项目1 单机 KV

在本项目中，您将在列系列的支持下构建一个独立的键/值存储 gRPC 服务。独立指的是只有一个节点，而不是分布式系统。列族（下文将缩写为 CF）是一个与键命名空间类似的术语，即同一键在不同列族中的值是不一样的。你可以简单地将多个列族视为独立的小型数据库。它用于支持项目4 中的事务模型，你就会知道为什么 TinyKV 需要 CF 的支持了。

该服务支持四种基本操作：输入/删除/获取/扫描。它维护一个简单的键/值对数据库。键和值都是字符串。`Put` 替换数据库中指定 CF 的特定键值，`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。

该项目可分为两个步骤，包括

1. 实施独立的存储引擎。
2. 执行原始键/值服务处理程序。

守则

`gRPC` 服务器在 `kv/main.go` 中初始化，它包含一个 `tinykv.Server`，提供名为 `TinyKv` 的 `gRPC` 服务。它由 `proto/proto/tinykvpb.proto` 中的 protocol-buffer 定义，rpc 请求和响应的细节在 `proto/proto/kvrpcpb.proto` 中定义。

一般来说，您不需要修改 proto 文件，因为所有必要的字段都已为您定义。但如果仍需修改，可以修改 proto 文件，然后运行 `make proto` 更新 `proto/pkg/xxx/xxx.pb.go` 中生成的相关 go 代码。

此外，服务器还依赖于 `Storage`，这是独立存储引擎需要实现的接口，位于 `kv/storage/standalone_storage/standalone_storage.go`。一旦在 `StandaloneStorage` 中实现了 `Storage` 接口，就可以用它为 `Server` 实现原始键/值服务。

实施独立存储引擎

第一个任务是实现獾键/值应用程序接口的封装。gRPC 服务器的服务依赖于 `kv/storage/storage.go`

中定义的存储。在这种情况下，独立的存储引擎只是獾键/值应用程序接口的封装器，由两个方法提供：

```
类型 存储接口 {
    // 其他东西
    Write(ctx *kvrpcpb.Context, batch []Modify) 错误
    Reader(ctx *kvrpcpb.Context) (StorageReader, error)
}
```

`Write` 应该提供一种对内部状态进行一系列修改的方法，在这种情况下，内部状态就是一个獾实例。

阅读器应返回一个存储阅读器，该阅读器支持对快照进行键/值点获取和扫描操作。

你现在不需要考虑 `kvrpcpb.Context`，它已在以下项目中使用。

> 提示
>
> - 您应该使用 badger.Txn 来实现读取器函数，因为 badger 提供的事务处理器可以提供键和值的
> - 一致快照。
>
>   Badger 不支持列族。 engine_util 软件包
>
>   (`kv/util/engine_util`) 通过为键添加前缀来模拟列族。例如，属于特定列族 `cf` 的键值会
>
>   存储为 `${cf}_${key}`。它封装了
> - 獾提供对 CF 的操作，还提供了许多有用的辅助函数。因此，你应该通过 `engine_util` 提供的
> - 方法进行所有读/写操作。请阅读
>
>   `util/engine_util/doc.go` 了解更多信息。
>
>   TinyKV 使用的是獾原始版本的分叉版，并做了一些修正，因此只需使用
>
>   `github.com/Connor1996/badger` 而不是 `github.com/dgraph-io/badger`。不要
>
>   忘记为 badger.Txn 调用 `Discard()` 并在丢弃前关闭所有迭代器。

执行服务处理程序

本项目的最后一步是使用已实现的存储引擎构建原始键/值服务处理程序，包括 RawGet/ RawScan/ RawPut/ RawDelete。处理程序已经为你定义好了，你只需在 `kv/server/raw_api.go` 中填写实现。完成后，记得运行 `make`
`project1` 才能通过测试套件。