从零实现一个 k-v 存储引擎

原创 roseduan roseduan写字的地方 2021-06-30 08:35

写这篇文章的目的，是为了帮助更多的人理解 rosedb，我会从零开始实现一个简单的包含 PUT、GET、DELETE 操作的 k-v 存储引擎。

你可以将其看做是一个简易版本的 rosedb，就叫它 minidb 吧（mini 版本的 rosedb）。

无论你是 Go 语言初学者，还是想进阶 Go 语言，或者是对 k-v 存储感兴趣，都可以尝试自己动手实现一下，我相信一定会对你帮助很大的。

说到存储，其实解决的一个核心问题就是，怎么存放数据，怎么取出数据。在计算机的世界里，这个问题会更加的多样化。

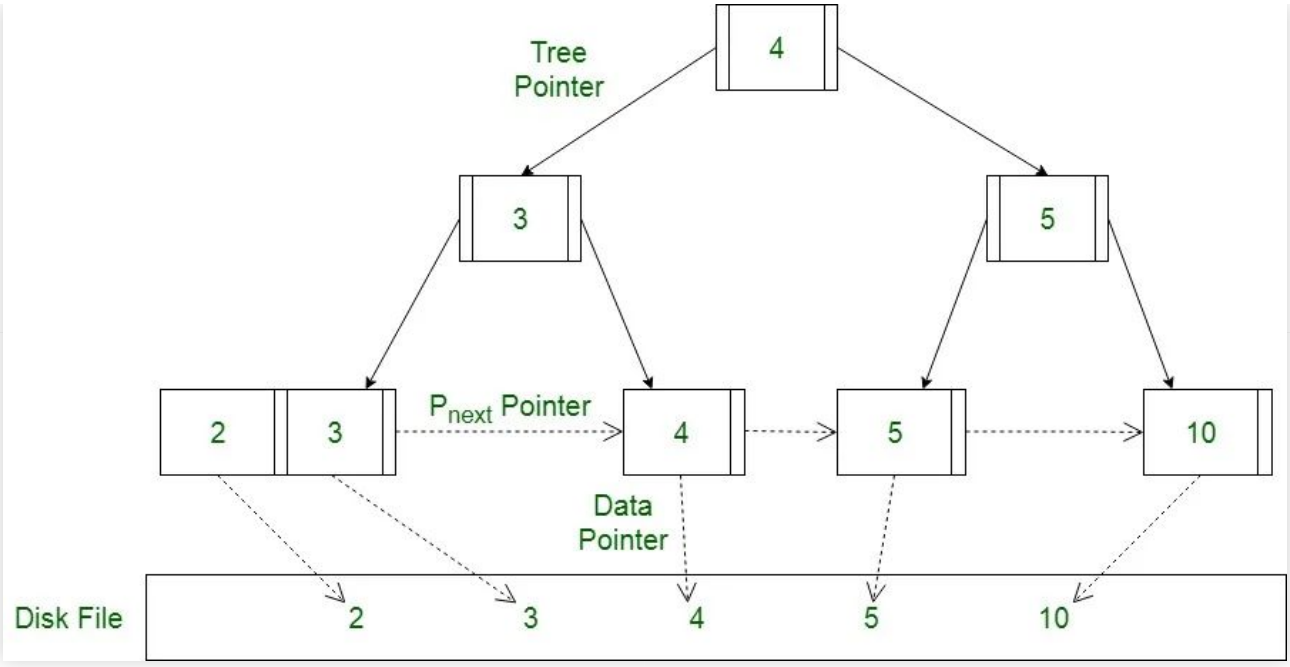
计算机当中有内存和磁盘，内存是易失性的，掉电之后存储的数据全部丢失，所以，如果想要系统崩溃再重启之后依然正常使用，就不得不将数据存储在非易失性介质当中，最常见的便是磁盘。

所以，针对一个单机版的 k-v，我们需要设计数据在内存中应该怎么存放，在磁盘中应该怎么存放。

当然，已经有很多优秀的前辈们去探究过了，并且已经有了经典的总结，主要将数据存储的模型分为了两类：B+ 树和 LSM 树。

本文的重点不是讲这两种模型，所以只做简单介绍。

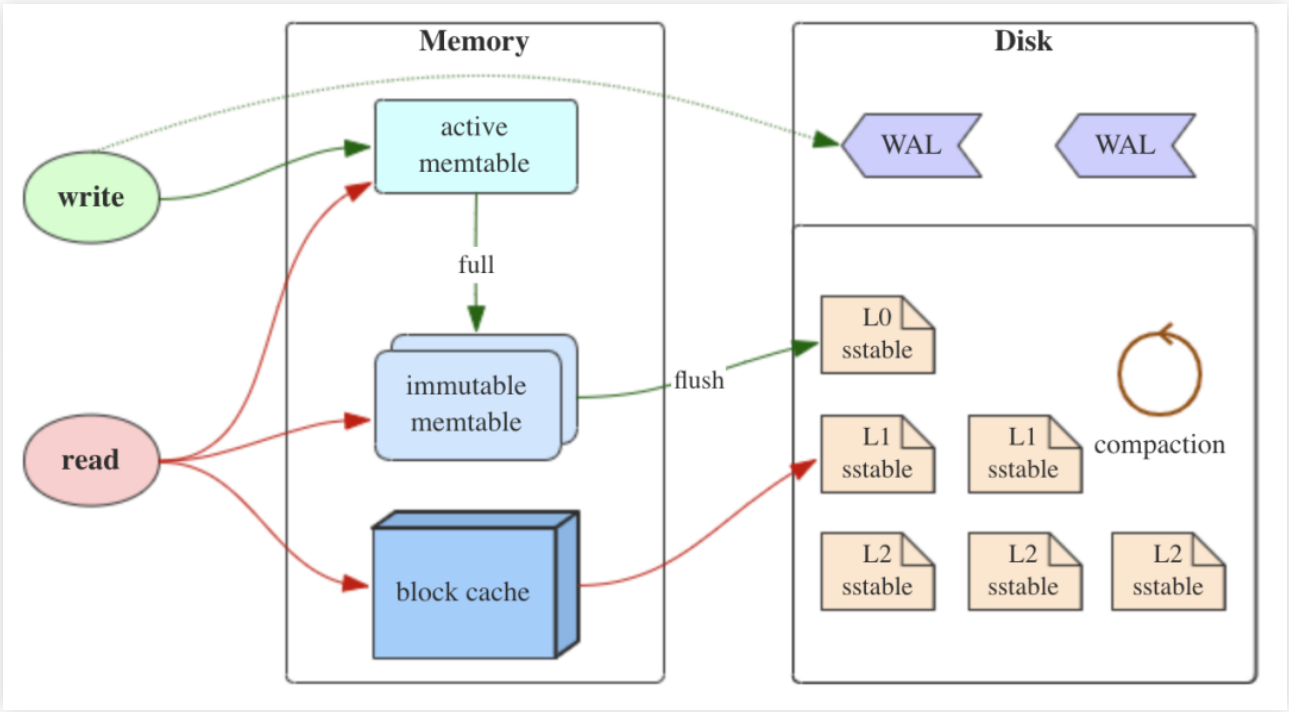
B+ 树



B+ 树由二叉查找树演化而来，通过增加每层节点的数量，来降低树的高度，适配磁盘的页，尽量减少磁盘 IO 操作。

B+ 树查询性能比较稳定，在写入或更新时，会查找并定位到磁盘中的位置并进行原地操作，注意这里是随机 IO，并且大量的插入或删除还有可能触发页分裂和合并，写入性能一般，因此 B+ 树适合读多写少的场景。

LSM 树



LSM Tree（Log Structured Merge Tree，日志结构合并树）其实并不是一种具体的

树类型的数据结构，而只是一种数据存储的模型，它的核心思想基于一个事实：顺序 IO 远快于随机 IO。

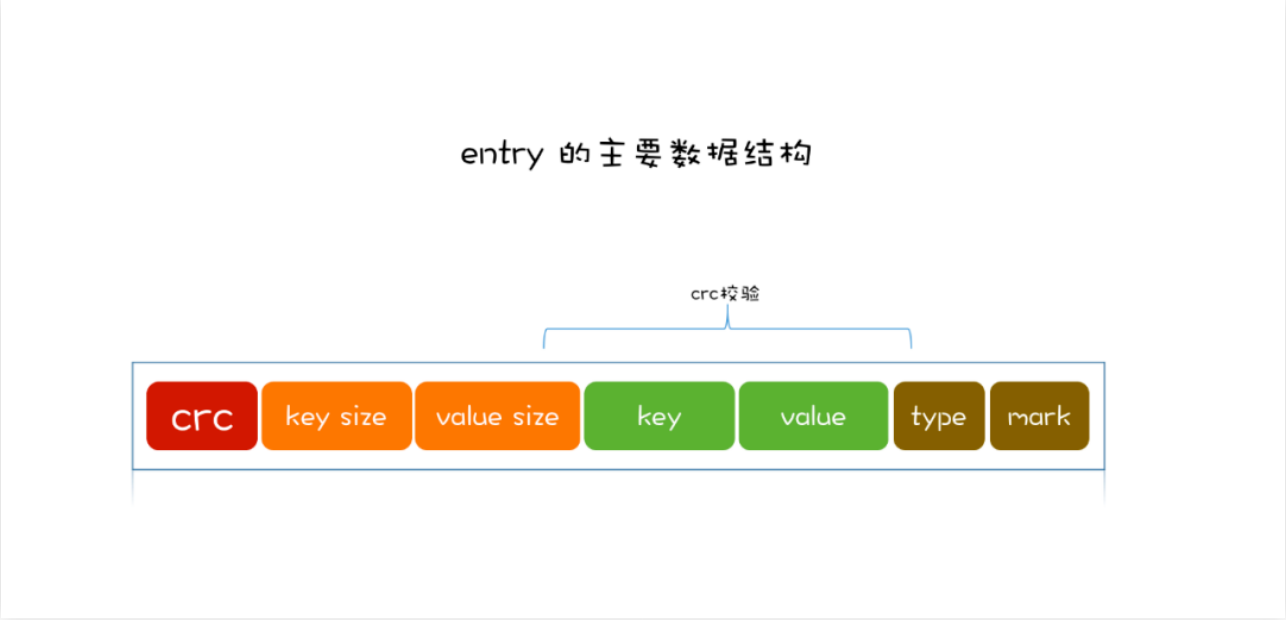
和 B+ 树不同，在 LSM 中，数据的插入、更新、删除都会被记录成一条日志，然后追加写入到磁盘文件当中，这样所有的操作都是顺序 IO，因此 LSM 比较适用于写多读少的场景。

看了前面的两种基础存储模型，相信你已经对如何存取数据有了基本的了解，而 minidb 基于一种更加简单的存储结构，总体上它和 LSM 比较类似。

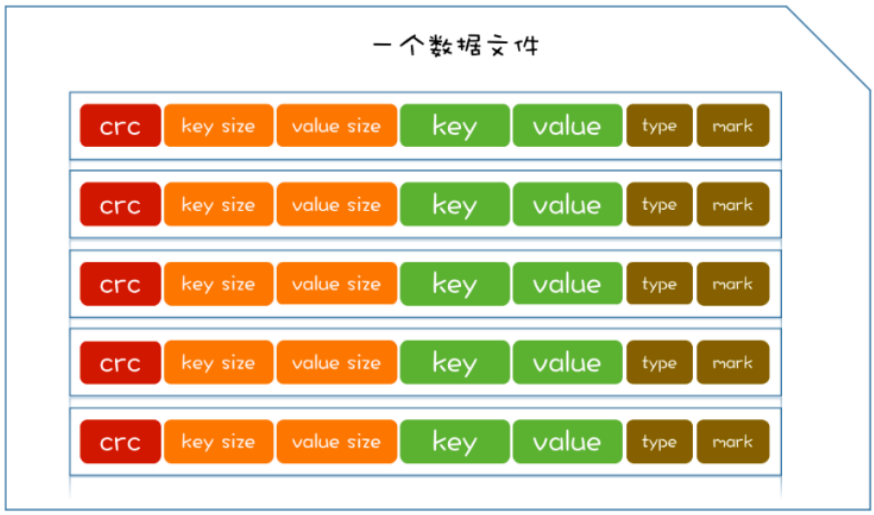
我先不直接干巴巴的讲这个模型的概念，而是通过一个简单的例子来看一下 minidb 当中数据 PUT、GET、DELETE 的流程，借此让你理解这个简单的存储模型。

PUT

我们需要存储一条数据，分别是 key 和 value，首先，为预防数据丢失，我们会将这个 key 和 value 封装成一条记录（这里把这条记录叫做 Entry），追加到磁盘文件当中。Entry 的里面的内容，大致是 key、value、key 的大小、value 的大小、写入的时间。



所以磁盘文件的结构非常简单，就是多个 Entry 的集合。



磁盘更新完了，再更新内存，内存当中可以选择一个简单的数据结构，比如哈希表。哈希表的 key 对应存放的是 Entry 在磁盘中的位置，便于查找时进行获取。

这样，在 minidb 当中，一次数据存储的流程就完了，只有两个步骤：一次磁盘记录的追加，一次内存当中的索引更新。

GET

再来看 GET 获取数据，首先在内存当中的哈希表查找到 key 对应的索引信息，这其中包含了 value 存储在磁盘文件当中的位置，然后直接根据这个位置，到磁盘当中去取出 value 就可以了。

DEL

然后是删除操作，这里并不会定位到原记录进行删除，而还是将删除的操作封装成 Entry，追加到磁盘文件当中，只是这里需要标识一下 Entry 的类型是删除。

然后在内存当中的哈希表删除对应的 key 的索引信息，这样删除操作便完成了。
可以看到，不管是插入、查询、删除，都只有两个步骤：一次内存中的索引更新，一次磁盘文件的记录追加。所以无论数据规模如何， minidb 的写入性能十分稳定。

Merge

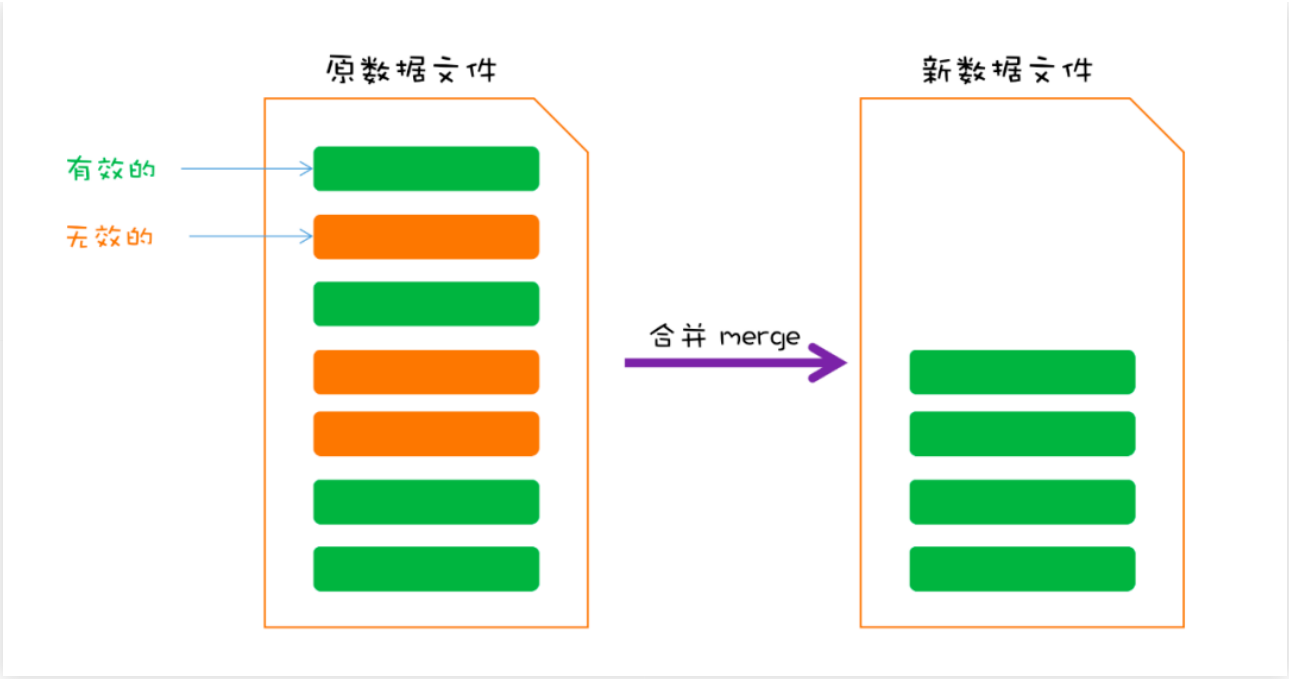
最后再来看一个比较重要的操作，前面说到，磁盘文件的记录是一直在追加写入的，这样会导致文件容量也一直在增加。并且对于同一个 key，可能会在文件中存在多条 Entry（回想一下，更新或删除 key 内容也会追加记录），那么在数据文件当中，其实存在冗余的 Entry 数据。

举一个简单的例子，比如针对 key A， 先后设置其 value 为 10、20、30，那么磁盘文件中就有三条记录：

此时 A 的最新值是 30，那么其实前两条记录已经是无效的了。

针对这种情况，我们需要定期合并数据文件，清理无效的 Entry 数据，这个过程一般叫做 merge。

merge 的思路也很简单，需要取出原数据文件的所有 Entry，将有效的 Entry 重新写入到一个新建的临时文件中，最后将原数据文件删除，临时文件就是新的数据文件了。



这就是 minidb 底层的数据存储模型，它的名字叫做 bitcask，当然 rosedb 采用的也是这种模型。它本质上属于类 LSM 的模型，核心思想是利用顺序 IO 来提升写性能，只不过在实现上，比 LSM 简单多了。

介绍完了底层的存储模型，就可以开始代码实现了，我将完整的代码实现放到了我的 Github 上面，地址：

https://github.com/roseduan/minidb

文章当中就截取部分关键的代码。

首先是打开数据库，需要先加载数据文件，然后取出文件中的 Entry 数据，还原索引状态，关键部分代码如下：

```
func Open(dirPath string) (*MiniDB, error) {
    // 如果数据库目录不存在，则新建一个
    if _, err := os.Stat(dirPath); os.IsNotExist(err) {
        if err := os.MkdirAll(dirPath, os.ModePerm); err != nil {
            return nil, err
        }
    }

    // 加载数据文件
    dbFile, err := NewDBFile(dirPath)
    if err != nil {
```

```
        return nil, err
    }

    db := &MiniDB{
        dbFile: dbFile,
        indexes: make(map[string]int64),
        dirPath: dirPath,
    }

    // 加载索引
    db.loadIndexesFromFile(dbFile)
    return db, nil
}
```

再来看看 PUT 方法，流程和上面的描述一样，先更新磁盘，写入一条记录，再更新内存：

```
func (db *MiniDB) Put(key []byte, value []byte) (err error) {

    offset := db.dbFile.Offset
    // 封装成 Entry
    entry := NewEntry(key, value, PUT)
    // 追加到数据文件当中
    err = db.dbFile.Write(entry)

    // 写到内存
    db.indexes[string(key)] = offset
    return
}
```

GET 方法需要先从内存中取出索引信息，判断是否存在，不存在直接返回，存在的话从磁盘当中取出数据。

```
func (db *MiniDB) Get(key []byte) (val []byte, err error) {
    // 从内存当中取出索引信息
    offset, ok := db.indexes[string(key)]
    // key 不存在
    if !ok {
        return
    }
```

```
    // 从磁盘中读取数据
    var e *Entry
    e, err = db.dbFile.Read(offset)
    if err != nil && err != io.EOF {
        return
    }
    if e != nil {
        val = e.Value
    }
    return
}
```

DEL 方法和 PUT 方法类似，只是 Entry 被标识为了 DEL ，然后封装成 Entry 写到文件当中：

```
func (db *MiniDB) Del(key []byte) (err error) {
    // 从内存当中取出索引信息
    _, ok := db.indexes[string(key)]
    // key 不存在，忽略
    if !ok {
        return
    }

    // 封装成 Entry 并写入
    e := NewEntry(key, nil, DEL)
    err = db.dbFile.Write(e)
    if err != nil {
        return
    }

    // 删除内存中的 key
    delete(db.indexes, string(key))
    return
}
```

最后是重要的合并数据文件操作，流程和上面的描述一样，关键代码如下：

```
func (db *MiniDB) Merge() error {
    // 读取原数据文件中的 Entry
    for {
        e, err := db.dbFile.Read(offset)
        if err != nil {
```

```
            if err == io.EOF {
                break
            }
            return err
        }
        // 内存中的索引状态是最新的，直接对比过滤出有效的 Entry
        if off, ok := db.indexes[string(e.Key)]; ok && off == offset {
            validEntries = append(validEntries, e)
        }
        offset += e.GetSize()
    }

    if len(validEntries) > 0 {
        // 新建临时文件
        mergeDBFile, err := NewMergeDBFile(db.dirPath)
        if err != nil {
            return err
        }
        defer os.Remove(mergeDBFile.File.Name())

        // 重新写入有效的 entry
        for _, entry := range validEntries {
            writeOff := mergeDBFile.Offset
            err := mergeDBFile.Write(entry)
            if err != nil {
                return err
            }

            // 更新索引
            db.indexes[string(entry.Key)] = writeOff
        }

        // 删除旧的数据文件
        os.Remove(db.dbFile.File.Name())
        // 临时文件变更为新的数据文件
        os.Rename(mergeDBFile.File.Name(), db.dirPath+string(os.PathSepa

        db.dbFile = mergeDBFile
    }
    return nil
}
```

除去测试文件，minidb 的核心代码只有 300 行，麻雀虽小，五脏俱全，它已经包含

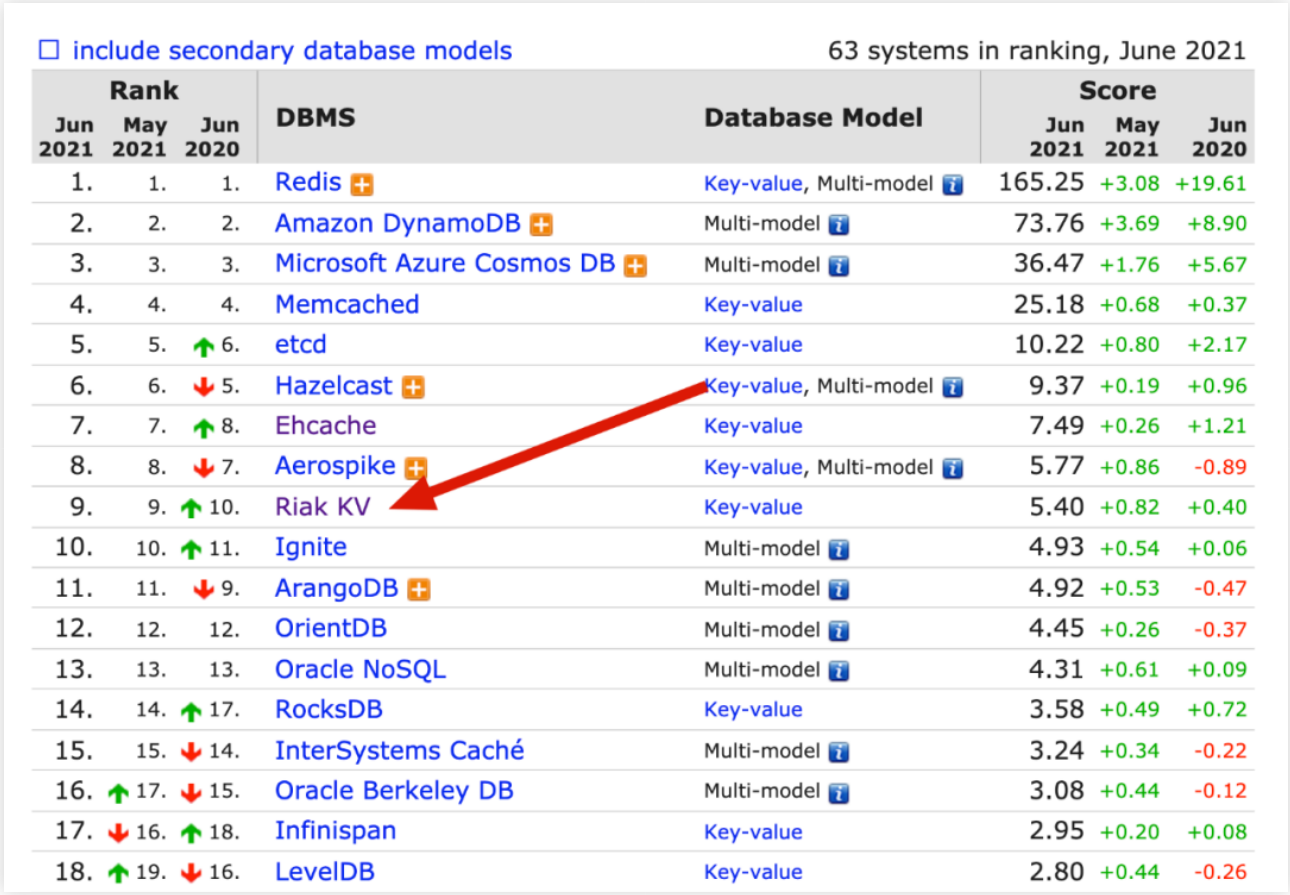
了 bitcask 这个存储模型的主要思想，并且也是 rosedb 的底层基础。

理解了 minidb 之后，基本上就能够完全掌握 bitcask 这种存储模型，多花点时间，相信对 rosedb 也能够游刃有余了。

进一步，如果你对 k-v 存储这方面感兴趣，可以更加深入的去研究更多相关的知识，bitcask 虽然简洁易懂，但是问题也不少，rosedb 在实践的过程当中，对其进行了一些优化，但目前还是有不少的问题存在。

有的人可能比较疑惑，bitcask 这种模型简单，是否只是一个玩具，在实际的生产环境中有应用吗？答案是肯定的。

bitcask 最初源于 Riak 这个项目的底层存储模型，而 Riak 是一个分布式 k-v 存储，在 NoSQL 的排名中也名列前茅：



☐ include secondary database models — 63 systems in ranking, June 2021

Rank Jun 2021	Rank May 2021	Rank Jun 2020	DBMS	Database Model	Score Jun 2021	Score May 2021	Score Jun 2020
1.	1.	1.	Redis	Key-value, Multi-model	165.25	+3.08	+19.61
2.	2.	2.	Amazon DynamoDB	Multi-model	73.76	+3.69	+8.90
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	36.47	+1.76	+5.67
4.	4.	4.	Memcached	Key-value	25.18	+0.68	+0.37
5.	5.	↑ 6.	etcd	Key-value	10.22	+0.80	+2.17
6.	6.	↓ 5.	Hazelcast	Key-value, Multi-model	9.37	+0.19	+0.96
7.	7.	↑ 8.	Ehcache	Key-value	7.49	+0.26	+1.21
8.	8.	↓ 7.	Aerospike	Key-value, Multi-model	5.77	+0.86	-0.89
9.	9.	↑ 10.	Riak KV	Key-value	5.40	+0.82	+0.40
10.	10.	↑ 11.	Ignite	Multi-model	4.93	+0.54	+0.06
11.	11.	↓ 9.	ArangoDB	Multi-model	4.92	+0.53	-0.47
12.	12.	12.	OrientDB	Multi-model	4.45	+0.26	-0.37
13.	13.	13.	Oracle NoSQL	Multi-model	4.31	+0.61	+0.09
14.	14.	↑ 17.	RocksDB	Key-value	3.58	+0.49	+0.72
15.	15.	↓ 14.	InterSystems Caché	Multi-model	3.24	+0.34	-0.22
16.	↑ 17.	↓ 15.	Oracle Berkeley DB	Multi-model	3.08	+0.44	-0.12
17.	↓ 16.	↑ 18.	Infinispan	Key-value	2.95	+0.20	+0.08
18.	↑ 19.	↓ 16.	LevelDB	Key-value	2.80	+0.44	-0.26

豆瓣所使用的的分布式 k-v 存储，其实也是基于 bitcask 模型，并对其进行了很多优化。目前纯粹基于 bitcask 模型的 k-v 并不是很多，所以你可以多去看看 rosedb 的代码，可以提出自己的意见建议，一起完善这个项目。

最后，附上相关项目地址：

minidb：https://github.com/roseduan/minidb
rosedb：https://github.com/roseduan/rosedb

参考资料：
https://riak.com/assets/bitcask-intro.pdf
https://medium.com/@arpitbhayani/bitcask-a-log-structured-fast-kv-store-c6c728a9536b

---

题图：from wallheaven.cc