

# Project 1 StandaloneKV

## Project overview

首先，需要阅读 TiKV 的相关指南，了解 TiKV 的基本设计、框架、各部分的主要功能：

### TiDB 整体架构

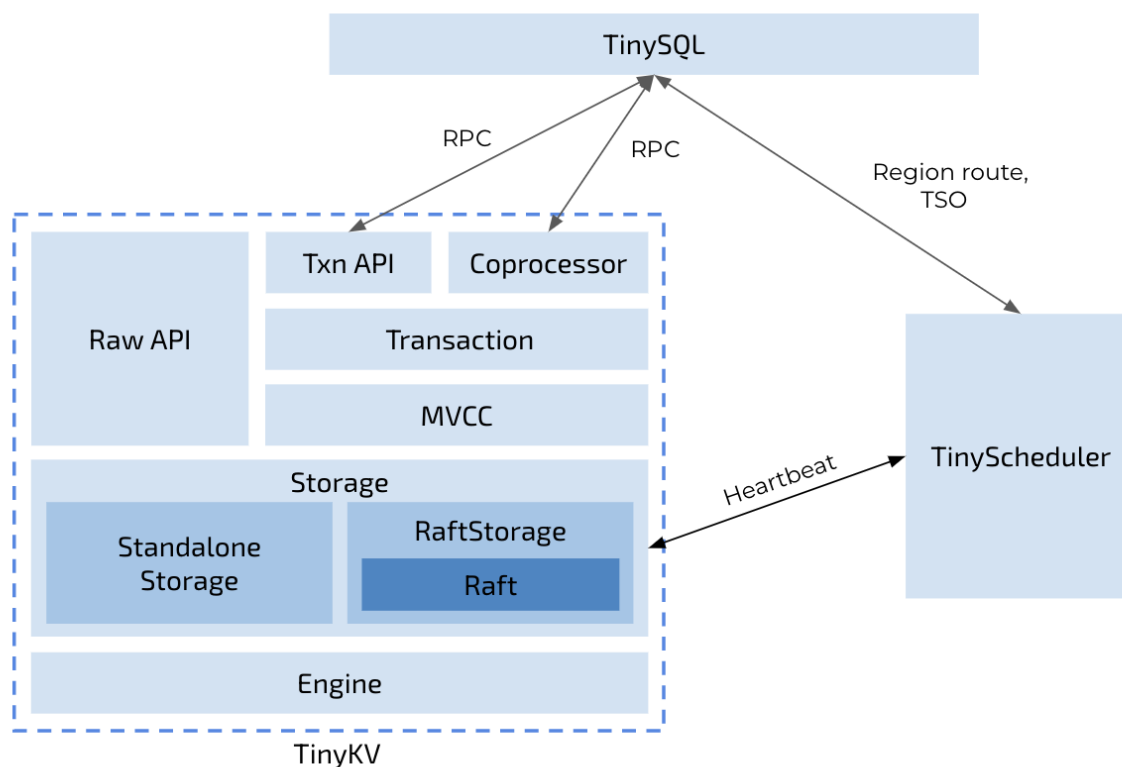
推荐先观看以下视频（时长约 14 分钟），快速了解 TiDB 的整体架构。与传统的单机数据库相比，TiDB 具有以下优势：纯分布式架构，拥有良好的扩展性，支持弹性的扩缩容支持 SQL，对外暴露

<https://docs.pingcap.com/zh/tidb/stable/tidb-architecture>



PingCAP

之后，需要了解 TinyKV 的基本框架。TinyKV 是 TiKV 的简化版，但框架基本一致。



最后了解 TinyKV handle 一个 query 的基本流程（以 Get 为例）：

- user 发送一个 Get RPC request 给 TinySQL 集群。
- TinySQL 集群 parse 这个 request，再 forward 给 TinyKV 集群。
- TinyKV 集群中 Get service handler 开始 handle 这个 request.
- service handler parse 这个 request，再 forward 给下层的 Storage 层。
- Storage 层再 forward 给底层的 Storage Engine 层，进行 Get.
- Get 得到的 value （如果 key 存在），再转回到 Storage 层，再转回到 service handler 。
- service handler 将这个 value 以及其他信息 wrap 到 Get RPC response 中，转给 TinySQL 集群。
- TinySQL 集群最终再以 RPC 的形式转给 user.

## Project Goal

在 project 1 中，我们要实现这样一个 TinyKV 集群：

- standalone: 仅由一个 node 组成，不需要 Raft。
- raw API: 所提供的 service 不需要 txn, MVCC 等 features，只需要完成基本功能。

也就是说，project 1 假定 user 只会发送 Raw API 对应的 RPC request，当 TinyKV 集群接收到 TinySQL forward 来的 request 时，Raw API 对应的 RPC/service handler 会 handle 它。之后调用 Standalone Storage (Storage interface 的单机 implementation) 层的接口。Standalone Storage 再调用 Engine 层的接口，进行真正的 query。

至于为什么 TinyKV 要这样分层设计，我想是为了模块化，分离 interface 与 implementation，使得代码更易写、易维护，框架更清晰、易懂，提供更大的 flexibility。

总结来说，project 1 需要我们实现：

- service handler 与 Storage 层的交互。
- Storage 层与 Engine 层的交互。

## Misc

### gRPC

在 TinyKV 的设计中，user 与集群，集群与集群之间的通信都是使用 RPC。TinyKV 项目选择使用 gRPC (Google RPC) 作为 RPC 框架/协议。一个RPC协议包含两个部分：序列化协议，传输协议。

谁能用通俗的语言解释一下什么是 RPC 框架？

没有邀请，强答一波。先说说原理。本地过程调用 RPC就是要像调用本地的函数一样去调远程函数。在研究RPC前，我们先看看本地调用是怎么调的。假设我们要调用函数Multiply来计算lvalue \* rvalue的

 <https://www.zhihu.com/question/25536695/answer/221638079>



gRPC 的序列化协议使用的是 Protocol Buffer 序列化协议，传输协议则使用的是 HTTP 协议。前者，是我们在项目中会接触到的。

Introduction to gRPC

An introduction to gRPC and protocol buffers. An introduction to gRPC and protocol buffers. This page introduces you to gRPC and protocol buffers. gRPC can use protocol buffers as both its Interface

 <https://grpc.io/docs/what-is-grpc/introduction/>



Protocol Buffer 协议提供了一个 proto 文件格式和 protoc 编译器。通过在 proto 文件中定义一系列 message，即 RPC request 或 response，再把这些文件输入给 protoc 编译器进行编译，会生成对应的 go 文件，包含 RPC request 和 response 的定义，可以被项目直接使用。

Overview | Protocol Buffers | Google Developers

Protocol buffers provide a language-neutral, platform-neutral, extensible mechanism for serializing structured data in a forward-compatible and backward-compatible way. It's like JSON, except it's

 <https://developers.google.com/protocol-buffers/docs/overview>

Google Developers

就像计算机网络中的其他协议一样，gRPC 会作为一个模块/软件运行在集群中的每一个 node 中。可以将 node 中运行的 gRPC 模块也看作一个 server，即 gRPC server。这个 server 可以自动接收 RPC request，但它不知道调用哪个 RPC handler。因此，需要先调用 gRPC 的接口，将我们所实现的 TinyKV 的 handlers 注册到 gRPC server。当然，这一部分，不需要我们自己写代码，`kv/main.go` 文件已经完成了这部分功能。

## Column Family

TinyKV 的另一个重要 feature 是：支持 column family。为了解释 column family，首先需要了解 key-value store 和 relational store，store 即 database。

在 relational store 中，数据库被抽象成表 (table)，每个表由一系列 row 组成，每个 row 则由一系列 column 组成。在存储的时候，这些 column 可以连续存储，也可以分开存储。

在 key-value store 中，数据库被抽象为一个 hash table，由一系列 key-value pair 组成。通常来说，value 的存储形式是字符串或字节串，即一个 key 对应的 value 是连续存储的。

很多时候，一个 key 对应的 value 由很多 field 组成。如果使用 relational store 的概念，key 可以看作 row，value 中的 field 则看成 column。但由于 field 是连续存储的，在 fetch 时，一个 key 对应的 value，即所有 field 都必须被整体 fetch。（这是因为，有些 field 的长度是可变的，不能预知某个 field 的 offset，所以必须整体 fetch）

显然，这会带来性能问题。因为对于某些 query，其只关注某个 key 的某些 field。如果有一种组织方式，能够将一个 key 对应的不同 field 进行分组组织，则在 fetch 时，只需 fetch 某一组 field，而不需要 fetch 所有 field。

column family 即是这样一种组织方式，其将 relational store 中的 column 映射到 key-value store 中的 field。key 对应的 value，即一连串 field，以 column family 为单位进行存储。

## Implementation

### Standalone Storage

切入点在 `standalone.go` 文件。根据 `storage.go` 文件中定义的 `Storage` interface，以及作业 spec，很容易知道 `Write` 方法的所需实现的功能：调用 Engine 层的接口，完成一系列写入。Engine 层的接口已经在 `util/engine_util/engines.go` 文件中给出了。值得一提的是，在我做 project 1 时，对于 TinyKV 的分层设计和代码结构，还没理解的这么清晰，因此我在实现 `Write` 时，调用的是 `util/engine_util/write_batch.go` 文件中的方法。虽然也能通过测试，但是与 TinyKV 的分层设计有冲突。

之后需要实现 `Reader` 方法。根据 spec 的说明，`Reader` 方法返回关于数据库的当前 snapshot 的一个 reader。既然与 snapshot 有关，因此我在阅读代码时猜想与 txn 有关。又根据 hints，需要使用 `badger.Txn`，则验证了我的猜想。

关于 reader，需要实现三个方法。其中值得一提的是 `IterCF` 的语义。我认为既然一个 reader 对应一个 snapshot，即对应一个 txn，则一个 reader 的 iterator 应该是复用的。并且 spec 中的 hints 也说，在 discard txn 之前，需要注意 close iterator。但是这个语义还是不清楚的，因为 project 1 只有一个测试用例调用了 `IterCF`，然而它在调用之后，又 defer 了 iterator 的 close。所以究竟是每次调用 `IterCF`，返回一个新的 iterator，然后由调用者 close；还是返回现有的 iterator（如果有的话），再在 close reader 时 close 这个 iterator。还需要商榷。

## Raw API

四个 service handler 的实现非常简单，且实现思路基本一致：首先解析 RPC request，再调用 Storage 层的接口。拿到 Storage 返回的结果后，将它们 wrap 到 RPC response 中，最后返回这些 response 即可。

## Conclusion

Project 1 主要工作量包括：

- 阅读 TiDB 和 TinyKV 的相关文档，了解基本框架。
- 阅读作业 spec 及相关，例如 gRPC, Protocol Buffer, Go 语言相关文档，了解作业要求。
- 阅读 TinyKV 代码，重点阅读 `kv` 文件夹下的代码，了解代码结构和每个文件的基本内容。

总结来说，project 1 比较简单，是关于 TinyKV 的 warp up 和关于 Go 语言的 refresh。

预计耗时 `2 ~ 6` 小时。

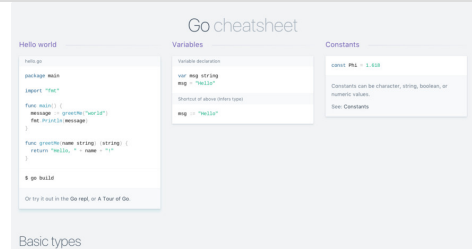
## Useful links

<https://go.dev/tour/welcome/1>

## Go cheatsheet


```
package main import "fmt" func main() { message :=  
greetMe("world") fmt.Println(message) } func greetMe(name string)  
string { return "Hello, " + name + "!" } Or try it out in the Go repl, or A
```

 <https://devhints.io/go>



tinykv/reading\_list.md at course · talent-plan/tinykv

A course to build distributed key-value service based on TiKV  
model - tinykv/reading\_list.md at course · talent-plan/tinykv

 [https://github.com/talent-plan/tinykv/blob/course/doc/reading\\_list.md](https://github.com/talent-plan/tinykv/blob/course/doc/reading_list.md)

talent-plan/**tinykv**

A course to build distributed key-value service  
based on TiKV model



 78  
Contributors

 31  
Issues

 2k  
Stars

 764  
Forks

