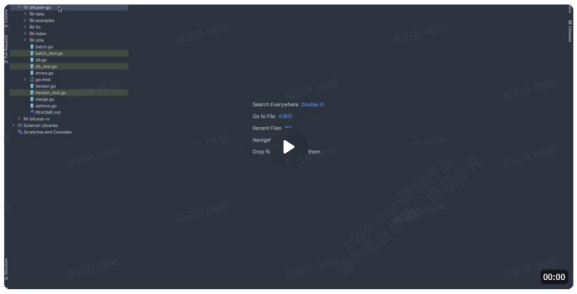
13 从零实现 KV 存储—内存索引优化

理论讲解

Go 编码



Go 语言代码修复，视频中遗漏了，seqNoFile 会一直追加写，所以加载后将这个文件删除掉：

```go
func (db *DB) loadSeqNo() error {
    fileName := filepath.Join(db.options.DirPath, data.SeqNoFileName)
    if _, err := os.Stat(fileName); os.IsNotExist(err) { return nil }

    seqNoFile, err := data.OpenSeqNoFile(db.options.DirPath)
    if err != nil { return err }
    record, _, err := seqNoFile.ReadLogRecord( offset: 0)
    seqNo, err := strconv.ParseUint(string(record.Value), base: 10, bitSize: 64)
    if err != nil { return err }
    db.seqNo = seqNo
    db.seqNoFileExists = true

    return os.Remove(fileName)
}
```

Rust 编码

多种索引数据结构

在最开始设计内存索引结构的时候，我们提到过，可以根据实际情况去实现不同的内存索引类型，我们设计了抽象的内存索引接口，一个的内存索引结构只需要实现这个抽象接口就可以接入了。

这里以一个简单的例子来向你展示如何来完成对不同索引结构的支持。

对于 Go 语言，可以使用自适应基数树：
https://github.com/plar/go-adaptive-radix-tree

自适应基数树可以看作是对前缀树的一个优化版本，如果子节点只有一个值，则会和父节点进行合并，减少空间占用。

> 这里是一篇专门介绍 ART 的论文：https://db.in.tum.de/~leis/papers/ART.pdf

而对于 Rust，我们可以使用较为常用的 crossbeam 库中的跳表来实现一个内存索引结构：

> https://crates.io/crates/crossbeam-skiplist

跳表是一个常用的内存有序的索引数据结构，LevelDB 和 RocksDB 中默认的 memtable 的数据结构就是跳表。

实现的逻辑也很简单，只需要填充我们的抽象索引的全部方法即可。

Go

```go
// Indexer 抽象索引接口，后续如果想要接入其他的数据结构，则直接实现这个接口即可
type Indexer interface {
    // Put 向索引中存储 key 对应的数据位置信息
    Put(key []byte, pos *data.LogRecordPos) bool

    // Get 根据 key 取出对应的索引位置信息
    Get(key []byte) *data.LogRecordPos

    // Delete 根据 key 删除对应的索引位置信息
    Delete(key []byte) bool

    // Size 索引中的数据量
    Size() int

    // Iterator 索引迭代器
    Iterator(reverse bool) Iterator
}
```

Rust

```rust
/// Indexer 抽象索引接口，后续如果想要接入其他的数据结构，则直接实现这个接口即可
pub trait Indexer: Sync + Send {
    /// 向索引中存储 key 对应的数据位置信息
    fn put(&self, key: Vec<u8>, pos: LogRecordPos) -> bool;

    /// 根据 key 取出对应的索引位置信息
    fn get(&self, key: Vec<u8>) -> Option<LogRecordPos>;

    /// 根据 key 删除对应的索引位置信息
    fn delete(&self, key: Vec<u8>) -> bool;

    /// 获取索引存储的所有的 key
    fn list_keys(&self) -> Result<Vec<Bytes>>;

    /// 返回索引迭代器
    fn iterator(&self, options: IteratorOptions) -> Box<dyn IndexIterator>;
}
```

索引存储空间优化

从 bitcask 论文中可以得知，其实这个存储模型最大的特点是所有的索引都只能在内存中维护，这样的特性带来了一个很大的好处，那就是只需要从内存中就能够直接获取到数据的索引信息，然后只通过一次磁盘 IO 操作就可以拿到数据了。

但是拥有这个好处的同时也带来了一个缺陷，那便是我们的存储引擎能维护多少索引，完全取决于内存容量，也就是说数据库能存储的 key+索引的数据量受到了内存容量的限制。

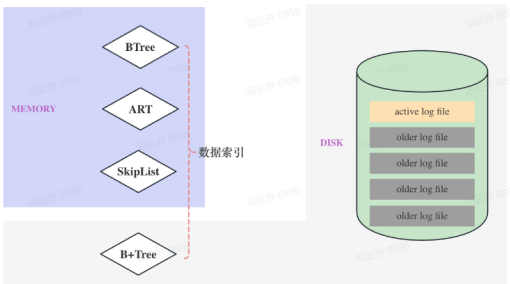
规避这个问题的方案其实有很多，我们可以选择一个节省内存空间的数据结构作为索引，或者直接将索引存储到磁盘当中，例如使用持久化的 B+ 树作为索引。

当然天下没有免费的午餐，如果将索引存储到了磁盘当中，好处是可以节省内存空间，突破存储引擎的数据量受内存容量的限制，但是随之而来的缺点也很明显，那就是读写性能会随之降低，因为需要从磁盘上获取索引，然后再去磁盘数据文件中获取 value。

如果有现成的 B+ 树实现的话，可以直接拿来使用，只需要实现我们的抽象索引接口就可以了。

对于 Go 语言，我们可以使用 boltdb 这个库，这是一个标准的 B+ 树实现，并且也是 Go 生态中较为知名的一个 KV 库；Rust 也可以使用对应的 B+ 树来作为索引结构。

> https://crates.io/crates/jammdb



对 WriteBatch 的影响

如果是 B+ 树索引，那么将不会从数据文件中加载索引，同时也拿不到最新的事务序列号，因为事务序列号是依次加载数据文件中的数据构建索引时获取到的。

解决这个问题的办法大致有如下几种：

- 还是加载数据文件，并且获取到最新的事务序列号
- 在 B+ 树索引模式下，直接禁用 WriteBatch 的功能
- 数据库 Close 的时候，将最新的序列号记录到一个文件当中，启动的时候，直接从这个文件中获取
  - 但是因为一些异常情况（比如调用者没有 Close），导致并没有将序列号记录到这个文件，可以考虑禁用 WriteBatch 的功能

扩展内容—索引锁粒度优化

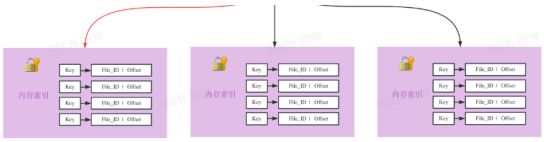
> 此部分内容将不做代码演示，学有余力的同学可以自己去实现。

在我们之前的设计中，内存中只有一个索引的结构，所有的写入和读取都会竞争这个索引数据结构的锁，在高并发的场景下，这可能会是一个性能瓶颈。

我们可以将这个锁的粒度减小，使用多个索引结构，然后将 key 通过 hash 取模映射到不同的索引数据结构中。

这样一来，只有映射到相同索引之中的 key 才会竞争同一把锁，这避免了去维护一个全局索引锁，大大减少了锁的粒度，并发性能能够得到提升。

具体分多少个索引，可以提供一个配置项，让调用者决定。



对迭代器的影响

这样修改之后，我们原来的迭代器就不能够直接使用了，因为它只代表了单个索引结构的数据，我们如何才能够取出多个索引结构的迭代器呢？

索引按照上述方法分区之后，每个索引内部的迭代器我们可以获取到，并且单个索引数据结构内部的数据是有序的。也就是说我们需要将多个有序的子分区合并起来，得到一个最终的迭代器。

我们可以建立一个最小堆，初始化的情况下，取出每个迭代器的第一个元素。

用户取出数据，我们就将堆顶的元素取出，因为堆顶的元素一定是最小的，然后再将对应的索引结构中再取出一个数据递补上去，插入到堆中。

如此循环往复，直到堆中的元素取完，则遍历结束。