CloudWeGo

Rust RPC框架使用介绍

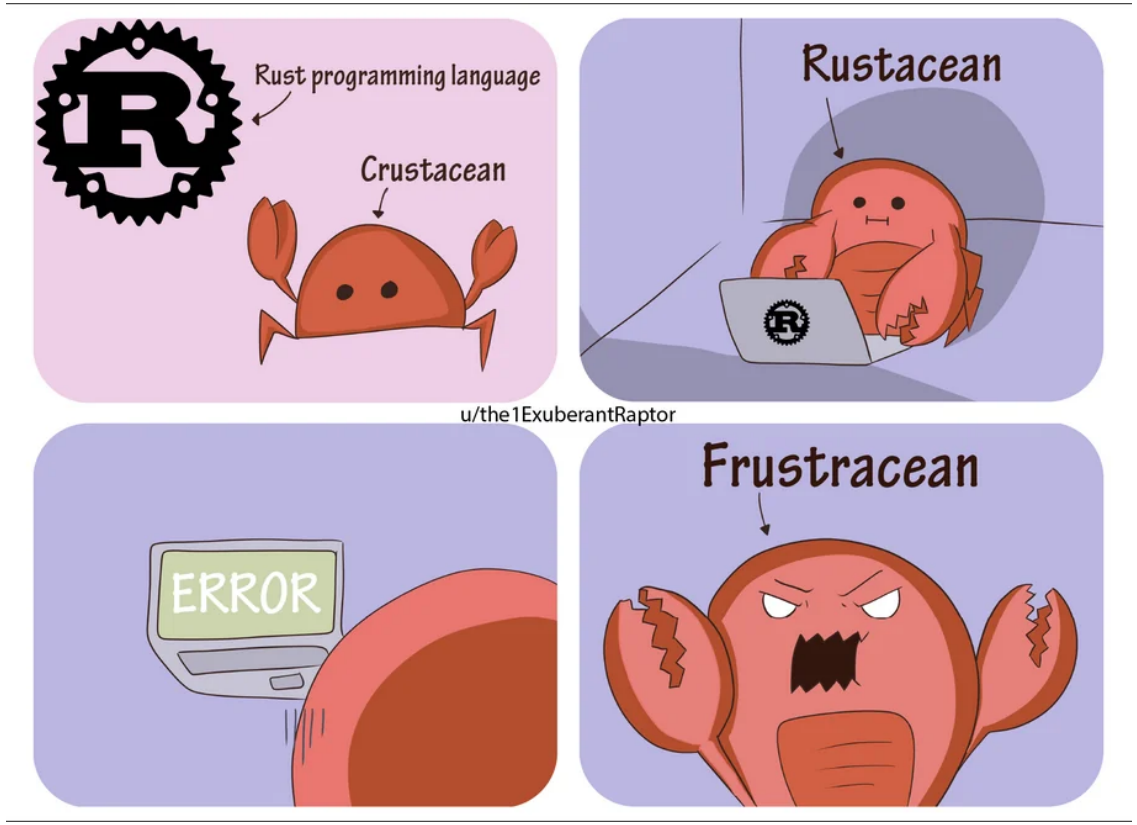
Rust 开发实训

字节跳动

2023/09/11

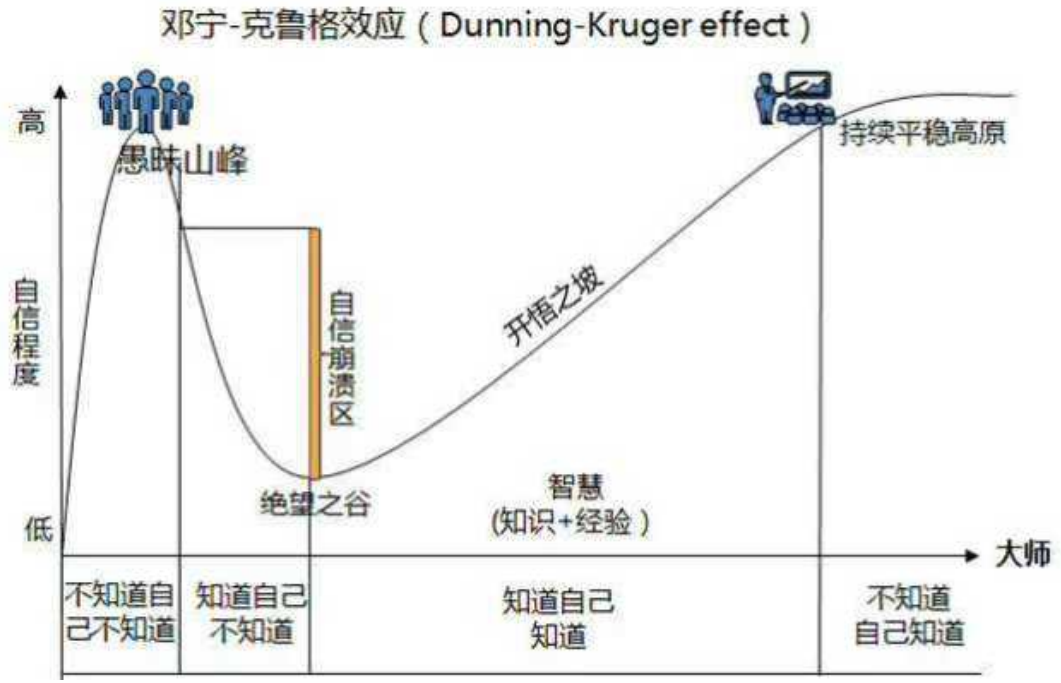
课前小回顾

Rust 这门语言学起来感觉如何?



[learning rust got me feeling like]

还记得我吗?



课前小回顾

```rust
impl<IL, OL, C, Req, Resp, MkT, MkC, LB> ClientBuilder<IL, OL, C, Req, Resp, MkT, MkC, LB>
where
    C: volo::client::MkClient<
        Client<
            BoxCloneService<
                ClientContext,
                Req,
                Option<Resp>,
                <OL::Service as Service<ClientContext, Req>>::Error,
            >,
        >,
    >,
    LB: MkLbLayer,
    LB::Layer: Layer<IL::Service>,
    <LB::Layer as Layer<IL::Service>>::Service: Service<ClientContext, Req, Response = Option<Resp>, Error = Error>
        + 'static
        + Send
        + Clone
        + Sync,
    for<'cx> <<LB::Layer as Layer<IL::Service>>::Service as Service<ClientContext, Req>>::Future<'cx>:
        Send,
    Req: EntryMessage + Send + 'static + Sync + Clone,
    Resp: EntryMessage + Send + 'static,
    IL: Layer<MessageService<Resp, MkT, MkC>>,
    IL::Service:
        Service<ClientContext, Req, Response = Option<Resp>> + Sync + Clone + Send + 'static,
    <IL::Service as Service<ClientContext, Req>>::Error: Send + Into<Error>,
    for<'cx> <IL::Service as Service<ClientContext, Req>>::Future<'cx>: Send,
    MkT: MakeTransport,
    MkC: MakeCodec<MkT::ReadHalf, MkT::WriteHalf> + Sync,
    OL: Layer<BoxCloneService<ClientContext, Req, Option<Resp>, Error>>,
    OL::Service:
        Service<ClientContext, Req, Response = Option<Resp>> + 'static + Send + Clone + Sync,
    for<'cx> <OL::Service as Service<ClientContext, Req>>::Future<'cx>: Send,
    <OL::Service as Service<ClientContext, Req>>::Error: Send + Sync + Into<Error>,
{
```

目录

CONTENT

01 RPC——灵魂三问

RPC 是什么?

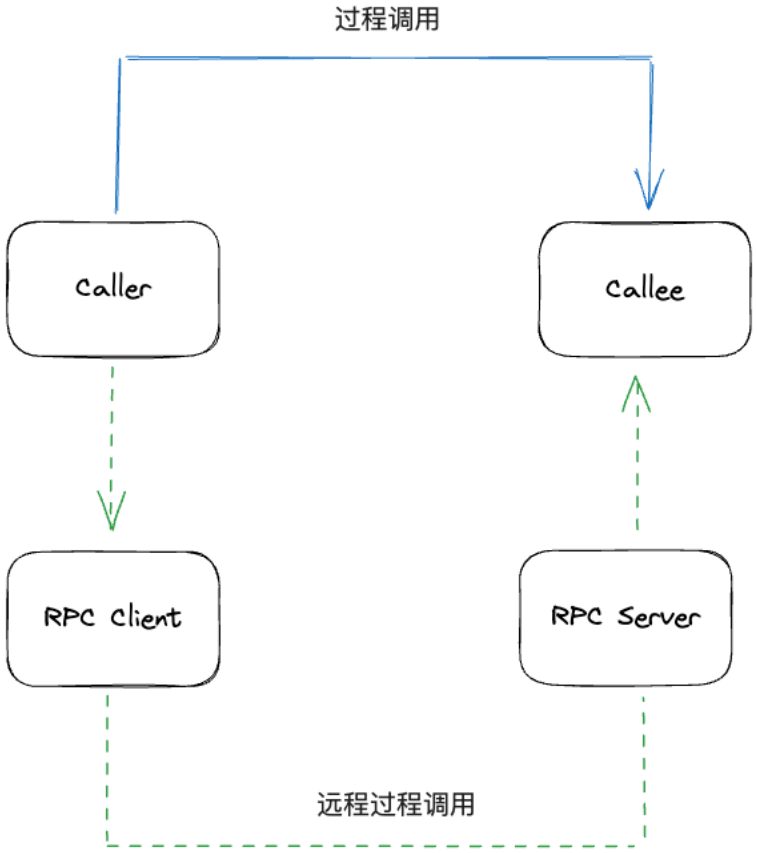
用一句话介绍一下RPC是什么

ChatGPT

RPC（Remote Procedure Call）是一种分布式系统中的通信机制，允许程序在不同计算机或进程之间像调用本地函数一样进行远程调用。

过程调用 VS 远程过程调用

- 相同点
  1. 调用方式（函数调用）
  2. 使用方式（let resp = self.call(req);）
- 不同点
  1. 执行位置（不同进程或不同机器）
  2. 通信机制（网络通信）
  3. 性能开销（网络和序列化开销）



为什么需要 RPC?

其实是在问RPC的应用场景?

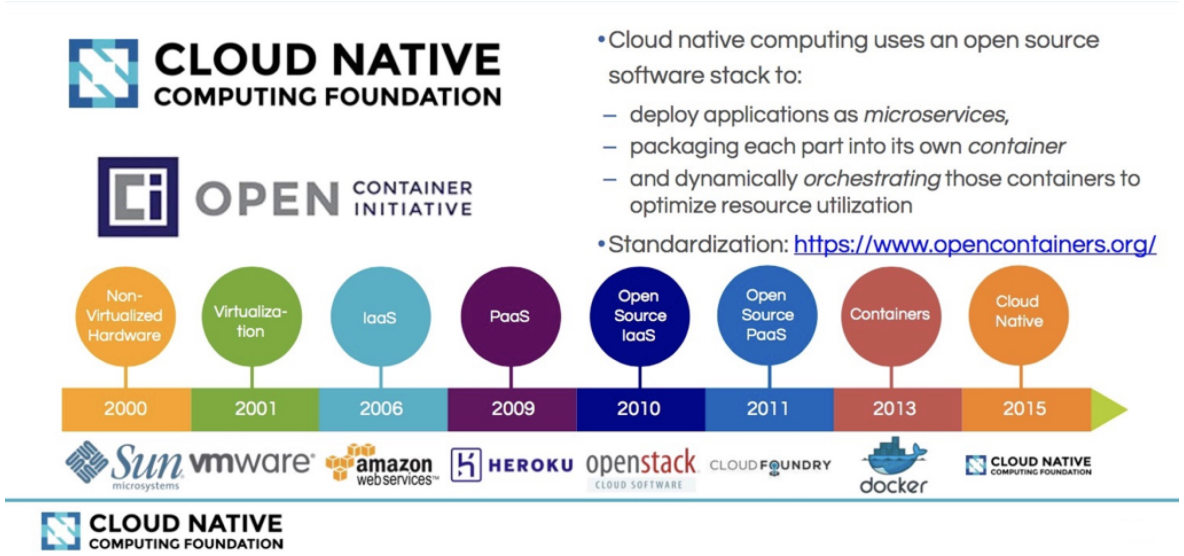
- 云原生的时代背景（敏捷开发）
  - DevOps、持续交付、容器化

- 单体服务 -> 微服务
  - 降低服务复杂性，便于维护
  - 独立技术栈，独立开发部署
  - 可扩展性和容错性

RPC VS REST

- 可读性与性能之间的取舍

天下大势，合久必分，分久必合。



Cloud Native

RPC 如何实现？

序列化：Thrift or Protobuf

网络传输：TCP or HTTP

Missing parts：

- IDL（Interface Define Language）
- 中间件（服务治理）
- 性能优化

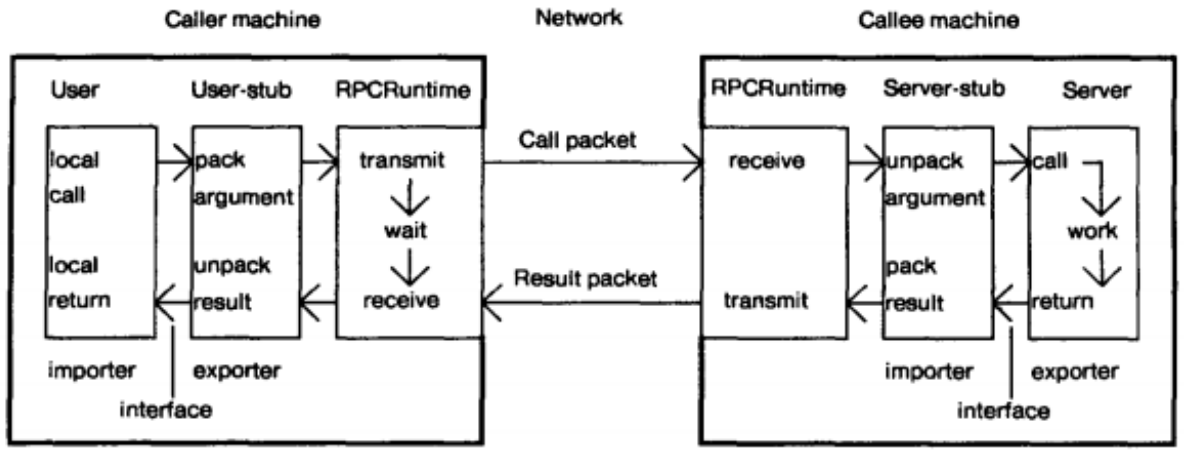


Fig. 1. The components of the system, and their interactions for a simple call.

Implementing Remote Procedure Calls

02 Volo——以点破面

为什么要做 Volo?

弥补 Rust RPC 框架开源生态的缺失：

- Thrift: 没有生产可用的 Async Thrift 实现
- gRPC: Tonic 实现服务治理功能偏弱，易用性不够强
- 抽象：没有基于 GAT 和 TAIT 的设计

Generic Associated Types

```
1 trait Foo {
2   type Bar<T>;
3
4   type Baz<'a>;
5 }
```

Type Alias Impl Trait

```
1 type Qux = impl Foo;
2
3 struct __Foo_alias;
4 type Qux = __Foo_alias;
```

GAT

比如实现一个迭代器，每次获取一段重叠的可变引用slice

不使用GAT会遇到什么问题?

```rust
pub trait Iterator {
    type Item;

    // Required method
    fn next(&mut self) -> Option<Self::Item>;
}
```

```rust
struct WindowsMut<'t, T> {
    slice: &'t mut [T],
    start: usize,
    window_size: usize,
}

impl<'t, T> Iterator for WindowsMut<'t, T> {
    type Item = &'t mut [T];

    fn next<'a>(&'a mut self) -> Option<Self::Item> {
        let retval = self.slice[self.start..].get_mut(..self.window_size)?;
        self.start += 1;
        Some(retval)
    }
}
```

GAT

```rust
trait LendingIterator {
    type Item<'a> where Self: 'a;

    fn next<'a>(&'a mut self) -> Option<Self::Item<'a>>;
}
```

https://blog.rust-lang.org/2021/08/03/GATs-stabilization-push.html

```rust
impl<'t, T> LendingIterator for WindowsMut<'t, T> {
    type Item<'a> where Self: 'a = &'a mut [T];

    fn next<'a>(&'a mut self) -> Option<Self::Item<'a>> {
        let retval = self.slice[self.start..].get_mut(..self.window_size)?;
        self.start += 1;
        Some(retval)
    }
}
```

TAIT

代码量减少一倍

逻辑更加清晰简单

```rust
#[derive(Debug, Clone)]
pub struct Timeout<T> {
    inner: T,
    timeout: Duration,
}

impl<S, Request> Service<Request> for Timeout<S>
where
    S: Service<Request>,
    S::Error: Into<crate::BoxError>,
{
    type Response = S::Response;
    type Error = crate::BoxError;
    type Future = ResponseFuture<S::Future>;

    fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>>
{
        match self.inner.poll_ready(cx) {
            Poll::Pending => Poll::Pending,
            Poll::Ready(r) => Poll::Ready(r.map_err(Into::into)),
        }
    }

    fn call(&mut self, request: Request) -> Self::Future {
        let response = self.inner.call(request);
        let sleep = tokio::time::sleep(self.timeout);

        ResponseFuture::new(response, sleep)
    }
}

pin_project! {
    #[derive(Debug)]
    pub struct ResponseFuture<T> {
        #[pin]
        response: T,
        #[pin]
        sleep: Sleep,
    }
}

impl<T> ResponseFuture<T> {
    pub(crate) fn new(response: T, sleep: Sleep) -> Self {
        ResponseFuture { response, sleep }
    }
}

impl<F, T, E> Future for ResponseFuture<F>
where
    F: Future<Output = Result<T, E>>,
    E: Into<crate::BoxError>,
{
    type Output = Result<T, crate::BoxError>;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let this = self.project();

        match this.response.poll(cx) {
            Poll::Ready(v) => return Poll::Ready(v.map_err(Into::into)),
            Poll::Pending => {}
        }

        match this.sleep.poll(cx) {
            Poll::Pending => Poll::Pending,
            Poll::Ready(_) => Poll::Ready(Err(Elapsed(()).into())),
        }
    }
}
```

```rust
#[derive(Clone)]
pub struct Timeout<S> {
    inner: S,
    duration: Duration,
}

#[service]
impl<Cx, Req, S> Service<Cx, Req> for Timeout<S>
where
    Req: 'static + Send,
    S: Service<Cx, Req> + 'static + Send,
    Cx: 'static + Send,
    S::Error: Send + Sync + Into<BoxError>,
{
    async fn call(&mut self, cx: &mut Cx, req: Req) -> Result<S::Response, BoxError> {
        let sleep = tokio::time::sleep(self.duration);
        tokio::select! {
            r = self.inner.call(cx, req) => {
                r.map_err(Into::into)
            },
            _ = sleep => Err(std::io::Error::new(std::io::ErrorKind::TimedOut, "service time
out").into()),
        }
    }
}
```

Volo 架构

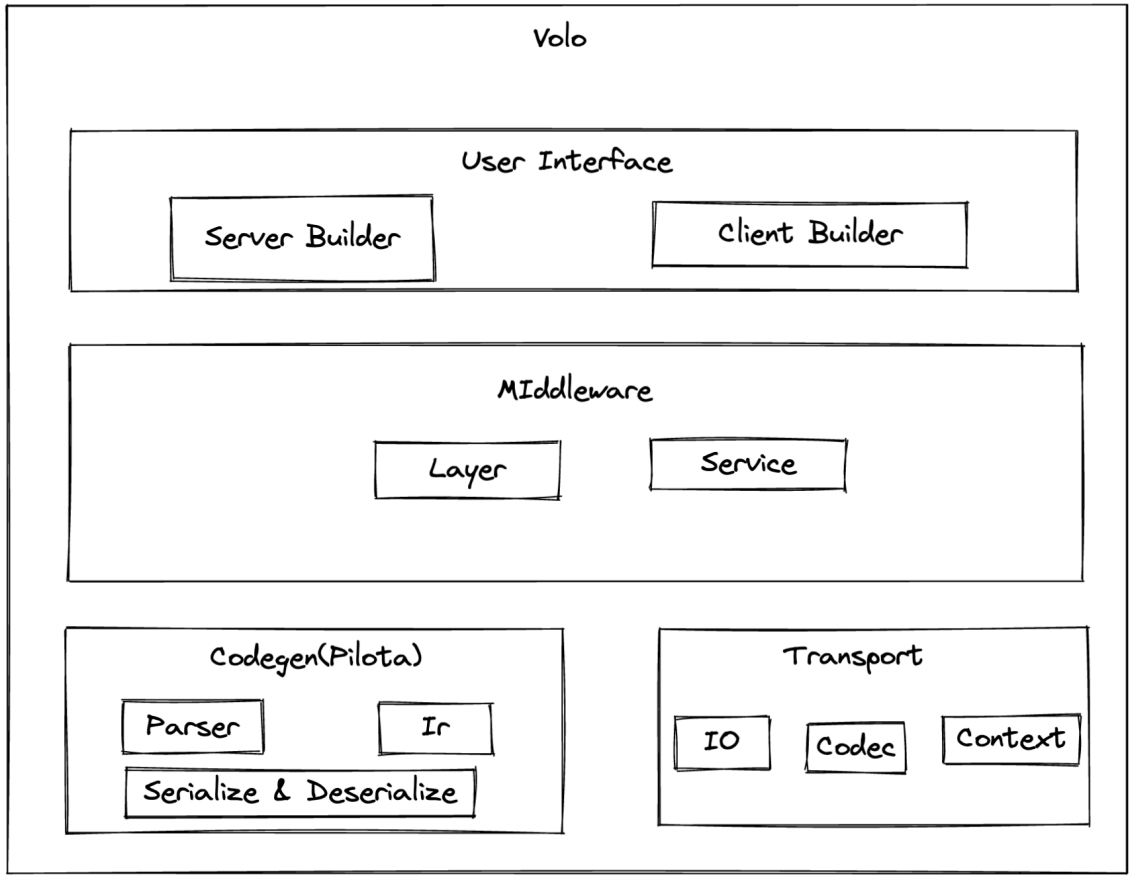
用户接口：配置 & 模块的插拔

中间件：服务治理的实现部分

Codec：Req & Resp -> Bytes -> Req & Resp

IO：Send & Recv Bytes

生成代码：连接用户代码和框架代码的桥梁

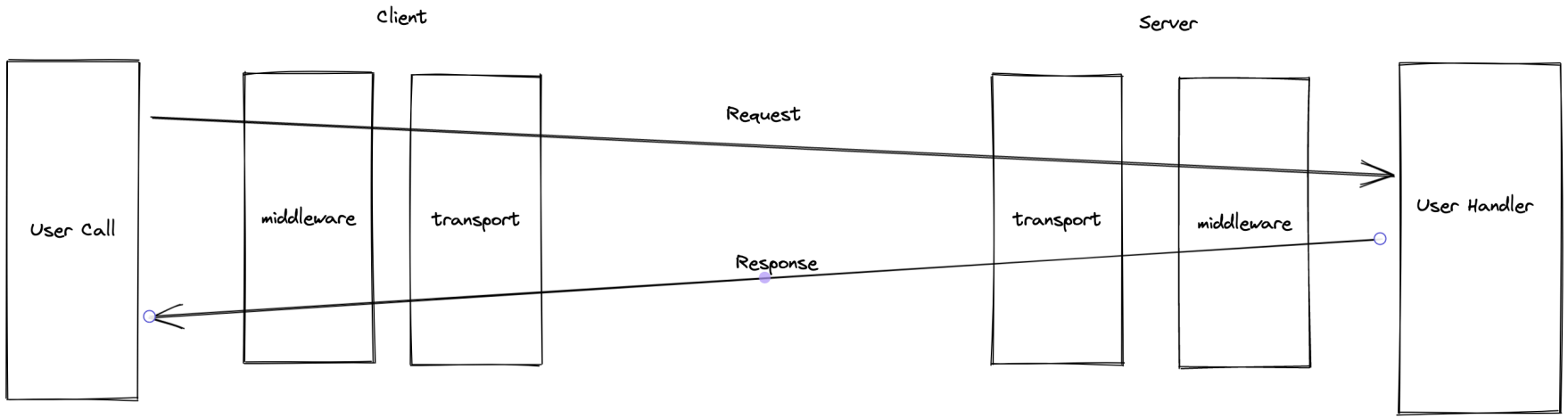


Volo 架构

RPC 调用流程

说简单也简单，说复杂也可以很复杂

- 简单在于主流程清晰明了
- 复杂在于隐藏的服务治理、性能优化、易用性
    - 比如调用失败怎么办?



一次 RPC 调用

定义 IDL

以 Thrift 格式为例：

- 定义 Service 接口方法
- 定义请求参数 Req
- 定义返回结果 Resp

根据 IDL 中的定义生成相对应的 Rust 代码

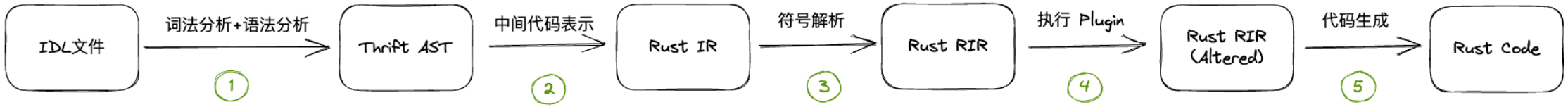
- build.rs 在编译时完成代码生成逻辑
- 通过 volo-build 来实现
- volo-build 依赖 pilota 的能力

```
namespace rs volo.example

struct Item {
    1: required i64 id,
    2: required string title,
    3: required string content,
}

struct GetItemRequest {
    1: required i64 id,
}

struct GetItemResponse {
    1: required Item item,
}

service ItemService {
    string HelloWorld (1: string req),
    GetItemResponse GetItem (1: GetItemRequest req)
}
```

Thrift Spec

Pilota

类似于完整编译器的实现过程：




```
struct Item {
    1: required i64 id,
    2: required string title,
    3: required string content,

    10: optional map<string, string> extra,
}
```

pilota

```
#[derive(Debug, Default, Clone, PartialEq)]
pub struct Item {
    pub id: i64,
    pub title: ::std::string::String,
    pub content: ::std::string::String,
    pub extra: ::std::option::Option<
        ::std::collections::HashMap<::std::string::String, ::std::string::String>,
    >,
}

#[::async_trait::async_trait]
impl ::pilota::thrift::Message for Item {
    fn encode<T: ::pilota::thrift::TOutputProtocol>(
      &self,
      protocol: &mut T,
    ) -> ::std::result::Result<(), ::pilota::thrift::Error> {
      xxx
    }

    fn decode<T: ::pilota::thrift::TOutputProtocol>(
      protocol: &mut T,
    ) -> ::std::result::Result<Self, ::pilota::thrift::Error> {
      xxx
    }
}
```

如何成为桥梁？

用户预期中的使用方式：

- Client 侧
  1. 提供配置参数得到 client
  2. 通过 client 直接调用相关方法
- Server 侧
  1. 定义 S 并实现相关方法
  2. 提供配置参数以及 S 给 server，开始提供服务

```
service ItemService {
    string HelloWorld (1: string req),
    GetItemResponse GetItem (1: GetItemRequest req),
}
```

用户接口

```
pub struct Client { }

impl Client {
  fn hello_world(&self, req: String) -> Result<String, Error> { }
  fn get_item(&self, req: GetItemRequest) -> Result<GetItemResponse, Error>
}}

pub struct Server<S: ItemService> {}
```

循环引用

Thrift

```
struct A {
    1: required B b,
}

struct B {
    1: A a,
}
```

Rust

```
struct A {
    b: B,
}

struct B {
    a: Option<A>,
}
```

Error: recursive types `A` and `B` have infinite size

循环引用

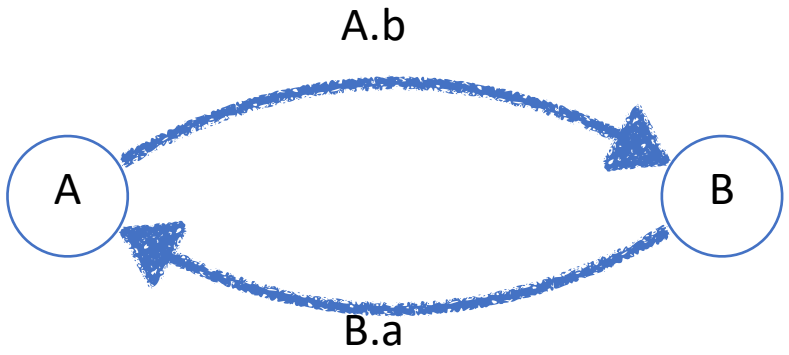
Thrift

```
struct A {
    1: required B b,
}

struct B {
    1: A a,
}
```

Rust

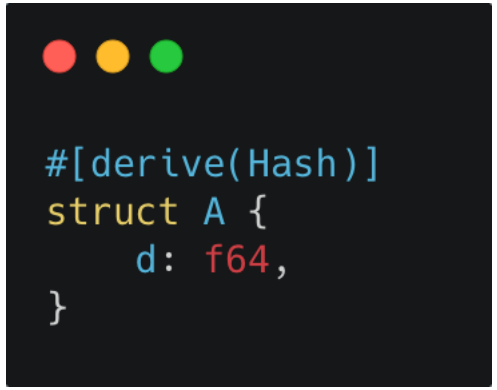
```
struct A {
    b: Box<B>
}

struct B {
    a: Option<Box<A>>
}
```

循环引用



循环依赖

用户需求: 请帮忙给所有我生成的结构带上 Hash Eq



```
#[derive(Hash)]
struct A {
    d: f64,
}
```

the trait bound `f64: std::hash::Hash` is not satisfied

循环依赖

Thrift

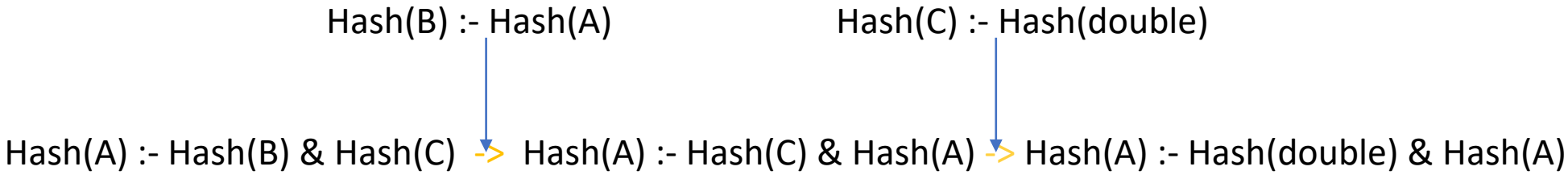
```
struct A {
  1: required B b,
  2: required C c,
}

struct B {
  1: optional A a,
}

struct C {
  1: required double d,
}
```

```
struct A {
    b: B,
    c: C,
}

struct B {
    a: Option<Box<A>>,
}

struct C {
    d: f64,
}
```

Hash(A) :- Hash(B) & Hash(C)
Hash(B) :- Hash(A)
Hash(C) :- Hash(double)

循环依赖

Hash(A) :- Hash(B) & Hash(C)
Hash(B) :- Hash(A)
Hash(C) :- Hash(double)



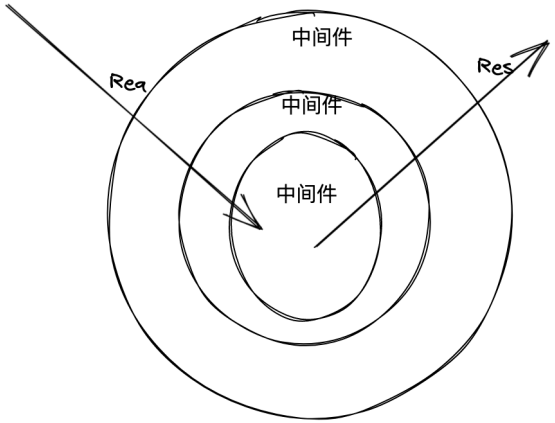
循环依赖

Hash(A) :- Hash(B) & Hash(C) -> Hash(A) :- Hash(C) & Hash(A) -> Hash(A) :- Hash(i64) & Hash(A)

Hash(A) :- Hash(A)

中间件 Service

总结来说就是洋葱模型，Service 包 Service




```rust
pub trait Service<Cx, Request> {
    type Response;
    type Error;
    type Future<'cx>: Future<Output = Result<Self::Response, Self::Error>> + Send +
'cx where
        Cx: 'cx,
        Self: 'cx;
    fn call<'cx, 's>(&'s self, cx: &'cx mut Cx, req: Request) -> Self::Future<'cx>
    where
        's: 'cx;
}
```

inventing the service trait

以 Timeout 为例

S -> Timeout<S>

S -> Log<S> -> Timeout<Log<S>>

怎么一层层的组装 Service 呢?

- Timeout::new(Log::new(S::new()))?

```rust
pub struct Timeout<S> {
    inner: S,
    duration: Duration,
}

impl<Cx, Req, S> Service<Cx, Req> for Timeout<S>
where
    Req: 'static + Send,
    S: Service<Cx, Req> + 'static + Send + Sync,
    Cx: 'static + Send,
    S::Error: Send + Sync + Into<BoxError>,
{
    type Response = S::Response;

    type Error = BoxError;

    type Future<'cx> = impl Future<Output = Result<S::Response, Self::Error>> + Send + 'cx;

    fn call<'cx, 's>(&'s self, cx: &'cx mut Cx, req: Req) -> Self::Future<'cx>
    where
        's: 'cx,
    {
        async move {
            let sleep = tokio::time::sleep(self.duration);
            tokio::select! {
                r = self.inner.call(cx, req) => {
                    r.map_err(Into::into)
                },
                _ = sleep => Err(std::io::Error::new(std::io::ErrorKind::TimedOut, "service time
out").into()),
            }
    }
}
```

组装器 Layer

引入 Layer 这个 trait

```rust
pub trait Layer<S> {
    /// The wrapped service
    type Service;

    /// Wrap the given service with the middleware, returning
    /// a new service that has been decorated with the middleware.
    fn layer(self, inner: S) -> Self::Service;
}
```

TimeoutLayer.layer(LogLayer.layer(S::new()))

```rust
#[derive(Clone)]
pub struct Stack<Inner, Outer> {
    inner: Inner,
    outer: Outer,
}

impl<S, Inner, Outer> Layer<S> for Stack<Inner, Outer>
where
    Inner: Layer<S>,
    Outer: Layer<Inner::Service>,
{
    type Service = Outer::Service;

    fn layer(self, service: S) -> Self::Service {
        let inner = self.inner.layer(service);

        self.outer.layer(inner)
    }
}
```

LogLayer -> Stack<LogLayer, TimeoutLayer> -> Stack<MetricsLayer, Stack<LogLayer, TimeoutLayer>>

Timeout<Log<Metrics<S>>>

Motore VS Tower

Motore 相比于 Tower 的改变：

- 去掉了 poll_ready 方法
  - 绝大多数时候并不需要
  - 在 call 方法里也能做到
- call 方法多了一个泛型 Cx 请求参数
  - 减少不必要的 clone
  - 减少 Box 开销

```rust
pub trait Service<Cx, Request> {
    type Response;
    type Error;
    type Future<'cx>: Future<Output = Result<Self::Response, Self::Error>> + Send +
'cx where
        Cx: 'cx,
        Self: 'cx;
    fn call<'cx, 's>(&'s self, cx: &'cx mut Cx, req: Request) -> Self::Future<'cx>
    where
        's: 'cx;
}
```

motore

```rust
pub trait Service<Request> {
    type Response;
    type Error;
    type Future: Future
    where
        <Self::Future as Future>::Output == Result<Self::Response,
Self::Error>;
    fn poll_ready(
        &mut self,
        cx: &mut Context<'_>
    ) -> Poll<Result<(), Self::Error>>;
    fn call(&mut self, req: Request) -> Self::Future;
}
```

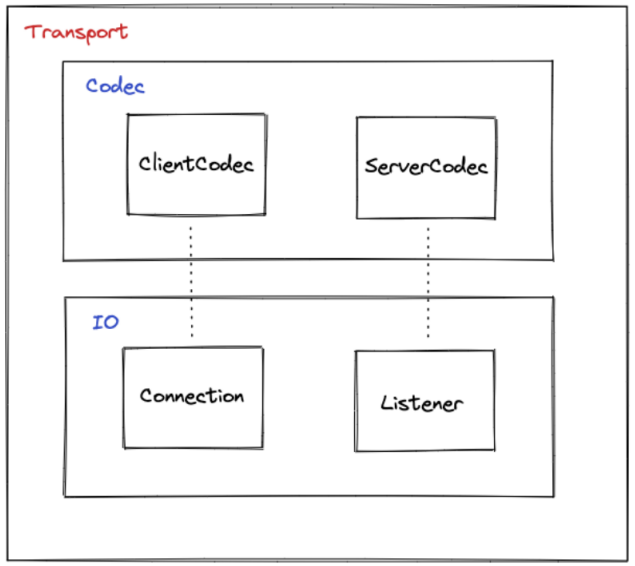
tower

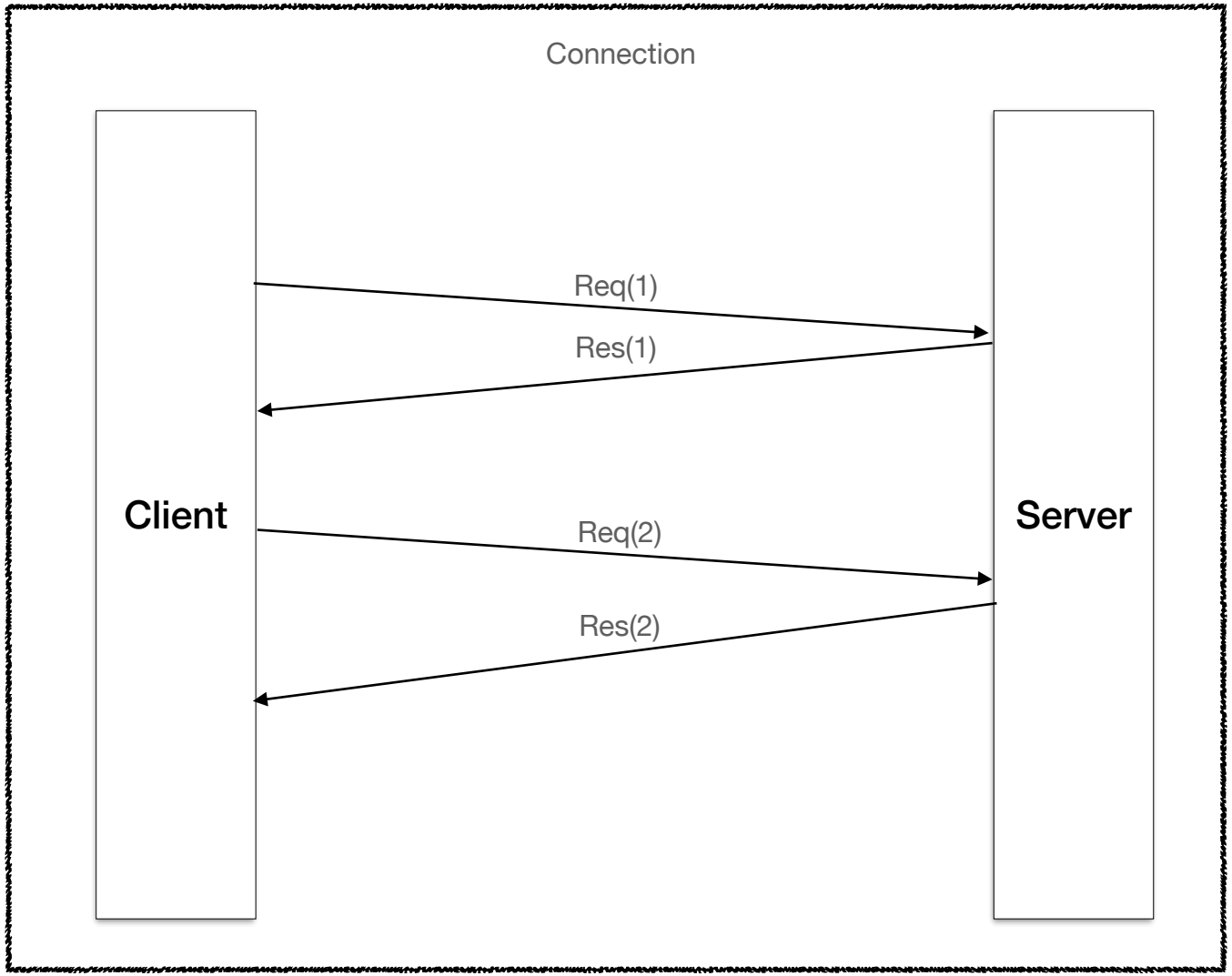
Transport = Codec + IO

Ping Pong 模型：一发一收

Oneway 模型：只发不收

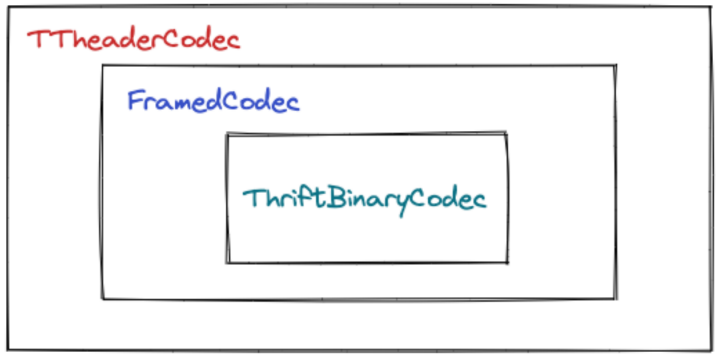
Multiplex 模型：一直发一直收





Codec

总结来说就是嵌套

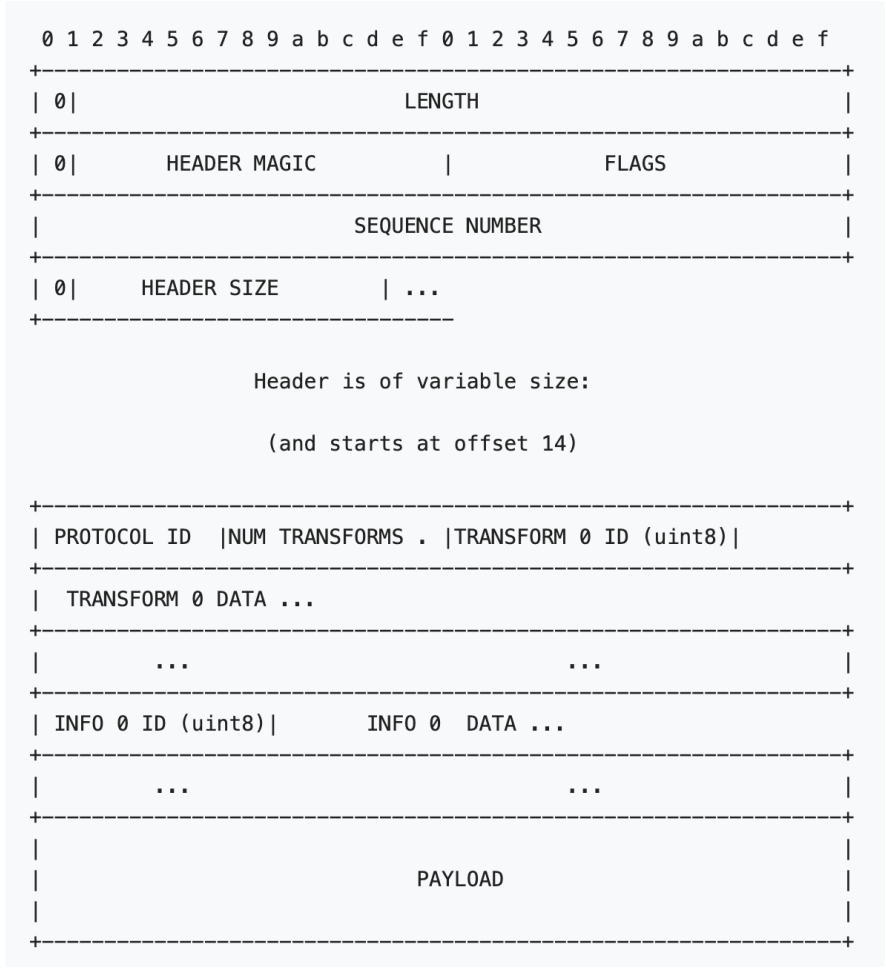



```rust
#[async_trait::async_trait]
pub trait Decoder: Send + 'static {
    async fn decode<Msg: Send + EntryMessage, Cx: ThriftContext>
(       &mut self,
        cx: &mut Cx,
    ) -> Result<Option<ThriftMessage<Msg>>, crate::Error>;
}

#[async_trait::async_trait]
pub trait Encoder: Send + 'static {
    async fn encode<Req: Send + EntryMessage, Cx: ThriftContext>
(       &mut self,
        cx: &mut Cx,
        msg: ThriftMessage<Req>,
    ) -> Result<(), crate::Error>;
}
```

- Client
  - 写入协议头，给一层Codec继续写入
- Server
  - 协议探测自适应
    - 读出并检测协议头，不是对应协议就原封不动给下一层Codec

TTHeader

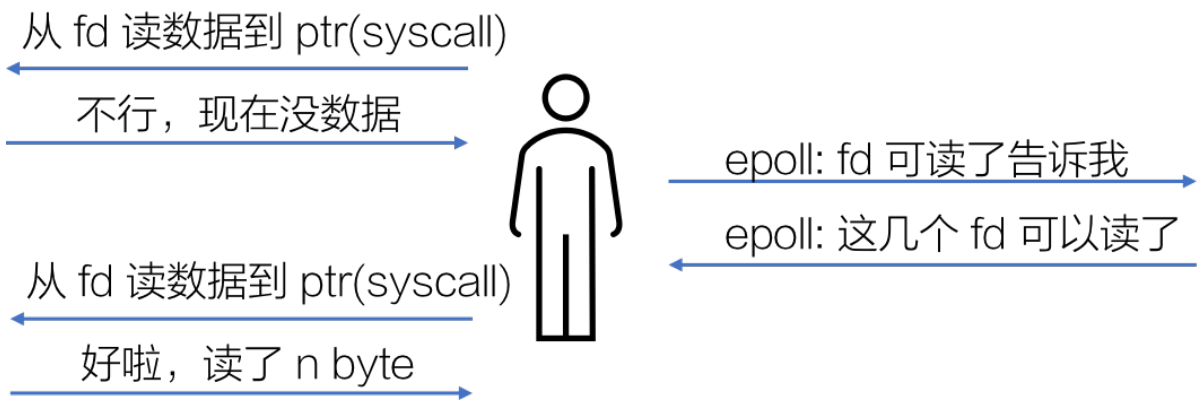
1. `LENGTH` 字段 32bits，包括数据包剩余部分的字节大小，不包含 `LENGTH` 自身长度
2. `HEADER MAGIC` 字段 16bits，值为：0x1000，用于标识 TTHeaderTransport
3. `FLAGS` 字段 16bits，为预留字段，暂未使用，默认值为 `0x0000`
4. `SEQUENCE NUMBER` 字段 32bits，表示数据包的 seqId，可用于多路复用，最好确保单个连接内递增
5. `HEADER SIZE` 字段 16bits，等于头部长度字节数 /4，头部长度计算从第 14 个字节开始计算，一直到 `PAYLOAD` 前（备注：header 的最大长度为 64K）
6. `PROTOCOL ID` 字段 uint8 编码，取值有：
   - ProtocolIDBinary = 0
   - ProtocolIDCompact = 2
7. `NUM TRANSFORMS` 字段 uint8 编码，表示 `TRANSFORM` 个数
8. `TRANSFORM ID` 字段 uint8 编码，具体取值参考下文
9. `INFO ID` 字段 uint8 编码，具体取值参考下文
10. `PAYLOAD` 消息内容



https://www.cloudwego.io/zh/docs/kitex/reference/transport_protocol_ttheader/

IO

Epoll 模型，采用 Tokio 抽象出的接口




```rust
pub trait AsyncRead {
    fn poll_read(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
        buf: &mut ReadBuf<'_>,
    ) -> Poll<io::Result<()>>;
}

pub trait AsyncWrite {
    fn poll_write(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
        buf: &[u8],
    ) -> Poll<Result<usize, io::Error>>;
```

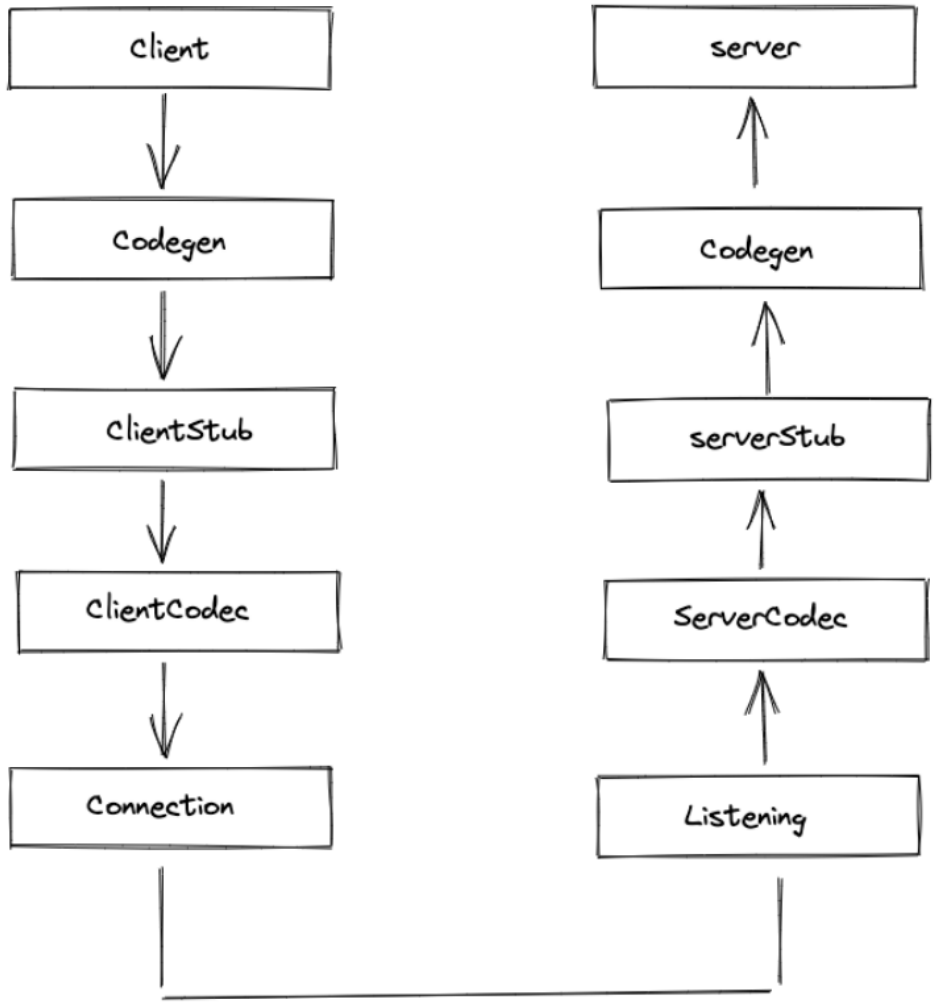
Put it together

用户代码

生成代码

框架代码

通过 Service 串起来



Volo 总览

Volo 是 RPC 框架的名字，包含了 Volo-Thrift 和 Volo-gRPC两部分

Volo-rs 组织：Volo 的相关生态

Pilota：Volo 使用的 Thrift 与 Protobuf 编译器及编解码的纯 Rust 实现

Motore：Volo 参考 Tower 设计的，使用了 GAT 和 TAIT 的中间件抽象层

03 Get Your Hands Dirty

从一个示例开始

https://www.cloudwego.io/zh/docs/volo/volo-thrift/getting-started/

源码解读

https://github.com/cloudwego/volo

上机实操

实现一个 mini-redis：

- 基础：
  - get set(expired) del ping
  - 写一个中间件filter，可以过滤部分请求
- 进阶：
  - publish subscribe

演示：client 请求 server 返回相对应结果（ bonus：封装一个 client-cli 进行请求 ）

https://redis.io/commands/

THANKS

CloudWeGo