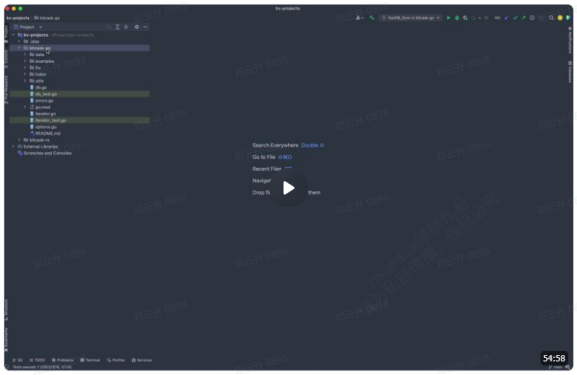
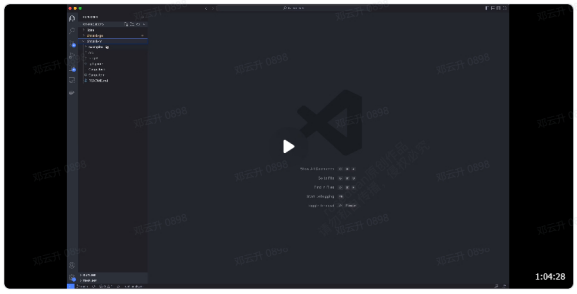
11 从零实现 KV 存储—WriteBatch 原子写

理论讲解

Go 编码部分



Rust 编码部分



事务的基本概念

事务的概念大家多少都接触过，一个事务表示的是一批操作要么全部成功，要么全部失败回滚，不会停留在中间状态，造成数据的不一致。

标准的事务定义有四个属性：A（Atomicity 原子性）C（Consistency 一致性）I（Isolation 隔离性）D（Durability 持久性）。

- 原子性：描述事务的不可分割性，一个事务中的操作要么全部执行完成，要么失败回滚
- 一致性：事务开始和结束后数据的完整性没有被破坏
- 隔离性：数据库可以支持多个并发执行的事务对数据进行修改和读取，隔离性可以防止多个事务执行时，由于交叉执行而导致数据的不一致
- 持久性：事务执行成功后，对数据的修改就是永久的

如何保证原子性?

原子性一般通过预写日志（WAL，Write Ahead Log）来实现，例如 Mysql 中，使用到了 undo log 来保存事务回滚的信息，当事务提交失败后，会利用 undo log 来将数据恢复到事务开始之前的状态。

如何实现隔离性?

隔离性是 ACID 四个属性当中最复杂、最难实现的，隔离性定义的标准几个隔离级别如下：

- 读未提交（存在的问题：脏读）
- 读提交（存在的问题：不可重复读）
- 可重复读（存在的问题：幻读）
- 串行化

并发控制的实现

隔离性主要控制多个并发执行事务的正确性，避免由于多个事务交叉执行，导致数据不一致。

两阶段锁（Two-Phase Locking）

在事务的执行过程当中，对需要操作的对象加锁，这样如果其他的事务也需要操作同一个对象时，将会等待另一个事务释放锁，或者直接因为未获取到锁而回滚，避免发生死锁。

在类 LSM 的存储引擎中，一般不会采用两阶段锁来实现事务，因为 LSM 存储模型中，数据本身具有多版本的特征，因此更倾向于使用基于多版本的并发控制。

多版本并发控制（MVCC）

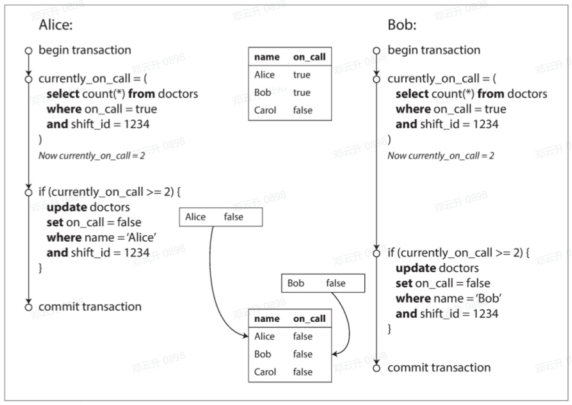
MVCC 理解起来其实并不难，在修改一条数据时，我们并不修改其原有的数据，而是增加一条新的数据，并标识其版本，这样可以实现读写事务之间互不阻塞，因为它们都在执行的过程当中维护了自己的数据版本。

主流的关系型数据库例如 MySQL、PostgreSQL、Oracle 都实现了 MVCC。

基于 MVCC 实现的事务隔离方式，一般叫做快照隔离（Snapshot Isolation，简称 SI），SI 并不是标准定义中事务的四种隔离级别，它的基本思路是每个事务都持有一个自己的快照，事务读数据时会基于事务开始时的一个快照，

其他事务的修改不会对读取有影响，当事务提交后，它的修改才会被其他的事务看到。

SI 基本上解决了脏读、不可重复读、幻读的问题，但是仍然存在写偏斜（Write Skew）的问题。



基于此，后面又有人提出了串行化快照隔离（Serializable Snapshot Isolation，简称 SSI），SI 只会检测写写冲突，而 SSI 则会对读取过的数据进行跟踪，并且在提交时进行冲突检测。

SSI 的大致实现思路：

写数据

- 将数据暂存到内存中，并记录每个 key 到一个集合中，便于冲突检测
- 提交事务
  - 加锁保证线程安全
  - 检测冲突，当前事务读取过的 key 是否被其他的事务修改过，如果是，则说明有冲突，事务放弃提交，回滚
  - 获取当前最新的事务序列号（序列号一般是全局递增的，每个事务都分配了一个序列号）
  - 将所有需要写入的数据，key 进行编码，加上事务序列号
  - 将数据批量写到存储引擎中，保证原子性和持久性
  - 写完后更新内存索引

读数据

- 从当前事务的数据集合中获取，如果获取到直接返回
- 未获取到，则 key+当前的事务序列号，从存储引擎中查找
- 将读过的数据记录下来，便于在提交事务时进行冲突检测

WriteBatch

前面介绍了事务隔离的概念及简单的实现，在我们的 bitcask 存储引擎的设计中，有一个很大的特点是我们需要将所有的 key 维护在内存中，如果在此基础上实现 MVCC，那么也会在内存中去维护所有的 key、位置索引、版本信息，那么这可能会造成内存容量的急剧膨胀。

鉴于此，为了兼顾大多数人能够学懂，我们在本课程中实现了一种更简单的事务，利用一个全局锁保证串行化，实现简单的满足 ACID 的事务。

对于学有余力的同学，如果想要实现基于 MVCC 的事务，除了上面提到的大致流程，还可以参考以下资料：

https://zhuanlan.zhihu.com/p/395229054

https://github.com/rfyiamcool/notes/blob/main/go_badger_transaction.md

https://www.infoq.cn/article/teJA7X43BO2alp6rLCWk

https://www.infoq.cn/article/KyZjpzySYHUYDJa2e1fS

https://www.infoq.cn/article/gaOh3me9PmJBiQFD2j15

https://catkang.github.io/2018/09/19/concurrency-control.html

https://dgraph.io/blog/post/badger-txn/

https://tech.ipalfish.com/blog/2020/03/26/isolation/

设计思路

之前我们增加了一个简单的存储和删除数据的接口 Put、Delete，如果我们在多次调用中，例如在一个循环中连续 Put 100 条数据，这一个批量的操作并不能保证原子性。

现在就来看看如何保证一个批量操作的原子性，也就是实现事务的功能，让这个批量操作要么全部成功，要么全部失败，而不会停留在中间状态，造成数据的不一致。

具体的实现方案如下：

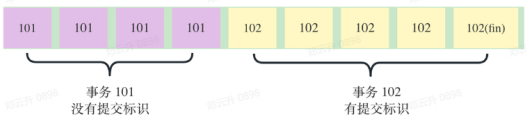
我们可以将用户的批量操作缓存起来，保存到一个内存数据结构中，然后提供一个提交的接口，可以叫做 Commit。Commit 意为提交事务，主要逻辑是将用户的批量操作全部写入到磁盘文件中，并更新内存索引。

正常情况下，批量操作全部写入到磁盘文件，然后直接更新内存索引。

但如果在写磁盘的过程中发生异常，例如系统崩溃了，进程异常 crash 了，我们的数据有可能只写了一部分。这样在数据库启动的时候，还是会识别到这一部分数据，并把它当做是有效的数据，这样就不能保证原子性了。

所以我们可以给这一批数据添加一个唯一标识，一般把它叫做序列号 Seq Number，你可以将其理解为事务 ID。这个 seq number 是全局递增的，每一个批次的数据在提交的时候都将获取一个 seq number，并且保证后面的 seq number 一定比前面获取的更大。

提交事务的时候，每一条日志记录 LogRecord 都有一个 seq number，并且写到数据文件中，然后我们可以在这一批次的最后增加一个标识事务完成的日志记录，此时数据文件状态大致如下：



如上图，假如有两个事务，序列号分别为 101 和 102，101 没有完成提交的标识，说明事务是不成功的；事务

102 在最后有一条完成提交的标识数据，说明事务是正常提交的。

在数据库启动的时候，如果判断到日志记录有序列号 seq number，那么我们先不直接更新内存索引，而是将它暂存起来，直到读到了一条标识事务完成的记录，说明事务是正常提交的，就可以将这一批数据都更新到内存索引中。

像事务 101，由于其没有标识事务完成的记录，所以这一批数据就不会应用到内存索引中，是对用户不可见的，这样就保证了这个操作的原子性。

> 注：没有完成提交的脏数据将会在后续的 merge 过程中被清理。

WriteBatch 代码

再来看看具体的代码实现。

我们可以定义一个 WriteBatch 结构体，字段大致如下：

```go
// WriteBatch 原子批量写数据，类似事务的功能
type WriteBatch struct {
    mu            *sync.RWMutex
    db            *DB
    opts          WriteBatchOptions
    pendingWrites map[string]*data.LogRecord // 待写入的数据
}
```

```rust
pub struct WriteBatch<'a> {
    pending_writes: Arc<Mutex<HashMap<Vec<u8>, LogRecord>>>,
    engine: &'a Engine,
    options: WriteBatchOptions,
}
```

我们可以在新建 WriteBatch 的时候传入一个配置项，目前有两个配置项。

第一个 MaxBatchNum 表示一个批次中，最多能写多少条数据，加入这个限制的原因是有可能批次数据量太大，导致内存溢出，默认值是 10000，可以根据实际的物理内存大小调整。

第二个是 Sync，表示提交的时候是否需要持久化数据，即强制刷盘，为了保证事务的持久性，这个值默认为 true。

```go
type WriteBatchOptions struct {
        // 一个批次中最大的数据量
        MaxBatchNum int
        // 每一次事务提交时是否持久化
        SyncWrites bool
}
```

```rust
pub struct WriteBatchOptions {
    // 一个 Batch 中最大的数据量
    pub max_batch_num: u32,

    // 提交时是否 sync 持久化
    pub sync_writes: bool,
}
```

注意我们在 WriteBatch 结构体中，使用了一个 map 来暂存待写入的数据，其实用一个数组也是可以的。但是为了防止多次写入同一个 key 的数据，例如我们对同一个 key 连续 Put 了 10 次，实际上只需要保存最后的那条数据即可，而不用将全部的数据都写到磁盘。

WriteBatch 提供了两个方法，分别是 Put 和 Delete，这两个方法和前面实现的存储和删除数据的方法的区别是，不会写磁盘，也不会更新内存，只是构造对应的 LogRecord，并暂存到 map 中。

Commit 方法是核心逻辑，我们需要拿到当前最新的 seq number，然后将其添加到 LogRecord 中，这里采取的办法是将 seq number 和 key 编码到一起，并且采用变长数组，尽量节省空间。实际上我们也可以在 LogRecord 中增加一个字段来专门存储这个 seq number，但是这样会涉及到改动 LogRecord 的编码和解码，稍微复杂点。

然后就是遍历待写入的数据，将其全部写入到磁盘，并且添加一条标识完成的日志记录。

最后再更新内存索引即可。

启动时的修改

启动数据库的时候，不能直接拿到 LogRecord 就去更新内存索引了，因为 LogRecord 有可能是无效的事务的数据，所以我们将其暂存起来，如果读到了一个标识事务完成的数据，才将暂存的对应的事务 id 的数据更新到内存索引。

还需要注意的是，我们在遍历数据的时候，还需要更新对应的 seqNo，并找到最大的那个值，方便数据库启动之后，新的 WriteBatch 能够拿到最新的事务序列号。

上面的修改完成之后，WriteBatch 的大致逻辑便实现了。