



下载APP

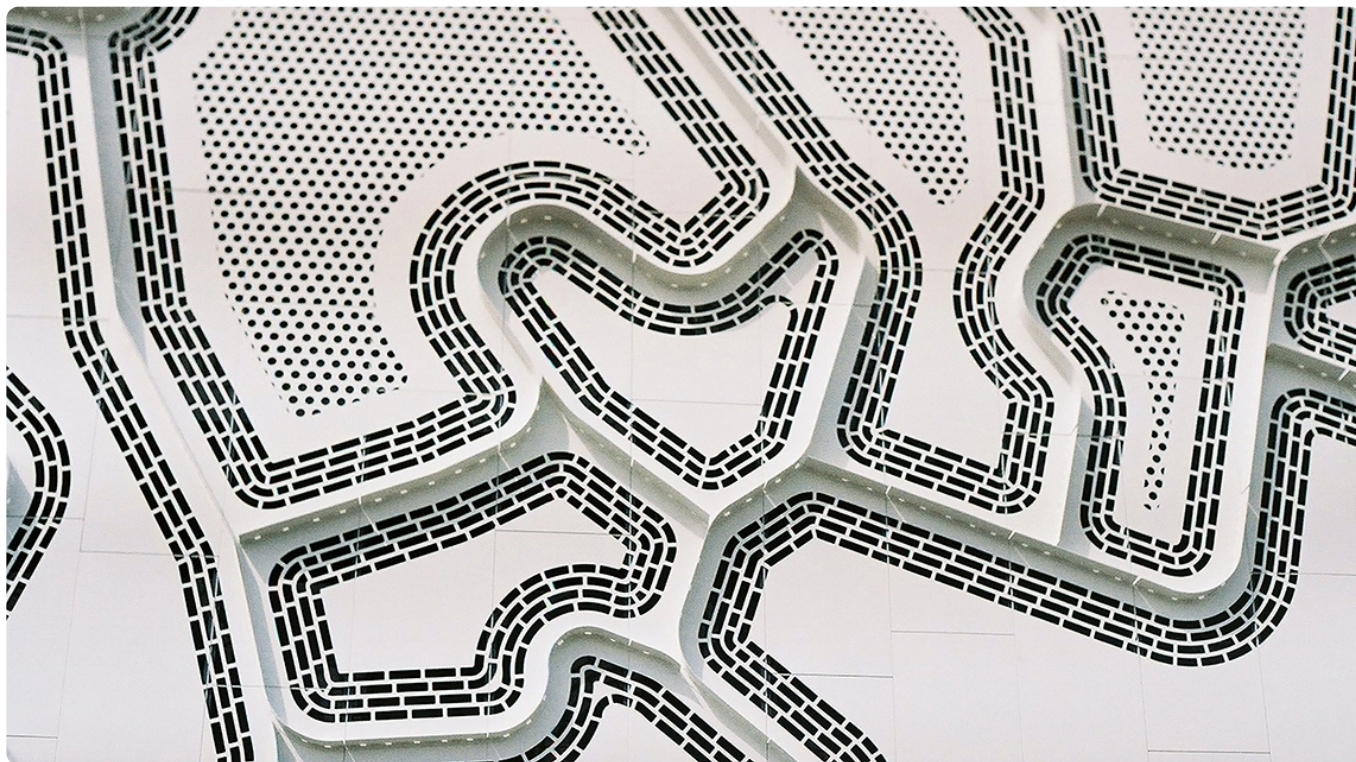


41 | 阶段实操（5）：构建一个简单的KV server-异步处理

2021-12-03 陈天

《陈天 · Rust 编程第一课》

课程介绍 >



讲述：陈天

时长 11:20 大小 10.38M



你好，我是陈天。

到目前为止，我们已经一起完成了一个相对完善的 KV server。还记得是怎么一步步构建这个服务的么？

基础篇学完，我们搭好了 KV server 的基础功能（[21 讲](#)、[22 讲](#)），构造了客户端和服务端间交互的 protobuf，然后设计了 CommandService trait 和 Storage trait，分别处理客户端命令和存储。



在进阶篇掌握了 trait 的实战使用技巧之后，（[26 讲](#)）我们进一步构造了 Service 数据结构，接收 CommandRequest，根据其类型调用相应的 CommandService 处理，并做合适的事件通知，最后返回 CommandResponse。

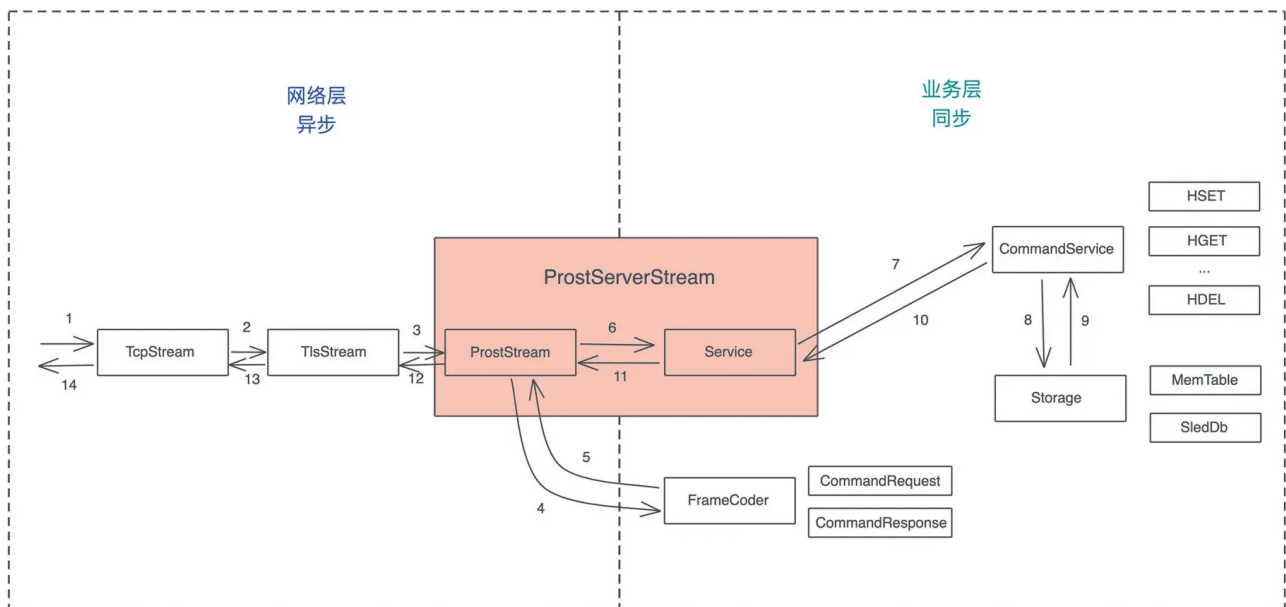
但所有这一切都发生在同步的世界：不管数据是怎么获得的，数据已经在那里，我们需要做的就是把一种数据类型转换成另一种数据类型的运算而已。

之后我们涉足网络的世界。（[🔗36讲](#)）为 KV server 构造了自己的 frame：一个包含长度和是否压缩的信息的 4 字节的头，以及实际的 payload；还设计了一个 FrameCoder 来对 frame 进行封包和拆包，这为接下来构造网络接口打下了坚实的基础。考虑到网络安全，（[🔗37讲](#)）我们提供了 TLS 的支持。

在构建 ProstStream 的时候，我们开始处理异步：ProstStream 内部的 stream 需要支持 AsyncRead + AsyncWrite，这可以让 ProstStream 适配包括 TcpStream 和 TlsStream 在内的一切实现了 AsyncRead 和 AsyncWrite 的异步网络接口。


至此，我们打通了从远端得到一个命令，历经 TCP、TLS，然后被 FrameCoder 解出来一个 CommandRequest，交由 Service 来处理的过程。**把同步世界和异步世界连接起来的，就是 ProstServerStream 这个结构。**

这个从收包处理到处理完成后发包的完整流程和系统结构，可以看下图：



今天做什么？


虽然我们很早就已经撰写了不少异步或者和异步有关的代码。但是最能体现 Rust 异步本质的 `poll()`、`poll_read()`、`poll_next()` 这样的处理函数还没有怎么写过，之前测试异步的 `read_frame()` 写过一个 `DummyStream`，算是体验了一下底层的异步处理函数的复杂接口。不过在 `DummyStream` 里，我们并没有做任何复杂的动作：

 复制代码

```
1 struct DummyStream {
2     buf: BytesMut,
3 }
4
5 impl AsyncRead for DummyStream {
6     fn poll_read(
7         self: std::pin::Pin<&mut Self>,
8         _cx: &mut std::task::Context<'_,>,
9         buf: &mut tokio::io::ReadBuf<'_,>,
10    ) -> std::task::Poll<std::io::Result<()>> {
11        // 看看 ReadBuf 需要多大的数据
12        let len = buf.capacity();
13        // split 出这么大的数据
14        let data = self.get_mut().buf.split_to(len);
15        // 拷贝给 ReadBuf
16        buf.put_slice(&data);
17        // 直接完工
18        std::task::Poll::Ready(Ok(()))
19    }
20 }
```

上一讲我们学习了异步 IO，这节课我们就学以致用，对现有的代码做些重构，让核心的 `ProstStream` 更符合 Rust 的异步 IO 接口逻辑。具体要做点什么呢？

看之前写的 `ProstServerStream` 的 `process()` 函数，比较一下它和 `async_prost` 库的 `AsyncProst` 的调用逻辑：

 复制代码

```
1 // process() 函数的内在逻辑
2 while let Ok(cmd) = self.recv().await {
3     info!("Got a new command: {:?}", cmd);
4     let res = self.service.execute(cmd);
5     self.send(res).await?;
6 }
7
8 // async_prost 库的 AsyncProst 的调用逻辑
9 while let Some(Ok(cmd)) = stream.next().await {
```

```
10     info!("Got a new command: {:?}", cmd);
11     let res = svc.execute(cmd);
12     stream.send(res).await.unwrap();
13 }
```

可以看到由于 AsyncProst 实现了 `Stream` 和 `Sink`，能更加自然地调用 `StreamExt` trait 的 `next()` 方法和 `SinkExt` trait 的 `send()` 方法，来处理数据的收发，而 `ProstServerStream` 则自己额外实现了函数 `recv()` 和 `send()`。

虽然从代码对比的角度，这两段代码几乎一样，但未来的可扩展性，和整个异步生态的融洽性上，`AsyncProst` 还是更胜一筹。

所以今天我们就构造一个 `ProstStream` 结构，让它实现 `Stream` 和 `Sink` 这两个 trait，然后让 `ProstServerStream` 和 `ProstClientStream` 使用它。

创建 ProstStream

在开始重构之前，先来简单复习一下 `Stream` trait 和 `Sink` trait：

[复制代码](#)

```
1 // 可以类比 Iterator
2 pub trait Stream {
3     // 从 Stream 中读取到的数据类型
4     type Item;
5
6     // 从 stream 里读取下一个数据
7     fn poll_next(
8         self: Pin<&mut Self>, cx: &mut Context<'_>
9     ) -> Poll<Option<Self::Item>>;
10 }
11
12 //
13 pub trait Sink<Item> {
14     type Error;
15     fn poll_ready(
16         self: Pin<&mut Self>,
17         cx: &mut Context<'_>
18     ) -> Poll<Result<(), Self::Error>>;
19     fn start_send(self: Pin<&mut Self>, item: Item) -> Result<(), Self::Error>
20     fn poll_flush(
21         self: Pin<&mut Self>,
22         cx: &mut Context<'_>
23     ) -> Poll<Result<(), Self::Error>>;
24     fn poll_close(
```



```
25     self: Pin<&mut Self>,
26     cx: &mut Context<'_>
27 ) -> Poll<Result<(), Self::Error>>;
28 }
```

那么 ProstStream 具体需要包含什么类型呢？

因为它的主要职责是从底下的 stream 中读取或者发送数据，所以一个支持 AsyncRead 和 AsyncWrite 的泛型参数 S 是必然需要的。

另外 Stream trait 和 Sink 都各需要一个 Item 类型，对于我们的系统来说，Item 是 CommandRequest 或者 CommandResponse，但为了灵活性，我们可以用 In 和 Out 这两个泛型参数来表示。

当然，在处理 Stream 和 Sink 时还需要 read buffer 和 write buffer。

综上所述，我们的 ProstStream 结构看上去是这样子的：

```
1 pub struct ProstStream<S, In, Out> {
2     // inner stream
3     stream: S,
4     // 写缓存
5     wbuf: BytesMut,
6     // 读缓存
7     rbuf: BytesMut,
8 }
```

[复制代码](#)

然而，Rust 不允许数据结构有超出需要的泛型参数。怎么办？别急，可以用 [PhantomData](#)，之前讲过它是一个零字节大小的占位符，可以让我们的数据结构携带未使用的泛型参数。

好，现在有足够的思路了，我们创建 src/network/stream.rs，添加如下代码（记得在 src/network/mod.rs 添加对 [stream.rs](#) 的引用）：

```
1 use bytes::BytesMut;
```

[复制代码](#)

```
2 use futures::{Sink, Stream};
3 use std::{
4     marker::PhantomData,
5     pin::Pin,
6     task::{Context, Poll},
7 };
8 use tokio::io::{AsyncRead, AsyncWrite};
9
10 use crate::{FrameCoder, KvError};
11
12 /// 处理 KV server prost frame 的 stream
13 pub struct ProstStream<S, In, Out> where {
14     // innner stream
15     stream: S,
16     // 写缓存
17     wbuf: BytesMut,
18     // 读缓存
19     rbuf: BytesMut,
20
21     // 类型占位符
22     _in: PhantomData<In>,
23     _out: PhantomData<Out>,
24 }
25
26 impl<S, In, Out> Stream for ProstStream<S, In, Out>
27 where
28     S: AsyncRead + AsyncWrite + Unpin + Send,
29     In: Unpin + Send + FrameCoder,
30     Out: Unpin + Send,
31 {
32     /// 当调用 next() 时,得到 Result<In, KvError>
33     type Item = Result<In, KvError>;
34
35     fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Option<Se
36         todo!()
37     }
38 }
39
40 /// 当调用 send() 时,会把 Out 发出去
41 impl<S, In, Out> Sink<Out> for ProstStream<S, In, Out>
42 where
43     S: AsyncRead + AsyncWrite + Unpin,
44     In: Unpin + Send,
45     Out: Unpin + Send + FrameCoder,
46 {
47     /// 如果发送出错,会返回 KvError
48     type Error = KvError;
49
50     fn poll_ready(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Result<(
51         todo!()
52     )>
53 }
```

```

54     fn start_send(self: Pin<&mut Self>, item: Out) -> Result<(), Self::Error>
55         todo!()
56     }
57
58     fn poll_flush(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>>
59         todo!()
60     }
61
62     fn poll_close(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>>
63         todo!()
64     }
65 }

```

这段代码包含了为 ProstStream 实现 Stream 和 Sink 的骨架代码。接下来我们就一个个处理。注意对于 In 和 Out 参数，还为其约束了 FrameCoder，这样，在实现里我们可以使用 decode_frame() 和 encode_frame() 来获取一个 Item 或者 encode 一个 Item。

Stream 的实现

先来实现 Stream 的 poll_next() 方法。

poll_next() 可以直接调用我们之前写好的 read_frame()，然后再用 decode_frame() 来解包：

 复制代码

```

1 fn poll_next(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Option<Self::Item>>
2     // 上一次调用结束后 rbuf 应该为空
3     assert!(self.rbuf.len() == 0);
4
5     // 从 rbuf 中分离出 rest (摆脱对 self 的引用)
6     let mut rest = self.rbuf.split_off(0);
7
8     // 使用 read_frame 来获取数据
9     let fut = read_frame(&mut self.stream, &mut rest);
10    ready!(Box::pin(fut).poll_unpin(cx))?;
11
12    // 拿到一个 frame 的数据, 把 buffer 合并回去
13    self.rbuf.unsplit(rest);
14
15    // 调用 decode_frame 获取解包后的数据
16    Poll::Ready(Some(In::decode_frame(&mut self.rbuf)))
17 }

```

这个不难理解，但中间这段需要稍微解释一下：

[复制代码](#)

```
1 // 使用 read_frame 来获取数据
2 let fut = read_frame(&mut self.stream, &mut rest);
3 ready!(Box::pin(fut).poll_unpin(cx));
```

因为 `poll_xxx()` 方法已经是 `async/await` 的底层 API 实现，所以我们在 `poll_xxx()` 方法中，是不能直接使用异步函数的，需要把它看作一个 `future`，然后调用 `future` 的 `poll` 函数。因为 `future` 是一个 `trait`，所以需要 `Box` 将其处理成一个在堆上的 `trait object`，这样就可以调用 `FutureExt` 的 `poll_unpin()` 方法了。`Box::pin` 会生成 `Pin<Box>`。

至于 `ready!` 宏，它会在 `Pending` 时直接 `return Pending`，而在 `Ready` 时，返回 `Ready` 的值：

[复制代码](#)

```
1 macro_rules! ready {
2     ($e:expr $(,)? ) => {
3         match $e {
4             $crate::task::Poll::Ready(t) => t,
5             $crate::task::Poll::Pending => return $crate::task::Poll::Pending,
6         }
7     };
8 }
```

Stream 我们就实现好了，是不是也没有那么复杂？

Sink 的实现

再写 Sink，看上去要实现好几个方法，其实也不算复杂。四个方法 `poll_ready`、`start_send()`、`poll_flush` 和 `poll_close` 我们再回顾一下。

`poll_ready()` 是做背压的，你可以根据负载来决定要不要返回 `Poll::Ready`。对于我们的网络层来说，可以先不关心背压，依靠操作系统的 TCP 协议栈提供背压处理即可，所以这里直接返回 `Poll::Ready(Ok(()))`，也就是说，上层想写数据，可以随时写。


```
1 fn poll_ready(self: Pin<&mut Self>, _cx: &mut Context<'_>) -> Poll<Result<(),  
2     Poll::Ready(Ok(()))  
3 }
```

[复制代码](#)

当 `poll_ready()` 返回 `Ready` 后, Sink 就走到 [start_send\(\)](#)。我们在 `start_send()` 里就把必要的数 据准备好。这里把 `item` 封包成字节流, 存入 `wbuf` 中:

```
1 fn start_send(self: Pin<&mut Self>, item: Out) -> Result<(), Self::Error> {  
2     let this = self.get_mut();  
3     item.encode_frame(&mut this.wbuf)?;  
4  
5     Ok(())  
6 }
```

[复制代码](#)

然后在 [poll_flush\(\)](#) 中, 我们开始写数据。这里需要记录当前写到哪里, 所以需要在 `ProstStream` 里加一个字段 `written`, 记录写入了多少字节:

```
1 /// 处理 KV server prost frame 的 stream  
2 pub struct ProstStream<S, In, Out> {  
3     // innner stream  
4     stream: S,  
5     // 写缓存  
6     wbuf: BytesMut,  
7     // 写入了多少字节  
8     written: usize,  
9     // 读缓存  
10    rbuf: BytesMut,  
11  
12    // 类型占位符  
13    _in: PhantomData<In>,  
14    _out: PhantomData<Out>,  
15 }
```

[复制代码](#)

有了这个 `written` 字段, 就可以循环写入:

```
1 fn poll_flush(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Result<(), S  
2     let this = self.get_mut();  
3
```

[复制代码](#)

```

4      // 循环写入 stream 中
5      while this.written != this.wbuf.len() {
6          let n = ready!(Pin::new(&mut this.stream).poll_write(cx, &this.wbuf[this.written..this.wbuf.len() - 1]));
7          this.written += n;
8      }
9
10     // 清除 wbuf
11     this.wbuf.clear();
12     this.written = 0;
13
14     // 调用 stream 的 poll_flush 确保写入
15     ready!(Pin::new(&mut this.stream).poll_flush(cx)?);
16     Poll::Ready(Ok(()))
17 }

```

最后是 `poll_close()`，我们只需要调用 stream 的 flush 和 shutdown 方法，确保数据写完并且 stream 关闭：

[复制代码](#)

```

1 fn poll_close(mut self: Pin<&mut Self>, cx: &mut Context<'_,_>) -> Poll<Result<(), Error>> {
2     // 调用 stream 的 poll_flush 确保写入
3     ready!(self.as_mut().poll_flush(cx))?;
4
5     // 调用 stream 的 poll_shutdown 确保 stream 关闭
6     ready!(Pin::new(&mut self.stream).poll_shutdown(cx))?;
7     Poll::Ready(Ok(()))
8 }

```

ProstStream 的创建

我们的 ProstStream 目前已经实现了 Stream 和 Sink，为了方便使用，再构建一些辅助方法，比如 new()：

[复制代码](#)

```

1 impl<S, In, Out> ProstStream<S, In, Out>
2 where
3     S: AsyncRead + AsyncWrite + Send + Unpin,
4 {
5     /// 创建一个 ProstStream
6     pub fn new(stream: S) -> Self {
7         Self {
8             stream,
9             written: 0,
10            wbuf: BytesMut::new(),

```

```

11         rbuf: BytesMut::new(),
12         _in: PhantomData::default(),
13         _out: PhantomData::default(),
14     }
15 }
16 }
17
18 // 一般来说, 如果我们的 Stream 是 Unpin, 最好实现一下
19 impl<S, Req, Res> Unpin for ProstStream<S, Req, Res> where S: Unpin {}


```

此外，我们还为其实现 Unpin trait，这会给别人在使用你的代码时带来很多方便。**一般来说，为异步操作而创建的数据结构，如果使用了泛型参数，那么只要内部没有自引用数据，就应该实现 Unpin。**

测试！

又到了重要的测试环节。我们需要写点测试来确保 ProstStream 能正常工作。因为之前在 src/network/ [frame.rs](#) 中写了个 DummyStream，实现了 AsyncRead，我们只需要扩展它，让它再实现 AsyncWrite。

为了让它可以被复用，我们将其从 [frame.rs](#) 中移出来，放在 src/network/mod.rs 中，并修改成下面的样子（记得在 [frame.rs](#) 的测试里 use 新的 DummyStream）：

 复制代码

```

1  #[cfg(test)]
2  pub mod utils {
3      use bytes::{BufMut, BytesMut};
4      use std::task::Poll;
5      use tokio::io::{AsyncRead, AsyncWrite};
6
7      pub struct DummyStream {
8          pub buf: BytesMut,
9      }
10
11      impl AsyncRead for DummyStream {
12          fn poll_read(
13              self: std::pin::Pin<&mut Self>,
14              _cx: &mut std::task::Context<'_>,
15              buf: &mut tokio::io::ReadBuf<'_>,
16          ) -> Poll<std::io::Result<()>> {
17              let len = buf.capacity();
18              let data = self.get_mut().buf.split_to(len);
19              buf.put_slice(&data);
20              Poll::Ready(Ok(()))

```

```

21     }
22 }
23
24 impl AsyncWrite for DummyStream {
25     fn poll_write(
26         self: std::pin::Pin<&mut Self>,
27         _cx: &mut std::task::Context<'_,>,
28         buf: &[u8],
29     ) -> Poll<Result<usize, std::io::Error>> {
30         self.get_mut().buf.put_slice(buf);
31         Poll::Ready(Ok(buf.len()))
32     }
33
34     fn poll_flush(
35         self: std::pin::Pin<&mut Self>,
36         _cx: &mut std::task::Context<'_,>,
37     ) -> Poll<Result<(), std::io::Error>> {
38         Poll::Ready(Ok(()))
39     }
40
41     fn poll_shutdown(
42         self: std::pin::Pin<&mut Self>,
43         _cx: &mut std::task::Context<'_,>,
44     ) -> Poll<Result<(), std::io::Error>> {
45         Poll::Ready(Ok(()))
46     }
47 }
48 }


```

好，这样我们就可以在 `src/network/stream.rs` 下写个测试了：

```

1  #[cfg(test)]
2  mod tests {
3      use super::*;
4      use crate::{utils::DummyStream, CommandRequest};
5      use anyhow::Result;
6      use futures::prelude::*;
7
8      #[tokio::test]
9      async fn prost_stream_should_work() -> Result<()> {
10         let buf = BytesMut::new();
11         let stream = DummyStream { buf };
12         let mut stream = ProstStream::new(<_, CommandRequest, CommandRequest>::new);
13         let cmd = CommandRequest::new_hdel("t1", "k1");
14         stream.send(cmd.clone()).await?;
15         if let Some(Ok(s)) = stream.next().await {
16             assert_eq!(s, cmd);
17         } else {

```

 复制代码

```
18         assert!(false);
19     }
20     Ok(())
21 }
22 }
```

运行 `cargo test`，一切测试通过！（如果你编译错误，可能缺少 `use` 的问题，可以自行修改，或者参考 GitHub 上的完整代码）。

使用 ProstStream

接下来，我们可以让 `ProstServerStream` 和 `ProstClientStream` 使用新定义的 `ProstStream` 了，你可以参考下面的对比，看看二者的区别：

[复制代码](#)

```
1 // 旧的接口
2 // pub struct ProstServerStream<S> {
3 //     inner: S,
4 //     service: Service,
5 // }
6
7 pub struct ProstServerStream<S> {
8     inner: ProstStream<S, CommandRequest, CommandResponse>,
9     service: Service,
10 }
11
12 // 旧的接口
13 // pub struct ProstClientStream<S> {
14 //     inner: S,
15 // }
16
17 pub struct ProstClientStream<S> {
18     inner: ProstStream<S, CommandResponse, CommandRequest>,
19 }
```

然后删除 `send()` / `recv()` 函数，并修改 `process()` / `execute()` 函数使其使用 `next()` 方法和 `send()` 方法。主要的改动如下：

[复制代码](#)

```
1 /// 处理服务器端的某个 accept 下来的 socket 的读写
2 pub struct ProstServerStream<S> {
3     inner: ProstStream<S, CommandRequest, CommandResponse>,
```

```
4     service: Service,
5 }
6
7 /// 处理客户端 socket 的读写
8 pub struct ProstClientStream<S> {
9     inner: ProstStream<S, CommandResponse, CommandRequest>,
10 }
11
12 impl<S> ProstServerStream<S>
13 where
14     S: AsyncRead + AsyncWrite + Unpin + Send,
15 {
16     pub fn new(stream: S, service: Service) -> Self {
17         Self {
18             inner: ProstStream::new(stream),
19             service,
20         }
21     }
22
23     pub async fn process(mut self) -> Result<(), KvError> {
24         let stream = &mut self.inner;
25         while let Some(Ok(cmd)) = stream.next().await {
26             info!("Got a new command: {:?}", cmd);
27             let res = self.service.execute(cmd);
28             stream.send(res).await.unwrap();
29         }
30
31         Ok(())
32     }
33 }
34
35 impl<S> ProstClientStream<S>
36 where
37     S: AsyncRead + AsyncWrite + Unpin + Send,
38 {
39     pub fn new(stream: S) -> Self {
40         Self {
41             inner: ProstStream::new(stream),
42         }
43     }
44
45     pub async fn execute(&mut self, cmd: CommandRequest) -> Result<CommandResp
46         let stream = &mut self.inner;
47         stream.send(cmd).await?;
48
49         match stream.next().await {
50             Some(v) => v,
51             None => Err(KvError::Internal("Didn't get any response".into())),
52         }
53     }
54 }
```


再次运行 `cargo test`，所有的测试应该都能通过。同样如果有编译错误，可能是缺少了引用。

我们也可以打开一个命令行窗口，运行：`RUST_LOG=info cargo run --bin kvs --quiet`。然后在另一个命令行窗口，运行：`RUST_LOG=info cargo run --bin kvc --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！

我们重构了 `ProstServerStream` 和 `ProstClientStream` 的代码，使其内部使用更符合 `futures` 库里 `Stream / Sink trait` 的用法，整体代码改动不小，但是内部实现的变更并不影响系统的其它部分！这简直太棒了！

小结

在实际开发中，进行重构来改善既有代码的质量是必不可少的。之前在开发 KV server 的过程中，我们在不断地进行一些小的重构。

今天我们做了个稍微大一些的重构，为已有的代码提供更加符合异步 IO 接口的功能。从对外使用的角度来说，它并没有提供或者满足任何额外的需求，但是从代码结构和质量的角度，它使得我们的 `ProstStream` 可以更方便和更直观地被其它接口调用，也更容易跟整个 Rust 的现有生态结合起来。

你可能会好奇，为什么可以这么自然地进行代码重构？这是因为我们有足够的单元测试覆盖来打底。

就像生物的进化一样，好的代码是在良性的重构中不断演进出来的，**而良性的重构，是在优秀的单元测试的监管下，使代码朝着正确方向迈出的步伐**。在这里，单元测试扮演着生物进化中自然环境的角色，把重构过程中的错误一一扼杀。

思考题

1. 为什么在创建 `ProstStream` 时，要在数据结构中放 `wbuf / rbuf` 和 `written` 字段？为什么不能用局部变量？
2. 仔细阅读 [Stream](#) 和 [Sink](#) 的文档。尝试写代码构造实现 `Stream` 和 `Sink` 的简单数据结构。

欢迎在留言区分享你的思考和学习收获，感谢你的收听，你已经完成了 Rust 学习的第 41 次打卡啦，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 6  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 40 | 异步处理：如何处理异步IO？

精选留言 (2)

 写留言



罗杰 

2021-12-03

前两节感觉理解有些吃力了，今天的实操突然又让我觉得好像也没那么难了。老师实操阶段的代码真是赏心悦目。我要动手好好理解一下。

展开 



 1



罗同学

2021-12-03

利用异步io 重构后 process ,性能会不会有一定提升呢？

