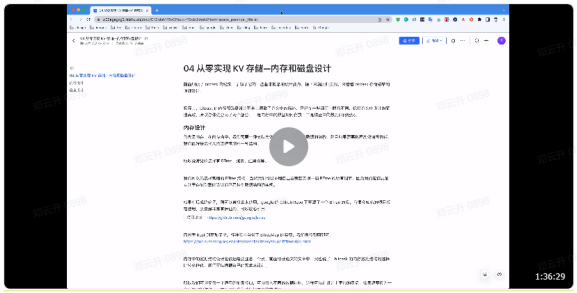
04 从零实现 KV 存储—内存和磁盘设计



前面细读了 bitcask 的论文，了解了它的一些基本概念和设计要点，接下来就回归实践，来看看 bitcask 存储引擎的详细设计。

实际上，bitcask 的内存和磁盘设计基本上遵循了论文中的描述，只是在一些细节中略有不同，保持论文中设计的简洁高效，所以总体就分为了两个部分，一是内存中的数据如何存放，二是磁盘中的数据如何组织。

内存设计

首先是内存，在内存当中，我们需要一种支持高效插入、读取、删除数据的结构，并且如果需要数据高效遍历的话，我们最好是选择天然支持有序的一种结构。

所以说常见的选择有 BTree、跳表、红黑树等。

我们可以先选择常用的 BTree 结构，当然我们可以不用自己去完整实现一遍 BTree 的所有细节，因为我们应该更加专注于存储引擎的设计而不是某个数据结构的实现。

如果有现成的轮子，则可以直接拿来使用，google 的 Github Repo 下开源了一个 BTree 的库，有很多知名的项目都在使用，质量是非常有保证的，可以放心引用。

> 项目地址：https://github.com/google/btree

而对于 Rust 则更加简单，标准库中自带了 BTreeMap 的实现，我们直接引用即可。
https://doc.rust-lang.org/stable/std/collections/struct.BTreeMap.html

内存中的数据结构设计应该还需要注意一个点，前面细读论文的文章中，我也说了，bitcask 的内存数据结构的选择比较多样化，我们可以根据自己的需求来设计。

所以我们可以提供一个通用的抽象接口，可以接入不同的数据结构，这样可以在设计上更加的灵活。如果想要接入一个新的数据结构，只需要实现我们抽象接口中的方法即可。

通用接口的定义大致如下：
Go

```go
// Indexer 通用索引接口
type Indexer interface {
        Put(key []byte, pos *data.LogRecordPos) bool
        Get(key []byte) *data.LogRecordPos
        Delete(key []byte) bool
}
```

Rust

```rust
// 通用索引接口
pub trait Indexer: Sync + Send {
    fn put(&self, key: Vec<u8>, pos: LogRecordPos) -> bool;
    fn delete(&self, key: Vec<u8>) -> bool;
    fn get(&self, key: Vec<u8>) -> Option<LogRecordPos>;
}
```

后续我将以自适应基数树（Adaptive Radix Tree）和跳表（SkipList）作为另一个内存索引，来展示如何支持多种内存索引结构，这样你可以根据自己的需要去实现其他的数据结构。

磁盘设计

内存设计完了，再来看看磁盘。

我们可以将标准文件操作 API 例如 read、write、close 等方法进行简单的封装，然后数据在磁盘上的读写可以使用这些标准的文件 API，我们可以加一个目录 fio，专门存放关于文件 IO 操作相关的代码。

目前我们只支持标准的系统文件 IO，但是如果后面有其他的 IO 类型，例如 MMap 内存映射，或者自己写一层文件 IO 系统，都可以进行接入。

因此我们可以定义一个 IOManager 接口，将 IO 操作的接口进行抽象，方便接入不同的 IO 类型。
Go

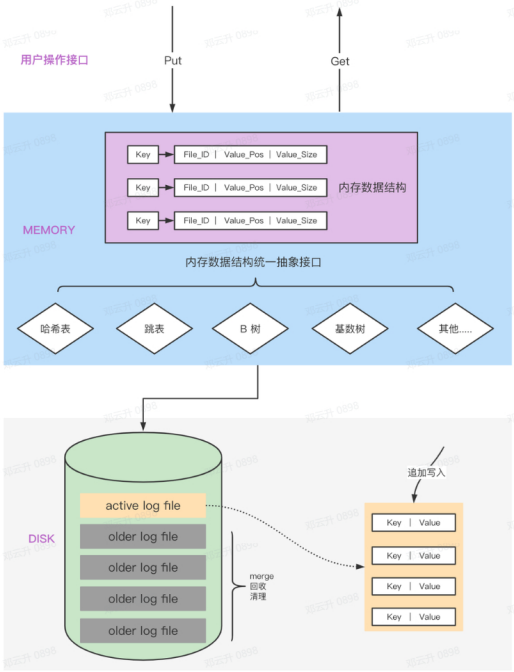
```go
type IOManager interface {
        Read([]byte, int64) (int, error)
        Write([]byte) (int, error)
        Sync() error
        Close() error
}
```

Rust

```rust
// 通用 IO 管理接口
pub trait IOManager: Sync + Send {
    fn write(&self, buf: &[u8]) -> Result<usize>;
    fn read(&self, buf: &mut [u8], offset: u64) -> Result<usize>;
    fn sync(&self) -> Result<()>;
}
```

然后对于数据文件的操作，例如新增、删除数据文件，从文件中读取记录，可以增加一个目录 data 来存放，表示数据文件、数据项等内容。

内存和磁盘都设计好之后，我们的 bitcask KV 存储引擎的架构就很清晰了，如下图：



可以看到我们的 bitcask 存储引擎总体来说是很简单的，架构比较简洁，但是在实现的过程当中，还是有很多的细节需要处理，后续章节将进入我们的数据读写流程等实践部分。

13 人点赞

全文评论

eot 3月30日 12:47
看完了之后有几个问题：

eot 3月30日 12:49
@eot 一：btree get 函数那里，为啥不用上读锁？难道是因为那个库支持并发读所以就不用上读锁了么？

eot 3月30日 12:56
@eot 二：如果索引的大小多到内存放不下的时候该怎么办？

roseduan 4月4日 17:08
@eot 是的

roseduan 4月4日 17:08
@eot 后面会讲

小何 4月7日 08:11
这节视频总是说加载中，不能播放，手机上，是哪的问题呢？

Lecarus 4月7日 19:21
@小何 网络问题，用电脑看，手机 ipad 看会很卡很慢的

小何 4月7日 22:43
@Lecarus 好的，我试试，谢谢！

陈桂铭 4月9日 23:44
go 语言里面的
file_io.go 里面的
func NewFileIOManager(fileName string) (*FileIO, error) {
fd, err := os.OpenFile(
fileName,
os.O_CREATE|os.O_RDWR|os.O_APPEND,
DataFilePerm,
)
if err != nil {
return nil, err
}
return &FileIO{fd: fd}, nil
}

为什么要在 io_manager.go 里面的这里调用啊。
func NewIOManager(fileName string) (IOManager, error) {
展开

roseduan 4月10日 08:50
@陈桂铭 后面会接入另一种 IO 类型

李东霜 4月11日 21:49 （编辑过）
google 的 btree 库实现的是 B 树还是 B+树啊，非叶节点带有数据吗

roseduan 4月11日 22:25
@李东霜 B 树，纯内存的

宋凌逸 5月5日 22:23
测试会报错：The process cannot access the file because it is being used by another process。文件 fio 要先关闭再被删除。

输入评论