



下载APP



37 | 阶段实操（4）：构建一个简单的KV server-网络安全

2021-11-24 陈天

《陈天·Rust 编程第一课》

课程介绍 >

**讲述：陈天**

时长 09:02 大小 8.29M



你好，我是陈天。

上一讲我们完成了 KV server 整个网络部分的构建。而安全是和网络密不可分的组成部分，在构建应用程序的时候，一定要把网络安全也考虑进去。当然，如果不考虑极致的性能，我们可以使用诸如 gRPC 这样的系统，在提供良好性能的基础上，它还通过 [🔗 TLS](#) 保证了安全性。

那么，当我们的应用架构在 TCP 上时，如何使用 TLS 来保证客户端和服务端间的安全呢？



生成 x509 证书

想要使用 TLS，我们首先需要 [🔗x509 证书](#)。TLS 需要 x509 证书让客户端验证服务器是否是一个受信的服务器，甚至服务器验证客户端，确认对方是一个受信的客户端。

为了测试方便，我们要有能力生成自己的 CA 证书、服务端证书，甚至客户端证书。证书生成的细节今天就不详细介绍了，我之前做了一个叫 [🔗certify](#) 的库，可以用来生成各种证书。我们可以在 Cargo.toml 里加入这个库：

[📄 复制代码](#)

```
1 [dev-dependencies]
2 ...
3 certify = "0.3"
4 ...
```

然后在根目录下创建 fixtures 目录存放证书，再创建 examples/gen_cert.rs 文件，添入如下代码：

[📄 复制代码](#)

```
1 use anyhow::Result;
2 use certify::{generate_ca, generate_cert, load_ca, CertType, CA};
3 use tokio::fs;
4
5 struct CertPem {
6     cert_type: CertType,
7     cert: String,
8     key: String,
9 }
10
11 #[tokio::main]
12 async fn main() -> Result<()> {
13     let pem = create_ca()?;
14     gen_files(&pem).await?;
15     let ca = load_ca(&pem.cert, &pem.key)?;
16     let pem = create_cert(&ca, &["kvserver.acme.inc"], "Acme KV server", false);
17     gen_files(&pem).await?;
18     let pem = create_cert(&ca, &[], "awesome-device-id", true)?;
19     gen_files(&pem).await?;
20     Ok(())
21 }
22
23 fn create_ca() -> Result<CertPem> {
24     let (cert, key) = generate_ca(
25         &["acme.inc"],
26         "CN",
27         "Acme Inc.",
```

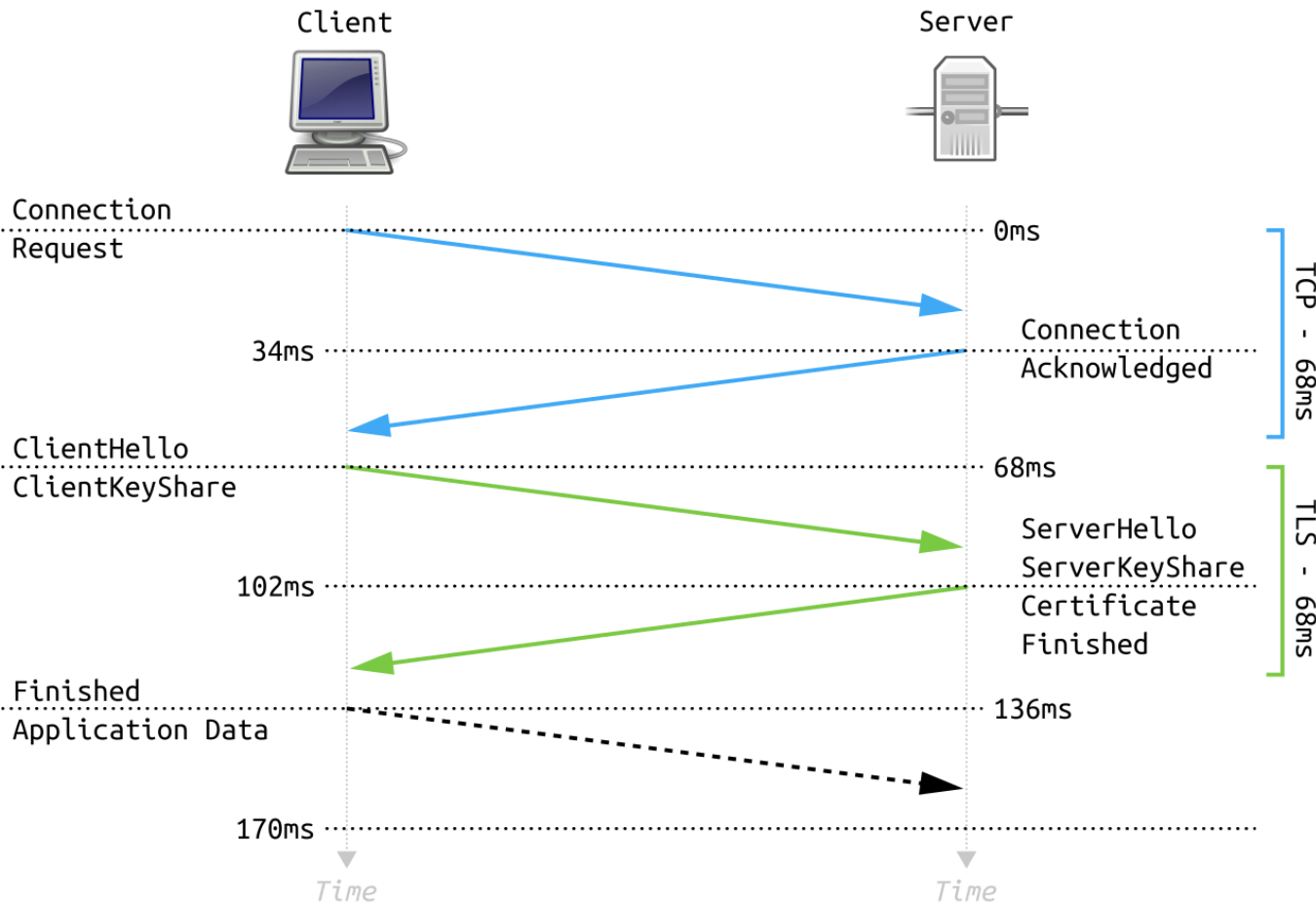
```
28     "Acme CA",
29     None,
30     Some(10 * 365),
31 );?;
32 Ok(CertPem {
33     cert_type: CertType::CA,
34     cert,
35     key,
36 })
37 }
38
39 fn create_cert(ca: &CA, domains: &[&str], cn: &str, is_client: bool) -> Result
40     let (days, cert_type) = if is_client {
41         (Some(365), CertType::Client)
42     } else {
43         (Some(5 * 365), CertType::Server)
44     };
45     let (cert, key) = generate_cert(ca, domains, "CN", "Acme Inc.", cn, None,
46
47     Ok(CertPem {
48         cert_type,
49         cert,
50         key,
51     })
52 }
53
54 async fn gen_files(pem: &CertPem) -> Result<()> {
55     let name = match pem.cert_type {
56         CertType::Client => "client",
57         CertType::Server => "server",
58         CertType::CA => "ca",
59     };
60     fs::write(format!("fixtures/{}.cert", name), pem.cert.as_bytes()).await?;
61     fs::write(format!("fixtures/{}.key", name), pem.key.as_bytes()).await?;
62     Ok(())
63 }
```

这个代码很简单，它先生成了一个 CA 证书，然后再生成服务器和客户端证书，全部存入刚创建的 fixtures 目录下。你需要 `cargo run --examples gen_cert` 运行一下这个命令，待会我们会在测试中用到这些证书和密钥。

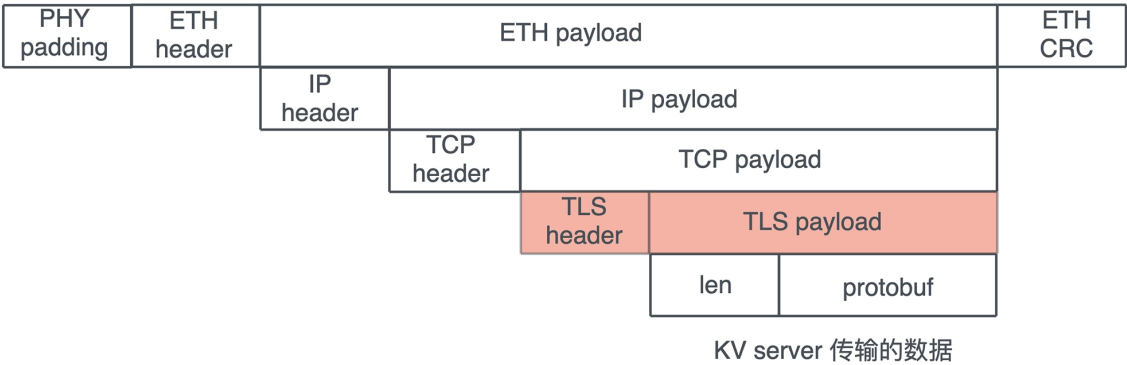
在 KV server 中使用 TLS

TLS 是目前最主要的应用层安全协议，被广泛用于保护架构在 TCP 之上的，比如 MySQL、HTTP 等各种协议。一个网络应用，即便是在内网使用，如果没有安全协议来保护，都是很危险的。

下图展示了客户端和服务端进行 TLS 握手的过程，来源 [wikimedia](#)：



对于 KV server 来说，使用 TLS 之后，整个协议的数据封装如下图所示：



所以今天要做的就是 在上一讲的网络处理的基础上，添加 TLS 支持，使得 KV server 的客户端服务器之间的通讯被严格保护起来，确保最大程度的安全，免遭第三方的偷窥、篡改以及伪造。

好，接下来我们看看 TLS 怎么实现。

估计很多人一听 TLS 或者 SSL，就头皮发麻，因为之前跟 [openssl](#) 打交道有过很多不好的经历。openssl 的代码库太庞杂，API 不友好，编译链接都很费劲。

不过，在 Rust 下使用 TLS 的体验还是很不错的，Rust 对 openssl 有很不错的 [封装](#)，也有不依赖 openssl 用 Rust 撰写的 [rustls](#)。tokio 进一步提供了符合 tokio 生态圈的 [tls 支持](#)，有 openssl 版本和 rustls 版本可选。

我们今天就用 [tokio-rustls](#) 来撰写 TLS 的支持。相信你在实现过程中可以看到，在应用程序中加入 TLS 协议来保护网络层，是多么轻松的一件事情。

先在 Cargo.toml 中添加 tokio-rustls：

[复制代码](#)

```
1 [dependencies]
2 ...
3 tokio-rustls = "0.22"
4 ...
```

然后创建 src/network/tls.rs，撰写如下代码（记得在 src/network/mod.rs 中引入这个文件哦）：

[复制代码](#)

```
1 use std::io::Cursor;
2 use std::sync::Arc;
3
4 use tokio::io::{AsyncRead, AsyncWrite};
5 use tokio_rustls::rustls::{internal::pemfile, Certificate, ClientConfig, Serve
6 use tokio_rustls::rustls::{AllowAnyAuthenticatedClient, NoClientAuth, PrivateK
7 use tokio_rustls::webpki::DNSNameRef;
8 use tokio_rustls::TlsConnector;
9 use tokio_rustls::{
10     client::TlsStream as ClientTlsStream, server::TlsStream as ServerTlsStream
11 };
12
13 use crate::KvError;
14
15 /// KV Server 自己的 ALPN (Application-Layer Protocol Negotiation)
16 const ALPN_KV: &str = "kv";
```



```
17
18 /// 存放 TLS ServerConfig 并提供方法 accept 把底层的协议转换成 TLS
19 #[derive(Clone)]
20 pub struct TlsServerAcceptor {
21     inner: Arc<ServerConfig>,
22 }
23
24 /// 存放 TLS Client 并提供方法 connect 把底层的协议转换成 TLS
25 #[derive(Clone)]
26 pub struct TlsClientConnector {
27     pub config: Arc<ClientConfig>,
28     pub domain: Arc<String>,
29 }
30
31 impl TlsClientConnector {
32     /// 加载 client cert / CA cert, 生成 ClientConfig
33     pub fn new(
34         domain: impl Into<String>,
35         identity: Option<(&str, &str)>,
36         server_ca: Option<&str>,
37     ) -> Result<Self, KvError> {
38         let mut config = ClientConfig::new();
39
40         // 如果有客户端证书, 加载之
41         if let Some((cert, key)) = identity {
42             let certs = load_certs(cert)?;
43             let key = load_key(key)?;
44             config.set_single_client_cert(certs, key)?;
45         }
46
47         // 加载本地信任的根证书链
48         config.root_store = match rustls_native_certs::load_native_certs() {
49             Ok(store) | Err((Some(store), _)) => store,
50             Err((None, error)) => return Err(error.into()),
51         };
52
53         // 如果有签署服务器的 CA 证书, 则加载它, 这样服务器证书不在根证书链
54         // 但是这个 CA 证书能验证它, 也可以
55         if let Some(cert) = server_ca {
56             let mut buf = Cursor::new(cert);
57             config.root_store.add_pem_file(&mut buf).unwrap();
58         }
59
60         Ok(Self {
61             config: Arc::new(config),
62             domain: Arc::new(domain.into()),
63         })
64     }
65
66     /// 触发 TLS 协议, 把底层的 stream 转换成 TLS stream
67     pub async fn connect<S>(&self, stream: S) -> Result<ClientTlsStream<S>, Kv
68     where
```

```

69     S: AsyncRead + AsyncWrite + Unpin + Send,
70     {
71         let dns = DNSNameRef::try_from_ascii_str(self.domain.as_str())
72             .map_err(|_| KvError::Internal("Invalid DNS name".into()))?;
73
74         let stream = TlsConnector::from(self.config.clone())
75             .connect(dns, stream)
76             .await?;
77
78         Ok(stream)
79     }
80 }
81
82 impl TlsServerAcceptor {
83     /// 加载 server cert / CA cert, 生成 ServerConfig
84     pub fn new(cert: &str, key: &str, client_ca: Option<&str>) -> Result<Self,
85         let certs = load_certs(cert)?;
86         let key = load_key(key)?;
87
88         let mut config = match client_ca {
89             None => ServerConfig::new(NoClientAuth::new()),
90             Some(cert) => {
91                 // 如果客户端证书是某个 CA 证书签发的, 则把这个 CA 证书加载到信任链中
92                 let mut cert = Cursor::new(cert);
93                 let mut client_root_cert_store = RootCertStore::empty();
94                 client_root_cert_store
95                     .add_pem_file(&mut cert)
96                     .map_err(|_| KvError::CertificateParseError("CA", "cert"))?;
97
98                 let client_auth = AllowAnyAuthenticatedClient::new(client_root
99                     ServerConfig::new(client_auth)
100             }
101         };
102
103         // 加载服务器证书
104         config
105             .set_single_cert(certs, key)
106             .map_err(|_| KvError::CertificateParseError("server", "cert"))?;
107         config.set_protocols(&[Vec::from(&ALPN_KV[..])]);
108
109         Ok(Self {
110             inner: Arc::new(config),
111         })
112     }
113
114     /// 触发 TLS 协议, 把底层的 stream 转换成 TLS stream
115     pub async fn accept<S>(&self, stream: S) -> Result<ServerTlsStream<S>, KvE
116     where
117         S: AsyncRead + AsyncWrite + Unpin + Send,
118         {
119         let acceptor = TlsAcceptor::from(self.inner.clone());
120         Ok(acceptor.accept(stream).await?)

```

```
121     }
122 }
123
124 fn load_certs(cert: &str) -> Result<Vec<Certificate>, KvError> {
125     let mut cert = Cursor::new(cert);
126     pemfile::certs(&mut cert).map_err(|_| KvError::CertificateParseError("serve
127 }
128
129 fn load_key(key: &str) -> Result<PrivateKey, KvError> {
130     let mut cursor = Cursor::new(key);
131
132     // 先尝试用 PKCS8 加载私钥
133     if let Ok(mut keys) = pemfile::pkcs8_private_keys(&mut cursor) {
134         if !keys.is_empty() {
135             return Ok(keys.remove(0));
136         }
137     }
138
139     // 再尝试加载 RSA key
140     cursor.set_position(0);
141     if let Ok(mut keys) = pemfile::rsa_private_keys(&mut cursor) {
142         if !keys.is_empty() {
143             return Ok(keys.remove(0));
144         }
145     }
146
147     // 不支持的私钥类型
148     Err(KvError::CertificateParseError("private", "key"))
149 }
```

这个代码创建了两个数据结构 TlsServerAcceptor / TlsClientConnector。虽然它有 100 多行，但主要的工作其实就是**根据提供的证书，来生成 tokio-tls 需要的 ServerConfig / ClientConfig**。

因为 TLS 需要验证证书的 CA，所以还需要加载 CA 证书。虽然平时在做 Web 开发时，我们都只使用服务器证书，但其实 TLS 支持双向验证，服务器也可以验证客户端的证书是否是它认识的 CA 签发的。

处理完 config 后，这段代码的核心逻辑其实就是客户端的 connect() 方法和服务器的 accept() 方法，它们都接受一个满足 AsyncRead + AsyncWrite + Unpin + Send 的 stream。类似上一讲，我们不希望 TLS 代码只能接受 TcpStream，所以这里提供了一个泛型参数 S：


```

1  /// 触发 TLS 协议,把底层的 stream 转换成 TLS stream
2  pub async fn connect<S>(&self, stream: S) -> Result<ClientTlsStream<S>, KvError
3  where
4      S: AsyncRead + AsyncWrite + Unpin + Send,
5      {
6          let dns = DNSNameRef::try_from_ascii_str(self.domain.as_str())
7              .map_err(|_| KvError::Internal("Invalid DNS name".into()))?;
8
9          let stream = TlsConnector::from(self.config.clone())
10             .connect(dns, stream)
11             .await?;
12
13         Ok(stream)
14     }
15
16  /// 触发 TLS 协议,把底层的 stream 转换成 TLS stream
17  pub async fn accept<S>(&self, stream: S) -> Result<ServerTlsStream<S>, KvError
18  where
19      S: AsyncRead + AsyncWrite + Unpin + Send,
20      {
21          let acceptor = TlsAcceptor::from(self.inner.clone());
22          Ok(acceptor.accept(stream).await?)
23      }

```


在使用 TlsConnector 或者 TlsAcceptor 处理完 connect/accept 后，我们得到了一个 TlsStream，它也满足 AsyncRead + AsyncWrite + Unpin + Send，后续的操作就可以在其上完成了。百来行代码就搞定了 TLS，是不是很简单？

我们来顺着往下写段测试：

```

1  #[cfg(test)]
2  mod tests {
3
4      use std::net::SocketAddr;
5
6      use super::*;
7      use anyhow::Result;
8      use tokio::{
9          io::{AsyncReadExt, AsyncWriteExt},
10         net::{TcpListener, TcpStream},
11     };
12
13     const CA_CERT: &str = include_str!("../../fixtures/ca.cert");
14     const CLIENT_CERT: &str = include_str!("../../fixtures/client.cert");
15     const CLIENT_KEY: &str = include_str!("../../fixtures/client.key");
16     const SERVER_CERT: &str = include_str!("../../fixtures/server.cert");

```

 复制代码

```
17     const SERVER_KEY: &str = include_str!("../../fixtures/server.key");
18
19     #[tokio::test]
20     async fn tls_should_work() -> Result<()> {
21         let ca = Some(CA_CERT);
22
23         let addr = start_server(None).await?;
24
25         let connector = TlsClientConnector::new("kvserver.acme.inc", None, ca)
26         let stream = TcpStream::connect(addr).await?;
27         let mut stream = connector.connect(stream).await?;
28         stream.write_all(b"hello world!").await?;
29         let mut buf = [0; 12];
30         stream.read_exact(&mut buf).await?;
31         assert_eq!(&buf, b"hello world!");
32
33         Ok(())
34     }
35
36     #[tokio::test]
37     async fn tls_with_client_cert_should_work() -> Result<()> {
38         let client_identity = Some((CLIENT_CERT, CLIENT_KEY));
39         let ca = Some(CA_CERT);
40
41         let addr = start_server(ca.clone()).await?;
42
43         let connector = TlsClientConnector::new("kvserver.acme.inc", client_id
44         let stream = TcpStream::connect(addr).await?;
45         let mut stream = connector.connect(stream).await?;
46         stream.write_all(b"hello world!").await?;
47         let mut buf = [0; 12];
48         stream.read_exact(&mut buf).await?;
49         assert_eq!(&buf, b"hello world!");
50
51         Ok(())
52     }
53
54     #[tokio::test]
55     async fn tls_with_bad_domain_should_not_work() -> Result<()> {
56         let addr = start_server(None).await?;
57
58         let connector = TlsClientConnector::new("kvserver1.acme.inc", None, So
59         let stream = TcpStream::connect(addr).await?;
60         let result = connector.connect(stream).await;
61
62         assert!(result.is_err());
63
64         Ok(())
65     }
66
67     async fn start_server(ca: Option<&str>) -> Result<SocketAddr> {
68         let acceptor = TlsServerAcceptor::new(SERVER_CERT, SERVER_KEY, ca)?;
```


```
69
70     let echo = TcpListener::bind("127.0.0.1:0").await.unwrap();
71     let addr = echo.local_addr().unwrap();
72
73     tokio::spawn(async move {
74         let (stream, _) = echo.accept().await.unwrap();
75         let mut stream = acceptor.accept(stream).await.unwrap();
76         let mut buf = [0; 12];
77         stream.read_exact(&mut buf).await.unwrap();
78         stream.write_all(&buf).await.unwrap();
79     });
80
81     Ok(addr)
82 }
83 }
```

这段测试代码使用了 `include_str!` 宏，在编译期把文件加载成字符串放在 `RODATA` 段。我们测试了三种情况：标准的 TLS 连接、带有客户端证书的 TLS 连接，以及客户端提供了错的域名的情况。运行 `cargo test`，所有测试都能通过。

让 KV client/server 支持 TLS

在 TLS 的测试都通过后，就可以添加 `kvs` 和 `kvc` 对 TLS 的支持了。


由于我们一路以来良好的接口设计，尤其是 `ProstClientStream` / `ProstServerStream` 都接受泛型参数，使得 TLS 的代码可以无缝嵌入。比如客户端：

 复制代码

```
1 // 新加的代码
2 let connector = TlsClientConnector::new("kvserver.acme.inc", None, Some(ca_cer
3
4 let stream = TcpStream::connect(addr).await?;
5
6 // 新加的代码
7 let stream = connector.connect(stream).await?;
8
9 let mut client = ProstClientStream::new(stream);
```

仅仅需要把传给 `ProstClientStream` 的 `stream`，从 `TcpStream` 换成生成的 `TlsStream`，就无缝支持了 TLS。

我们看完整的代码，src/server.rs：

 复制代码

```
1 use anyhow::Result;
2 use kv3::{MemTable, ProstServerStream, Service, ServiceInner, TlsServerAccepto
3 use tokio::net::TcpListener;
4 use tracing::info;
5
6 #[tokio::main]
7 async fn main() -> Result<()> {
8     tracing_subscriber::fmt::init();
9     let addr = "127.0.0.1:9527";
10
11     // 以后从配置文件取
12     let server_cert = include_str!("../fixtures/server.cert");
13     let server_key = include_str!("../fixtures/server.key");
14
15     let acceptor = TlsServerAccepto::new(server_cert, server_key, None)?;
16     let service: Service = ServiceInner::new(MemTable::new()).into();
17     let listener = TcpListener::bind(addr).await?;
18     info!("Start listening on {}", addr);
19     loop {
20         let tls = acceptor.clone();
21         let (stream, addr) = listener.accept().await?;
22         info!("Client {:?} connected", addr);
23         let stream = tls.accept(stream).await?;
24         let stream = ProstServerStream::new(stream, service.clone());
25         tokio::spawn(async move { stream.process().await });
26     }
27 }
```

src/client.rs：

 复制代码

```
1 use anyhow::Result;
2 use kv3::{CommandRequest, ProstClientStream, TlsClientConnector};
3 use tokio::net::TcpStream;
4 use tracing::info;
5
6 #[tokio::main]
7 async fn main() -> Result<()> {
8     tracing_subscriber::fmt::init();
9
10     // 以后用配置替换
11     let ca_cert = include_str!("../fixtures/ca.cert");
12
13     let addr = "127.0.0.1:9527";
```

```
14 // 连接服务器
15 let connector = TlsClientConnector::new("kvserver.acme.inc", None, Some(ca
16 let stream = TcpStream::connect(addr).await?;
17 let stream = connector.connect(stream).await?;
18
19 let mut client = ProstClientStream::new(stream);
20
21 // 生成一个 HSET 命令
22 let cmd = CommandRequest::new_hset("table1", "hello", "world".to_string()).
23
24 // 发送 HSET 命令
25 let data = client.execute(cmd).await?;
26 info!("Got response {:?}", data);
27
28 Ok(())
29 }
```

和上一讲的代码项目相比，更新后的客户端和服务端代码，各自仅仅多了一行，就把 `TcpStream` 封装成了 `TlsStream`。这就是使用 `trait` 做面向接口编程的巨大威力，系统的各个组件可以来自不同的 `crates`，但只要其接口一致（或者我们创建 `adapter` 使其接口一致），就可以无缝插入。

完成之后，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。

现在，我们的 KV server 已经具备足够的安全性了！以后，等我们使用配置文件，就可以根据配置文件读取证书和私钥。这样可以在部署的时候，才从 `vault` 中获取私钥，既保证灵活性，又能保证系统自身的安全。

小结

网络安全是开发网络相关的应用程序中非常重要的一个环节。虽然 KV Server 这样的服务基本上会运行在云端受控的网络环境中，不会对 `internet` 提供服务，然而云端内部的安全性也不容忽视。你不希望数据在流动的过程中被篡改。

TLS 很好地解决了安全性的问题，可以保证整个传输过程中数据的机密性和完整性。如果使用客户端证书的话，还可以做一定程度的客户端合法性的验证。比如你可以在云端为所

有有权访问 KV server 的客户端签发客户端证书，这样，只要客户端的私钥不泄露，就只有拥有证书的客户端才能访问 KV server。

不知道你现在有没有觉得，在 Rust 下使用 TLS 是非常方便的一件事情。并且，我们构建的 ProstServerStream / ProstClientStream，因为**有足够好的抽象，可以在 TcpStream 和 TlsStream 之间游刃有余地切换**。当你构建好相关的代码，只需要把 TcpStream 换成 TlsStream，KV server 就可以无缝切换到一个安全的网络协议栈。

思考题

1. 目前我们的 kvc / kvs 只做了单向的验证，如果服务器要验证客户端的证书，该怎么做？如果你没有头绪，可以再仔细看看测试 TLS 的代码，然后改动 kvc/kvs 使得双向验证也能通过吧。
2. 除了 TLS，另外一个被广泛使用的处理应用层安全的协议是 [noise protocol](#)。你可以阅读我的 [这篇文章](#) 了解 noise protocol。Rust 下有 [snow](#) 这个很优秀的库处理 noise protocol。对于有余力的同学，你们可以看看它的文档，尝试着写段类似 [tls.rs](#) 的代码，让我们的 kvs / kvc 可以使用 noise protocol。

欢迎在留言区分享你的思考，感谢你的收听，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。

恭喜你完成了第 37 次打卡，我们的 Rust 学习之旅已经过一大半啦，曙光就在前方，坚持下去，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 7  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 [36 | 阶段实操（3）：构建一个简单的KV server-网络处理](#)

精选留言 (2)

[写留言](#)**罗杰**

2021-11-24

生成证书这块是我比较欠缺的知识，可以好好补充一下了。

**罗同学**

2021-11-24

ca 证书和 tls什么关系呢？

另外为何以前做网站的时候证书都要向运营商购买申请？那个是什么证书

展开 ∨

