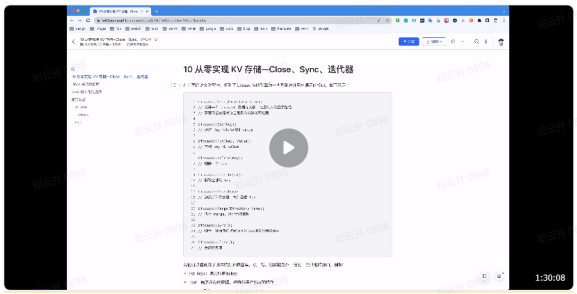
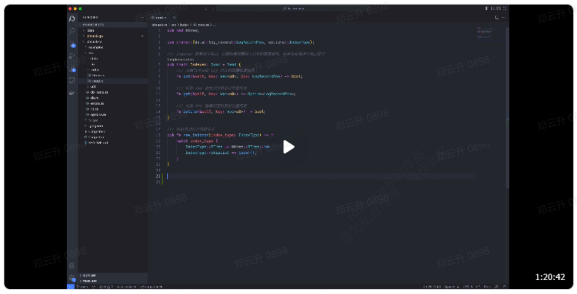
10 从零实现 KV 存储—Close、Sync、迭代器

理论讲解+Go 编码部分：



Rust 编码部分：



在前面的论文讲解中，提到了 bitcask 存储引擎的一些基础对外用户调用的接口，如下所示：

```
bitcask::Open(Directory Name);
// 打开一个 bitcask 数据库实例，使用传入的目录路径
// 需要保证进程对该目录具有可读可写权限

bitcask::Get(Key);
// 通过 Key 获取存储的 value

bitcask::Put(Key, Value);
// 存储 key 和 value

bitcask::Delete(Key);
// 删除一个 key

bitcask::List_keys();
// 获取全部的 key

bitcask::Fold(Fun);
// 遍历所有的数据，执行函数 Fun

bitcask::Merge(Directory Name);
// 执行 merge，清理无效数据

bitcask::Sync();
// 刷盘，将所有内核缓冲区的写入持久化到磁盘中

bitcask::Close();
// 关闭数据库
```

我们可以看到除了基本的打开数据库，读、写、删数据之外，还有一些其他的接口，例如：

- list_keys：遍历所有的 key
- flod：遍历所有的数据，并执行用户指定的操作
- sync：刷盘
- close：关闭数据库

其中 close、sync 这两个接口都比较简单，而关于遍历数据的接口，会涉及到我们的迭代器，在这一章中就来将它们补全，Merge 方法会专门在后面的章节中讲述。

Close 关闭数据库

close 方法的主要功能是在数据库使用完毕的时候，清理或释放相关的资源。

目前来说就是将打开的文件描述符关闭，一是需要将当前活跃文件关闭；二是将旧的数据文件关闭。当然在后续的过程中，也有可能会清理或释放其他的资源，都可以加到这个 Close 方法的逻辑中。

需要注意的是在 Rust 中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，所以我们并不需要显式的关闭文件，rust 中的 File 也没有提供这样的方法，所以这里的逻辑暂时和 Sync 一样，只需要持久化当前活跃文件。

Sync 持久化数据库

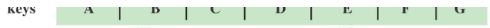
持久化主要负责将数据文件在缓冲区的内容刷到磁盘，能够保证数据不丢失。

只需要调用 IOManager 提供的 Sync 方法，并且只针对活跃文件即可，因为活跃文件写满之后，转换为旧的数据文件时，已经将其持久化过了，所以这里并不需要持久化，它所有内容一定是安全的存储在磁盘上的。

遍历数据

接下来的两个方法 list_keys 和 fold 其实比较类似，因为都需要对 key 进行遍历，而 fold 方法还需要取出对应的 value。

实现起来也比较简单，由于我们的 key 信息全部都保存在内存当中，所以直接从内存索引中取出全部的数据即可。



由于索引类型可能有多种，我们可以定义一个抽象的迭代器接口，让每一个具体的索引去实现这个迭代器，然后我们只需要调用这个迭代器获取索引中的数据。

Go

索引迭代器接口的定义大致如下：

```go
type Iterator interface {
    Rewind()
    Seek(key []byte)
    Next()
    Valid() bool
    Key() []byte
    Value() *data.LogRecordPos
    Close()
}
```

接口的几个方法简单解释：

- Rewind：重新回到迭代器的起点，即第一个数据
- Seek：根据传入的 key 查找到第一个大于（或小于）等于的目标 key，根据从这个 key 开始遍历
- Next：跳转到下一个 key
- Valid：是否有效，即是否已经遍历完了所有的 key，用于退出遍历
- Key：当前遍历位置的 Key 数据
- Value：当前遍历位置的 Value 数据
- Close：关闭迭代器，释放相应资源

Rust

在 Rust 中，对迭代器的定义可以更简单一点，如下所示，其中 rewind 和 seek 的定义和 Go 语言的描述一致，next 方法主要返回了对应的 key 和内存索引位置信息 LogRecordPos，这是一个 Option 包裹的值，如果它返回 None，则说明所有的数据都遍历完成了。

```rust
/// 抽象索引迭代器
pub trait IndexIterator: Sync + Send {
    /// Rewind 重新回到迭代器的起点，即第一个数据
    fn rewind(&mut self);

    /// Seek 根据传入的 key 查找到第一个大于（或小于）等于的目标 key，根据从这个 key 开始遍历
    fn seek(&mut self, key: Vec<u8>);

    /// Next 跳转到下一个 key，返回 None 则说明迭代完毕
    fn next(&mut self) -> Option<(&Vec<u8>, &LogRecordPos)>;
}
```

然后我们需要在对应的索引数据结构中，去实现这个接口定义的几个方法。

Iterator

索引的迭代器实现之后，我们可以在数据库层面增加一个迭代器，提供给用户使用，这样在遍历数据的时候，可以更加灵活的获取和控制遍历数据的流程。

用户可以传入一个 IteratorOptions 配置项，目前可以指定需要遍历的 Key 前缀，以及指定是否是反向遍历。

```go
type IteratorOptions struct {
        // 遍历前缀为指定值的 Key，默认为空
        Prefix []byte
        // 是否反向遍历，默认 false 是正向
        Reverse bool
}
```

这个结构体的方法和前面定义的索引迭代器接口基本一样，我们只需要调用索引迭代器的接口，并且在 Next 方法中加上对 Prefix 前缀的处理，以及在 Value 方法中加上从磁盘获取数据的逻辑。

ListKeys

针对 ListKeys 方法，因为只需要 Key，所以可以获取索引的迭代器接口，然后遍历获得所有的 Key 即可。

Fold

Fold 方法相较于 ListKeys，只是多了一个从磁盘获取 Value 的步骤，所以也是拿到索引的迭代器，然后遍历 Key 的时候，需要根据索引信息去磁盘中拿到对应的 value，然后再处理即可。

这里我们会定义将一个函数传进来，这个函数是用户自定义的，如果这个函数返回了 false，则遍历终止。