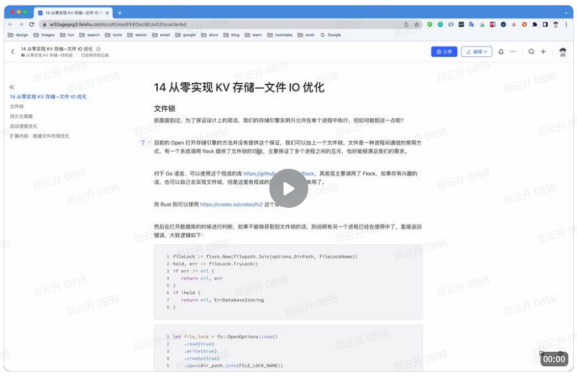
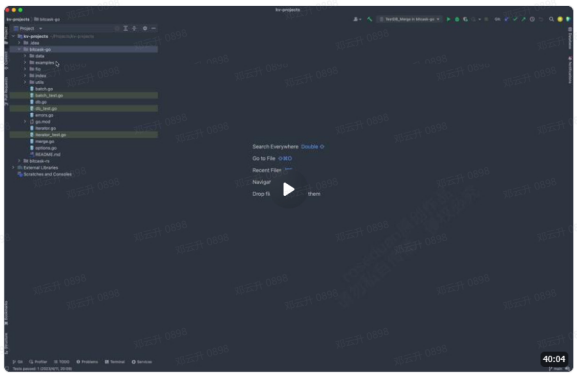
14 从零实现 KV 存储—文件 IO 优化

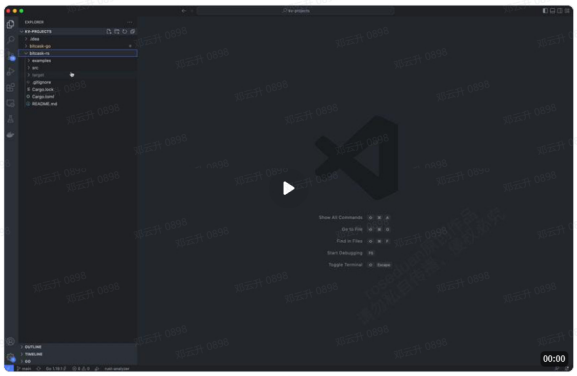
理论讲解



Go 编码部分



Rust 编码部分



文件锁

前面提到过，为了保证设计上的简洁，我们的存储引擎实例只允许在单个进程中执行，但如何做到这一点呢？

目前的 Open 打开存储引擎的方法并没有提供这个保证，我们可以加上一个文件锁，文件是一种进程间通信的常用方式，有一个系统调用 flock 提供了文件锁的功能，主要保证了多个进程之间的互斥，恰好能够满足我们的需求。

对于 Go 语言，可以使用这个现成的库 https://github.com/gofrs/flock，其底层主要调用了 Flock，如果你有兴趣的话，也可以自己去实现文件锁，但是这里有现成的我们就直接拿过来用了。

而 Rust 则可以使用 https://crates.io/crates/fs2 这个库。

然后在打开数据库的时候进行判断，如果不能够获取到文件锁的话，则说明有另一个进程已经在使用中了，直接返回错误，大致逻辑如下：

```go
fileLock := flock.New(filepath.Join(options.DirPath, fileLockName))
hold, err := fileLock.TryLock()
if err != nil {
    return nil, err
}
if !hold {
    return nil, ErrDatabaseIsUsing
}
```

```rust
let file_lock = fs::OpenOptions::new()
    .read(true)
    .write(true)
    .create(true)
    .open(dir_path.join(FILE_LOCK_NAME))
    .unwrap();
if let Err(_) = file_lock.try_lock_exclusive() {
    return Err(Errors::DatabaseIsUsing);
}
```

需要注意的是，最后在 Close 方法中加上释放文件锁的逻辑。

持久化策略

之前提供了一个 SyncWrites 的选项，主要是控制是否每条数据都持久化到磁盘，目前的默认值是 false，也就是将持久化这个行为交给了操作系统进行调度。

但是在实际环境中，我们可以再提供另一个选项，用户可以设置积累到多少个字节之后，再进行一次持久化。这相当于提供了一个折中的方案，相较于之前的要么每条都持久化，要么完全交给操作系统，这样能够让用户自己灵活配置。

具体的做法也比较简单，可以在打开数据库的时候，增加一个配置项 BytesPerSync，每次写数据的时候，都记录一下累计写了多少个字节，如果累计值达到了 BytesPerSync，则进行持久化。

```go
// 根据用户配置决定是否持久化
var needSync = db.options.SyncWrites
if !needSync && db.options.BytesPerSync > 0 && db.bytesWrite >= db.options.BytesPerSync {
    needSync = true
}
if needSync {
    if err := db.activeFile.Sync(); err != nil {
        return nil, err
    }
    if db.bytesWrite > 0 {
        db.bytesWrite = 0
    }
}
```

```rust
let bytes = self
    .bytes_write
    .fetch_add(enc_record.len(), Ordering::SeqCst);
// 根据配置项决定是否持久化
let mut need_sync = self.options.sync_writes;
if !need_sync && self.options.bytes_per_sync >= bytes + enc_record.len() {
    need_sync = true;
}
if need_sync {
    active_file.sync()?;
    if bytes > 0 {
        self.bytes_write.store(0, Ordering::SeqCst);
    }
}
```

最后需要将维护的累计值清零，开启下一次的统计。

启动速度优化

我们知道，bitcask 在启动的时候，会全量加载所有的数据并构建内存索引，在数据量较大的情况下，这个构建的过程可能会非常漫长。这带来的问题是重启时间过长，例如数据库崩溃之后需要立即启动，并马上接受用户请求，这时候用户等待的时间就会比较长。

在之前的默认文件 IO 下，会涉及到和内核进行交互，执行文件系统调用接口，操作系统会将内核态的数据拷贝到用户态。

所以我们可以使用内存文件映射（MMap） IO 来进行启动的加速，mmap 指的是将磁盘文件映射到内存地址空间，操作系统在读文件的时候，会触发缺页中断，将数据加载到内存中，这个内存是用户可见的，相较于传统的文件 IO，避免了从内核态到用户态的数据拷贝。

mmap 有一些现成的库我们可以直接使用：

- Go：https://pkg.go.dev/golang.org/x/exp/mmap
- Rust：https://github.com/RazrFalcon/memmap2-rs

> Rust crate 地址：https://crates.io/crates/memmap2

我们可以提供一个配置项，让用户决定是否在启动的时候使用 MMap，如果是的话，则打开数据文件的时候，我们将按照 MMap IO 的方式初始化 IOManager，加载索引时读数据都会使用 mmap。

加载索引完成后，我们需要重置我们的 IOManager，因为 MMap 只是用于数据库启动，启动完成之后，要将 IOManager 切换到原来的 IO 类型。

扩展内容：数据文件布局优化

> 此部分不做代码演示，学有余力的同学可以自行研究。

参考 LevelDB 或者 RocksDB 的 WAL 文件的格式，将数据文件的组织形式改为 block（32KB） 的方式，加速数据读写的效率。

https://leveldb-handbook.readthedocs.io/zh/latest/journal.html

https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log-%28WAL%29