



下载APP



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

2021-10-27 陈天

《陈天 · Rust 编程第一课》

[课程介绍 >](#)**讲述：陈天**

时长 16:06 大小 14.75M



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：

1. Storage trait 的 `get_iter()` 方法没有实现；
2. 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：

[复制代码](#)

```
1  #[test]
2  fn memtable_iter_should_work() {
3      let store = MemTable::new();
4      test_get_iter(store);
5  }
```

然后在 `src/storge/memory.rs` 里尝试实现它。

[复制代码](#)

```
1  impl Storage for MemTable {
2      ...
3      fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>
4          // 使用 clone() 来获取 table 的 snapshot
5          let table = self.get_or_create_table(table).clone();
6          let iter = table
7              .iter()
8              .map(|v| Kvpair::new(v.key(), v.value().clone()));
9          Ok(Box::new(iter)) // <-- 编译出错
10     }
11 }
```

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](#)）：

[📄 复制代码](#)

```
1 pub trait IntoIterator {  
2     type Item;  
3     type IntoIter: Iterator<Item = Self::Item>;  
4  
5     fn into_iter(self) -> Self::IntoIter;  
6 }
```

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`：

[📄 复制代码](#)

```
1 impl Storage for MemTable {  
2     ...  
3     fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>  
4         // 使用 clone() 来获取 table 的 snapshot  
5         let table = self.get_or_create_table(table).clone();  
6         let iter = table.into_iter().map(|data| data.into());  
7         Ok(Box::new(iter))  
8     }  
9 }
```

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 `From` trait：

[📄 复制代码](#)

```
1 impl From<(String, Value)> for Kvpair {  
2     fn from(data: (String, Value)) -> Self {  
3         Kvpair::new(data.0, data.1)  
4     }  
5 }
```

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：

1. 拿到一个关于某个 table 下的拥有所有权的 Iterator
2. 对 Iterator 做 map
3. 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`：

 复制代码

```
1  /// 提供 Storage iterator，这样 trait 的实现者只需要
2  /// 把它们的 iterator 提供给 StorageIter，然后它们保证
3  /// next() 传出的类型实现了 Into<Kvpair> 即可
4  pub struct StorageIter<T> {
5      data: T,
6  }
7
8  impl<T> StorageIter<T> {
9      pub fn new(data: T) -> Self {
10         Self { data }
11     }
12 }
13
14 impl<T> Iterator for StorageIter<T>
15 where
16     T: Iterator,
17     T::Item: Into<Kvpair>,
18 {
19     type Item = Kvpair;
20
21     fn next(&mut self) -> Option<Self::Item> {
22         self.data.next().map(|v| v.into())
23     }
24 }
```

```
23     }  
24 }
```

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：

[复制代码](#)

```
1 impl Storage for MemTable {  
2     ...  
3     fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>  
4         // 使用 clone() 来获取 table 的 snapshot  
5         let table = self.get_or_create_table(table).clone();  
6         let iter = StorageIter::new(table.into_iter()); // 这行改掉了  
7         Ok(Box::new(iter))  
8     }  
9 }
```

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 的例子中，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：

[复制代码](#)

```
1 pub fn execute(&self, cmd: CommandRequest) -> CommandResponse {  
2     debug!("Got request: {:?}", cmd);  
3     // TODO: 发送 on_received 事件  
4     let res = dispatch(cmd, &self.inner.store);  
5     debug!("Executed response: {:?}", res);  
6     // TODO: 发送 on_executed 事件  
7 }
```



```
8     res
9 }
```

为了解决这些 TODO，我们需要提供事件通知的机制：

1. 在创建 Service 时，注册相应的事件处理函数；
2. 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：


```
1  /// Service 内部数据结构
2  pub struct ServiceInner<Store> {
3      store: Store,
4      on_received: Vec<fn(&CommandRequest)>,
5      on_executed: Vec<fn(&CommandResponse)>,
6      on_before_send: Vec<fn(&mut CommandResponse)>,
7      on_after_send: Vec<fn()>,
8  }
```

[复制代码](#)

按照 21 讲的设计，我们提供了四个事件：

1. on\_received：当服务器收到 CommandRequest 时触发；
2. on\_executed：当服务器处理完 CommandRequest 得到 CommandResponse 时触发；
3. 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。
4. 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：

 复制代码


```

1  #[test]
2  fn event_registration_should_work() {
3      fn b(cmd: &CommandRequest) {
4          info!("Got {:?}", cmd);
5      }
6      fn c(res: &CommandResponse) {
7          info!("{:?}", res);
8      }
9      fn d(res: &mut CommandResponse) {
10         res.status = StatusCode::CREATED.as_u16() as _;
11     }
12     fn e() {
13         info!("Data is sent");
14     }
15
16     let service: Service = ServiceInner::new(MemTable::default())
17         .fn_received(|_: &CommandRequest| {})
18         .fn_received(b)
19         .fn_executed(c)
20         .fn_before_send(d)
21         .fn_after_send(e)
22         .into();
23
24     let res = service.execute(CommandRequest::new_hset("t1", "k1", "v1".into())
25     assert_eq!(res.status, StatusCode::CREATED.as_u16() as _);
26     assert_eq!(res.message, "");
27     assert_eq!(res.values, vec![Value::default()]);
28 }

```

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：

 复制代码

```

1  impl<Store: Storage> ServiceInner<Store> {
2      pub fn new(store: Store) -> Self {
3          Self {
4              store,

```

```

5         on_received: Vec::new(),
6         on_executed: Vec::new(),
7         on_before_send: Vec::new(),
8         on_after_send: Vec::new(),
9     }
10 }
11
12 pub fn fn_received(mut self, f: fn(&CommandRequest)) -> Self {
13     self.on_received.push(f);
14     self
15 }
16
17 pub fn fn_executed(mut self, f: fn(&CommandResponse)) -> Self {
18     self.on_executed.push(f);
19     self
20 }
21
22 pub fn fn_before_send(mut self, f: fn(&mut CommandResponse)) -> Self {
23     self.on_before_send.push(f);
24     self
25 }
26
27 pub fn fn_after_send(mut self, f: fn()) -> Self {
28     self.on_after_send.push(f);
29     self
30 }
31 }

```

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：

[复制代码](#)

```

1 impl<Store: Storage> From<ServiceInner<Store>> for Service<Store> {
2     fn from(inner: ServiceInner<Store>) -> Self {
3         Self {
4             inner: Arc::new(inner),
5         }
6     }
7 }

```

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：

[复制代码](#)

```

1 // 我们需要一个 service 结构至少包含 Storage

```



```
2 // let service = Service::new(MemTable::default());
3 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：

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

[复制代码](#)

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：

```
1 /// 事件通知（不可变事件）
2 pub trait Notify<Arg> {
3     fn notify(&self, arg: &Arg);
4 }
5
6 /// 事件通知（可变事件）
7 pub trait NotifyMut<Arg> {
8     fn notify(&self, arg: &mut Arg);
9 }
```


[复制代码](#)

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：

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


[复制代码](#)

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：

 复制代码

```
1 impl<Arg> Notify<Arg> for Vec<fn(&Arg)> {
2     #[inline]
3     fn notify(&self, arg: &Arg) {
4         for f in self {
5             f(arg)
6         }
7     }
8 }
9
10 impl<Arg> NotifyMut<Arg> for Vec<fn(&mut Arg)> {
11     #[inline]
12     fn notify(&self, arg: &mut Arg) {
13         for f in self {
14             f(arg)
15         }
16     }
17 }
```

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

 复制代码

```
1 impl<Store: Storage> Service<Store> {
2     pub fn execute(&self, cmd: CommandRequest) -> CommandResponse {
3         debug!("Got request: {:?}", cmd);
4         self.inner.on_received.notify(&cmd);
5         let mut res = dispatch(cmd, &self.inner.store);
6         debug!("Executed response: {:?}", res);
7         self.inner.on_executed.notify(&res);
8         self.inner.on_before_send.notify(&mut res);
9         if !self.inner.on_before_send.is_empty() {
10             debug!("Modified response: {:?}", res);
11         }
12
13         res
14     }
15 }
```

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：

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

[复制代码](#)

然后创建 src/storage/sleddb.rs，并添加如下代码：

```
1 use sled::{Db, IVec};
2 use std::{convert::TryInto, path::Path, str};
3
4 use crate::{KvError, Kvpair, Storage, StorageIter, Value};
5
6 #[derive(Debug)]
7 pub struct SledDb(Db);
8
9 impl SledDb {
10     pub fn new(path: impl AsRef<Path>) -> Self {
11         Self(sled::open(path).unwrap())
12     }
13
14     // 在 sleddb 里，因为它可以 scan_prefix，我们用 prefix
15     // 来模拟一个 table。当然，还可以用其它方案。
16     fn get_full_key(table: &str, key: &str) -> String {
17         format!("{:}:{:}", table, key)
18     }
19 }
```

[复制代码](#)

```
19
20 // 遍历 table 的 key 时,我们直接把 prefix: 当成 table
21 fn get_table_prefix(table: &str) -> String {
22     format!("{}", table)
23 }
24 }
25
26 /// 把 Option<Result<T, E>> flip 成 Result<Option<T>, E>
27 /// 从这个函数里,你可以看到函数式编程的优雅
28 fn flip<T, E>(x: Option<Result<T, E>>) -> Result<Option<T>, E> {
29     x.map_or(Ok(None), |v| v.map(Some))
30 }
31
32 impl Storage for SledDb {
33     fn get(&self, table: &str, key: &str) -> Result<Option<Value>, KvError> {
34         let name = SledDb::get_full_key(table, key);
35         let result = self.0.get(name.as_bytes())?.map(|v| v.as_ref().try_into())
36         flip(result)
37     }
38
39     fn set(&self, table: &str, key: String, value: Value) -> Result<Option<Value>, KvError> {
40         let name = SledDb::get_full_key(table, &key);
41         let data: Vec<u8> = value.try_into()?;
42
43         let result = self.0.insert(name, data)?.map(|v| v.as_ref().try_into())
44         flip(result)
45     }
46
47     fn contains(&self, table: &str, key: &str) -> Result<bool, KvError> {
48         let name = SledDb::get_full_key(table, &key);
49
50         Ok(self.0.contains_key(name)?)
51     }
52
53     fn del(&self, table: &str, key: &str) -> Result<Option<Value>, KvError> {
54         let name = SledDb::get_full_key(table, &key);
55
56         let result = self.0.remove(name)?.map(|v| v.as_ref().try_into())
57         flip(result)
58     }
59
60     fn get_all(&self, table: &str) -> Result<Vec<Kvpair>, KvError> {
61         let prefix = SledDb::get_table_prefix(table);
62         let result = self.0.scan_prefix(prefix).map(|v| v.into()).collect();
63
64         Ok(result)
65     }
66
67     fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError> {
68         let prefix = SledDb::get_table_prefix(table);
69         let iter = StorageIter::new(self.0.scan_prefix(prefix));
70         Ok(Box::new(iter))
```

```

71     }
72 }
73
74 impl From<Result<(Ivec, Ivec), sled::Error>> for Kvpair {
75     fn from(v: Result<(Ivec, Ivec), sled::Error>) -> Self {
76         match v {
77             Ok((k, v)) => match v.as_ref().try_into() {
78                 Ok(v) => Kvpair::new(ivec_to_key(k.as_ref()), v),
79                 Err(_) => Kvpair::default(),
80             },
81             _ => Kvpair::default(),
82         }
83     }
84 }
85
86 fn ivec_to_key(ivec: &[u8]) -> &str {
87     let s = str::from_utf8(ivec).unwrap();
88     let mut iter = s.split(":");
89     iter.next();
90     iter.next().unwrap()
91 }

```

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：

```

1 mod sleddb;
2
3 pub use sleddb::SledDb;
4
5 #[cfg(test)]
6 mod tests {
7     use tempfile::tempdir;
8
9     use super::*;
10
11     ...
12
13     #[test]
14     fn sleddb_basic_interface_should_work() {
15         let dir = tempdir().unwrap();
16         let store = SledDb::new(dir);
17         test_basi_interface(store);
18     }

```

[复制代码](#)

```

19     #[test]
20     fn sleddb_get_all_should_work() {
21         let dir = tempdir().unwrap();
22         let store = SledDb::new(dir);
23         test_get_all(store);
24     }
25
26     #[test]
27     fn sleddb_iter_should_work() {
28         let dir = tempdir().unwrap();
29         let store = SledDb::new(dir);
30         test_get_iter(store);
31     }
32 }
33

```

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：

[复制代码](#)

```

1 [dev-dependencies]
2 ...
3 tempfile = "3" # 处理临时目录和临时文件
4 ...

```

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：

[复制代码](#)

```

1 // let service: Service = ServiceInner::new(MemTable::new()).into();
2 let service: Service<SledDb> = ServiceInner::new(SledDb::new("/tmp/kvserver"))
3     .fn_before_send(|res| match res.message.as_ref() {
4         "" => res.message = "altered. Original message is empty.".into(),
5         s => res.message = format!("altered: {}", s),
6     })
7     .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：

[复制代码](#)

```
1 use anyhow::Result;
2 use async_prost::AsyncProstStream;
3 use futures::prelude::*;
4 use kv1::{CommandRequest, CommandResponse, Service, ServiceInner, SledDb};
5 use tokio::net::TcpListener;
6 use tracing::info;
7
8 #[tokio::main]
9 async fn main() -> Result<()> {
10     tracing_subscriber::fmt::init();
11     let service: Service<SledDb> = ServiceInner::new(SledDb::new("/tmp/kvserve
12         .fn_before_send(|res| match res.message.as_ref() {
13             "" => res.message = "altered. Original message is empty.".into(),
14             s => res.message = format!("altered: {}", s),
15         })
16         .into());
17     let addr = "127.0.0.1:9527";
18     let listener = TcpListener::bind(addr).await?;
19     info!("Start listening on {}", addr);
20     loop {
21         let (stream, addr) = listener.accept().await?;
22         info!("Client {:?} connected", addr);
23         let svc = service.clone();
24         tokio::spawn(async move {
25             let mut stream =
26                 AsyncProstStream::<_, CommandRequest, CommandResponse, _>::fro
27             while let Some(Ok(cmd)) = stream.next().await {
28                 info!("Got a new command: {:?}", cmd);
29                 let res = svc.execute(cmd);
30                 stream.send(res).await.unwrap();
31             }
32             info!("Client {:?} disconnected", addr);
33         });
34     }
35 }
```

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 message 包含了“altered. Original message is empty.”：

[复制代码](#)

```
1 > RUST_LOG=info cargo run --example client --quiet
2 Sep 23 22:09:12.215 INFO client: Got response CommandResponse { status: 200,
```

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：

[复制代码](#)

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

```
2     #[inline]
3     fn notify(&self, arg: &Arg) {
4         for f in self {
5             f(arg)
6         }
7     }
8 }
```

其实之前还见到过：

```
1 impl<T, U> Into<U> for T where U: From<T>,
2 {
3     fn into(self) -> U {
4         U::from(self)
5     }
6 }
```

[复制代码](#)

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

1. 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。
2. 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。
3. [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~

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 3     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](#)    [加餐 | Rust 2021 版次问世了！](#)

1024 活动特惠

VIP 年卡直降 ¥2000

新课上线即解锁，享 365 天畅看全场

超值拿下 ¥999



精选留言 (1)

写留言



罗杰

2021-10-27

哈 昨天终于完成 21 , 22 , 今天继续  
展开



1