

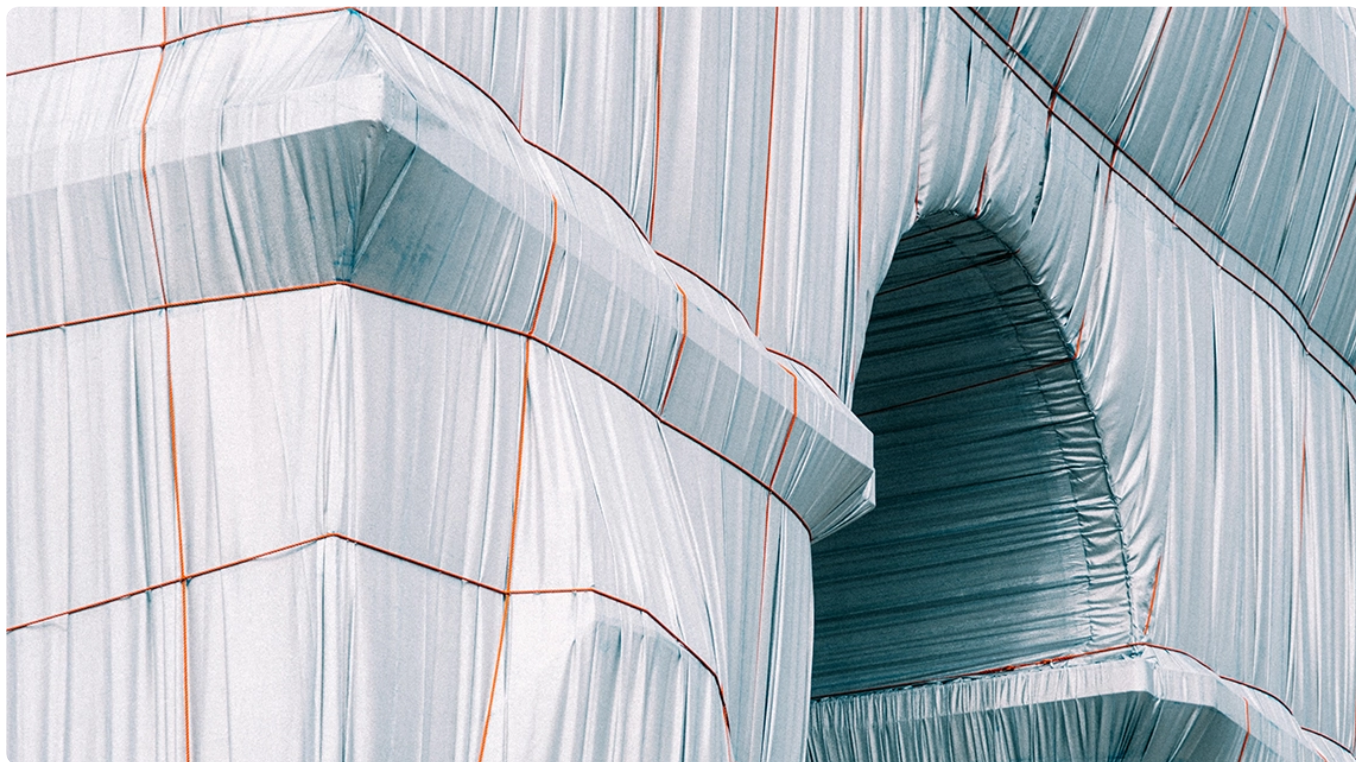


## 21 | 阶段实操：构建一个简单的 KV server (1) -基本流程（上）

2021-10-11 陈天

《陈天 · Rust 编程第一课》

[课程介绍 >](#)



讲述：陈天

时长 11:46 大小 10.79M



你好，我是陈天。

从第七讲开始，我们一路过关斩将，和所有权、生命周期死磕，跟类型系统和 trait 反复拉锯，为的是啥？就是为了能够读懂别人写的代码，进而让自己也能写出越来越复杂且优雅的代码。

今天就到检验自身实力的时候了，毕竟 talk is cheap，知识点掌握得再多，自己写不出来也白搭，所以我们将之前学的知识都运用起来，一起写个简单的 KV server。



不过这次和 get hands dirty 重感性体验的代码不同，我会带你一步步真实打磨，讲得比较细致，所以内容也会比较多，我分成了上下两篇文章，希望你能耐心看完，认真感受 Rust best practice 在架构设计以及代码实现思路上的体现。

为什么选 KV server 来实操呢？因为它是一个**足够简单又足够复杂**的服务。参考工作中用到的 Redis / Memcached 等服务，来梳理它的需求。

最核心的功能是根据不同的命令进行诸如数据存贮、读取、监听等操作；


而客户端要能通过网络访问 KV server，发送包含命令的请求，得到结果；

数据要能根据需要，存储在内存中或者持久化到磁盘上。

## 先来一个短平糙的实现

如果是为了完成任务构建 KV server，其实最初的版本两三百行代码就可以搞定，但是这样的代码以后维护起来就是灾难。

我们看一个省却了不少细节的意大利面条式的版本，你可以随着我的注释重点看流程：

 复制代码

```
1 use anyhow::Result;
2 use async_prost::AsyncProstStream;
3 use dashmap::DashMap;
4 use futures::prelude::*;
5 use kv::{
6     command_request::RequestData, CommandRequest, CommandResponse, Hset, KvErr
7 };
8 use std::sync::Arc;
9 use tokio::net::TcpListener;
10 use tracing::info;
11
12 #[tokio::main]
13 async fn main() -> Result<> {
14     // 初始化日志
15     tracing_subscriber::fmt::init();
16
17     let addr = "127.0.0.1:9527";
18     let listener = TcpListener::bind(addr).await?;
19     info!("Start listening on {}", addr);
20
21     // 使用 DashMap 创建放在内存中的 kv store
22     let table: Arc<DashMap<String, Value>> = Arc::new(DashMap::new());
23
24     loop {
25         // 得到一个客户端请求
26         let (stream, addr) = listener.accept().await?;
27         info!("Client {:?} connected", addr);
28
29         // 复制 db，让它在 tokio 任务中可以使用
```


```

30     let db = table.clone();
31
32     // 创建一个 tokio 任务处理这个客户端
33     tokio::spawn(async move {
34         // 使用 AsyncProstStream 来处理 TCP Frame
35         // Frame: 两字节 frame 长度, 后面是 protobuf 二进制
36         let mut stream =
37             AsyncProstStream::<_, CommandRequest, CommandResponse, _>::fro
38
39         // 从 stream 里取下一个消息 (拿出来后已经自动 decode 了)
40         while let Some(Ok(msg)) = stream.next().await {
41             info!("Got a new command: {:?}", msg);
42             let resp: CommandResponse = match msg.request_data {
43                 // 为演示我们就处理 HSET
44                 Some(RequestData::Hset(cmd)) => hset(cmd, &db),
45                 // 其它暂不处理
46                 _ => unimplemented!(),
47             };
48
49             info!("Got response: {:?}", resp);
50             // 把 CommandResponse 发送给客户端
51             stream.send(resp).await.unwrap();
52         }
53     });
54 }
55 }
56
57 // 处理 hset 命令
58 fn hset(cmd: Hset, db: &DashMap<String, Value>) -> CommandResponse {
59     match cmd.pair {
60         Some(Kvpair {
61             key,
62             value: Some(v),
63         }) => {
64             // 往 db 里写入
65             let old = db.insert(key, v).unwrap_or_default();
66             // 把 value 转换成 CommandResponse
67             old.into()
68         }
69         v => KvError::InvalidCommand(format!("hset: {:?}", v)).into(),
70     }
71 }

```

这段代码非常地平铺直叙，从输入到输出，一蹴而就，如果这样写，任务确实能很快完成，但是它有“完成之后，哪管洪水滔天”的感觉。

你复制代码后，打开两个窗口，分别运行 “cargo run --example naive\_server” 和 “cargo run --example client”，就可以看到运行 server 的窗口有如下打印：

 复制代码

```
1 Sep 19 22:25:34.016 INFO naive_server: Start listening on 127.0.0.1:9527
2 Sep 19 22:25:38.401 INFO naive_server: Client 127.0.0.1:51650 connected
3 Sep 19 22:25:38.401 INFO naive_server: Got a new command: CommandRequest { re
4 Sep 19 22:25:38.401 INFO naive_server: Got response: CommandResponse { status
```

虽然整体功能算是搞定了，不过以后想继续为这个 KV server 增加新的功能，就需要来来回回改这段代码。

此外，也不好做单元测试，因为所有的逻辑都被压缩在一起了，没有“单元”可言。虽然未来可以逐步把不同的逻辑分离到不同的函数，使主流程尽可能简单一些。但是，它们依旧是耦合在一起的，如果不做大的重构，还是解决不了实质性的问题。

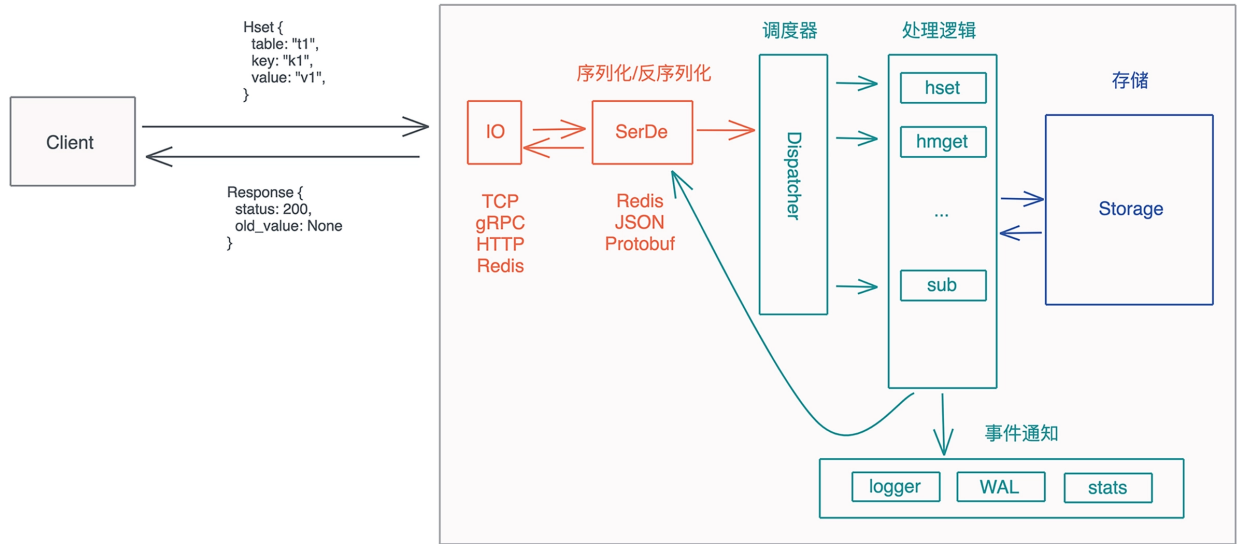
所以不管用什么语言开发，这样的代码都是我们要极力避免的，不光自己不要这么写，code review 遇到别人这么写也要严格地揪出来。

## 架构和设计

那么，怎样才算是好的实现呢？

好的实现应该是在分析完需求后，首先从系统的主流程开始，搞清楚从客户端的请求到最终客户端收到响应，都会经过哪些主要的步骤；然后根据这些步骤，思考哪些东西需要延迟绑定，构建主要的接口和 trait；等这些东西深思熟虑之后，最后再考虑实现。也就是所谓的“谋定而后动”。

开头已经分析 KV server 这个需求，现在我们来梳理主流程。你可以先自己想想，再参考示意图看看有没有缺漏：



极客时间

这个流程中有一些关键问题需要进一步探索：

1. 客户端和服务端用什么协议通信？TCP？gRPC？HTTP？支持一种还是多种？
2. 客户端和服务端之间交互的应用层协议如何定义？怎么做序列化 / 反序列化？是用 Protobuf、JSON 还是 Redis RESP？或者也可以支持多种？
3. 服务端都支持哪些命令？第一版优先支持哪些？
4. 具体的处理逻辑中，需不需要加 hook，在处理过程中发布一些事件，让其他流程可以得到通知，进行额外的处理？这些 hook 可不可以提前终止整个流程的处理？
5. 对于存储，要支持不同的存储引擎么？比如 MemDb（内存）、RocksDb（磁盘）、SledDb（磁盘）等。对于 MemDb，我们考虑支持 WAL（Write-Ahead Log）和 snapshot 么？
6. 整个系统可以配置么？比如服务使用哪个端口、哪个存储引擎？
7. ...

**如果你想做好架构，那么，问出这些问题，并且找到这些问题的答案就很重要。**值得注意的是，这里面很多问题产品经理并不能帮你回答，或者 TA 的回答会把你带入歧路。作为一个架构师，我们需要对系统未来如何应对变化负责。



下面是我的思考，你可以参考：

1. 像 KV Server 这样需要高性能的场景，通信应该优先考虑 TCP 协议。所以我们暂时只支持 TCP，未来可以根据需要支持更多的协议，如 HTTP2/gRPC。还有，未来可能对安全性有额外的要求，所以我们要保证 TLS 这样的安全协议可以即插即用。总之，**网络层需要灵活**。

2. 应用层协议我们可以用 protobuf 定义。protobuf 直接解决了协议的定义以及如何序列化和反序列化。Redis 的 [RESP](#) 固然不错，但它的短板也显而易见，命令需要额外的解析，而且大量的 `\r\n` 来分隔命令或者数据，也有些浪费带宽。使用 JSON 的话更加浪费带宽，且 JSON 的解析效率不高，尤其是数据量很大的时候。

protobuf 就很适合 KV server 这样的场景，灵活、可向后兼容式升级、解析效率很高、生成的二进制非常省带宽，唯一的缺点是需要额外的工具 protoc 来编译成不同的语言。虽然 protobuf 是首选，但也许未来为了和 Redis 客户端互通，还是要支持 RESP。

3. 服务器支持的命令我们可以参考 [Redis 的命令集](#)。第一版先来支持 [HXXX 命令](#)，比如 HSET、HMSET、HGET、HMGET 等。从命令到命令的响应，可以做个 trait 来抽象。

4. 处理流程中计划加这些 hook：收到客户端的命令后 OnRequestReceived、处理完客户端的命令后 OnRequestExecuted、发送响应之前 BeforeResponseSend、发送响应之后 AfterResponseSend。这样，**处理过程中的主要步骤都有事件暴露出去，让我们的 KV server 可以非常灵活，方便调用者在初始化服务的时候注入额外的处理逻辑**。

5. 存储必然需要足够灵活。可以对存储做个 trait 来抽象其基本的行为，一开始可以就只做 MemDb，未来肯定需要有支持持久化的存储。

6. 需要支持配置，但优先级不高。等基本流程搞定，使用过程中发现足够的痛点，就可以考虑配置文件如何处理了。

当这些问题都敲定下来，系统的基本思路就有了。我们可以先把几个重要的接口定义出来，然后仔细审视这些接口。

最重要的几个接口就是**三个主体交互的接口：客户端和服务器的接口或者说协议、服务器和命令处理流程的接口、服务器和存储的接口。**

## 客户端和服务器的协议

首先是客户端和服务端之间的协议。来试着用 protobuf 定义一下我们第一版支持的客户端命令：

[复制代码](#)

```
1 syntax = "proto3";
2
3 package abi;
4
5 // 来自客户端的命令请求
6 message CommandRequest {
7     oneof request_data {
8         Hget hget = 1;
9         Hgetall hgetall = 2;
10        Hmget hmget = 3;
11        Hset hset = 4;
12        Hmset hmset = 5;
13        Hdel hdel = 6;
14        Hmdel hmdel = 7;
15        Hexist hexist = 8;
16        Hmexist hmexist = 9;
17    }
18 }
19
20 // 服务器的响应
21 message CommandResponse {
22     // 状态码；复用 HTTP 2xx/4xx/5xx 状态码
23     uint32 status = 1;
24     // 如果不是 2xx，message 里包含详细的信息
25     string message = 2;
26     // 成功返回的 values
27     repeated Value values = 3;
28     // 成功返回的 kv pairs
29     repeated Kvpair pairs = 4;
30 }
31
32 // 从 table 中获取一个 key，返回 value
33 message Hget {
34     string table = 1;
35     string key = 2;
36 }
37
38 // 从 table 中获取所有的 Kvpair
39 message Hgetall { string table = 1; }
```

```
40 // 从 table 中获取一组 key , 返回它们的 value
41 message Hmget {
42     string table = 1;
43     repeated string keys = 2;
44 }
45
46 // 返回的值
47 message Value {
48     oneof value {
49         string string = 1;
50         bytes binary = 2;
51         int64 integer = 3;
52         double float = 4;
53         bool bool = 5;
54     }
55 }
56
57 // 返回的 kvpair
58 message Kvpair {
59     string key = 1;
60     Value value = 2;
61 }
62
63 // 往 table 里存一个 kvpair ,
64 // 如果 table 不存在就创建这个 table
65 message Hset {
66     string table = 1;
67     Kvpair pair = 2;
68 }
69
70 // 往 table 中存一组 kvpair ,
71 // 如果 table 不存在就创建这个 table
72 message Hmset {
73     string table = 1;
74     repeated Kvpair pairs = 2;
75 }
76
77 // 从 table 中删除一个 key , 返回它之前的值
78 message Hdel {
79     string table = 1;
80     string key = 2;
81 }
82
83 // 从 table 中删除一组 key , 返回它们之前的值
84 message Hmdel {
85     string table = 1;
86     repeated string keys = 2;
87 }
88
89 // 查看 key 是否存在
90 message Hexist {
91
```



```
92     string table = 1;
93     string key = 2;
94 }
95
96 // 查看一组 key 是否存在
97 message Hmexist {
98     string table = 1;
99     repeated string keys = 2;
100 }
```

通过 [prost](#)，这个 protobuf 文件可以被编译成 Rust 代码（主要是 struct 和 enum），供我们使用。你应该还记得，之前在 [第 5 讲](#)谈到 thumbor 的开发时，已经见识到了 prost 处理 protobuf 的方式了。

## CommandService trait

客户端和服务端间的协议敲定之后，就要思考如何处理请求的命令，返回响应。

我们目前打算支持 9 种命令，未来可能支持更多命令。所以最好**定义一个 trait 来统一处理所有的命令，返回处理结果**。在处理命令的时候，需要和存储发生关系，这样才能根据请求中携带的参数读取数据，或者把请求中的数据存入存储系统中。所以，这个 trait 可以这么定义：

[复制代码](#)

```
1 /// 对 Command 的处理的抽象
2 pub trait CommandService {
3     /// 处理 Command，返回 Response
4     fn execute(self, store: &impl Storage) -> CommandResponse;
5 }
```

有了这个 trait，并且每一个命令都实现了这个 trait 后，dispatch 方法就可以是类似这样的代码：

[复制代码](#)

```
1 // 从 Request 中得到 Response，目前处理 HGET/HGETALL/HSET
2 pub fn dispatch(cmd: CommandRequest, store: &impl Storage) -> CommandResponse
3     match cmd.request_data {
4         Some(RequestData::Hget(param)) => param.execute(store),
5         Some(RequestData::Hgetall(param)) => param.execute(store),
6         Some(RequestData::Hset(param)) => param.execute(store),
```

```


7      None => KvError::InvalidCommand("Request has no data".into()).into(),
8      _ => KvError::Internal("Not implemented".into()).into(),
9  }
10 ~

```

这样，未来我们支持新命令时，只需要做两件事：为命令实现 `CommandService`、在 `dispatch` 方法中添加新命令的支持。

## Storage trait

再来看为不同的存储而设计的 `Storage trait`，它提供 KV store 的主要接口：

 复制代码


```

1  /// 对存储的抽象，我们不关心数据存在哪儿，但需要定义外界如何和存储打交道
2  pub trait Storage {
3      /// 从一个 HashTable 里获取一个 key 的 value
4      fn get(&self, table: &str, key: &str) -> Result<Option<Value>, KvError>;
5      /// 从一个 HashTable 里设置一个 key 的 value，返回旧的 value
6      fn set(&self, table: &str, key: String, value: Value) -> Result<Option<Val
7      /// 查看 HashTable 中是否有 key
8      fn contains(&self, table: &str, key: &str) -> Result<bool, KvError>;
9      /// 从 HashTable 中删除一个 key
10     fn del(&self, table: &str, key: &str) -> Result<Option<Value>, KvError>;
11     /// 遍历 HashTable，返回所有 kv pair（这个接口不好）
12     fn get_all(&self, table: &str) -> Result<Vec<Kvpair>, KvError>;
13     /// 遍历 HashTable，返回 kv pair 的 Iterator
14     fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>
15 }

```

在 `CommandService trait` 中已经看到，在处理客户端请求的时候，与之打交道的是 `Storage trait`，而非具体的某个 store。这样做的好处是，**未来根据业务的需要，在不同的场景下添加不同的 store，只需要为其实现 `Storage trait` 即可，不必修改 `CommandService` 有关的代码。**

比如在 `HGET` 命令的实现时，我们使用 `Storage::get` 方法，从 `table` 中获取数据，它跟某个具体的存储方案无关：

 复制代码

```

1  impl CommandService for Hget {
2      fn execute(self, store: &impl Storage) -> CommandResponse {
3          match store.get(&self.table, &self.key) {

```

```
4         Ok(Some(v)) => v.into(),
5         Ok(None) => KvError::NotFound(self.table, self.key).into(),
6         Err(e) => e.into(),
7     }
8 }
9 }
```

Storage trait 里面的绝大多数方法相信你可以定义出来，但 `get_iter()` 这个接口可能你会比较困惑，因为它返回了一个 `Box<dyn Iterator>`，为什么？

之前（[第 13 讲](#)）讲过这是 trait object。

这里我们想返回一个 iterator，调用者不关心它具体是什么类型，只要可以不停地调用 `next()` 方法取到下一个值就可以了。不同的实现，可能返回不同的 iterator，如果要用同一个接口承载，我们需要使用 trait object。在使用 trait object 时，因为 `Iterator` 是个带有关联类型的 trait，所以这里需要指明关联类型 `Item` 是什么类型，这样调用者才好拿到这个类型进行处理。

你也许会有疑问，`set / del` 明显是个会导致 `self` 修改的方法，为什么它的接口依旧使用的是 `&self` 呢？

我们思考一下它的用法。对于 Storage trait，最简单的实现是 in-memory 的 `HashMap`。由于我们支持的是 `HSET / HGET` 这样的命令，**它们可以从不同的表中读取数据，所以需要嵌套的 `HashMap`**，类似 `HashMap<String, HashMap<String, Value>>>`。

另外，由于**要在多线程 / 异步环境下读取和更新内存中的 `HashMap`**，所以我们需要类似 `Arc<RwLock<HashMap<String, Arc<RwLock<HashMap<String, Value>>>>>>` 的结构。这个结构是一个多线程环境下具有内部可变性的数据结构，所以 `get / set` 的接口是 `&self` 就足够了。

## 小结

到现在，我们梳理了 KV server 的主要需求和主流程，思考了流程中可能出现的问题，也敲定了三个重要的接口：客户端和服务器的协议、`CommandService trait`、`Storage`

trait。下一讲继续实现 KV server，在看讲解之前，你可以先想一想自己平时是怎么开发的。

## 思考题

想一想，对于 Storage trait，为什么返回值都用了 Result<T, E>？在实现 MemTable 的时候，似乎所有返回都是 Ok(T) 啊？

欢迎在留言区分享你的思考。我们下篇见~

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 5 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 20 | 4 Steps：如何更好地阅读Rust源码？

下一篇 22 | 阶段实操：构建一个简单的 KV server (1) -基本流程（下）

4 周年庆限定

# 299元随心畅学卡

五门课程任你选，总价值高达千元

超值拿下



## 精选留言 (4)

[写留言](#)**Marvichov**

2021-10-14

这次的选图发生在最近的一个大地艺术: 包裹凯旋门!

想了想为什么需要序列化/反序列化, http, json或者xml已经是structured的数据了啊; 后来意识到, 需要把text data转化成某种rust的数据结构(data + algorithm)才能使用与数据绑定的algorithm (OOP)...

展开

**D. D**

2021-10-12

定义Storage trait就是为了以后可以灵活的使用不同的具体的存储方案。如果之后要求持久化, 其中涉及到例如I/O之类的操作, 就很有可能要返回Err了。

此外, 在并发场景下, 也会有例如获取锁失败之类的情況。

展开

**pedro**

2021-10-11

想一想, 对于 Storage trait, 为什么返回值都用了 Result<T, E> ? 在实现 MemTable 的时候, 似乎所有返回都是 Ok(T) 啊?

Result 这个枚举有两个类型T, E, 当查询出错时, 通过 E 给出出错原因, 方便客户端及时做出纠错和调整, 而 Ok 只有 1 和 0 的区分。

展开

**给我点阳光就灿烂**

2021-10-11

老师在以后的章节中可不可以讲一下, 如何把server实现类似docker 一样的socket 守护进程



