TinyKV Project1 设计思路和实现

一.目标

总体目标

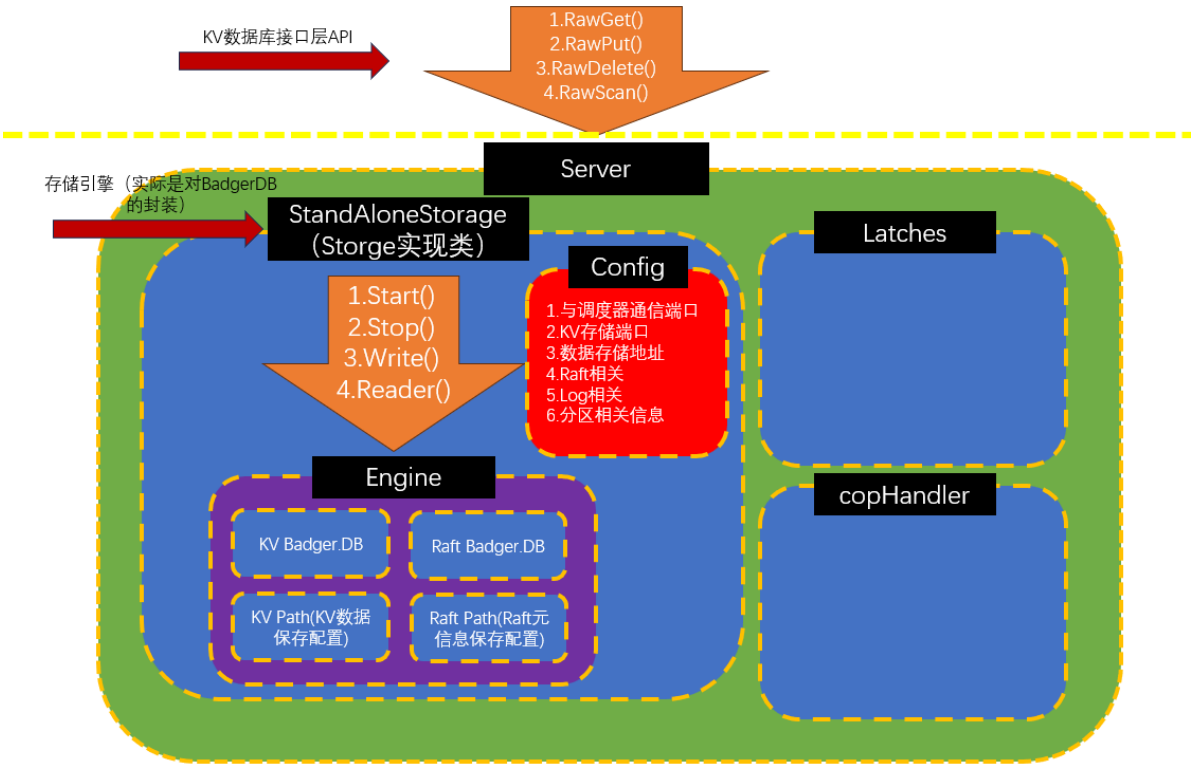
在Project1中，需要构建支持列族的单机键值存储的grpc server，列族称为column familly(CF)，CF 类似于一个命名空间，不同列族中的同名key是不同的，可以认为每一个列族对应一个小型数据库。好像Project4针对列族的作用做了更详细的探讨（负载均衡？）。该服务支持四种基本操作：Put/Delete/Get/Scan。它维护一个简单的KV数据库。键和值是字符串。Put 替换数据库中指定 CF 中特定键的值，Delete 删除指定 CF 的键值，Get 获取指定 CF 的键的当前值，Scan 获取指定 CF 的一系列键的当前值。

目标细化

1. 实现一个底层的单机存储引擎
2. 实现KV数据库操作层接口

二.设计思路

架构设计



涉及到的核心代码

Server结构体

```
type Server struct {
    storage storage.Storage

    // (Used in 4A/4B)
    Latches *latches.Latches

    // coprocessor API handler, out of course scope
    copHandler *coprocessor.CopHandler
}
```

对外有`RawGet()`、`RawPut()`、`RawDelete()`、`RawScan()`几个主要的方法，借助`Storage`的`Reader()`和`Write()`方法调用底层BadgerDB实现持久化存储。

Storage是一个抽象类，它的定义如下

```
type Storage interface {
    Start() error
    Stop() error
    Write(ctx *kvrpcpb.Context, batch []Modify) error
    Reader(ctx *kvrpcpb.Context) (StorageReader, error)
}
```

StandAloneStorage 则是`Storage`接口的一个实现，它的定义如下：

```
type StandAloneStorage struct {
    // Your Data Here (1).
    //存储引擎（对BadgerDB的封装，对应Engine结构体）
    engines *engine_util.Engines
    //配置信息
    config  *config.Config
}
```

Reader()和Write()方法的实现用到了engine_util.Engines的方法，engine_util.Engines`实际上是对Badger键值API的一个封装:

```
type Engines struct {
    // Data, including data which is committed (i.e., committed across other
nodes) and un-committed (i.e., only present
    // locally).
    Kv     *badger.DB
    KvPath string
    // Metadata used by Raft.
    Raft     *badger.DB
    RaftPath string
}
```

Engines中键值的获取、插入、删除等操作，以及事务、迭代器BadgerIterator在kv/util/engine_util中实现

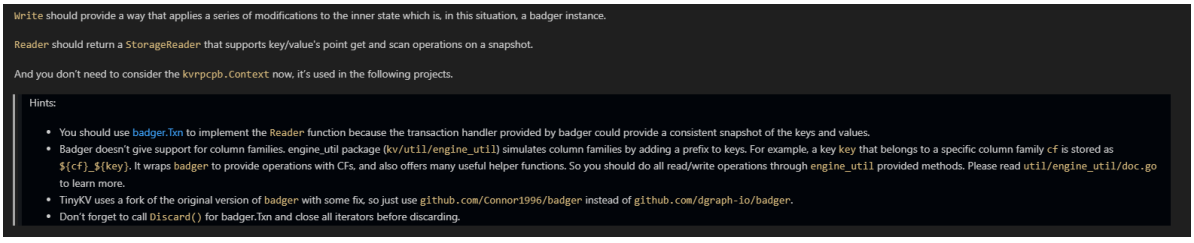
三.代码实现

StandAloneStorage和StandAloneStorageReader实现

```
type StandAloneStorage struct {
    // Your Data Here (1).
    // 底层的存储引擎
    // TODO:追一下util包下Engines的实现
    storeEngine *engine_util.Engines
    // 单机KV引擎的配置信息
    config *config.Config
}
func NewStandAloneStorage(conf *config.Config) *StandAloneStorage {
    // Your Code Here (1).
    // 配置初始化，主要看config中需要更新哪些
    // 注意这里底层持久化是通过badger数据库实现的，（见指导手册）
    // 这里单机kv引擎只需要起一个badger数据库,持久化KV数据即可，raft模式下要起两个badger数据库，一个存KV数据，一个存Raft的MetaData(见engine_util.Engines结构体实现)
    kvPath := conf.DBPath + "/kv_data"
    raftPath := conf.DBPath + "/raft_data"
    var kvEngine * badger.DB
    // kvEngine := engine_util.CreateDB(kvPath,false)
    var raftEngine * badger.DB
    // if conf.Raft {
    //  raftEngine = engine_util.CreateDB(raftPath, true)
    // }
    engines := engine_util.NewEngines(kvEngine, raftEngine, kvPath, raftPath)

    return &StandAloneStorage{
        storeEngine:   engines,
        config:        conf,
    }
}
```

再实现Start，Stop，Reader，Write方法。注意，这里指导手册推荐，Reader使用badger.Txn来实现，最后要返回一个StorageReader，由于StorageReader是一个抽象类，所以要实现一个StandAloneStorageReader实例




```
type StorageReader interface {
    // When the key doesn't exist, return nil for the value
    // StandAloneStorageReader要实现如下三种方法
    GetCF(cf string, key []byte) ([]byte, error)
    IterCF(cf string) engine_util.DBIterator
    Close()
}
```

StandAloneStorageReader具体实现如下：结构体中包含badger.Txn事务句柄

```
type StandAloneStorageReader struct {
    txn *badger.Txn
}

func (s *StandAloneStorageReader) GetCF(cf string, key []byte) ([]byte, error) {
    value, err := engine_util.GetCFFromTxn(s.txn, cf, key)
    if err == badger.ErrKeyNotFound {
        return nil, nil
    }
    return value, err
}

func (s *StandAloneStorageReader) IterCF(cf string) engine_util.DBIterator {
    return engine_util.NewCFIterator(cf, s.txn)
}

func (s *StandAloneStorageReader) Close() {
    s.txn.Discard()
}
```

实现Start，Stop，Reader，Write方法，注意这里Reader要开启事务读，最终返回一个StandAloneStorageReader，上册RawGet，RawScan等都需要调用StandAloneStorageReader实现真正意义上的读取KV

```
func (s *StandAloneStorage) Start() error {
    // Your Code Here (1).
    // 调用start时才初始化数据库
    s.storeEngine.Kv = engine_util.CreateDB(s.storeEngine.KvPath,false)
    if s.config.Raft {
        s.storeEngine.Raft = engine_util.CreateDB(s.storeEngine.RaftPath,true)
    } else {
        s.storeEngine.Raft = engine_util.CreateDB(s.storeEngine.RaftPath,false)
    }
    // TODO:很奇怪CreateDB没有返回错误信息，这里只能返回nil了，细看一下badger数据库使用
    return nil
}

func (s *StandAloneStorage) Stop() error {
    // Your Code Here (1).
    return s.storeEngine.Close()
}

func (s *StandAloneStorage) Reader(ctx *kvrpcpb.Context) (storage.StorageReader,
error) {
    // Your Code Here (1).
    txn := s.storeEngine.Kv.NewTransaction(false)
    return &StandAloneStorageReader{
        txn: txn,
    }, nil
}

func (s *StandAloneStorage) Write(ctx *kvrpcpb.Context, batch []storage.Modify)
error {
    // Your Code Here (1).
```

```
    for _ , modify := range batch {
        switch modify.Data.(type) {
        case storage.Put:
            err :=
engine_util.PutCF(s.storeEngine.Kv,modify.Cf(),modify.Key(),modify.Value())
            if err != nil {
                log.Fatal("write with error: %v\n", err)
                return err
            }
        case storage.Delete:
            err :=
engine_util.DeleteCF(s.storeEngine.Kv,modify.Cf(),modify.Key())
            if err != nil {
                log.Fatal("delete with error: %v\n", err)
                return err
            }
        default :
            log.Fatal("illegal operation in Write handler")
            return nil
        }
    }
    return nil
}
```

RawGet，RawPut，RawDelete，RawScan API的实现

注意这里需要参考grpc request发过来的信息，结果注释读取。同时还需要理解一下Modify这个泛型的作用，本质上是对Put和Delete操作绑定的数据类型。依靠断言实现自动绑定

```
// The functions below are Server's Raw API. (implements TinyKvServer).
// Some helper methods can be found in sever.go in the current directory

// RawGet return the corresponding Get response based on RawGetRequest's CF and
Key fields
func (server *Server) RawGet(_ context.Context, req *kvrpcpb.RawGetRequest)
(*kvrpcpb.RawGetResponse, error) {
    // Your Code Here (1).
    key := req.GetKey()
    cf := req.GetCf()
    reader , _ := server.storage.Reader(req.Context)
    // 底层调用GetCF实现单个读取
    value , err := reader.GetCF(cf, key)
    // 封装grpc response
    response := &kvrpcpb.RawGetResponse{
        Value: value,
        NotFound: false,
    }
    if value == nil {
        response.Value = nil
        response.NotFound = true
    }
    if err != nil {
        log.Fatal("RawGet error:", err)
        response.Error = err.Error()
    }
```

```
    return response, err
}

// RawPut puts the target data into storage and returns the corresponding
response
func (server *Server) RawPut(_ context.Context, req *kvrpcpb.RawPutRequest)
(*kvrpcpb.RawPutResponse, error) {
    // Your Code Here (1).
    // Hint: Consider using Storage.Modify to store data to be modified
    cf := req.GetCf()
    key := req.GetKey()
    value := req.GetValue()
    // Write时需要追一下Modify内部实现，本质上用的是一个泛型。将Modify内部Data要绑定到一种
操作类型的结构体上，结构体内部包含操作的数据
    // 调用Write接口，传入Modify实现自动绑定到Put类型，然后写入调用的底层PutCF
    err := server.storage.Write(nil, []storage.Modify{
        {
            Data: storage.Put{
                Cf:  cf,
                Key: key,
                Value: value,
            },
        },
    })
    putResponse := &kvrpcpb.RawPutResponse{}
    if err != nil {
        log.Fatal("RawPut error:", err)
        putResponse.Error = err.Error()
    }
    return putResponse, err

}

// RawDelete delete the target data from storage and returns the corresponding
response
func (server *Server) RawDelete(_ context.Context, req
*kvrpcpb.RawDeleteRequest) (*kvrpcpb.RawDeleteResponse, error) {
    // Your Code Here (1).
    // Hint: Consider using Storage.Modify to store data to be deleted
    key := req.GetKey()
    cf := req.GetCf()

    err := server.storage.Write(nil, []storage.Modify{
        {
            Data: storage.Delete{
                Cf: cf,
                Key: key,
            },
        },
    })

    delResponse := &kvrpcpb.RawDeleteResponse{}
    if err != nil {
        log.Fatal("RawDelete error:", err)
        delResponse.Error = err.Error()
```

```
    }
    return delResponse, err
}

// RawScan scan the data starting from the start key up to limit. and return the
corresponding result
// RawScan稍微复杂一些，需要看一下RawScanRequest传过来什么
func (server *Server) RawScan(_ context.Context, req *kvrpcpb.RawScanRequest)
(*kvrpcpb.RawScanResponse, error) {
    // Your Code Here (1).
    // Hint: Consider using reader.IterCF
    startKey := req.GetStartKey()
    limit := req.GetLimit()
    cf := req.GetCf()
    reader , _ := server.storage.Reader(nil)
    // 迭代器BadgerIterator的Valid函数是判断当前位置是否有效，而不是判断Next是否有效
    iterator := reader.IterCF(cf)

    var kvs []*kvrpcpb.KvPair
    var err error
    iterator.Seek(startKey)
    for i := 0; uint32(i) < limit; i++ {
        if !iterator.Valid() {
            break
        }
        item := iterator.Item()
        key := item.Key()
        value , err := item.Value()
        if err != nil {
            log.Fatal("ERROR: Failed to get Value from key:", key)
            break;
        }
        pair := kvrpcpb.KvPair{
            Key: key,
            Value: value,
        }
        kvs = append(kvs, &pair)
        iterator.Next()
    }
    response := &kvrpcpb.RawScanResponse{
        Kvs: kvs,
        // err为空时不能调用err.Error(),特判一下
        //Error : err.Error(),
    }
    if err != nil {
        response.Error = err.Error()
    }
    return response, err
}
```

迭代器实现追一下这段代码就知道底层调用的还是Badger迭代器

```
func NewCFIterator(cf string, txn *badger.Txn) *BadgerIterator {
    return &BadgerIterator{
        iter:   txn.NewIterator(badger.DefaultIteratorOptions),
        prefix: cf + "_",
    }
}
```

看一下迭代器遍历后存储的结构体KVPair,通过RawScanResponse发现最后要返回一个KVPairs的切片，所以看一下KVPair怎么存的

```
type RawScanResponse struct {
    RegionError *errorpb.Error
`protobuf:"bytes,1,opt,name=region_error,json=regionError"
json:"region_error,omitempty"`
    // An error which affects the whole scan. Per-key errors are included in
kvs.
    Error                string    `protobuf:"bytes,2,opt,name=error,proto3"
json:"error,omitempty"`
    Kvs                  []*KvPair `protobuf:"bytes,3,rep,name=kvs"
json:"kvs,omitempty"`
    XXX_NoUnkeyedLiteral struct{}  `json:"-"`
    XXX_unrecognized     []byte    `json:"-"`
    XXX_sizecache        int32     `json:"-"`
}
```

```
type KvPair struct {
    Error                *KeyError `protobuf:"bytes,1,opt,name=error"
json:"error,omitempty"`
    Key                  []byte    `protobuf:"bytes,2,opt,name=key,proto3"
json:"key,omitempty"`
    Value                []byte    `protobuf:"bytes,3,opt,name=value,proto3"
json:"value,omitempty"`
    XXX_NoUnkeyedLiteral struct{}  `json:"-"`
    XXX_unrecognized     []byte    `json:"-"`
    XXX_sizecache        int32     `json:"-"`
}
```

四.实验结果

```
GO111MODULE=on go test -v --count=1 --parallel=1 -p=1 ./kv/server -run 1
=== RUN   TestRawGet1
--- PASS: TestRawGet1 (1.04s)
=== RUN   TestRawGetNotFound1
2023/07/20 12:49:32 log.go:77: [info] [key not found when GetCFFrom Txn]

--- PASS: TestRawGetNotFound1 (0.90s)
=== RUN   TestRawPut1
--- PASS: TestRawPut1 (1.00s)
=== RUN   TestRawGetAfterRawPut1
--- PASS: TestRawGetAfterRawPut1 (1.00s)
=== RUN   TestRawGetAfterRawDelete1
2023/07/20 12:49:36 log.go:77: [info] [key not found when GetCFFrom Txn]

--- PASS: TestRawGetAfterRawDelete1 (1.11s)
=== RUN   TestRawDelete1
2023/07/20 12:49:37 log.go:77: [info] [key not found when GetCFFrom Txn]

--- PASS: TestRawDelete1 (1.01s)
=== RUN   TestRawScan1
--- PASS: TestRawScan1 (0.70s)
=== RUN   TestRawScanAfterRawPut1
--- PASS: TestRawScanAfterRawPut1 (1.03s)
=== RUN   TestRawScanAfterRawDelete1
--- PASS: TestRawScanAfterRawDelete1 (0.82s)
=== RUN   TestIterWithRawDelete1
--- PASS: TestIterWithRawDelete1 (0.98s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/server   9.633s
```