26｜阶段实操：构建一个简单的 KV server (2) - 高级 trait 技巧

time.geekbang.org/column/article/429666

00:00

1.0x
讲述：陈天大小：14.75M时长：16:06

你好，我是陈天。

到现在，泛型的基础知识、具体如何使用以及设计理念，我们已经学得差不多了，也和函数作了类比帮助你理解，泛型就是数据结构的函数。

如果你觉得泛型难学，是因为它的抽象层级比较高，需要足够多的代码阅读和撰写的历练。所以，通过学习，现阶段你能够看懂包含泛型的代码就够了，至于使用，只能靠你自己在后续练习中不断体会总结。如果实在觉得不好懂，某种程度上说，你缺乏的不是泛型的能力，而是设计和架构的能力。

今天我们就用之前 1.0 版简易的 KV store 来历练一把，看看怎么把之前学到的知识融入代码中。

在 21 讲、22 讲中，我们已经完成了 KV store 的基本功能，但留了两个小尾巴：
Storage trait 的 get_iter() 方法没有实现；

Service 的 execute() 方法里面还有一些 TODO，需要处理事件的通知。

我们一个个来解决。先看 get_iter() 方法。

处理 Iterator

在开始撰写代码之前，先把之前在 src/storage/mod.rs 里注掉的测试，加回来：

```
#[test]

fn memtable_iter_should_work() {

let store = MemTable::new();

test_get_iter(store);

}
```

然后在 src/storge/memory.rs 里尝试实现它。

```
impl Storage for MemTable {

...

fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError> {

// 使用 clone() 来获取 table 的 snapshot

let table = self.get_or_create_table(table).clone();

let iter = table

.iter()

.map(|v| Kvpair::new(v.key(), v.value().clone()));

Ok(Box::new(iter)) // <-- 编译出错

}

}
```

很不幸的，编译器提示我们 Box::new(iter) 不行，“cannot return value referencing local variable table” 。这让人很不爽，究其原因，table.iter() 使用了 table 的引用，我们返回 iter，但 iter 引用了作为局部变量的 table，所以无法编译通过。

此刻，我们需要有一个能够完全占有 table 的迭代器。Rust 标准库里提供了一个 trait IntoIterator，它可以把数据结构的所有权转移到 Iterator 中，看它的声明（代码）：

```
pub trait IntoIterator {

type Item;
```

```
type IntoIter: Iterator<Item = Self::Item>;

fn into_iter(self) -> Self::IntoIter;

}
```

绝大多数的集合类数据结构都实现了它。DashMap 也实现了它，所以我们可以用 table.into_iter() 把 table 的所有权转移给 iter：

```
impl Storage for MemTable {

...

fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError> {

// 使用 clone() 来获取 table 的 snapshot

let table = self.get_or_create_table(table).clone();

let iter = table.into_iter().map(|data| data.into());

Ok(Box::new(iter))

}

}
```

这里又遇到了数据转换，从 DashMap 中 iterate 出来的值 (String, Value) 需要转换成 Kvpair，我们依旧用 into() 来完成这件事。为此，需要为 Kvpair 实现这个简单的 Fromtrait：

```
impl From<(String, Value)> for Kvpair {

fn from(data: (String, Value)) -> Self {

Kvpair::new(data.0, data.1)

}

}
```

这两段代码都放在 src/storage/memory.rs 下。

Bingo！这个代码可以编译通过。现在如果运行 cargo test 进行测试的话，对 get_iter() 接口的测试也能通过。

虽然这个代码可以通过测试，并且本身也非常精简，我们还是有必要思考一下，如果以后想为更多的 data store 实现 Storage trait，都会怎样处理 get_iter() 方法？

我们会：

拿到一个关于某个 table 下的拥有所有权的 Iterator

对 Iterator 做 map

将 map 出来的每个 item 转换成 Kvpair

这里的第 2 步对于每个 Storage trait 的 get_iter() 方法的实现来说，都是相同的。有没有可能把它封装起来呢？使得 Storage trait 的实现者只需要提供它们自己的拥有所有权的 Iterator，并对 Iterator 里的 Item 类型提供 Into<Kvpair> ？

来尝试一下，在 src/storage/mod.rs 中，构建一个 StorageIter，并实现 Iterator trait：

```
/// 提供 Storage iterator，这样 trait 的实现者只需要
/// 把它们的 iterator 提供给 StorageIter，然后它们保证
/// next() 传出的类型实现了 Into<Kvpair> 即可
pub struct StorageIter<T> {
data: T,
}
impl<T> StorageIter<T> {
pub fn new(data: T) -> Self {
Self { data }
}
}
impl<T> Iterator for StorageIter<T>
where
T: Iterator,
T::Item: Into<Kvpair>,
{
type Item = Kvpair;
fn next(&mut self) -> Option<Self::Item> {
self.data.next().map(|v| v.into())
}
```

```
}
```

这样，我们在 src/storage/memory.rs 里对 get_iter() 的实现，就可以直接使用 StorageIter 了。不过，还要为 DashMap 的 Iterator 每次调用 next() 得到的值 (String, Value)，做个到 Kvpair 的转换：

```
impl Storage for MemTable {

...

fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError> {

// 使用 clone() 来获取 table 的 snapshot

let table = self.get_or_create_table(table).clone();

let iter = StorageIter::new(table.into_iter()); // 这行改掉了

Ok(Box::new(iter))

}

}
```

我们可以再次使用 cargo test 测试，同样通过！

如果回顾刚才撰写的代码，你可能会哑然一笑：我辛辛苦苦又写了 20 行代码，创建了一个新的数据结构，就是为了 get_iter() 方法里的一行代码改得更漂亮？何苦呢？

的确，在这个 KV server 的例子里，这样的抽象收益不大。但是，如果刚才那个步骤不是 3 步，而是 5 步 /10 步，其中大量的步骤都是相同的，也就是说，我们每实现一个新的 store，就要撰写相同的代码逻辑，那么，这个抽象就非常有必要了。

支持事件通知

好，我们再来看事件通知。在 src/service/mod.rs 中（以下代码，如无特殊声明，都是在 src/service/mod.rs 中），目前的 execute() 方法还有很多 TODO 需要解决：

```
pub fn execute(&self, cmd: CommandRequest) -> CommandResponse {

debug!("Got request: {:?}", cmd);

// TODO: 发送 on_received 事件

let res = dispatch(cmd, &self.inner.store);

debug!("Executed response: {:?}", res);

// TODO: 发送 on_executed 事件
```

```
res
}
```

为了解决这些 TODO，我们需要提供事件通知的机制：

在创建 Service 时，注册相应的事件处理函数；

在 execute() 方法执行时，做相应的事件通知，使得注册的事件处理函数可以得到执行。

先看事件处理函数如何注册。

如果想要能够注册，那么倒推也就是，Service/ServiceInner 数据结构就需要有地方能够承载事件注册函数。可以尝试着把它加在 ServiceInner 结构里：

```
/// Service 内部数据结构
pub struct ServiceInner<Store> {
store: Store,
on_received: Vec<fn(&CommandRequest)>,
on_executed: Vec<fn(&CommandResponse)>,
on_before_send: Vec<fn(&mut CommandResponse)>,
on_after_send: Vec<fn()>,
}
```

按照 21 讲的设计，我们提供了四个事件：

on_received：当服务器收到 CommandRequest 时触发；

on_executed：当服务器处理完 CommandRequest 得到 CommandResponse 时触发；

on_before_send：在服务器发送 CommandResponse 之前触发。注意这个接口提供的是 &mut CommandResponse，这样事件的处理者可以根据需要，在发送前，修改 CommandResponse。

on_after_send：在服务器发送完 CommandResponse 后触发。

在撰写事件注册的代码之前，还是先写个测试，从使用者的角度，考虑如何进行注册：

```
#[test]
fn event_registration_should_work() {
fn b(cmd: &CommandRequest) {
```

```
info!("Got {:?}", cmd);
}
fn c(res: &CommandResponse) {
info!("{:?}", res);
}
fn d(res: &mut CommandResponse) {
res.status = StatusCode::CREATED.as_u16() as _;
}
fn e() {
info!("Data is sent");
}
let service: Service = ServiceInner::new(MemTable::default())
.fn_received(|_: &CommandRequest| {})
.fn_received(b)
.fn_executed(c)
.fn_before_send(d)
.fn_after_send(e)
.into();
let res = service.execute(CommandRequest::new_hset("t1", "k1", "v1".into()));
assert_eq!(res.status, StatusCode::CREATED.as_u16() as _);
assert_eq!(res.message, "");
assert_eq!(res.values, vec![Value::default()]);
}
```

从测试代码中可以看到，我们希望通过 ServiceInner 结构，不断调用 fn_xxx 方法，为 ServiceInner 注册相应的事件处理函数；添加完毕后，通过 into() 方法，我们再把 ServiceInner 转换成 Service。这是一个经典的构造者模式（Builder Pattern），在很多 Rust 代码中，都能看到它的身影。

那么，诸如 fn_received() 这样的方法有什么魔力呢？它为什么可以一路做链式调用呢？答案很简单，它把 self 的所有权拿过来，处理完之后，再返回 self。所以，我们继续添加如下代码：

```
impl<Store: Storage> ServiceInner<Store> {
pub fn new(store: Store) -> Self {
Self {
store,
on_received: Vec::new(),
on_executed: Vec::new(),
on_before_send: Vec::new(),
on_after_send: Vec::new(),
}
}
pub fn fn_received(mut self, f: fn(&CommandRequest)) -> Self {
self.on_received.push(f);
self
}
pub fn fn_executed(mut self, f: fn(&CommandResponse)) -> Self {
self.on_executed.push(f);
self
}
pub fn fn_before_send(mut self, f: fn(&mut CommandResponse)) -> Self {
self.on_before_send.push(f);
self
}
pub fn fn_after_send(mut self, f: fn()) -> Self {
self.on_after_send.push(f);
```

```
self

}

}
```

这样处理之后呢，Service 之前的 new() 方法就没有必要存在了，可以把它删除。同时，我们需要为 Service 类型提供一个 From<ServiceInner> 的实现：

```
impl<Store: Storage> From<ServiceInner<Store>> for Service<Store> {

fn from(inner: ServiceInner<Store>) -> Self {

Self {

inner: Arc::new(inner),

}

}

}
```

目前，代码中几处使用了 Service::new() 的地方需要改成使用 ServiceInner::new()，比如：

```
// 我们需要一个 service 结构至少包含 Storage

// let service = Service::new(MemTable::default());

let service: Service = ServiceInner::new(MemTable::default()).into();
```

全部改动完成后，代码可以编译通过。

然而，如果运行 cargo test，新加的测试会失败：

```
test service::tests::event_registration_should_work ... FAILED
```

这是因为，我们虽然完成了事件处理函数的注册，但现在还没有发事件通知。

另外因为我们的事件包括不可变事件（比如 on_received）和可变事件（比如 on_before_send），所以事件通知需要把二者分开。来定义两个 trait：Notify 和 NotifyMut：

```
/// 事件通知（不可变事件）

pub trait Notify<Arg> {

fn notify(&self, arg: &Arg);

}
```

```
/// 事件通知（可变事件）
pub trait NotifyMut<Arg> {
fn notify(&self, arg: &mut Arg);
}
```

这两个 trait 是泛型 trait，其中的 Arg 参数，对应事件注册函数里的 arg，比如：

```
fn(&CommandRequest);
```

由此，我们可以特地为 Vec<fn(&Arg)> 和 Vec<fn(&mut Arg)> 实现事件处理，它们涵盖了目前支持的几种事件：

```
impl<Arg> Notify<Arg> for Vec<fn(&Arg)> {
#[inline]
fn notify(&self, arg: &Arg) {
for f in self {
f(arg)
}
}
}
impl<Arg> NotifyMut<Arg> for Vec<fn(&mut Arg)> {
#[inline]
fn notify(&self, arg: &mut Arg) {
for f in self {
f(arg)
}
}
}
```

Notify / NotifyMut trait 实现好之后，我们就可以修改 execute() 方法了：

```
impl<Store: Storage> Service<Store> {
```

```
pub fn execute(&self, cmd: CommandRequest) -> CommandResponse {
debug!("Got request: {:?}", cmd);
self.inner.on_received.notify(&cmd);
let mut res = dispatch(cmd, &self.inner.store);
debug!("Executed response: {:?}", res);
self.inner.on_executed.notify(&res);
self.inner.on_before_send.notify(&mut res);
if !self.inner.on_before_send.is_empty() {
debug!("Modified response: {:?}", res);
}
res
}
}
```

现在，相应的事件就可以被通知到相应的处理函数中了。这个通知机制目前还是同步的函数调用，未来如果需要，我们可以将其改成消息传递，进行异步处理。

好，现在测试应该可以工作了，cargo test 所有的测试都通过。

为持久化数据库实现 Storage trait

到目前为止，我们的 KV store 还都是一个在内存中的 KV store。一旦终止应用程序，用户存储的所有 key / value 都会消失。我们希望存储能够持久化。

一个方案是为 MemTable 添加 WAL 和 disk snapshot 支持，让用户发送的所有涉及更新的命令都按顺序存储在磁盘上，同时定期做 snapshot，便于数据的快速恢复；另一个方案是使用已有的 KV store，比如 RocksDB，或者 sled。
RocksDB 是 Facebook 在 Google 的 levelDB 基础上开发的嵌入式 KV store，用 C++ 编写，而 sled 是 Rust 社区里涌现的优秀的 KV store，对标 RocksDB。二者功能很类似，从演示的角度，sled 使用起来更简单，更加适合今天的内容，如果在生产环境中使用，RocksDB 更加合适，因为它在各种复杂的生产环境中经历了千锤百炼。

所以，我们今天就尝试为 sled 实现 Storage trait，让它能够适配我们的 KV server。

首先在 Cargo.toml 里引入 sled：

```
sled = "0.34" # sled db
```

然后创建 src/storage/sleddb.rs，并添加如下代码：

```
use sled::{Db, IVec};
use std::{convert::TryInto, path::Path, str};
use crate::{KvError, Kvpair, Storage, StorageIter, Value};
#[derive(Debug)]
pub struct SledDb(Db);
impl SledDb {
pub fn new(path: impl AsRef<Path>) -> Self {
Self(sled::open(path).unwrap())
}
// 在 sleddb 里，因为它可以 scan_prefix，我们用 prefix
// 来模拟一个 table。当然，还可以用其它方案。
fn get_full_key(table: &str, key: &str) -> String {
format!("{}:{}", table, key)
}
// 遍历 table 的 key 时，我们直接把 prefix: 当成 table
fn get_table_prefix(table: &str) -> String {
format!("{}:", table)
}
}
/// 把 Option<Result<T, E>> flip 成 Result<Option<T>, E>
/// 从这个函数里，你可以看到函数式编程的优雅
fn flip<T, E>(x: Option<Result<T, E>>) -> Result<Option<T>, E> {
x.map_or(Ok(None), |v| v.map(Some))
}
impl Storage for SledDb {
```

```
fn get(&self, table: &str, key: &str) -> Result<Option<Value>, KvError> {
let name = SledDb::get_full_key(table, key);
let result = self.0.get(name.as_bytes())?.map(|v| v.as_ref().try_into());
flip(result)
}
fn set(&self, table: &str, key: String, value: Value) -> Result<Option<Value>, KvError> {
let name = SledDb::get_full_key(table, &key);
let data: Vec<u8> = value.try_into()?;
let result = self.0.insert(name, data)?.map(|v| v.as_ref().try_into());
flip(result)
}
fn contains(&self, table: &str, key: &str) -> Result<bool, KvError> {
let name = SledDb::get_full_key(table, &key);
Ok(self.0.contains_key(name)?)
}
fn del(&self, table: &str, key: &str) -> Result<Option<Value>, KvError> {
let name = SledDb::get_full_key(table, &key);
let result = self.0.remove(name)?.map(|v| v.as_ref().try_into());
flip(result)
}
fn get_all(&self, table: &str) -> Result<Vec<Kvpair>, KvError> {
let prefix = SledDb::get_table_prefix(table);
let result = self.0.scan_prefix(prefix).map(|v| v.into()).collect();
Ok(result)
}
fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError> {
```

```
let prefix = SledDb::get_table_prefix(table);
let iter = StorageIter::new(self.0.scan_prefix(prefix));
Ok(Box::new(iter))
}
}
impl From<Result<(IVec, IVec), sled::Error>> for Kvpair {
fn from(v: Result<(IVec, IVec), sled::Error>) -> Self {
match v {
Ok((k, v)) => match v.as_ref().try_into() {
Ok(v) => Kvpair::new(ivec_to_key(k.as_ref()), v),
Err(_) => Kvpair::default(),
},
_ => Kvpair::default(),
}
}
}
fn ivec_to_key(ivec: &[u8]) -> &str {
let s = str::from_utf8(ivec).unwrap();
let mut iter = s.split(":");
iter.next();
iter.next().unwrap()
}
```

这段代码主要就是在实现 Storage trait。每个方法都很简单，就是在 sled 提供的功能上增加了一次封装。如果你对代码中某个调用有疑虑，可以参考 sled 的文档。

在 src/storage/mod.rs 里引入 sleddb，我们就可以加上相关的测试，测试新的 Storage 实现啦：

```
mod sleddb;
```

```
pub use sleddb::SledDb;

#[cfg(test)]

mod tests {

use tempfile::tempdir;

use super::*;

...

#[test]

fn sleddb_basic_interface_should_work() {

let dir = tempdir().unwrap();

let store = SledDb::new(dir);

test_basi_interface(store);

}

#[test]

fn sleddb_get_all_should_work() {

let dir = tempdir().unwrap();

let store = SledDb::new(dir);

test_get_all(store);

}

#[test]

fn sleddb_iter_should_work() {

let dir = tempdir().unwrap();

let store = SledDb::new(dir);

test_get_iter(store);

}

}
```

因为 SledDb 创建时需要指定一个目录，所以要在测试中使用 tempfile 库，它能让文件资源在测试结束时被回收。我们在 Cargo.toml 中引入它：

```
[dev-dependencies]

...

tempfile = "3" # 处理临时目录和临时文件

...
```

代码目前就可以编译通过了。如果你运行 cargo test 测试，会发现所有测试都正常通过！

构建新的 KV server

现在完成了 SledDb 和事件通知相关的实现，我们可以尝试构建支持事件通知，并且使用 SledDb 的 KV server 了。把 examples/server.rs 拷贝出 examples/server_with_sled.rs，然后修改 let service 那一行：

```
// let service: Service = ServiceInner::new(MemTable::new()).into();

let service: Service<SledDb> = ServiceInner::new(SledDb::new("/tmp/kvserver"))

.fn_before_send(|res| match res.message.as_ref() {

"" => res.message = "altered. Original message is empty.".into(),

s => res.message = format!("altered: {}", s),

})

.into();
```

当然，需要引入 SledDb 让编译通过。你看，只需要在创建 KV server 时使用 SledDb，就可以实现 data store 的切换，未来还可以进一步通过配置文件，来选择使用什么样的 store。非常方便。

新的 examples/server_with_sled.rs 的完整的代码：

```
use anyhow::Result;

use async_prost::AsyncProstStream;

use futures::prelude::*;

use kv1::{CommandRequest, CommandResponse, Service, ServiceInner, SledDb};

use tokio::net::TcpListener;

use tracing::info;
```

```
#[tokio::main]
async fn main() -> Result<()> {
tracing_subscriber::fmt::init();
let service: Service<SledDb> = ServiceInner::new(SledDb::new("/tmp/kvserver"))
.fn_before_send(|res| match res.message.as_ref() {
"" => res.message = "altered. Original message is empty.".into(),
s => res.message = format!("altered: {}", s),
})
.into();
let addr = "127.0.0.1:9527";
let listener = TcpListener::bind(addr).await?;
info!("Start listening on {}", addr);
loop {
let (stream, addr) = listener.accept().await?;
info!("Client {:?} connected", addr);
let svc = service.clone();
tokio::spawn(async move {
let mut stream =
AsyncProstStream::<_, CommandRequest, CommandResponse,
_>::from(stream).for_async();
while let Some(Ok(cmd)) = stream.next().await {
info!("Got a new command: {:?}", cmd);
let res = svc.execute(cmd);
stream.send(res).await.unwrap();
}
info!("Client {:?} disconnected", addr);
```

```
});

}

}
```

它和之前的 server 几乎一样，只有 11 行生成 service 的代码应用了新的 storage，并且引入了事件通知。

完成之后，我们可以打开一个命令行窗口，运行：RUST_LOG=info cargo run --example server_with_sled --quiet。然后在另一个命令行窗口，运行：RUST_LOG=info cargo run --example client --quiet。

此时，服务器和客户端都收到了彼此的请求和响应，并且处理正常。如果你停掉服务器，再次运行，然后再运行客户端，会发现，客户端在尝试 HSET 时得到了服务器旧的值，我们的新版 KV server 可以对数据进行持久化了。

此外，如果你注意看 client 的日志，会发现原本应该是空字符串的 messag 包含了“altered. Original message is empty.”：

```
› RUST_LOG=info cargo run --example client --quiet

Sep 23 22:09:12.215 INFO client: Got response CommandResponse { status: 200, message: "altered. Original message is empty.", values: [Value { value: Some(String("world")) }], pairs: [] }
```

这是因为，我们的服务器注册了 fn_before_send 的事件通知，对返回的数据做了修改。未来我们可以用这些事件做很多事情，比如监控数据的发送，甚至写 WAL。

小结

今天的课程我们进一步认识到了 trait 的威力。当为系统设计了合理的 trait ，整个系统的可扩展性就大大增强，之后在添加新的功能的时候，并不需要改动多少已有的代码。

在使用 trait 做抽象时，我们要衡量，这么做的好处是什么，它未来可以为实现者带来什么帮助。就像我们撰写的 StorageIter，它实现了 Iterator trait，并封装了 map 的处理逻辑，让这个公共的步骤可以在 Storage trait 中复用。

除此之外，也进一步熟悉了如何为带泛型参数的数据结构实现 trait。我们不仅可以为具体的数据结构实现 trait，也可以为更笼统的泛型参数实现 trait。除了文中这个例子：

```
impl<Arg> Notify<Arg> for Vec<fn(&Arg)> {

#[inline]

fn notify(&self, arg: &Arg) {

for f in self {
```

```
f(arg)

}

}

}
```

其实之前还见到过：

```
impl<T, U> Into<U> for T where U: From<T>,

{

fn into(self) -> U {

U::from(self)

}

}
```

也是一样的道理。

如果结合这一讲和第 21、22讲，你会发现，我们目前完成了一个功能比较完整的 KV server 的核心逻辑，但是，整体的代码似乎没有太多复杂的生命周期标注，或者太过抽象的泛型结构。
是的，别看我们在介绍 Rust 的基础知识时，扎的比较深，但是大多数写代码的时候，并不会用到那么深的知识。Rust 编译器会尽最大的努力，让你的代码简单。如果你用 clippy 这样的 linter 的话，它还会进一步给你提一些建议，让你的代码更加简单。

那么，为什么我们还要讲那么深入呢？

这是因为我们在写代码的时候不可避免地要引入第三方库，你也看到了，在写这个项目的时候用了不少依赖，当你使用这些库的时候，又不可避免地要阅读一些它们的源码，而这些源码，可能有各种各样复杂的写法。这也是为什么在开头我会说，现阶段能看懂包含泛型的代码就可以了。

深入地了解 Rust 的基础知识，可以帮我们更快更清晰地阅读源码，而更快更清晰地读懂别人的源码，又可以更快地帮助我们用好别人的库，从而写好我们的代码。

思考题

如果你在 21 讲已经完成了 KV server 其它的 6 个命令，可以对照着我在 GitHub repo 里的代码和测试，看看你写的结果。
我们的 Notify 和 NotifyMut trait 目前只能做到通知，无法告诉 execute 提前结束处理并直接给客户端返回错误。试着修改一下这两个 trait，让它具备提前结束整个 pipeline 的能力。

RocksDB 是一个非常优秀的 KV DB，它有对应的 rust 库。尝试着为 RocksDB 实现 Storage trait，然后写个 example server 应用它。
感谢你的收听，你已经完成了 Rust 学习的第 26 次打卡，如果你觉得有收获，也欢迎你分享给身边的朋友，邀他一起讨论。我们下节课见~

给文章提建议


良师益友

Command + Enter 发表

0/2000字

提交留言