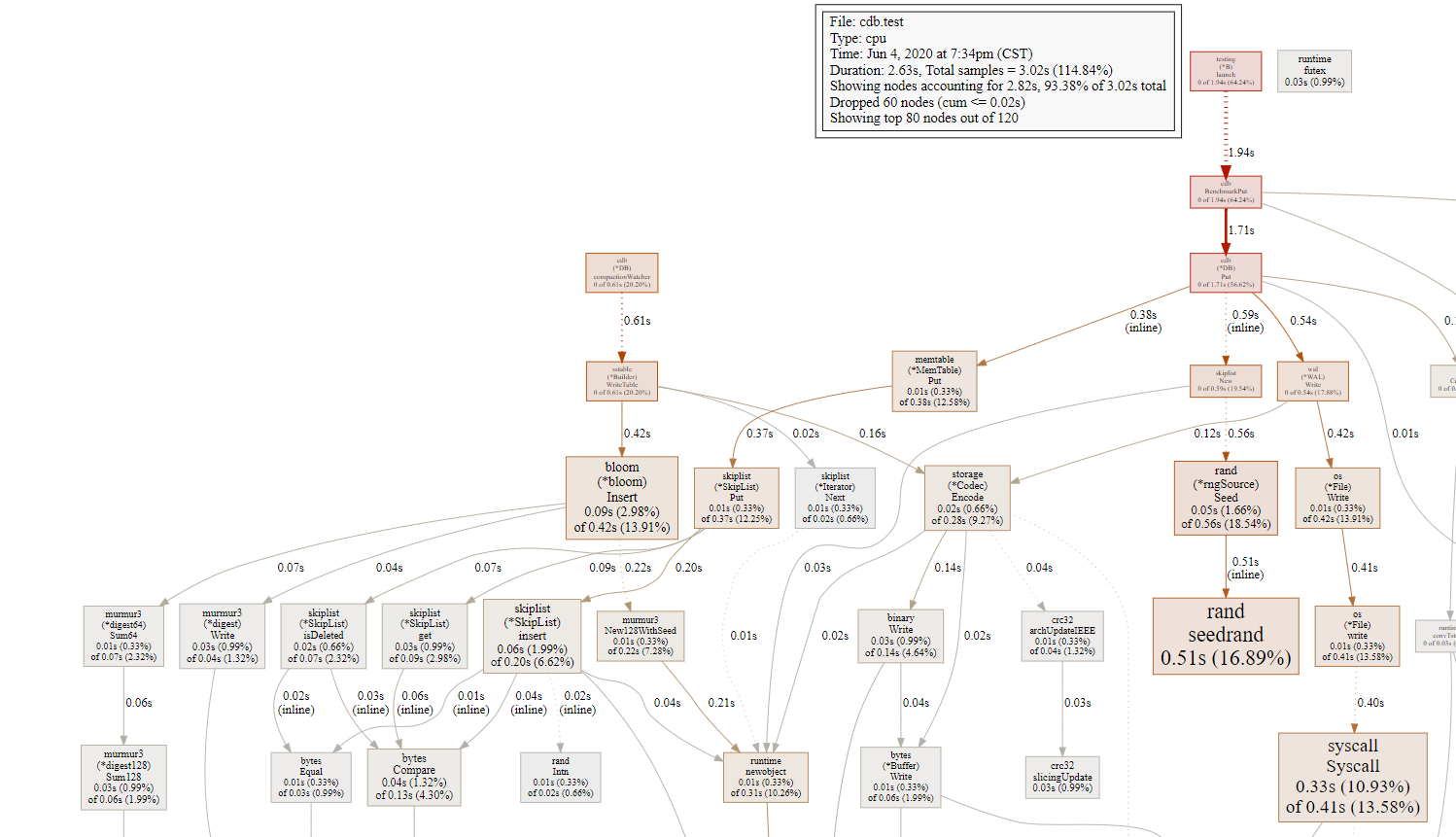


图5-3 性能分析局部火焰图

### 5.3.2 redis-benchmark基准测试

性能测试部分也采用了基于redis-benchmark的性能测试工具来测试代码。该工具可以基于不同的存储系统构建Redids服务。将本系统与github上一个采用B-tree储引擎的键值存储系统进行性能对比。测试采用256字节作为数据值的大小，9个字节作为数据键的大小。

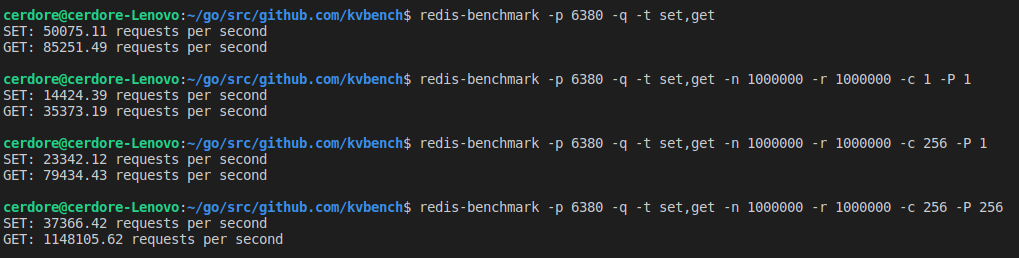


图5-4 本系统性能测试

利用四种测试方案将本键值存储系统与基于B-tree储引擎的键值存储系统进行性能测试对比分析。方案1是对系统插入读取接口的常规测试；方案2是插入1M个键值对并随机读取1M个键值对；方案3是在方案2的基础上使用256个并发连接发送操作请求；方案4是在方案3的基础上采用管道机制。测试效果分别如图5-4、图5-5所示。

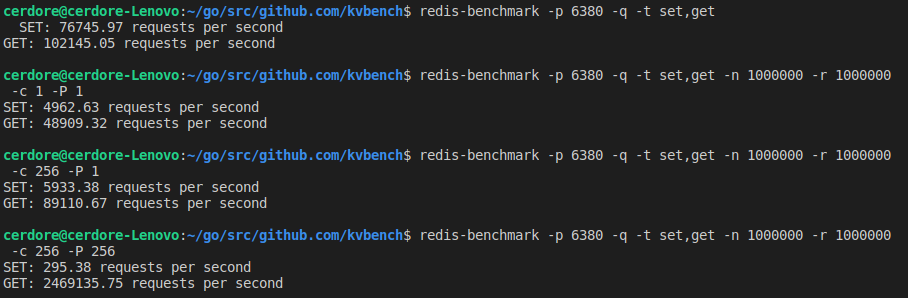


图5-5 B-tree引擎存储系统性能测试

两系统的写入性能对比如图5-6所示。可以看出本次测试使用的基于B-tree发的键值存储系统只有在较小数据量的常规测试下比较有优势，在大规模数据量和并发环境下性能表现不如本系统。

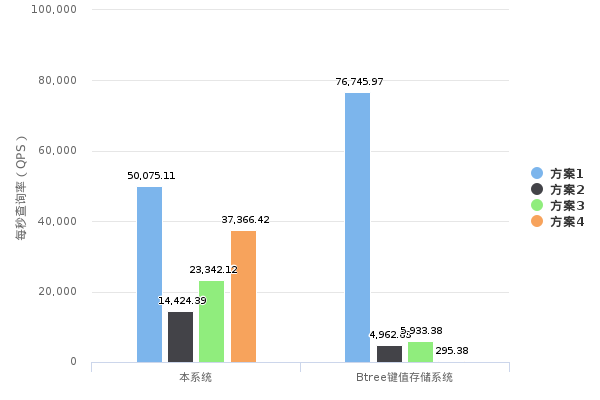


图5-6 系统写入性能对比

两系统的读取性能对比如图5-7所示。可以看出本次测试使用的基于B-tree发的键值存储系统拥有比本系统更高的读取性能。

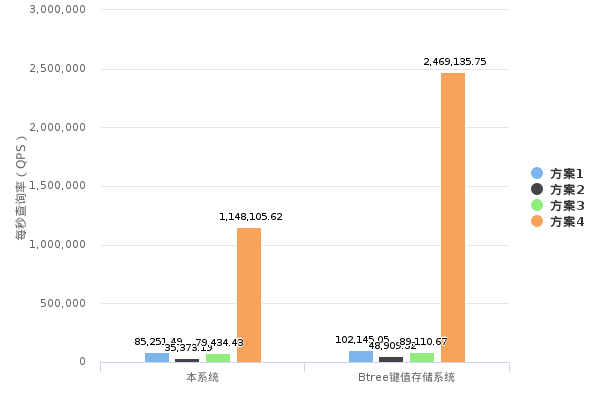


图5-7 系统读取性能对比

# 结 论

互联网的快速发展带来了海量数据，NoSQL数据库以易扩展、高效的读写性能的优势，得到了快速发展。键值存储便是NoSQL数据库中的热门方向之一。日志结构合并树是键值存储系统的典型结构。日志结构合并树通过批量写入规避了磁盘的随机写入问题，有着优秀的写入性能。

本文设计并实现了压缩合并、内存表、硬盘表、数据日志和文件存储等模块，并将其整合在一起实现了基于日志结构合并树的嵌入式（无网络服务封装）键值存储系统，并引入布隆过滤器提升系统读取性能。

最后，对本键值存储系统进行了细致的功能测试，统计了各模块的测试覆盖率。并与基于B-tree现的一个键值存储系统作读写性能测试对比，证明了基于日志结构合并树在大数据量以及并发环境下具有更优秀的写入性能。

由于本人能力与时间问题，本键值存储系统仍有许多可以完善的方向。如：采用更优秀的布隆过滤器方案，目前使用的布隆过滤器的信息保存在了文件元信息中；当前系统支持同时维护两个内存表，未来可以维护一个待写入到硬盘表第0层的内存表队列；尝试键值分离来减小系统压缩合并带来的写放大问题。

# 参考文献

1. CODD E F. A relational model of data for large shared data banks[J]. Commun. ACM, 1970, 13(6): 377-387.
2. O’Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351-385.
3. Sears R, Ramakrishnan R. bLSM: a general purpose log structured merge tree[C]//Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. 2012: 217-228.
4. Bloom B H. Space/time trade-offs in hash coding with allowable errors[J]. Communications of the ACM, 1970, 13(7): 422-426.
5. Wang J, Zhang Y, Gao Y, et al. PLSM: a highly efficient LSM-tree index supporting real-time big data analysis[C]//2013 IEEE 37th Annual Computer Software and Applications Conference. IEEE, 2013: 240-245.
6. Wu X, Xu Y, Shao Z, et al. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items[C]//2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15). 2015: 71-82.
7. Ongaro D, Ousterhout J. In search of an understandable consensus algorithm (extended version)[J]. 2013.
8. Pugh W. Skip lists: a probabilistic alternative to balanced trees[J]. Communications of the ACM, 1990, 33(6): 668-676.
9. 齐俊维. 基于LSM-Tree的持久化存储系统设计与实现[D].电子科技大学,2019.
10. 桂清鑫. 一种轻量级键值存储系统的研究与实现[D].华中科技大学,2018.
11. 马文龙,朱妤晴,蒋德钧,等.Key-Value型NoSQL本地存储系统研究[J].计算机学报,2018,41(08):1722-1751.
12. 罗军,陈席林,李文生.高效Key-Value持久化缓存系统的实现[J].计算机工程,2014,40(03):33-38.

# 致 谢

衷心感谢导师王海鹏副教授对我的精心指导，感谢王海鹏老师在我迷茫的时候对我的指引与对我毕业论文的指导。导师严谨的治学态度是我今后工作、学习的榜样。

感谢刘晓红、张先伟与张立红等ACM实验室的指导教师对我的关怀与帮助。在实验室中打磨出的品性是我在大学得到的宝贵财富。

感谢室友在宿舍中为我营造的学习氛围，让我克制在宿舍的时长。感谢在参加比赛时许多同学的帮助。感谢在学习过程中朋友的帮助。感谢某位异性同学的陪伴。感谢国家让我有了继续深造的机会。

# 附录A 程序主要部分源代码

package cdb

import (

…

)

type DB struct {

name string

DataDir string

mutex sync.RWMutex

memTable \*memtable.MemTable

walog \*wal.WAL

manifest \*manifest.Manifest

compactor \*compaction.Compactor

compactingMemTable \*memtable.MemTable

compactingWAL \*wal.WAL

compact chan bool

stopWatching chan bool

MtSizeLimit uint32

wg sync.WaitGroup

}

// TODO: allow configuration

const (

// Makes sense on Linux, may not elsewhere

DataDir = "/home/cerdore/cdb"

lockFile = "\_\_DB\_LOCK\_\_"

// Limit memtable to 4 MBs before flushing

//MtSizeLimit = uint32(4194304)

MtSizeLimit = uint32(2 \* 1024 \* 1024)//2MB

)

type DBOpts struct {

DataDir string

MtSizeLimit uint32

}

func (o \*DBOpts) applyDefaults() {

if o.DataDir == "" {

o.DataDir = DataDir

}

if o.MtSizeLimit == 0 {

o.MtSizeLimit = MtSizeLimit

}

}

// New creates a new database

func New(name string, opts DBOpts) (\*DB, error) {

opts.applyDefaults()

if err := os.MkdirAll(opts.DataDir, 0755); err != nil {

return nil, fmt.Errorf("could not create data dir %s: %w", opts.DataDir, err)

}

dbPath := path.Join(opts.DataDir, name)

if exists, err := exists(name, opts.DataDir); !exists {

if err := os.Mkdir(dbPath, 0755); err != nil {

return nil, fmt.Errorf("failed creating data directory for database %s: %w", name, err)

}

} else if err != nil {

return nil, fmt.Errorf("could not create new database: %w", err)

} else {

return nil, fmt.Errorf("database %s already exists. use DB#Open instead", name)

}

return Open(name, opts)

}

func lock(name string, DataDir string) error {

pid := os.Getpid()

lockPath := path.Join(DataDir, name, lockFile)

lock, err := os.Open(lockPath)

// Database is not currently locked, attempt to acquire

if os.IsNotExist(err) {

if lockFile, err := os.OpenFile(lockPath, os.O\_RDWR|os.O\_CREATE|os.O\_EXCL, 0666); os.IsExist(err) {

return fmt.Errorf("cannot lock database. already locked by another process")

} else if err != nil {

return fmt.Errorf("failure attempting to lock database: %w", err)

} else {

pidBytes := []byte(strconv.Itoa(pid))

if n, err := lockFile.Write(pidBytes); n < len(pidBytes) {

return fmt.Errorf("failure writing owner pid to lock file. wrote %d bytes, expected %d",

n, len(pidBytes))

} else if err != nil {

return fmt.Errorf("failure writing owner pid to lock file: %w", err)

}

return nil

}

} else if err != nil {

return fmt.Errorf("failure attempting to lock database: %w", err)

} else {

// Database currently locked

scanner := bufio.NewScanner(lock)

scanner.Scan()

lockPid, err := strconv.Atoi(scanner.Text())

if err != nil {

return fmt.Errorf("failed attempting to read lockfile: %w", err)

}

if lockPid == pid {

return nil

} else {

return fmt.Errorf("cannot lock database. already locked by another process (%d)", lockPid)

}

}

}

// Open opens a database

func Open(name string, opts DBOpts) (\*DB, error) {

opts.applyDefaults()

if exists, err := exists(name, opts.DataDir); !exists {

if err == nil {

return nil, fmt.Errorf("failed opening database %s. does not exist", name)

} else {

return nil, fmt.Errorf("failed opening database %s: %v", name, err)

}

}

if err := lock(name, opts.DataDir); err != nil {

return nil, fmt.Errorf("could not lock database: %w", err)

}

mem := memtable.New()

// load WAL

found, walog, err := wal.FindExisting(name, opts.DataDir)

if err != nil {

return nil, fmt.Errorf("failed attempting to look for existing WAL file: %w", err)

}

if !found {

waf, err := wal.CreateFile(name, opts.DataDir)

if err != nil {

return nil, fmt.Errorf("could not create WAL file: %w", err)

}

walog = wal.New(waf)

} else {

if err = walog.Restore(mem); err != nil {

return nil, fmt.Errorf("failed attempting to restore WAL: %w", err)

}

}

found, man, err := manifest.LoadLatest(name, opts.DataDir)

if err != nil {

return nil, fmt.Errorf("failed attempting to load manifest file: %w", err)

} else if !found {

maf, err := manifest.CreateManifestFile(name, opts.DataDir)

if err != nil {

return nil, fmt.Errorf("could not create manifest file: %w", err)

}

man = manifest.NewManifest(maf)

}

db := &DB{

memTable: mem,

walog: walog,

manifest: man,

compactor: compaction.New(man, opts.DataDir, name),

name: name,

DataDir: opts.DataDir,

compact: make(chan bool, 1),

stopWatching: make(chan bool),

MtSizeLimit: opts.MtSizeLimit,

}

db.wg.Add(1)

go db.compactionWatcher()

return db, nil

}

func exists(name string, DataDir string) (bool, error) {

dbPath := path.Join(DataDir, name)

if \_, err := os.Stat(dbPath); os.IsNotExist(err) {

return false, nil

} else if err != nil {

return false, fmt.Errorf("failure checking to see if database already exists: %w", err)

} else {

return true, nil

}

}

// OpenOrNew opens the DB if it exists or creates it if it doesn't

func OpenOrNew(name string, opts DBOpts) (\*DB, error) {

opts.applyDefaults()

dbExists, err := exists(name, opts.DataDir)

if err != nil {

return nil, fmt.Errorf("failed checking if database %s already exists: %v", name, err)

}

if dbExists {

return Open(name, opts)

} else {

return New(name, opts)

}

}

// Close ensures that any resources used by the DB are tidied up

func (d \*DB) Close() error {

// TODO: flush to memtable here

close(d.stopWatching)

d.wg.Wait()

log.Info("Already closed db: ", d.name)

return d.unlock()

}

func (d \*DB) unlock() error {

lockPath := path.Join(d.DataDir, d.name, lockFile)

return os.Remove(lockPath)

}

// Get returns the value of record

func (d \*DB) Get(key []byte) ([]byte, error) {

d.mutex.RLock()

defer d.mutex.RUnlock()

val := d.memTable.Get(key)

if val == nil && d.compactingMemTable != nil {

val = d.compactingMemTable.Get(key)

}

if val == nil {

// 255 == uint8 max == max number of levels based on value used for encoding level information on disk

levelTraversal:

for i := 0; i < 255; i++ {

for \_, meta := range d.manifest.MetadataForLevel(i) {

if meta == nil {

break levelTraversal

}

if meta.ContainsKey(key) {

bloom := bloom.RecoverBloom(meta.Bits)

if !bloom.Check(key) {

return nil, nil

}

val, err := d.searchSSTable(key, meta)

if err != nil {

return nil, fmt.Errorf("failed attempting to scan sstable for key %s: %w", string(key), err)

}

if val != nil {

return val, nil

}

}

}

}

}

return val, nil

}

func (d \*DB) searchSSTable(key []byte, meta \*sstable.Metadata) ([]byte, error) {

// TODO: cache

sstHandle, err := os.Open(path.Join(d.DataDir, d.name, meta.Filename))

if err != nil {

return nil, fmt.Errorf("failed attempting to open sstable for reading: %w", err)

}

defer sstHandle.Close()

return sstable.Search(key, sstHandle)

}

// Put inserts or updates the value if the key already exists

func (d \*DB) Put(key []byte, value []byte, syncW bool) error {

d.mutex.Lock()

defer d.mutex.Unlock()

if err := d.walog.Write(storage.NewRecord(key, value, false), syncW); err != nil {

return fmt.Errorf("failed attempting write put to WAL: %w", err)

}

d.memTable.Put(key, value)

if d.memTable.Size() > d.MtSizeLimit && d.compactingMemTable == nil {

//fmt.Println("begin to Compact ")

d.compactingMemTable = d.memTable

d.compactingWAL = d.walog

d.memTable = memtable.New()

waf, err := wal.CreateFile(d.name, d.DataDir)

if err != nil {

// Abort compaction attempt

d.memTable = d.compactingMemTable

d.walog = d.compactingWAL

// if err := d.compactingWAL.Close(); err != nil {

// return fmt.Errorf("compactingWAL closed error : %w", err)

// }

d.compactingMemTable = nil

d.compactingWAL = nil

return fmt.Errorf("could not create WAL file: %w", err)

}

d.walog = wal.New(waf)

d.compact <- true

}

return nil

}

// Deletes the specified key from the data store

func (d \*DB) Delete(key []byte, syncW bool) error {

d.mutex.Lock()

defer d.mutex.Unlock()

if err := d.walog.Write(storage.NewRecord(key, nil, true), syncW); err != nil {

return fmt.Errorf("failed attempting write delete to WAL: %w", err)

}

d.memTable.Delete(key)

return nil

}

func (d \*DB) compactionWatcher() {

defer d.wg.Done()

for {

select {

case <-d.compact:

//fmt.Println("watcher found chanel changed!!")

d.wg.Add(1)

if err := d.doCompaction(); err != nil {

log.Errorf("error performing compaction: %v", err)

}

case <-d.stopWatching:

return

}

}

}

func (d \*DB) flushMemTable(tableName string, writer \*os.File, memNum uint32) error {

iter := d.compactingMemTable.InternalIterator()

//fmt.Println(d.compactingMemTable.Num())

builder := sstable.NewBuilder(tableName, iter, 0, writer)

metadata, err := builder.WriteTable(memNum)

if err != nil {

return fmt.Errorf("could not write memtable to level 0 sstable: %w", err)

}

return d.manifest.AddEntry(manifest.NewEntry(metadata, false))

}

func (d \*DB) doCompaction() error {

defer d.wg.Done()

if d.compactingMemTable != nil {

file, err := sstable.CreateFile(d.name, d.DataDir)

if err != nil {

return fmt.Errorf("failed attempt to create new sstable file: %w", err)

}

defer file.Close()

err = d.flushMemTable(filepath.Base(file.Name()), file, d.compactingMemTable.Num())

if err == nil {

if err = file.Sync(); err != nil {

return fmt.Errorf("error flushing sstable to disk: %w", err)

}

d.mutex.Lock()

defer d.mutex.Unlock()

if err = d.compactingWAL.Close(); err != nil {

return fmt.Errorf("failed attempt to close WAL: %w", err)

}

d.compactingMemTable = nil

d.compactingWAL = nil

} else {

fmt.Println("Closed error %w: ", err)

}

}

if err := d.compactor.Compact(); err != nil {

return fmt.Errorf("failed attempting to compact: %w", err)

}

return nil

}

# 附录B 程序使用说明书

import “github.com/cerdore/cdb” //导入包即可使用本键值存储系统

func main(){

//采用默认设置，新建一个名称为”chen”的键值存储系统

db, err := cdb.New("chen", cdb.DBOpts{DataDir: “”, MtSizeLimit: 0})

if err != nil{

fmt.Println(err)

}

//以不立即刷盘的方式，插入一个键为“sdut”，值为“2019”的记录

db.Put([]byte(“sdut”), []byte(“2019”), false)

//以立即刷盘的方式，插入一个键为“sdut”，值为“2020”的记录

db.Put([]byte(“sdut”), []byte(“2020”), true)

//读取键为”sdut”对应的值

ans, err := db.Get([]byte(“sdut”))

if err != nil{

fmt.Println(err)

}

//以不立即刷盘的方式，删除键”sdut”对应的记录

err = db.Delete([]byte(“sdut”),false)

if err != nil{

fmt.Println(err)

}

//以立即刷盘的方式，删除键”sdut”对应的记录

err = db.Delete([]byte(“sdut”),true)

if err != nil{

fmt.Println(err)

}

db.Close() // 关闭系统

d1, err := db.Open(“chen”, cdb.DBOpts{DataDir: “”, MtSizeLimit: 0})

if err != nil{

fmt.Println(err)

}

d1.Close() // 关闭系统

}