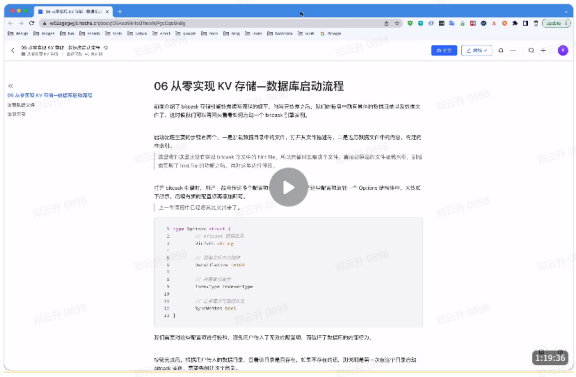
# 06 从零实现 KV 存储—数据库启动流程




前面介绍了 bitcask 存储引擎数据读写流程的细节，当写完数据之后，我们的磁盘中就有具体的数据目录以及数据文件了，这时候我们可以再回头看看如何启动一个 bitcask 引擎实例。

启动流程主要的步骤有两个，一是加载数据目录中的文件，打开其文件描述符，二是遍历数据文件中的内容，构建内存索引。

> 这里我们这里还没有实现 bitcask 论文中的 hint file，所以先暂时忽略这个文件，直接读原始的文件加载索引，到后面实现了 hint file 的功能之后，再对这里进行修改。

打开 bitcask 引擎时，用户一般会传递多个配置项进来，我们可以把这些配置项放到一个 Options 结构体中，大致如下所示，后续有新的配置项再添加即可。

> 上一节课程中已经将其定义出来了。

```
type Options struct {
        // bitcask 数据目录
        DirPath string

        // 数据文件大小阈值
        DataFileSize int64

        // 内存索引类型
        IndexType IndexerType

        // 是否每次写都持久化
        SyncWrites bool
}
```

我们需要对这些配置项进行校验，避免用户传入了无效的配置项，而破坏了数据库的内部行为。

校验完成后，根据用户传入的数据目录，查看该目录是否存在，如果不存在的话，则说明是第一次在这个目录启动 bitcask 实例，需要先创建这个目录。

我们需要定义一个表示 bitcask 实例的结构体，里面存放索引、数据文件等内容，在启动的时候，需要初始化这个结构体。

我们可以把这个结构体定义为 DB，其大致结构如下：

> Rust 中由于调用 crate 路径的问题，文件名已经叫 db 了，所以这个结构体可以命名为 Engine，意为 bitcask 存储引擎。

```
type DB struct {
        options     Options                 // 打开 bitcask 时的配置选项
        activeFile *data.DataFile           // 当前活跃的数据文件，用于追加写
        olderFiles map[uint32]*data.DataFile // 旧的数据文件，不能写，只能用于读
        index      index.Indexer            // 数据内存索引
        mu         *sync.RWMutex
}
```

```
/// bitcask 存储引擎实例结构体
pub struct Engine {
    options: Arc<Options>,
    active_file: Arc<RwLock<DataFile>>, // 当前活跃数据文件
    older_files: Arc<RwLock<HashMap<u32, DataFile>>>, // 旧的数据文件
    index: Box<dyn index::Indexer>,     // 数据内存索引
}
```

DB 结构体初始化之后，就到了启动流程最重要的两个步骤。

## 加载数据文件

首先是加载数据文件，我们可以定义一个方法 `loadDataFiles`，全部写到这个方法里面。

我们先打开用户传递的数据目录，如果判断到这个目录下没有文件的话，表示是一个空的数据库，则直接返回。

否则，加载这个目录下的数据文件，我们可以约定一个 bitcask 存储引擎的数据文件的文件扩展名，然后加载数据文件的时候，可以直接加载拥有这个后缀名的文件。

我们可以将这个后缀名定为 .data，表示的是数据文件。

所以我们直接遍历查看这个目录下的文件，然后找出所有以 .data 结尾的文件，并且取出文件的 id，文件 id 我们可以以文件名称来标识，当有很多数据存在的时候，我们将这个 id 递增，作为数据文件的名称，大致如下图所示：

```
bitcask ll
total 814264
-rw-r--r--  1 roseduan  wheel    64M Dec 25 12:04 000000000.data
-rw-r--r--  1 roseduan  wheel    64M Dec 25 12:04 000000001.data
-rw-r--r--  1 roseduan  wheel    64M Dec 25 12:04 000000002.data
-rw-r--r--  1 roseduan  wheel    64M Dec 25 12:04 000000003.data
-rw-r--r--  1 roseduan  wheel    64M Dec 25 12:04 000000004.data
```

```
-rw-r--r--  1 roseduan  wheel    64M Dec 25 12:04 000000005.data
-rw-r--r--  1 roseduan  wheel    10M Dec 25 12:04 000000006.data
```

取出所有的文件 id 之后，我们将其遍历，然后构造对应的文件路径+文件名，再调用 DataFile 的方法，打开对应的文件即可。
需要注意的是，这里我们需要标识其中最大的 id，因为最大的 id 所代表的数据文件就是当前活跃文件。

## 加载索引

有了数据文件之后，接下来就是构建索引了。

这里需要注意的是，我们必须依次从小到大遍历每个文件 id，其对应的逻辑就是将数据文件从旧到新进行加载，这个顺序一定不能乱，因为数据是一直追加写入的，最新的数据一定在更靠前的位置。

我们可以对文件 id 进行排序，然后只需要遍历每个文件 id，根据 id 找到其对应的数据文件，这里可以写一个 for 循环，遍历这个数据文件中的所有记录。

这里我们可以定义一个 offset 变量，表示读取到当前文件的哪个位置了。然后直接调用数据文件的读取 LogRecord 方法，拿到 LogRecord 信息即可。

最后再根据当前遍历的文件 id，以及遍历的位置 offset，构造出内存索引位置信息 LogRecordPos，并且将其存储到内存索引当中。

基本的逻辑大致如下：

```rust
fn load_index_from_data_files(&self) -> Result<()> {
    // 没有任何数据文件，直接返回
    if self.file_ids.len() == 0 {
        return Ok(());
    }

    let active_file = self.active_file.read();
    let older_files = self.older_files.read();
    for (i, file_id) in self.file_ids.iter().enumerate() {
        let mut offset = 0;

        loop {
            let log_record_res = match *file_id == active_file.get_file_id() {
                true => active_file.read_log_record(offset),
                false => {
                    let data_file = older_files.get(file_id).unwrap();
                    data_file.read_log_record(offset)
                }
            };

            let (log_record, size) = match log_record_res {
                Ok(result) => (result.record, result.size),
                Err(e) => {
                    if e == Errors::DataFileReadEOF {
                        break;
                    }
                    return Err(e);
                }
            };

            // 构造索引并存储
            let pos = LogRecordPos {
                fid: *file_id,
                offset,
            };
            match log_record.rec_type {
                LogRecordType::NORMAL => self.index.put(log_record.key.to_vec(), pos),
                LogRecordType::DELETED => self.index.delete(log_record.key.to_vec()),
            };

            offset += size as u64;
        }

        // 设置最后一个文件的 offset
        if i == self.file_ids.len() - 1 {
            active_file.set_write_off(offset);
        }
    }
    Ok(())
}
```