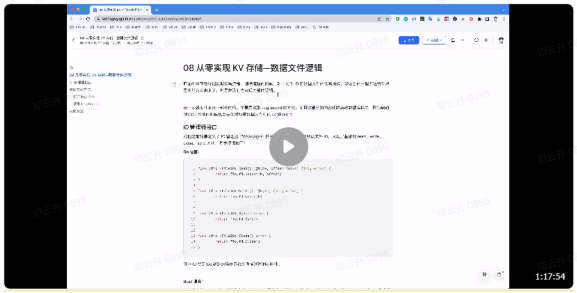
08 从零实现 KV 存储—数据文件逻辑



前面的章节在介绍数据读写流程、删除流程的时候，对于文件 IO 和数据文件的读写操作，涉及到的一些方法我们只是将其定义出来了，但是并没有去实现内部的逻辑。

这一节就来补全这一部分内容，主要是读取 LogRecord 的方法，而且这部分的内容还跟启动数据库相关，因为启动 bitcask 引擎的时候也会去读取所有数据文件中的 LogRecord。

IO 管理器接口

我们之前简单定义了 IO 管理器 `IOManager` 的接口，然后调用系统默认文件 IO，实现了基础的 read、write、close、sync 方法，基本逻辑如下：

Go 语言：

```go
func (fio *FileIO) Read(b []byte, offset int64) (int, error) {
	return fio.fd.ReadAt(b, offset)
}

func (fio *FileIO) Write(b []byte) (int, error) {
	return fio.fd.Write(b)
}

func (fio *FileIO) Sync() error {
	return fio.fd.Sync()
}

func (fio *FileIO) Close() error {
	return fio.fd.Close()
}
```

其中 fd 就是 Go 封装的操作系统文件描述符的结构体。

Rust 语言：

需要注意的是 Rust 由于所有权机制的存在，变量在离开其作用域的时候，会自动释放相关的资源，所以 rust 的文件中并没有 close 方法：

```rust
impl IOManager for FileIO {
    fn write(&self, buf: &[u8]) -> Result<usize> {
        let mut write_guard = self.fd.write();
        let n_bytes = match write_guard.write(buf) {
            Ok(n) => n,
            Err(e) => {
                error!("write data file err: {}", e);
                panic!("failed to write to data file");
            },
        };
        Ok(n_bytes)
    }

    fn read(&self, buf: &mut [u8], offset: u64) -> Result<usize> {
        let read_guard = self.fd.read();
        let n_bytes = match read_guard.read_at(buf, offset) {
            Ok(n) => n,
            Err(e) => {
                error!("read data file err: {}", e);
                panic!("failed to read from data file");
            }
        };
        Ok(n_bytes)
    }

    fn sync(&self) -> Result<()> {
        let read_guard = self.fd.read();
        if let Err(e) = read_guard.sync_all() {
            error!("sync data file err: {}", e);
            panic!("failed to sync data file");
        }
        Ok(())
    }
}
```

至于为什么要封装一层，并且提供了一个抽象的接口，主要是为了屏蔽上层的调用者，并且方便我们后续接入不同的 IO 类型，我们目前只实现了基础的文件 IO 接口，后续可以再实现其他的 IO 类型，比如 `MMap`，或者自定义 IO 系统等等。

需要注意我们在打开文件的时候，需要加上一个 O_APPEND 的选项，因为我们的数据文件是只允许追加进行写入的，一般的编程语言的文件操作也都会提供 `O_APPEND` 的选项，打开文件的的时候设置这个 flag 就可以了。

```go
fd, err := os.OpenFile(
    fileName,
    os.O_CREATE|os.O_RDWR|os.O_APPEND,
    DataFilePerm,
)
```

```rust
match OpenOptions::new()
    .create(true)
    .read(true)
    .write(true)
    .append(true)
    .open(file_name)
```

然后我们需要在 IOManager 的抽象接口中新增打开对应 IO 的方法，由于目前只实现了标准文件 IO，因此只需要调用之前定义的 `NewFileIOManager` 方法。

数据文件方法

这里主要需要补全两个方法：

- 打开数据文件（`OpenDataFile`）
- 从数据文件中读取 LogRecord（`ReadLogRecord`）

还有其他的方法，例如向数据文件中写入数据，以及数据文件的 Sync、Close 方法，可以直接调用 IOManager 的接口，都比较简单。

打开数据文件

打开数据文件，需要传入文件目录的路径，以及对应的文件 id，根据路径和 id，并且拼上数据文件的后缀名 `.data`，构造出完整的数据文件名称。

然后调用 IOManager 的创建方法打开文件，拿到 IOManager 的对象。

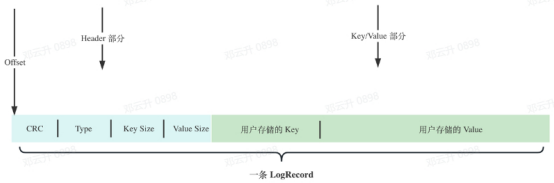
最后填充数据文件的结构体，数据文件的结构体定义大致如下：

```go
type DataFile struct {
    WriteOff  int64         // 文件写偏移，记录文件写到什么位置了，下次 Write 时使用
    FileId    uint32        // 文件编号
    IoManager fio.IOManager // 文件操作接口
}
```

```rust
// 数据文件
pub struct DataFile {
    file_id: Arc<RwLock<u32>>,          // 数据文件id
    write_off: Arc<RwLock<u64>>,        // 当前写偏移，记录该数据文件写到哪个位置了
    io_manager: Box<dyn fio::IOManager>, // IO 管理接口
}
```

读取 LogRecord

这是一个比较重要的方法，因为读取数据、打开 bitcask 实例的时候都会用到，这个方法的目的很简单，就是根据一个偏移 offset 读取指定位置的 LogRecord 信息。



上图是存储在数据文件当中的一条 LogRecord 的结构，我们可以将其分成两部分：

- 一是头部信息，存储了一些元数据，例如 crc 校验值、Type 类型、Key 的大小、Value 的大小
- 二是包含用户实际的 Key、Value 部分

这里需要注意的是，我们存储了用户实际的 Key/Value 之外，为什么还要存储它对应的大小 key size 和 value size?

这是因为用户传递过来的 key 和 value 的长度是不确定的，这样我们在读取对应数据的时候，并不知道应该读取多少个字节，所以可以加上一个对应的大小，我们先读取 header 部分的内容，这部分的长度是定长的，读取上来之后，我们就知道对应的 key/value 大小了，然后就可以再根据这个大小去读取用户实际存储的数据。

Header 部分的数据中，crc 占 4 字节，type 占一个字节，key size 和 value size 是变长的，从数据文件中读取的时候，我们会取最大的 header 字节数，反序列化的时候，如果解码 key size 和 value size 之后还有多余的字节，会自动忽略。

> 为什么要设计成变长的？主要是为了节省空间，如果 key size 是 u32 类型的话，如果不使用变长，将固定占据 4 字节，但是有的时候我们 key 的长度很小，例如长度为 5，那么只需要一个字节就够了。

读的时候，需要判断读取的偏移加上 LogRecord 的最大头部字节数，是不是超过了文件的大小，针对这个 case，需要特殊处理一下。

> 在前面的删除流程的文章中已经介绍过了，这里不再赘述。
>
> Rust 中由于 BytesMut 在读取的时候不会报 EOF 错误，所以可以不用处理这个 case。

LogRecord 的头信息，读出来之后仍然是二进制编码的，我们需要将其解压缩，得到具体的头信息，目前主要包含四个字段

- CRC：数据校验值
- Type：LogRecord 的类型
- Key Size：key 的长度
- Value Size：value 的长度

拿到 header 之后，如果判断到 key size 和 value size 均为 0，则说明读取到了文件的末尾，我们直接返回一个 EOF 的错误。

否则，再根据 header 中的 key 和 value 的长度信息，判断其值是否大于 0，如果是的话，则说明存在 key 或者 value。

我们就将该偏移 offset 加上 keySize 和 valueSize 的总和，读出一个字节数组，这其中就是实际的 Key/Value 数据，填充到 LogRecord 结构体中。

最后，需要根据读出的信息，获取到其对应的校验值 CRC，判断和 header 中的 CRC 是否相等，只有完全相等，才说明这是一条完整有效的数据，否则说明数据可能被破坏了。

其他方法

这里主要是 write、close、sync 方法。

这几个方法其实都是直接调用底层封装的 IOManager 中对应的方法即可，例如 Close 方法，如下：

```go
func (df *DataFile) Close() error {
    return df.IoManager.Close()
}
```

只是 write 方法，在调用之后，需要更新我们自己维护的 WriteOff 字段，表示当前写到哪个位置了，内存索引中需要保存这个信息，大致逻辑如下：

```go
func (df *DataFile) Write(buf []byte) error {
    nBytes, err := df.IoManager.Write(buf)
    if err != nil {
        return err
    }
    df.WriteOff += int64(nBytes)
    return nil
}
```

8 人点赞

输入评论