东北大学秦皇岛分校
Northeastern University at Qinhuangdao

毕业论文

基于 LSM-Tree 结构和 Raft 算法的分布式存储系统

院　　别	计算机与通信工程学院
专业名称	计算机科学与技术
班级学号	1901-20197897
学生姓名	华令楠
指导教师	吕艳霞

2023 年 5 月 20 日

郑 重 声 明

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名：　　日期：

基于 LSM-Tree 结构和 Raft 算法的分布式存储系统

摘　要

随着互联网的发展，网络用户的激增导致互联网服务提供公司要存储极大规模的数据，而对于一些复杂场景下的数据复制以及分布式系统下数据库的可靠性仍然是一个巨大的挑战。传统的互联网架构时代，单机数据库如 MySQL，Oracle 占领很大的市场份额，而单机数据库有很多缺点，成本高：它们通常比分布式数据库系统更加复杂和成本更高，因为它们需要专门的硬件、存储设备和管理软件。维护困难：由于它们是单独的系统，因此一旦出现问题，修复它们可能需要很长时间和高昂的成本。不易扩展：当需要增加新功能或存储容量时，单机数据库系统可能无法轻松地扩展。数据共享和备份困难：由于它们是单独的系统，数据不能轻松地在它们之间共享或备份。安全性差：由于它们是单独的系统，数据很容易受到黑客攻击和数据泄露。而分布式的数据存储恰恰解决了单机数据库的性能瓶颈问题和单机数据库在实现面向用户系统时的各种痛点。

由于一个可靠的分布式存储系统设计复杂，挑战较大，本文致力于系统的存储策略，数据压缩方法，日志复制同步过程，Raft 算法的可达性分析。我们利用 Golang 编程语言的天然并发的特性，来开发 Raft 共识算法和 LSM-tree 数据结构以实现分布式数据存储系统。我们的系统通过跨节点集群复制数据并使用 Raft 共识算法确保副本之间的一致性来提供容错性、可扩展性和高可用性。LSM-tree 数据结构通过优化磁盘访问和减少随机查找次数来实现高效的读写。通过不断精读原文和阅读参考并 demo 出 Google 开源 C++ 语言的 leveldb 的 LSM-Tree 实现以便尽最大限度复现 LSM-Tree 和 Raft 论文中提及的所有关键点，我们的评估表明，我们的系统在保持强一致性保证的同时实现了高性能和可扩展性，同时支持跨多平台的服务端部署，跨多个平台的客户端调用和多种语言的 API。

关键词： 日志结构归并树，Raft 共识性算法，键值型数据存储，分布式系统

Distributed storage system based on LSM-Tree structure and Raft algorithm

Abstract

With the development of the Internet, the surge of network users has led Internet service providers to store extremely large-scale data, but data replication in some complex scenarios and the reliability of databases in distributed systems are still a huge challenge. In the era of traditional Internet architecture, stand-alone databases such as MySQL and Oracle occupy a large market share, but stand-alone databases have many shortcomings. High cost: They are usually more complex and costly than distributed database systems because they require specialized hardware, storage devices, and management software. Difficult to maintain: Since they are separate systems, it can take a long time and be costly to fix if something goes wrong. Not easy to expand: When new functions or storage capacity need to be added, a stand-alone database system may not be easily expanded. Difficulty in data sharing and backup: Since they are separate systems, data cannot be easily shared or backed up between them. Poor security: Since they are separate systems, the data is vulnerable to hacking and data breaches. The distributed data storage just solves the performance bottleneck problem of the stand-alone database and the various pain points of the stand-alone database when implementing the user-oriented system.

Since the design of a reliable distributed storage system is complex and challenging, this paper focuses on the system's storage strategy, data compression method, log replication synchronization process, and the reachability analysis of the Raft algorithm. We use the natural concurrency of the Golang programming language to develop the Raft consensus algorithm and the LSM-tree data structure to implement a distributed data storage system. Our system provides fault tolerance, scalability, and high availability by replicating data across a cluster of nodes and using the Raft consensus algorithm to ensure consistency between replicas. The LSM-tree data structure enables efficient reads and writes by optimizing disk access and reducing the number of random lookups. By continuously intensively reading the original text and reading references, and demoing the LSM-Tree implementation of leveldb in Google's open source C++ language, in order to reproduce all the key points mentioned in the LSM-Tree and Raft pa-

pers as much as possible, Our evaluation shows that our system achieves high performance and scalability while maintaining strong consistency guarantees, while supporting server-side deployment across multiple platforms, client calls across multiple platforms, and APIs in multiple languages.

Keywords: Log-Structured Merge-Tree , Raft consensus algorithm , Key-Value Data Storage , Distributed System

目录

1 绪论 **1**

1.1 课题的背景和意义 1

1.2 分布式存储系统的发展状况 1

1.3 课题研究的主要方法及内容 2

1.4 论文组织结构 2

2 相关背景知识介绍 **3**

2.1 开发工具和环境 3

2.2 LSM-Tree 存储结构 4

2.3 Raft 共识性算法 5

2.4 涉及的开源库 5

2.5 本章小结 6

3 Radds 存储系统需求分析 **7**

3.1 存储系统需求概述 7

3.2 存储系统功能需求分析 7

3.3 存储系统性能需求分析 8

3.4 存储系统可行性分析 9

3.5 本章小结 9

4 Radds 存储系统总体设计 **10**

4.1 存储系统架构设计 11

4.2 存储层功能总体设计 11

4.2.1 内存可变数据结构 memtable 总体设计 12

4.2.2 内存不可变数据结构 immutable memtable 总体设计 12

4.2.3 日志文件数据结构 journal 总体设计 12

4.2.4 磁盘持久化数据结构 sstable 总体设计 13

4.2.5 文件元数据 manifest 总体设计 13

4.2.6 版本号 current 总体设计 14
4.3 共识层功能总体设计 15
4.4 客户端功能总体设计 15
4.4.1 客户端服务平台总体设计 15
4.4.2 gRPC API 客户端总体设计 15
4.4.3 RESTful API 客户端总体设计 15
4.4.4 CLI 客户端总体设计 15
4.5 本章小结 15

5 Radds 存储系统详细设计与实现 16
5.1 基础层详细设计与实现 16
5.1.1 错误处理的实现 16
5.1.2 日志系统的实现 16
5.1.3 工具库的实现 20
5.2 存储层详细设计与实现 20
5.2.1 写数据的实现 20
5.2.2 读数据的实现 25
5.2.3 内存数据库的实现 27
5.3 共识层详细设计与实现 31
5.4 客户端层详细设计与实现 31
5.4.1 API 客户端服务平台的实现 31
5.4.2 gRPC API 客户端的实现 31
5.4.3 RESTful API 客户端的实现 31
5.4.4 CLI 命令行客户端的实现 31
5.5 本章小结 31

6 Radds 存储系统部署、日志分析与客户端测试 32
6.1 分布式存储系统部署 32
6.1.1 在 X86-64 GNU/Linux Ubuntu22.04 操作系统部署 32
6.1.2 在 X86-64 Windows11 操作系统部署 32

6.1.3 在 Arm64 Darwin MacOS 操作系统部署 32
6.2 数据存储系统日志分析 32
6.3 共识性系统日志分析 32
6.4 客户端服务平台日志分析 32
6.5 客户端测试 32
6.5.1 gRPC API 客户端测试 32
6.5.2 RESTful API 客户端测试 32
6.5.3 CLI 客户端测试 32
6.6 本章小结 32

结论 **33**

致 谢 **34**

附 录 **35**
附录 A 35
附录 B 35

参考文献 **35**

1 绪论

1.1 课题的背景和意义

分布式存储系统是一个重要且不断发展的领域，它涉及到许多学科和技术，如计算机科学、网络安全、数据库管理等。分布式存储系统的研究主要集中在三个方面：分布式存储系统的架构设计、底层协议的研究以及应用场景的拓展。其中，架构设计包括存储系统的硬件架构、软件框架等；底层协议的研究则包括各种存储接口、协议栈等；应用场景的拓展则涵盖了企业级、消费级、个人级等不同类型的存储系统。分布式存储系统的研究具有广泛的应用前景，它可以应用于企业级存储系统、云存储服务和分布式应用等方面。在企业级存储系统中，分布式存储系统可以用于存储大量的数据，提高数据存储的效率和可靠性。在云存储服务中，分布式存储系统可以用于存储云端的数据，提供高效的数据存储和管理服务。在分布式应用中，分布式存储系统可以用于存储和管理大量的数据，提高应用的可靠性和性能。在分布式存储系统的研究中，需要解决的关键问题包括数据一致性、数据持久性、数据备份和恢复等。在解决这些问题的过程中，需要采用一些新的技术和方法，如分布式数据库、分布式文件系统、分布式对象存储等。同时，需要考虑分布式存储系统的安全性和可靠性问题，采用一些新的安全技术和机制，如安全协议、数据加密、访问控制等，以保证分布式存储系统的安全性和可靠性。

1.2 分布式存储系统的发展状况

分布式存储系统的发展历史可以追溯到上世纪 90 年代，当时出现了一些基于局部存储器的分布式存储系统，如 Lustre 和 Xanadu 等。这些系统主要用于文件服务器等领域。随着网络技术的发展，基于网络的分布式存储系统出现了，如 Hadoop 和 HDFS 等。这些系统将数据存储在分布式节点上，并通过网络进行数据的访问和管理。近年来，随着云计算的发展，分布式存储系统开始广泛应用于云存储服务中。云存储服务将数据存储在云端，并通过互联网提供数据的访问和管理。这些系统通常采用分布式文件系统，如 AFS、HDFS 和 Gluster 等。随着大数据时代的到来，分布式存储系统的研究和应用也越来越广泛。基于分布式文件系统的分布式存储系统可以存储海量的数据，并支持高效的数据存储和管理。同时，基于块存储的分布式存储系统也得到了广泛的研究和应用，它可以实现数据的高效复制和同步，并支持大规模的数据存储和管理。总之，分布式存储系统的发展历史可以分为三个阶段：基于局部存储器的分布式存储系统、基于网络的

分布式存储系统和基于块存储的分布式存储系统。随着大数据时代的到来，分布式存储系统的研究和应用也将越来越广泛和深入。

1.3 课题研究的主要方法及内容

本课题主要工作是...

本课题主要包含以下几个方面内容：

1、xxx

2、xxxx

3、xxxxx

1.4 论文组织结构

本文主要围绕相关技术选型，需求分析，系统整体设计、详细设计，部署与测试等方面来进行论述，共分为 6 章，各章内容如下：

第 1 章...

第 2 章...

第 3 章...

第 4 章...

第 5 章...

第 6 章...

为了更好的理解...

2 相关背景知识介绍

2.1 开发工具和环境

1、开发工具：VS Code

Visual Studio Code（简称 VS Code）是一款由微软开发且跨平台的免费集成开发环境。该软件支持语法高亮、代码自动补全（又称 IntelliSense）、代码重构功能，并且内置了命令行工具和 Git 版本控制系统。用户可以更改主题和键盘快捷方式实现个性化设置，也可以通过内置的扩展程序商店安装扩展以拓展软件功能。VS Code 使用 Monaco Editor 作为其底层的代码编辑器。Visual Studio Code 的源代码以 MIT 许可证在 GitHub 上释出，而可执行文件使用了专门的许可证。

2、开发环境

（1） X86-64 GNU/Linux-Ubuntu22.04

Linux 是一种自由和开放源码的类 UNIX 操作系统。该操作系统的内核由林纳斯·托瓦兹在 1991 年 10 月 5 日首次发布，再加上用户空间的应用程序之后，就成为了 Linux 操作系统。Linux 严格来说是单指操作系统的内核，因操作系统中包含了许多用户图形接口和其他实用工具。如今 Linux 常用来指基于 Linux 的完整操作系统，内核则改以 Linux 内核称之。由于这些支持用户空间的系统工具和库主要由理查德·斯托曼于 1983 年发起的 GNU 计划提供，自由软件基金会提议将其组合系统命名为 GNU/Linux。

Ubuntu 是基于 Debian，以桌面应用为主的 Linux 发行版。Ubuntu 有三个正式版本，包括桌面版、服务器版及用于物联网设备和机器人的 Core 版。前述三个版本既能安装于实体电脑，也能安装于虚拟环境。

（2） Golang1.20

Go（又称 Golang[4]）是 Google 开发的一种静态强类型、编译型、并发型，并具有垃圾回收功能的编程语言。罗伯特·格瑞史莫、罗勃·派克及肯·汤普逊于 2007 年 9 月开始设计 Go，稍后伊恩·兰斯·泰勒（Ian Lance Taylor）、拉斯·考克斯（Russ Cox）加入项目。Go 是基于 Inferno 操作系统所开发的。Go 于 2009 年 11 月正式宣布推出，成为开放源代码项目，支持 Linux、macOS、Windows 等操作系统。

3、测试环境

（1） X86-64 Windows11

Windows 11 是微软于 2021 年推出的 Windows NT 系列操作系统，为 Windows 10 的后继者。出于安全考虑，Windows 11 的系统需求比 Windows 10 有所提高。微软仅支持使用英特尔酷睿第 8 代或更新的处理器、AMD Zen+ 或更新的处理器及高通骁龙 850 或更新的处理器的设备。Windows 11 不再支持 32 位 x86 架构或使用 BIOS 固件的设备。

（2） X86-64 Windows11 WSL2-GNU/Linux-Ubuntu20.04

适用于 Linux 的 Windows 子系统（英语：Windows Subsystem for Linux，简称 WSL）是一个为在 Windows 10 和 Windows Server 2019 以上能够原生运行 Linux 二进制可执行文件（ELF 格式）的兼容层。WSL 提供了一个由微软开发的 Linux 兼容的内核接口（不包含 Linux 内核代码），然后可以在其上运行 GNU 用户空间，例如 Ubuntu，openSUSE，SUSE Linux Enterprise Server，Debian 和 Kali Linux。这样的用户空间可能包含 Bash shell 和命令语言，使用本机 GNU/Linux 命令行工具（sed，awk 等），编程语言解释器（Ruby，Python 等），甚至是图形应用程序（使用主机端的 X 窗口系统）。

（3） X86-64 GNU/Linux-Ubuntu22.04

前文已经提及

（4） Arm64 Darwin MacOS Ventura13.3

Darwin 是由苹果公司于 2000 年所发布的一个开放源代码操作系统。Darwin 是 macOS 和 iOS 操作环境的操作系统部分。苹果公司于 2000 年把 Darwin 发布给开放源代码社群。Darwin 是一种类 Unix 操作系统，包含开放源代码的 XNU 内核，其以微核心为基础的核心架构来实现 Mach，而操作系统的服务和用户空间工具则以 BSD 为基础。类似其他类 Unix 操作系统，Darwin 也有对称多处理器的优点，高性能的网络设施和支持多种集成的文件系统。Darwin 的内核是 XNU，它是一种混合内核，它采用了来自 OSF 的 OSFMK 7.3（Open Software Foundation Mach Kernel）和 FreeBSD 的各种要素（包括过程模型，网络堆栈和虚拟文件系统），还有一个称为 I/O Kit 的面向对象的设备驱动程序 API。混合内核设计使其具备了了微内核的灵活性和宏内核的性能。

2.2 LSM-Tree 存储结构

在计算机科学中，日志结构合并树（也称为 LSM 树或 LSMT）是一种具有一定性能特征的数据结构，可以为具有高插入量的文件（例如事务日志）提供索引访问数据。LSM 树和其他搜索树一样，维护键值对。LSM 树将数据保存在两个或多个独立的结构中，每个结构都针对其各自的底层存储介质进行了优化；数据在两个结构之间有效地、

批量地同步。

LSM 树的一个简单版本是两级 LSM 树。两级 LSM 树包含两个树状结构，称为 C0 和 C1。C0 较小，完全驻留在内存中，而 C1 驻留在磁盘上。新记录被插入到内存驻留的 C0 组件中。如果插入导致 C0 组件超过某个大小阈值，则从 C0 中删除一个连续的条目段，并合并到磁盘上的 C1 中。LSM 树的性能特征源于这样一个事实，即每个组件都根据其底层存储介质的特性进行调整，并且使用一种让人联想到归并排序的算法，数据可以滚动批次高效地跨介质迁移。

实践中使用的大多数 LSM 树都采用多个级别。0 级保存在主内存中，可以用树表示。磁盘上的数据被组织成排序的数据运行。每次运行都包含按索引键排序的数据。一次运行可以在磁盘上表示为单个文件，或者表示为具有非重叠键范围的文件集合。要对特定键执行查询以获取其关联值，必须在 Level 0 树中进行搜索，并且每次都运行。LSM 树的 Stepped-Merge 版本是 LSM 树的变体，它支持多层次，每一层次都有多个树结构。一个特定的键可能会出现在多次运行中，这对查询意味着什么取决于应用程序。一些应用程序只需要具有给定键的最新键值对。某些应用程序必须以某种方式组合这些值以获得要返回的正确聚合值。例如，在 Apache Cassandra 中，每个值代表数据库中的一行，不同版本的行可能有不同的列集。为了降低查询成本，系统必须避免运行次数过多的情况。随着越来越多的读写工作负载在 LSM-tree 存储结构下共存，由于 LSM-tree 压缩操作经常使缓冲区缓存中的缓存数据失效，读取数据访问可能会遇到高延迟和低吞吐量。为了重新启用有效的缓冲区缓存以实现快速数据访问，提出并实现了一种日志结构缓冲合并树（LSbM-tree）。

2.3 Raft 共识性算法

Raft 是一种用于替代 Paxos 的共识算法。相比于 Paxos，Raft 的目标是提供更清晰的逻辑分工使得算法本身能被更好地理解，同时它安全性更高，并能提供一些额外的特性。Raft 能为在计算机集群之间部署有限状态机提供一种通用方法，并确保集群内的任意节点在某种状态转换上保持一致。Raft 算法的开源实现众多，在 Go、C++、Java 以及 Scala 中都有完整的代码实现。

2.4 涉及的开源库

1、golang 跨平台文件系统通知库 fsnotify

项目地址：https://github.com/fsnotify/fsnotify

fsnotify 是一个 Go 库，用于在 Windows、Linux、macOS、BSD 和 illumos 上提供跨平台文件系统通知。

2、golang 文件压缩库 snappy

项目地址：https://github.com/golang/snappy

Snappy 是一个压缩/解压库。它不以最大压缩或与任何其他压缩库兼容为目标；相反，它以非常高的速度和合理的压缩为目标。例如，与 zlib 的最快模式相比，Snappy 对大多数输入来说要快一个数量级，但由此产生的压缩文件要大 20 快速：压缩速度为 250 MB/秒及以上，没有汇编代码。稳定：在过去几年里，Snappy 在谷歌的生产环境中压缩和解压缩了 PB 的数据。Snappy bitstream 格式是稳定的，不会在版本之间更改。稳健：Snappy 解压器旨在在遇到损坏或恶意输入时不会崩溃。golang/snappy 是 google/snappy 的官方 golang 实现。

3、golang 测试库 Ginkgo ｜ Gomega

项目地址：https://github.com/onsi/ginkgo

Ginkgo 是 Go 的测试框架，旨在帮助您编写富有表现力的测试。它与 Gomega 匹配器库搭配使用。结合使用时，Ginkgo 和 Gomega 为编写测试提供了丰富且富有表现力的 DSL（领域特定语言）。Ginkgo 有时被描述为“行为驱动开发”（BDD）框架。实际上，Ginkgo 是一个通用测试框架，在各种测试环境中得到积极使用：单元测试、集成测试、验收测试、性能测试等。

4、golang 性能度量库 go-metrics

项目地址：github.com/armon/go-metrics

go-metrics 是一个 Go 应用性能度量指标的库，go-metrics 提供的 meter、histogram 可以覆盖 Go 应用基本性能指标需求（吞吐性能、延迟数据分布等）。go-metrics 是模仿 Coda Hale 的 JVM Metrics 库开发的 golang 运行时性能度量程序。

5、golang 断言库 testify

项目地址：https://github.com/stretchr/testify

testify 是一个具有常见断言和模拟的工具包，可以与标准库无缝贴合使用

2.5 本章小结

3 Radds 存储系统需求分析

3.1 存储系统需求概述

作为一个即时通讯系统，省略一段文字... 性能需求[?]，整体需求如图 3.1 所示。

图 3.1 Radds 存储系统需求概述

3.2 存储系统功能需求分析

系统的功能性需求主要...，详细情况如下：

1、功能需求 1

（1） 注册。省略一段文字

（2） 登录。省略一段文字

（3） 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

2、功能需求 2

（1） 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段

文字省略一段文字省

（2） 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

3、功能需求 3

（1） 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

（2） 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

4、功能需求 4

（1） 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

（2） 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

5、功能需求 5

（1） 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

（2） 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

3.3 存储系统性能需求分析

1、性能需求 1

省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字

省略一段文字省

2、性能需求 2

省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

3、性能需求 3

省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

4、性能需求 4

省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

5、性能需求 5

省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

3.4 存储系统可行性分析

3.5 本章小结

4 Radds 存储系统总体设计

Radds 存储系统总体架构组成如图 4.1 所示，省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

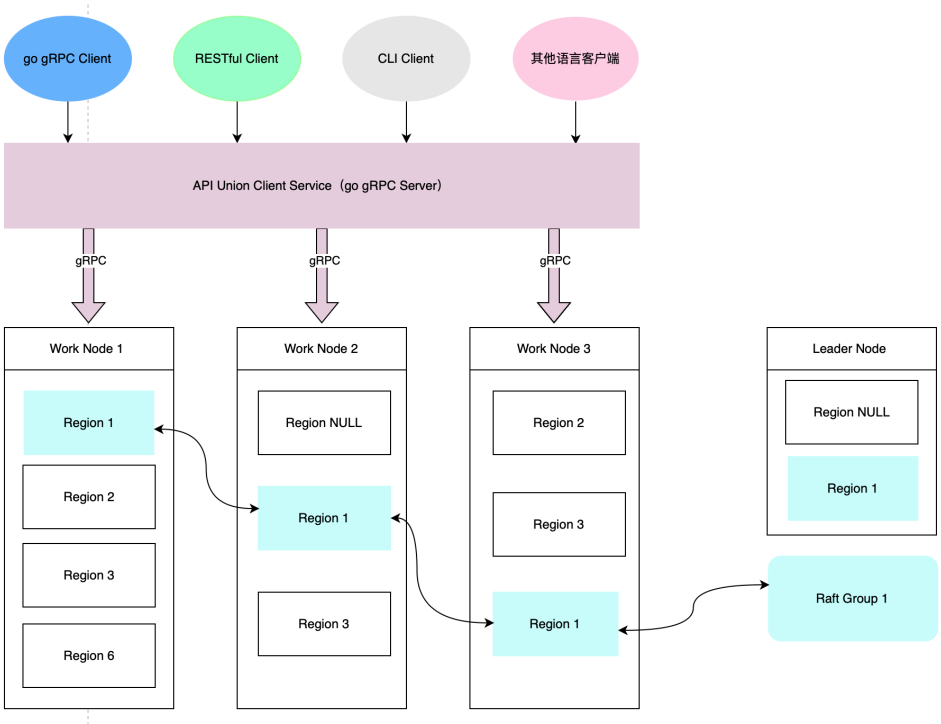

图 4.1 Radds 存储系统总体架构图

4.1 存储系统架构设计

4.2 存储层功能总体设计

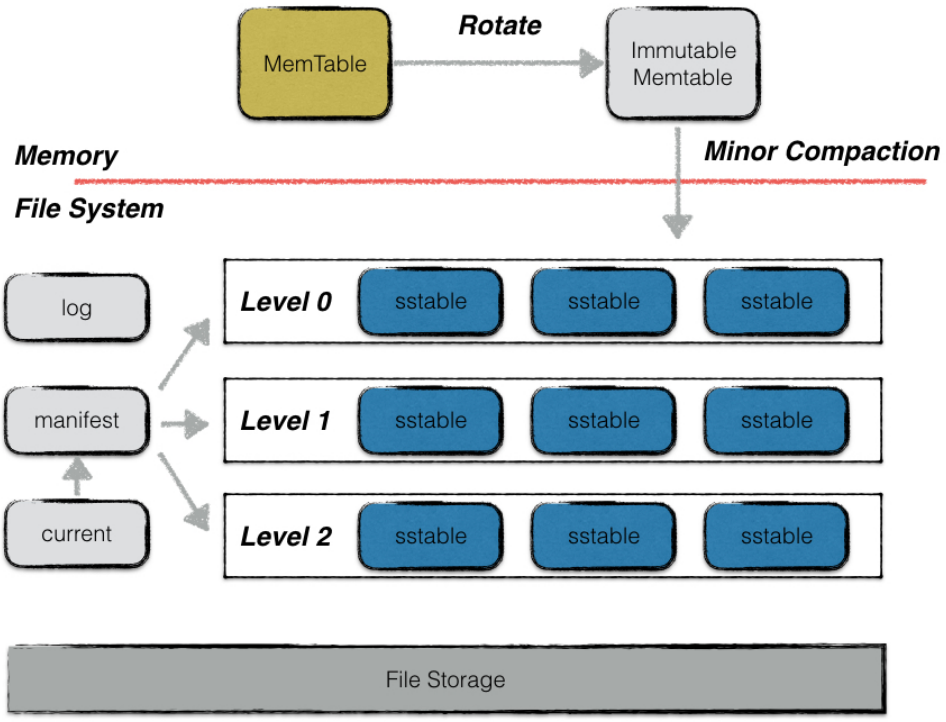

图 4.2 Radds 存储层架构设计

1、存储引擎设计概述

我们要实现一个写性能优秀的存储引擎，则必须实现一个 LSM 树 (Log Structured-Merge Tree)。LSM 树的核心思想就是放弃部分读的性能，换取最大的写入能力。

LSM 树写性能极高的原理，简单地来说就是尽量减少随机写的次数。对于每次写入操作，并不是直接将最新的数据驻留在磁盘中，而是将其拆分成

（1）一次日志文件的顺序写

（2）一次内存中的数据插入

存储架构正是实践了这种思想，将数据首先更新在内存中，当内存中的数据达到一定的阈值，将这部分数据真正刷新到磁盘文件中，因而获得了极高的写性能（顺序写 60MB/s, 随机写 45MB/s）。

4.2.1 内存可变数据结构 memtable 总体设计

之前提到，存储引擎的一次写入操作并不是直接将数据刷新到磁盘文件，而是首先写入到内存中作为代替，memtable 就是一个在内存中进行数据组织与维护的结构。memtable 中，所有的数据按用户定义的排序方法排序之后按序存储，等到其存储内容的容量达到阈值时（默认为 4MB），便将其转换成一个不可修改的 memtable，与此同时创建一个新的 memtable，供用户继续进行读写操作。memtable 底层使用了一种跳表数据结构[?]，这种数据结构效率可以比拟二叉查找树，绝大多数操作的时间复杂度为 O(log n)。

4.2.2 内存不可变数据结构 immutable memtable 总体设计

memtable 的容量到达阈值时，便会转换成一个不可修改的 memtable，也称为 immutable memtable。这两者的结构定义完全一样，区别只是 immutable memtable 是只读的。当一个 immutable memtable 被创建时，存储系统的后台压缩进程便会将利用其中的内容，创建一个 sstable，持久化到磁盘文件中。

4.2.3 日志文件数据结构 journal 总体设计

存储系统的写操作并不是直接写入磁盘的，而是首先写入到内存。假设写入到内存的数据还未来得及持久化，存储系统进程发生了异常，抑或是宿主机器发生了宕机，会造成用户的写入发生丢失。因此存储系统在写内存之前会首先将所有的写操作写到日志文件中，也就是 log 文件。当以下异常情况发生时，均可以通过日志文件进行恢复：

1、写 log 期间进程异常；

2、写 log 完成，写内存未完成；

3、write 动作完成（即 log、内存写入都完成）后，进程异常；

4、immutable memtable 持久化过程中进程异常；

5、其他压缩异常（较为复杂，首先不在这里介绍）；

异常发生时，处理的情况分两种：

1、当第一类情况发生时，数据库重启读取 log 时，发现异常日志数据，抛弃该条日志数据，即视作这次用户写入失败，保障了数据库的一致性；

2、当第二类，第三类，第四类情况发生了，均可以通过 redo 日志文件中记录的写入操作完成数据库的恢复。

每次日志的写操作都是一次顺序写，因此写效率高，整体写入性能较好。此外，存储系统的用户写操作的原子性同样通过日志来实现。

4.2.4 磁盘持久化数据结构 sstable 总体设计

虽然存储系统采用了先写内存的方式来提高写入效率，但是内存中数据不可能无限增长，且日志中记录的写入操作过多，会导致异常发生时，恢复时间过长。因此内存中的数据达到一定容量，就需要将数据持久化到磁盘中。除了某些元数据文件，存储系统的数据主要都是通过 sstable 来进行存储。

虽然在内存中，所有的数据都是按序排列的，但是当多个 memetable 数据持久化到磁盘后，对应的不同的 sstable 之间是存在交集的，在读操作时，需要对所有的 sstable 文件进行遍历，严重影响了读取效率。因此存储系统后台会“定期“整合这些 sstable 文件，该过程也称为 compaction。随着 compaction 的进行，sstable 文件在逻辑上被分成若干层，由内存数据直接 dump 出来的文件称为 level 0 层文件，后期整合而成的文件为 level i 层文件，这也是以 leveldb 为原型的存储系统的这个名字的由来。

注意，所有的 sstable 文件本身的内容是不可修改的，这种设计带来了许多优势，简化了很多设计。

4.2.5 文件元数据 manifest 总体设计

存储系统中有个版本的概念，一个版本中主要记录了每一层中所有文件的元数据，元数据包括（1）文件大小（2）最大 key 值（3）最小 key 值。该版本信息十分关键，除了在查找数据时，利用维护的每个文件的最大／小 key 值来加快查找，还在其中维护了一些进行 compaction 的统计值，来控制 compaction 的进行。

一个文件的元数据主要包括了最大最小 key，文件大小等信息；代码清单 4.2

代码清单 4.1 tFile

```
type tFile struct {
fd          storage.FileDesc
seekLeft    int32
size        int64
imin, imax internalKey
}
```

4.2.6 版本号 current 总体设计

一个版本信息主要维护了每一层所有文件的元数据。

代码清单 4.2 tFile

```
type version struct {
s *session // session - version
levels []tFiles // file meta
cLevel int // next level
cScore float64 // current score
cSeek unsafe.Pointer
closing  bool
ref      int
released bool
}
```

当每次 compaction 完成（或者换一种更容易理解的说法，当每次 sstable 文件有新增或者减少），l eveldb 都会创建一个新的 version，创建的规则是:

versionNew = versionOld + versionEdit

versionEdit 指代的是基于旧版本的基础上，变化的内容（例如新增或删除了某些 sstable 文件）。

manifest 文件就是用来记录这些 versionEdit 信息的。一个 versionEdit 数据，会被编码成一条记录，写入 manifest 文件中。如图 4.3 便是一个 manifest 文件的示意图，其中包含了 3 条 versionEdit 记录，每条记录包括（1）新增哪些 sst 文件（2）删除哪些 sst 文件（3）当前 compaction 的下标（4）日志文件编号（5）操作 seqNumber 等信息。通过这些信息，存储系统便可以在启动时，基于一个空的 version，不断 apply 这些记录，最终得到一个上次运行结束时的版本信息。

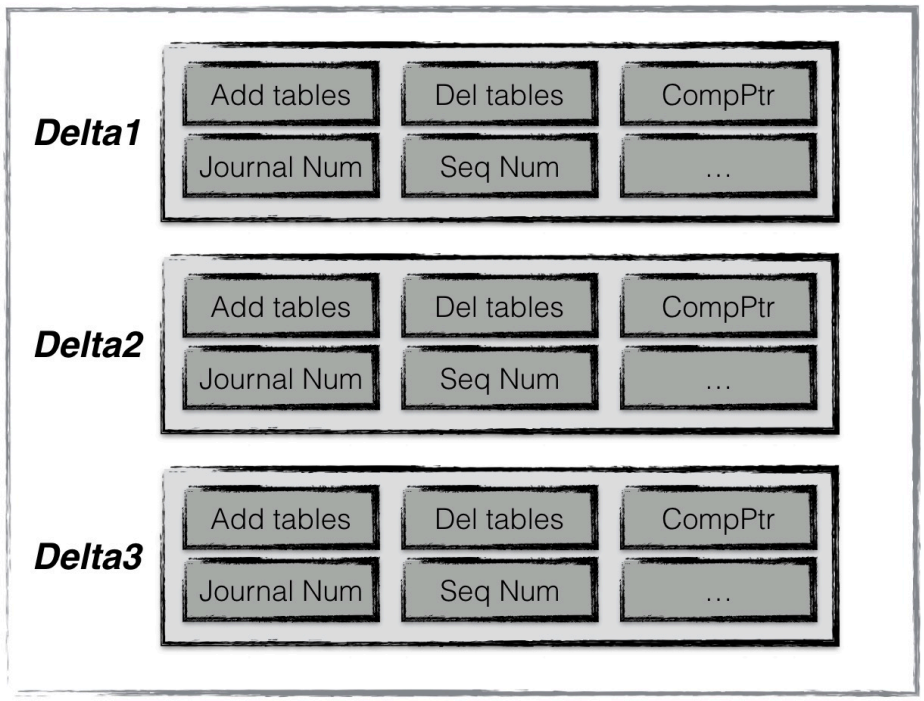

图 4.3 版本信息记录数据结构

4.3 共识层功能总体设计

4.4 客户端功能总体设计

4.4.1 客户端服务平台总体设计

4.4.2 gRPC API 客户端总体设计

4.4.3 RESTful API 客户端总体设计

4.4.4 CLI 客户端总体设计

4.5 本章小结

5 Radds 存储系统详细设计与实现

本章对存储系统的各层进行详细设计与实现，针对各层内部的子系统，各层之间的接口进行详细定义。

5.1 基础层详细设计与实现

5.1.1 错误处理的实现

针对 go 语言本身的特性，错误处理成为整个系统程序开发的首要项目，我们以轻量化，插件化的形式进行错误处理。

代码清单 5.1 Errors

5.1.2 日志系统的实现

为了防止写入内存的数据库因为进程异常、操作系统掉电等情况发生丢失，存储系统在写内存之前会将本次写操作的内容写入日志文件中。

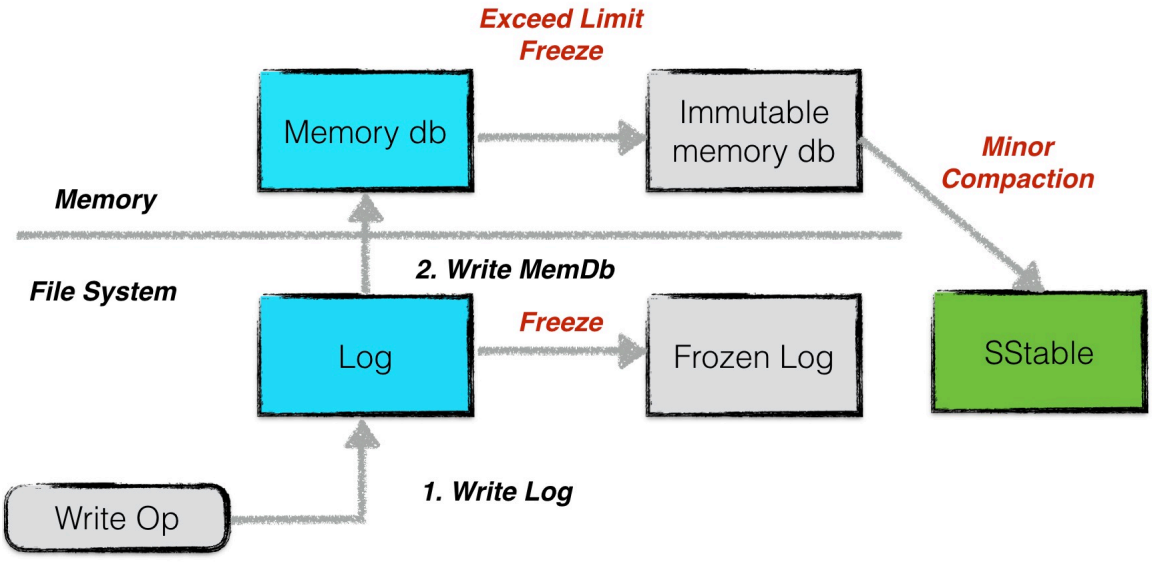

图 5.1 日志系统架构图

存储系统中，有两个 memory db，以及对应的两份日志文件。其中一个 memory db 是可读写的，当这个 db 的数据量超过预定的上限时，便会转换成一个不可写的 memory

db，与此同时，与之对应的日志文件也变成一份 frozen log。

而新生成的 immutable memory db 则会由后台的 minor compaction 进程将其转换成一个 sstable 文件进行持久化，持久化完成，与之对应的 frozen log 被删除。

1、日志结构

图 5.2 日志文件存储结构图

为了增加读取效率，日志文件中按照 block 进行划分，每个 block 的大小为 32KiB。每个 block 中包含了若干个完整的 chunk。

一条日志记录包含一个或多个 chunk。每个 chunk 包含了一个 7 字节大小的 header，前 4 字节是该 chunk 的校验码，紧接的 2 字节是该 chunk 数据的长度，以及最后一个字节是该 chunk 的类型。其中 checksum 校验的范围包括 chunk 的类型以及随后的 data 数据。

chunk 共有四种类型：full，first，middle，last。一条日志记录若只包含一个 chunk，则该 chunk 的类型为 full。若一条日志记录包含多个 chunk，则这些 chunk 的第一个类型为 first, 最后一个类型为 last，中间包含大于等于 0 个 middle 类型的 chunk。

由于一个 block 的大小为 32KiB，因此当一条日志文件过大时，会将第一部分数据写在第一个 block 中，且类型为 first，若剩余的数据仍然超过一个 block 的大小，则第二部分数据写在第二个 block 中，类型为 middle，最后剩余的数据写在最后一个 block 中，类型为 last。

2、日志内容

日志的内容为写入的 batch 编码后的信息。

具体的格式为：

图 5.3 日志文件格式图

一条日志记录的内容包含：Header 和 Data 其中 Header 中有（1）当前 db 的 sequence number（2）本次日志记录中所包含的 put/del 操作的个数。

紧接着写入所有 batch 编码后的内容。

3、日志文件写

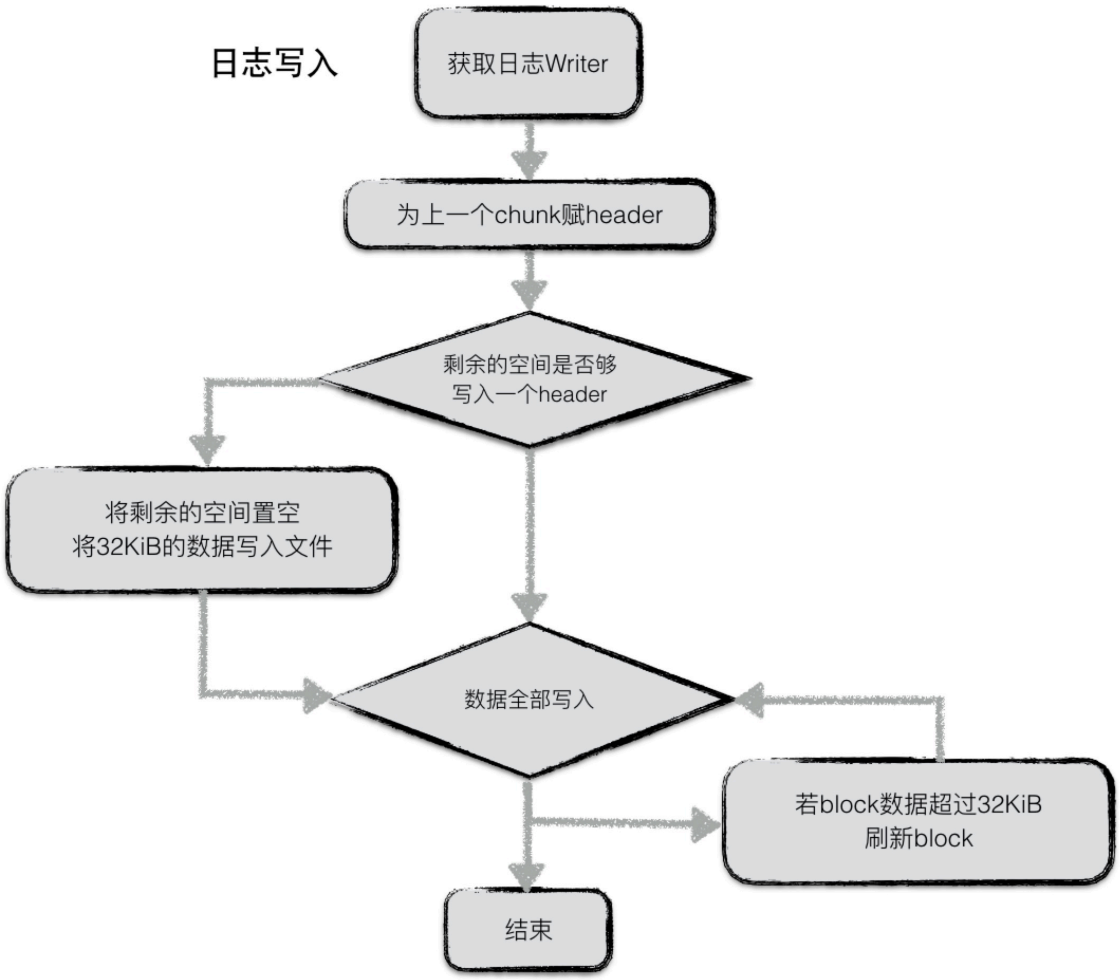

图 5.4 日志文件写流程图

日志写入流程较为简单，在存储系统内部，实现了一个 journal 的 writer。首先调用 Next 函数获取一个 singleWriter，这个 singleWriter 的作用就是写入一条 journal 记录。

singleWriter 开始写入时，标志着第一个 chunk 开始写入。在写入的过程中，不断判断 writer 中 buffer 的大小，若超过 32KiB，将 chunk 开始到现在做为一个完整的 chunk，为其计算 header 之后将整个 chunk 写入文件。与此同时 reset buffer，开始新的 chunk 的写入。

若一条 journal 记录较大，则可能会分成几个 chunk 存储在若干个 block 中。

4、日志文件读

同样，日志读取也较为简单。为了避免频繁的 IO 读取，每次从文件中读取数据时，按 block（32KiB）进行块读取。

每次读取一条日志记录，reader 调用 Next 函数返回一个 singleReader。singleReader 每次调用 Read 函数就返回一个 chunk 的数据。每次读取一个 chunk，都会检查这批数据的校验码、数据类型、数据长度等信息是否正确，若不正确，且用户要求严格的正确性，则返回错误，否则丢弃整个 chunk 的数据。

循环调用 singleReader 的 read 函数，直至读取到一个类型为 Last 的 chunk，表示整条日志记录都读取完毕，返回。

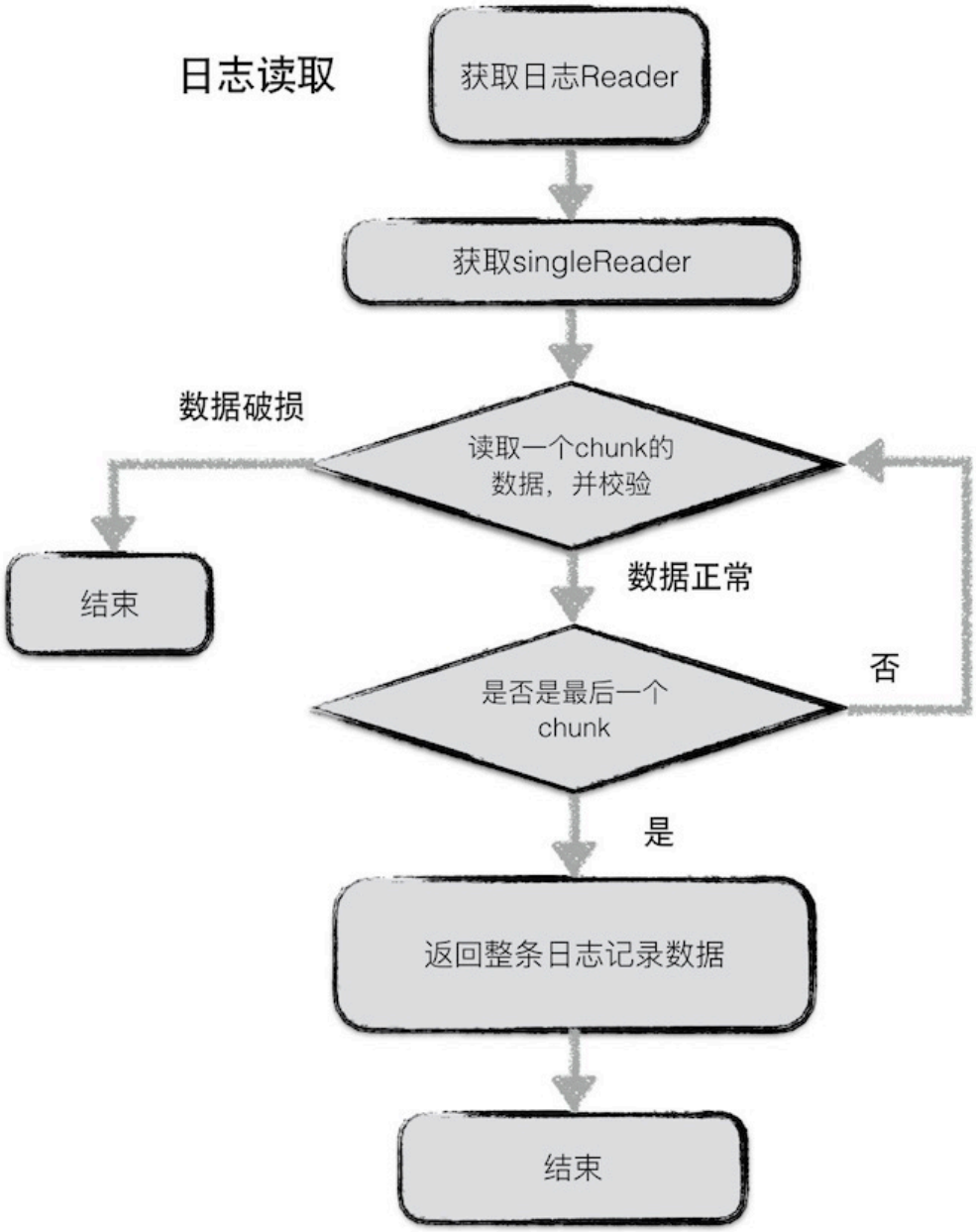

图 5.5 日志文件读流程图

5.1.3 工具库的实现

5.2 存储层详细设计与实现

5.2.1 写数据的实现

1、写入数据的整体流程

先来分析一下存储系统整个写入的流程，底层数据结构的支持以及为何能够优化我们的写入性能。

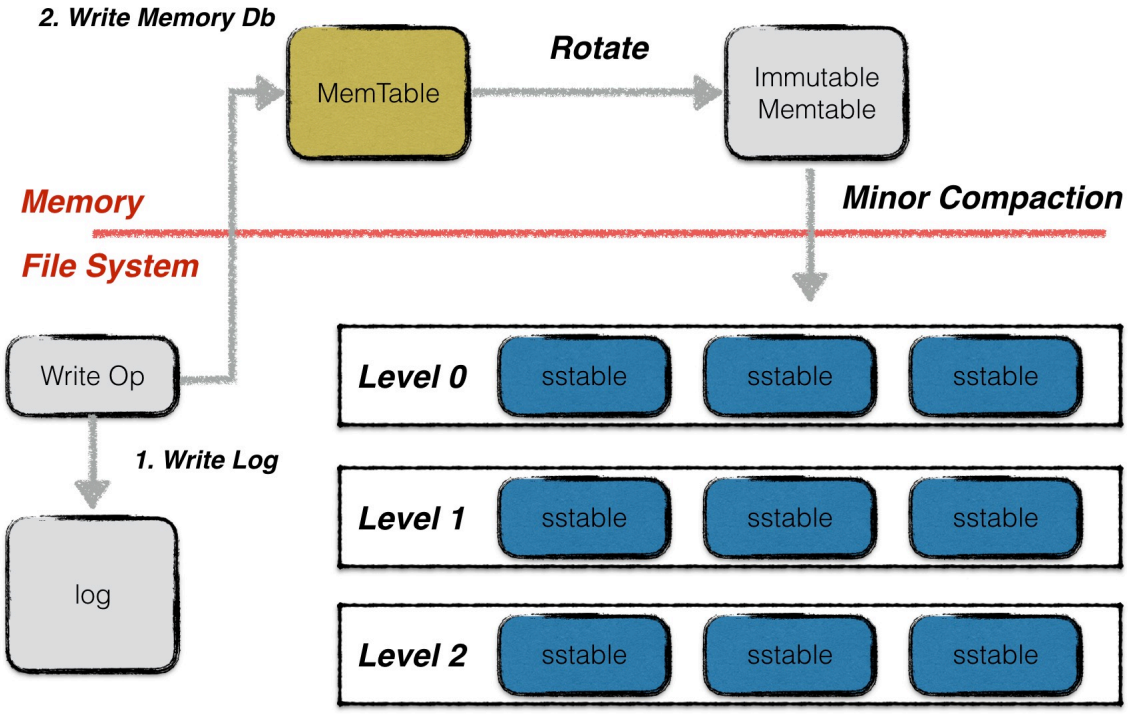

图 5.6 存储系统写数据流程

数据的一次写入分为两部分：

将写操作写入日志；将写操作应用到内存数据库中；之前已经阐述过为何这样的操作可以优化写入性能，以及通过先写日志的方法能够保障用户的写入不丢失。

其实仍然存在写入丢失的隐患。在写设置为非同步的情况下，在写完日志文件以后，操作系统并不是直接将这些数据真正落到磁盘中，而是暂时留在操作系统缓存中，因此当用户写入操作完成，操作系统还未来得及落盘的情况下，发生系统宕机，就会造成写丢失；但是若只是进程异常退出，则不存在该问题。

2、写类型

由于是键值型非关系型数据存储，存储系统对外提供的写入接口有：（1）Put（2）Delete 两种。这两种本质对应同一种操作，Delete 操作同样会被转换成一个 value 为空的 Put 操作。

除此以外，我们还提供了一个批量处理的工具 Batch，用户可以依据 Batch 来完成批量的数据库更新操作，且这些操作是原子性的。

3、batch 结构

无论是 Put/Del 操作，还是批量操作，底层都会为这些操作创建一个 batch 实例作为一个数据库操作的最小执行单元。因此首先介绍一下 batch 的组织结构。

图 5.7 batch 的数据结构

在 batch 中，每一条数据项都按照上图格式进行编码。每条数据项编码后的第一位是这条数据项的类型（更新还是删除），之后是数据项 key 的长度，数据项 key 的内容；若该数据项不是删除操作，则再加上 value 的长度，value 的内容。

batch 中会维护一个 size 值，用于表示其中包含的数据量的大小。该 size 值为所有数据项 key 与 value 长度的累加，以及每条数据项额外的 8 个字节。这 8 个字节用于存储一条数据项额外的一些信息。

4、key 值编码

当数据项从 batch 中写入到内存数据库中时，需要将一个 key 值的转换，即在存储系统内部，所有数据项的 key 是经过特殊编码的，这种格式称为 internalKey。

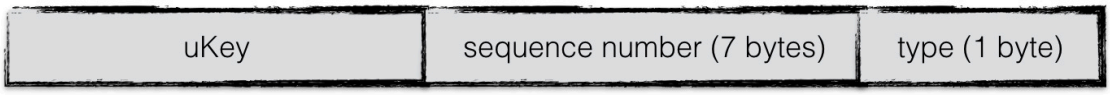

图 5.8 internalkey 的数据结构

internalkey 在用户 key 的基础上，尾部追加了 8 个字节，用于存储（1）该操作对应的 sequence number（2）该操作的类型。

其中，每一个操作都会被赋予一个 sequence number。该计时器是在存储系统内部维护，每进行一次操作就做一个累加。由于在存储系统中，一次更新或者一次删除，采用的是 append 的方式，并非直接更新原数据。因此对应同样一个 key，会有多个版本的数据记录，而最大的 sequence number 对应的数据记录就是最新的。

此外，存储系统的快照（snapshot）也是基于这个 sequence number 实现的，即每一个 sequence number 代表着数据库的一个版本。

5、数据合并写入

存储系统中，在面对并发写入时，做了一个处理的优化。在同一个时刻，只允许一个写入操作将内容写入到日志文件以及内存数据库中。为了在写入进程较多的情况下，减少日志文件的小写入，增加整体的写入性能，存储系统将一些“小写入”合并成一个“大写入”。

当前写操作

（1）第一个写入操作获取到写入锁；

（2）在当前写操作的数据量未超过合并上限，且有其他写操作 pending 的情况下，将其他写操作的内容合并到自身；

（3）若本次写操作的数据量超过上限，或者无其他 pending 的写操作了，将所有内容统一写入日志文件，并写入到内存数据库中；

（4）通知每一个被合并的写操作最终的写入结果，释放或移交写锁；

其它写操作

（1）等待获取写锁或者被合并；

（2）若被合并，判断是否合并成功，若成功，则等待最终写入结果；

（3）反之，则表明获取锁的写操作已经 oversize 了，此时，该操作直接从上个占有锁的写操作中接过写锁进行写入；

（4）若未被合并，则继续等待写锁或者等待被合并；

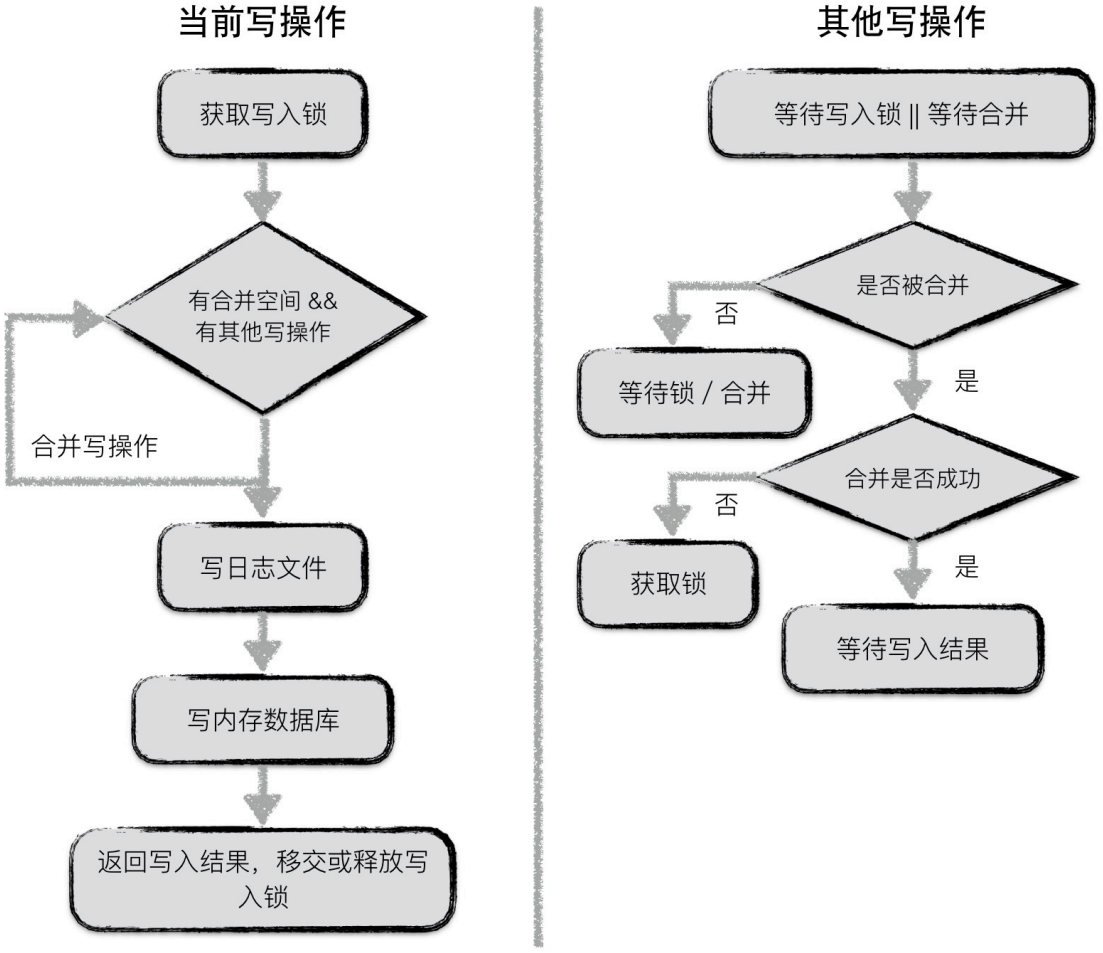

图 5.9 写合并的流程

6、原子性

存储系统的任意一个写操作（无论包含了多少次写），其原子性都是由日志文件实现的。一个写操作中所有的内容会以一个日志中的一条记录，作为最小单位写入。

考虑以下两种异常情况：

（1）写日志未开始，或写日志完成一半，进程异常退出；（2）写日志完成，进程异常退出；

前者中可能存储一个写操作的部分写已经被记载到日志文件中，仍然有部分写未被记录，这种情况下，当数据库重新启动恢复时，读到这条日志记录时，发现数据异常，直接丢弃或退出，实现了写入的原子性保障。

后者，写日志已经完成，写入日志的数据未真正持久化，存储系统启动恢复时通过 redo 日志实现数据写入，仍然保障了原子性。

5.2.2 读数据的实现

存储系统提供给用户两种进行读取数据的接口：

直接通过 Get 接口读取数据；首先创建一个 snapshot，基于该 snapshot 调用 Get 接口读取数据；两者的本质是一样的，只不过第一种调用方式默认地以当前数据库的状态创建了一个 snapshot，并基于此 snapshot 进行读取。

读者可能不了解 snapshot（快照）到底是什么？简单地来说，就是数据库在某一个时刻的状态。基于一个快照进行数据的读取，读到的内容不会因为后续数据的更改而改变。

由于两种方式本质都是基于快照进行读取的，因此在介绍读操作之前，首先介绍快照。

1、snapshot（快照）

快照代表着数据库某一个时刻的状态，在存储系统中，巧妙地用一个整型数来代表一个数据库状态。

在存储系统中，用户对同一个 key 的若干次修改（包括删除）是以维护多条数据项的方式进行存储的（直至进行 compaction 时才会合并成同一条记录），每条数据项都会被赋予一个序列号，代表这条数据项的新旧状态。一条数据项的序列号越大，表示其中代表的内容为最新值。

因此，每一个序列号，其实就代表着存储系统的一个状态。换句话说，每一个序列号都可以作为一个状态快照。

当用户主动或者被动地创建一个快照时，存储系统会以当前最新的序列号对其赋值。例如图中用户在序列号为 98 的时刻创建了一个快照，并且基于该快照读取 key 为“name”的数据时，即便此刻用户将”name” 的值修改为”dog”，再删除，用户读取到的内容仍然是“cat”。

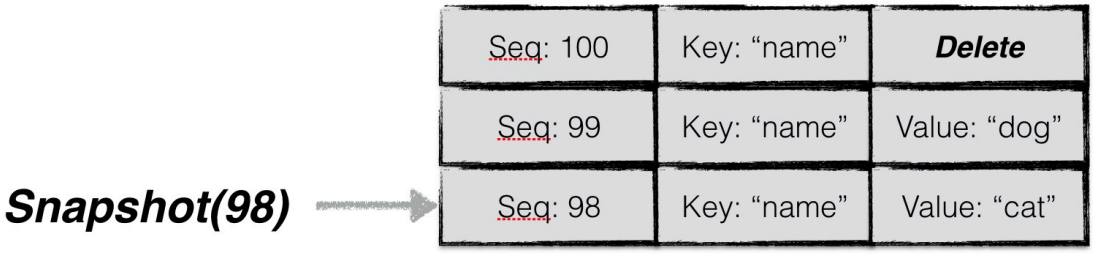

图 5.10 快照数据示例图

所以，利用快照能够保证数据库进行并发的读写操作。

在获取到一个快照之后，存储系统会为本次查询的 key 构建一个 internalKey（格式如上文所述），其中 internalKey 的 seq 字段使用的便是快照对应的 seq。通过这种方式可以过滤掉所有 seq 大于快照号的数据项。

2、读数据流程

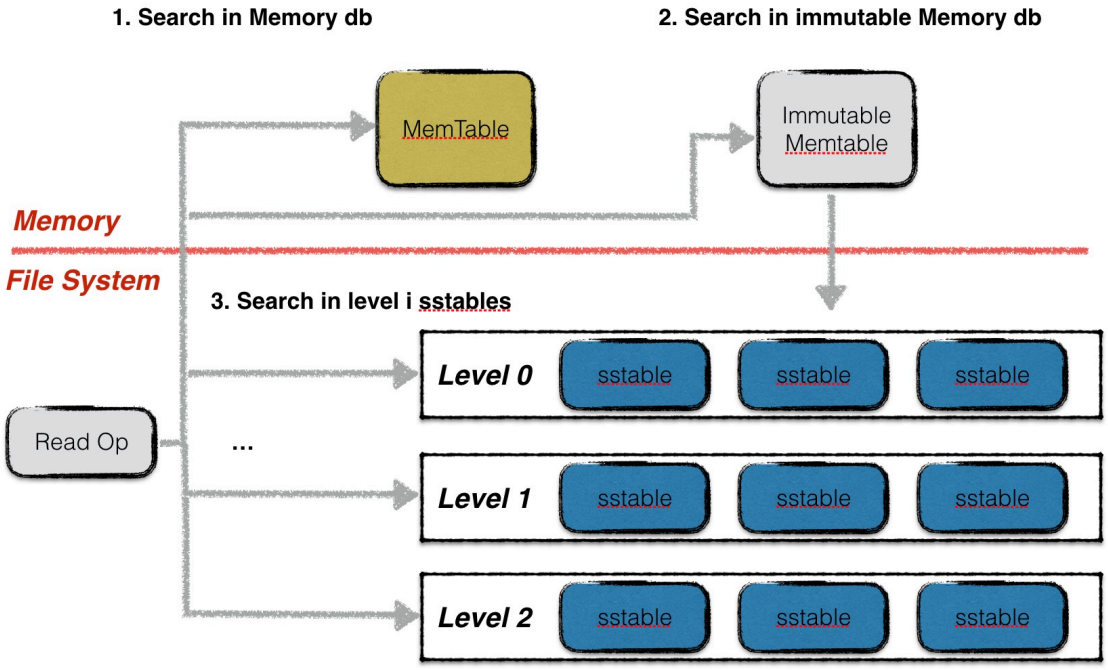

图 5.11 读数据流程

存储系统读取分为三步：

（1）在 memory db 中查找指定的 key，若搜索到符合条件的数据项，结束查找；

（2）在冻结的 memory db 中查找指定的 key，若搜索到符合条件的数据项，结束查找；

（3）按低层至高层的顺序在 level i 层的 sstable 文件中查找指定的 key，若搜索到符合条件的数据项，结束查找，否则返回 Not Found 错误，表示数据库中不存在指定的数据；

注意存储系统在每一层 sstable 中查找数据时，都是按序依次查找 sstable 的。

0 层的文件比较特殊。由于 0 层的文件中可能存在 key 重合的情况，因此在 0 层中，文件编号大的 sstable 优先查找。理由是文件编号较大的 sstable 中存储的总是最新的数据。

非 0 层文件，一层中所有文件之间的 key 不重合，因此存储系统可以借助 sstable 的元数据（一个文件中最小与最大的 key 值）进行快速定位，每一层只需要查找一个 sstable 文件的内容。

在 memory db 或者 sstable 的查找过程中，需要根据指定的序列号拼接一个 internalKey，查找用户 key 一致，且 seq 号不大于指定 seq 的数据，

5.2.3 内存数据库的实现

内存数据库用来维护有序的 key-value 对，其底层是利用跳表实现，绝大多数操作（读／写）的时间复杂度均为 O(log n)，有着与平衡树相媲美的操作效率，但是从实现的角度来说简单许多，接下来将介绍一下内存数据库的实现细节。

1、跳表对实现内存数据库的分析

跳表（SkipList）是由 William Pugh 提出的。他在论文《Skip lists: a probabilistic alternative to balanced trees》中详细地介绍了有关跳表结构、插入删除操作的细节。

这种数据结构是利用概率均衡技术，加快简化插入、删除操作，且保证绝大大多操作均拥有 O(log n) 的良好效率。

原文的一段话道出了跳表在数据结构中运用离散数学知识的精髓：数据结构是离散的，计算机的本质是离散的。

Skip lists are a data structure that can be used in place of balanced trees. Skip lists use probabilistic balancing rather than strictly enforced balancing and as a result the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.

图 5.12 跳表的影响

平衡树（以红黑树为代表）是一种非常复杂的数据结构，为了维持树结构的平衡，获取稳定的查询效率，平衡树每次插入可能会涉及到较为复杂的节点旋转等操作。作者设计跳表的目的就是借助概率平衡，来构建一个快速且简单的数据结构，取代平衡树。

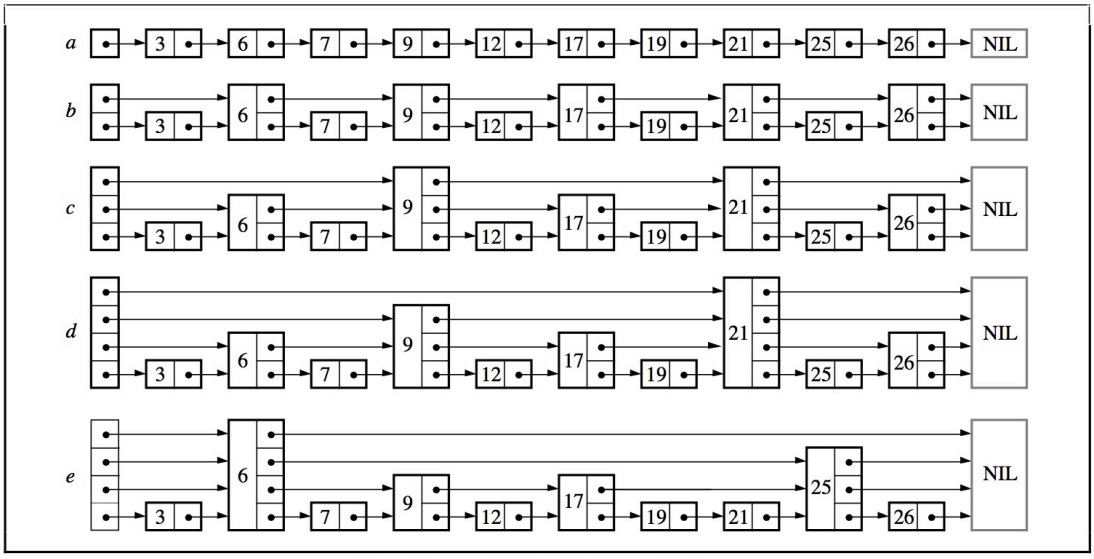

FIGURE 1 - Linked lists with additional pointers

图 5.13 一个跳表的图

作者从链表讲起，一步步引出了跳表这种结构的由来。

图 a 中，所有元素按序排列，被存储在一个链表中，则一次查询之多需要比较 N 个链表节点；

图 b 中，每隔 2 个链表节点，新增一个额外的指针，该指针指向间距为 2 的下一个节点，如此以来，借助这些额外的指针，一次查询至多只需要 □n/2□ + 1 次比较；

图 c 中，在图 b 的基础上，每隔 4 个链表节点，新增一个额外的指针，指向间距为 4 的下一个节点，一次查询至多需要 □n/4□ + 2 次比较；

作者推论，若每隔 2 个节点，新增一个辅助指针，最终一次节点的查询效率为 O(log n)。但是这样不断地新增指针，使得一次插入、删除操作将会变得非常复杂。

一个拥有 k 个指针的结点称为一个 k 层结点（level k node）。按照上面的逻辑，50% 的结点为 1 层节点，25% 的结点为 2 层节点，12.5%。若保证每层节点的分布如上述概率所示，则仍然能够相同的查询效率。图 e 便是一个示例。

维护这些辅助指针将会带来较大的复杂度，因此作者将每一层中，每个节点的辅助指针指向该层中下一个节点。故在插入删除操作时，只需跟操作链表一样，修改相关的前后两个节点的内容即可完成，作者将这种数据结构称为跳表。

2、跳表的结构

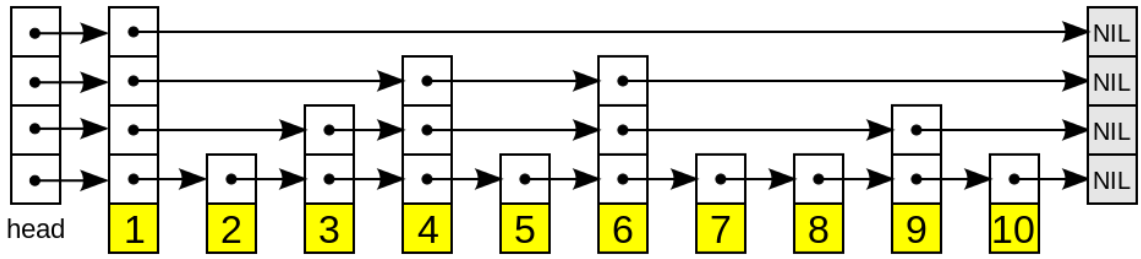

图 5.14 跳表的结构

跳跃列表是按层建造的。底层是一个普通的有序链表。每个更高层都充当下面链表的” 快速通道”，这里在层 i 中的元素按某个固定的概率 p (通常为 0.5 或 0.25) 出现在层 i+1 中。平均起来，每个元素都在 1/(1-p) 个列表中出现，而最高层的元素（通常是在跳跃列表前端的一个特殊的头元素）在 O(log1/p n) 个列表中出现。

3、跳表的查找

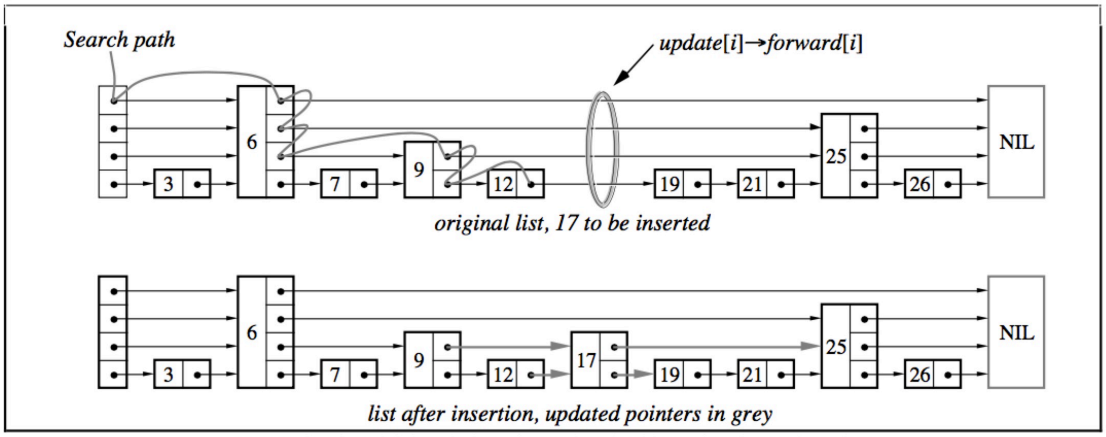

FIGURE 3 - Pictorial description of steps involved in performing an insertion

图 5.15 跳表的查找图

在介绍插入和删除操作之前，我们首先介绍查找操作，该操作是上述两个操作的基础。

例如图中，需要查找一个值为 17 的链表节点，查找的过程为：

首先根据跳表的高度选取最高层的头节点；

若跳表中的节点内容小于查找节点的内容，则取该层的下一个节点继续比较；

若跳表中的节点内容等于查找节点的内容，则直接返回；

若跳表中的节点内容大于查找节点的内容，且层高不为 0，则降低层高，且从前一个节点开始，重新查找低一层中的节点信息；若层高为 0，则返回当前节点，该节点的 key 大于所查找节点的 key。

综合来说，就是利用稀疏的高层节点，快速定位到所需要查找节点的大致位置，再利用密集的底层节点，具体比较节点的内容。

4、基于跳表实现内存数据库 memtable

5.3 共识层详细设计与实现

5.4 客户端层详细设计与实现

5.4.1 API 客户端服务平台的实现

5.4.2 gRPC API 客户端的实现

5.4.3 RESTful API 客户端的实现

5.4.4 CLI 命令行客户端的实现

5.5 本章小结

6 Radds 存储系统部署、日志分析与客户端测试

6.1 分布式存储系统部署

6.1.1 在 X86-64 GNU/Linux Ubuntu22.04 操作系统部署

6.1.2 在 X86-64 Windows11 操作系统部署

6.1.3 在 Arm64 Darwin MacOS 操作系统部署

6.2 数据存储系统日志分析

6.3 共识性系统日志分析

6.4 客户端服务平台日志分析

6.5 客户端测试

6.5.1 gRPC API 客户端测试

6.5.2 RESTful API 客户端测试

6.5.3 CLI 客户端测试

6.6 本章小结

结论

1、调研了...

2、调研...

3、设计...

4、依据...

5、对整个系统...

致　谢

附　录

附录 A

In Search of an Understandable Consensus Algorithm(Extended Version)

中文译文 A

Raft 寻找一种易于理解的共识性算法

附录 B

The Log-Structured Merge-Tree

中文译文 B

日志结构归并树