CloudWeGo

代码生成工具

为什么如此复杂

CloudWeGo 开源团队出品

2023/05/28

个人介绍

刘翼飞

- CloudWeGo-Volo Maintainer
- CloudWeGo-Pilota Maintainer

CONTENT

目录

01 为什么需要代码生成工具

IDL

Thrift

```
namespace rs hello

struct HelloRequest {
    1: required string name,
}

struct HelloResponse {
    1: required string message,
}

service HelloService {
    HelloResponse Hello (1: HelloRequest req),
}
```

IDL

Thrift

```
namespace rs hello

struct HelloRequest {
    1: required string name,
}

struct HelloResponse {
    1: required string message,
}

service HelloService {
    HelloResponse Hello (1: HelloRequest req),
}
```

Rust

```
struct HelloRequest {
    name: String,
}

struct HelloResponse {
    message: String,
}

trait HelloService {
    fn hello(req: HelloRequest),
}
```

编解码逻辑

```
trait Message {
   fn encode<B: bytes::BufMut>(&self, buf: B) -> Result<(), Error>;
   fn decode<B: bytes::BufMut>(buf: B) -> Result<Self, Error> where Self: Sized;
}
```

开发者的使用方式

```rust
// build.rs

fn main() {
    volo_build::Builder::thrift().add_service("hello.thrift").write();
}
```

```rust
// main.rs

volo::include_service!();

fn main() {
    let client = volo_gen::HelloServiceClient::new("hello");
    client.hello(volo_gen::HelloRequest { name: "hello".to_string() });
}
```

这样调用 RPC 也太方便了吧！

02 生成 Rust 代码的挑战

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

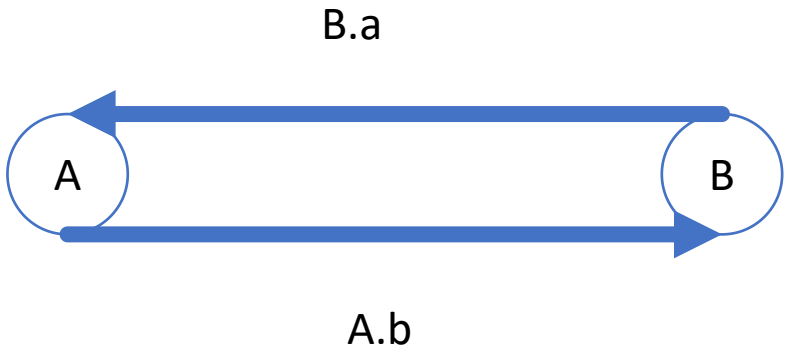
Thrift

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

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

Rust

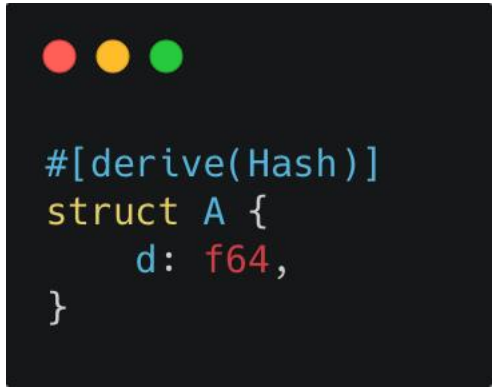
```
struct A {
    b: Box<B>
}

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

循环引用



还是循环依赖

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

还是循环依赖

Thrift

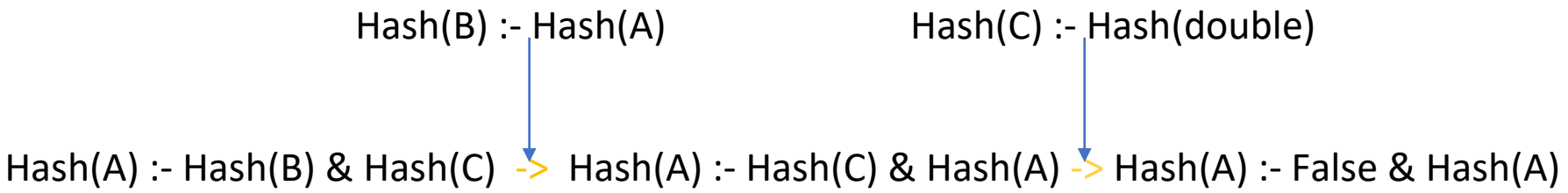
```
struct A {
  1: required B b,
  2: required C c,
}

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)

还是循环依赖

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



还是循环依赖

Hash(C) :- Hash(i64)

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

Hash(A) :- Hash(A)

Static Value

Thrift

```
const String A = "hello world";
```

Static Value

Thrift

```
const String A = "hello world";
```

Rust

```
const A: &str = "hello world";
```

Static Value

Thrift

```
const map<i32, list<string>> TEST_MAP_LIST = {
    1: ["hello"]
}
```

Rust

```
::pilota::lazy_static::lazy_static! {
    pub static ref TEST_MAP_LIST: ::std::collections::HashMap<i32, ::std::vec::Vec<&'static str>> = {
        let mut map = ::std::collections::HashMap::with_capacity(1);
        map.insert(1i32, ::std::vec!["hello"]);
        map
    };
}
```

Static Value

Thrift

```
const HELLO string = "hello world";

struct A {
    1: required string a = "hello world",
}
```

Rust

const: string -> &’static str
non-const: string -> String?

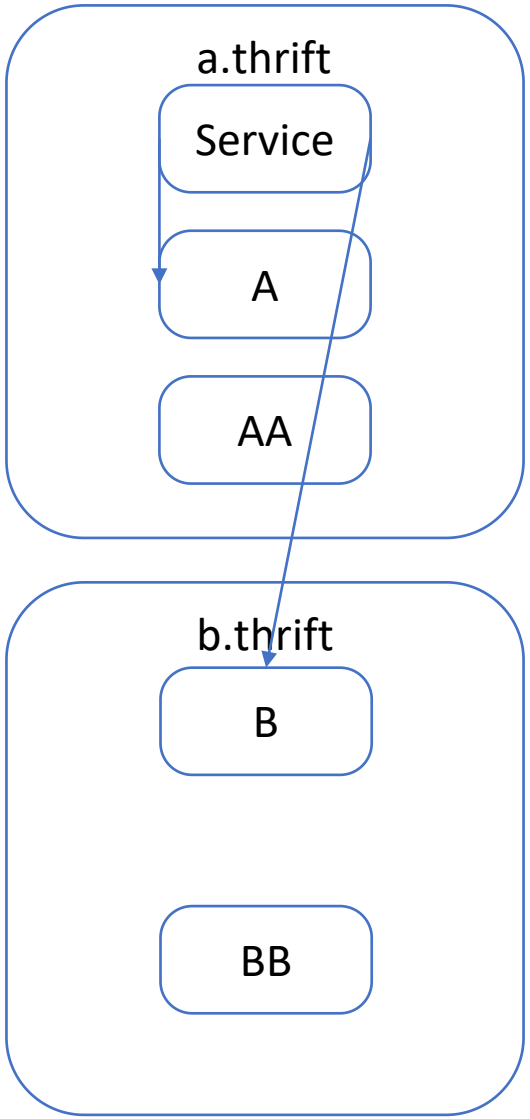
“hello world” → &’static str -> “hello world”

“hello world” → String -> “hello world”.to_owned()

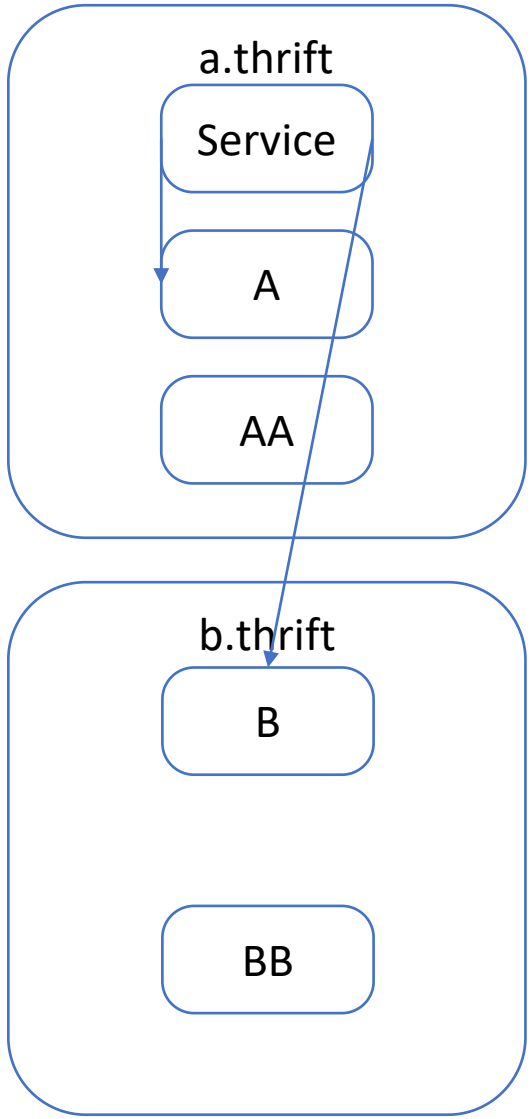
庞大的代码生成量

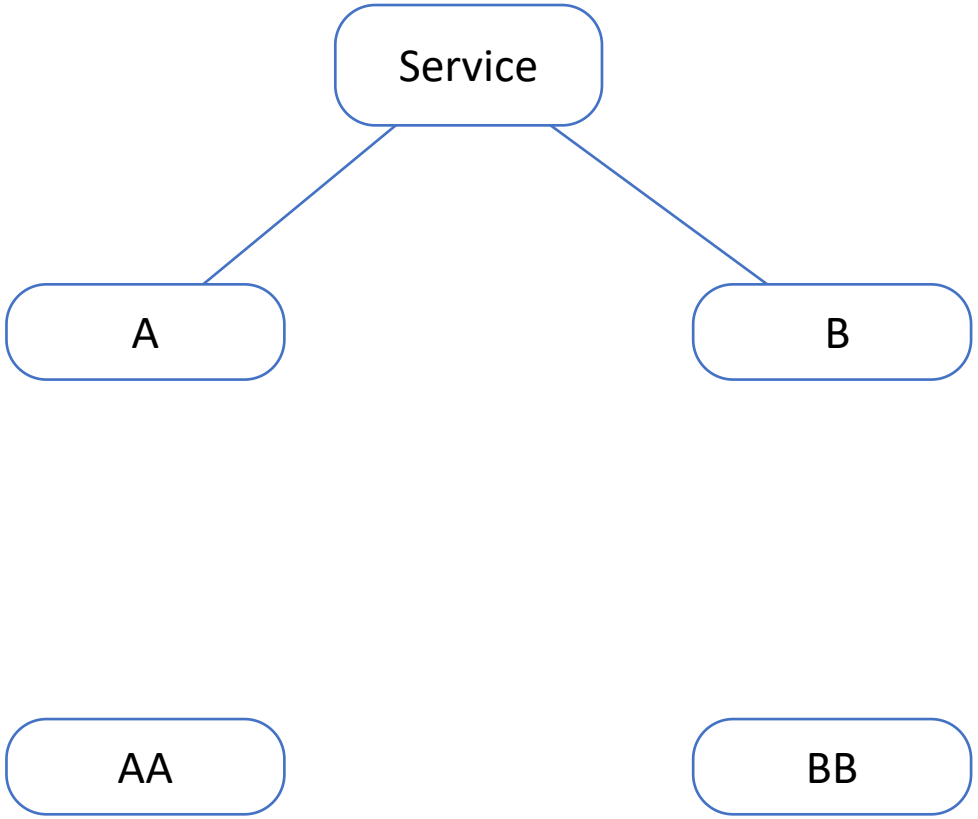
IDL 生成了150万行代码，cargo check 用了10分钟

庞大的代码生成量

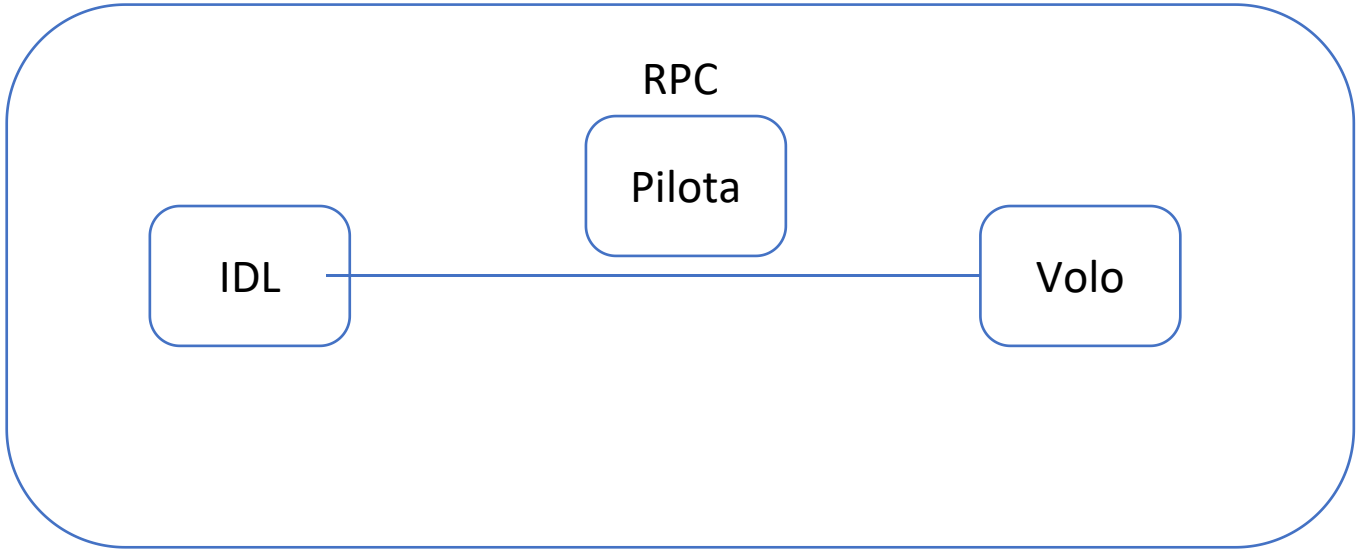


庞大的代码生成量





所以怎么解决这一堆问题呢

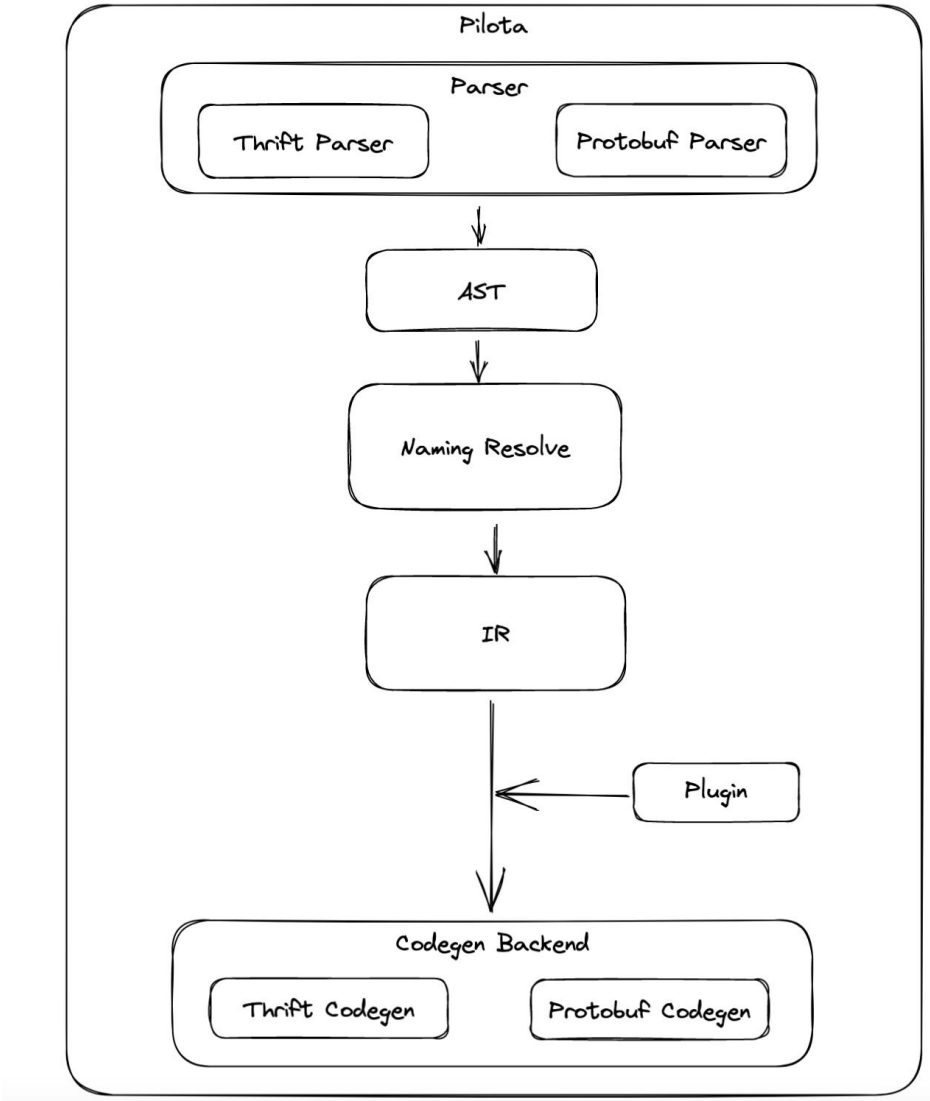


Volo: Volo 是字节跳动服务框架团队研发的 **高性能、可扩展性强** 的 Rust RPC 框架

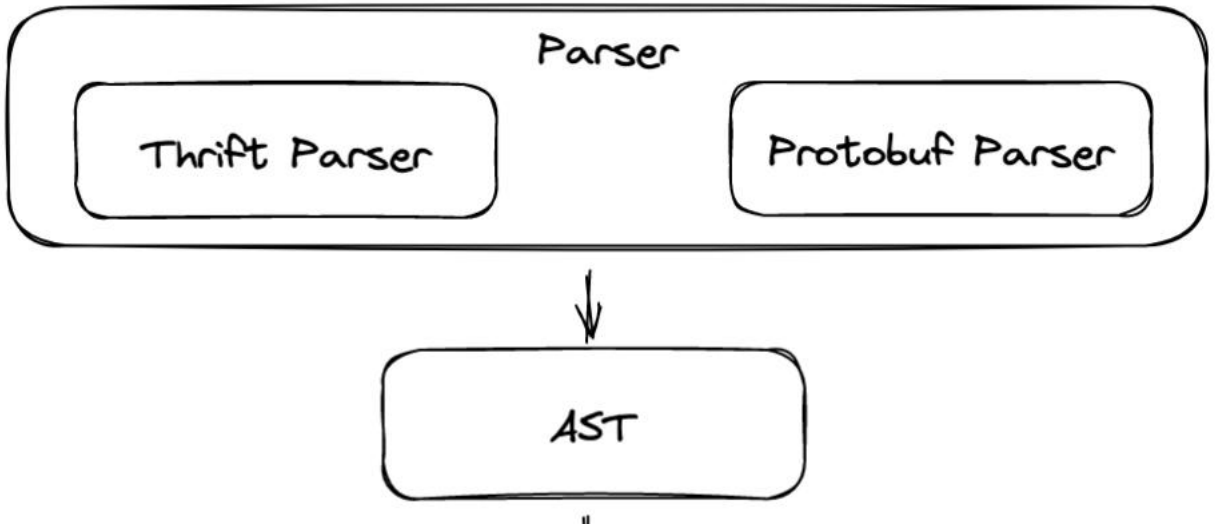
Pilota: 通过 IDL 生成 Rust 代码，提供给 Volo 框架 和 用户使用

03 Pilota 的设计架构

设计结构


Pilota
Parser
Thrift Parser
Protobuf Parser
AST
Naming Resolve
IR
Plugin
Codegen Backend
Thrift Codegen
Protobuf Codegen

多种 IDL



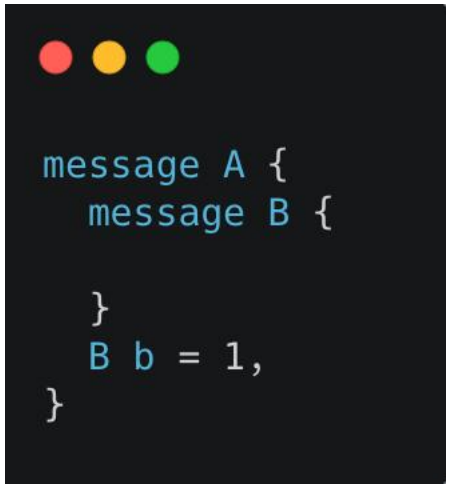
只要转换成 Pilota 自有的 AST 格式，即可兼容任意形式的IDL

Naming Resolve

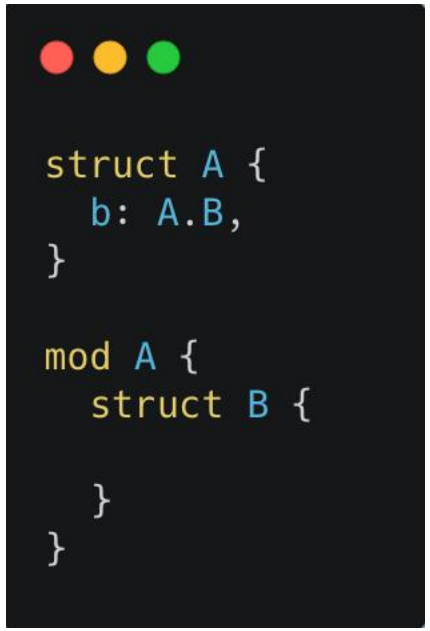
```
// file b.ast

struct B {

}


// file a.ast

mod Test {

  struct B {

  }

  struct A {
    b: B,
  }
}

struct B {

}

struct Test {
  b1: Test.B,
  b2: b.B,
  b3: B,
}
```

为什么存在同名的 `Test` mod 和 `Test` struct?

符号的 NameSpace

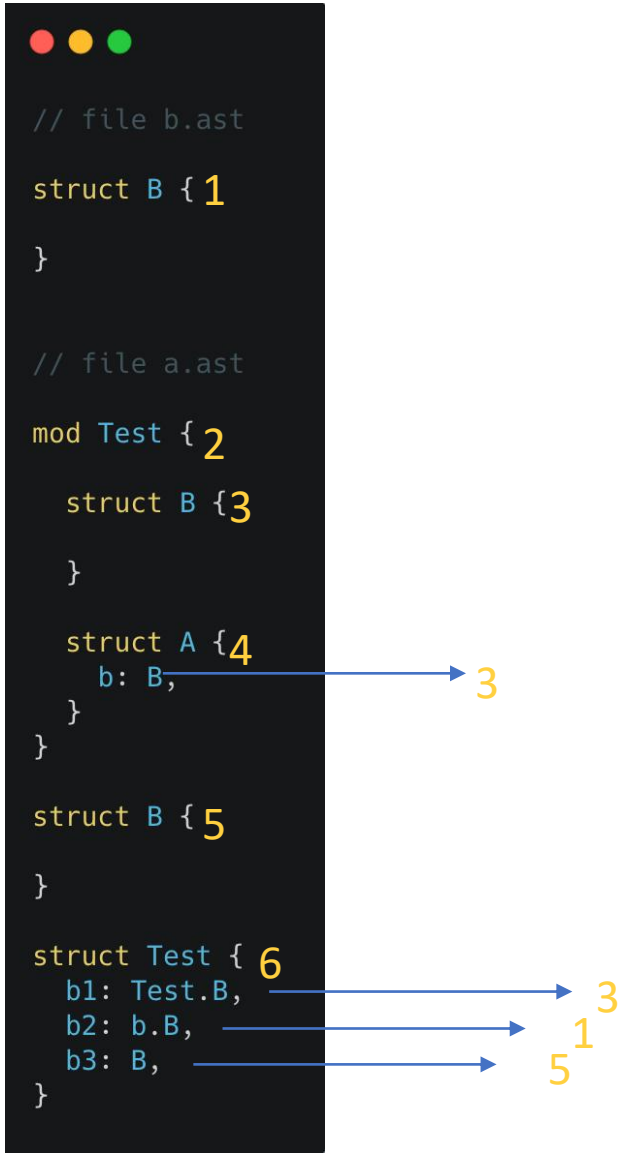


```
message A {
  message B {

  }
  B b = 1,
}
```



```
struct A {
  b: A.B,
}

mod A {
  struct B {

  }
}
```

Naming Resolve



Plugin

用户：我有个需求，我希望所有的结构都带上 Serde 的Derive，并且还要自定义 Serde attribute

```
struct A {
    1: required string a(pilota.serde_attribute = "#[serde(rename = \"AA\")]"),
    2: required i32 b,
}
```

```
#[derive(serde::Serialize, serde::Deserialize)]
struct A {
    #[serde(rename = "AA")]
    a: string,
    b: i32,
}
```

Plugin

Struct, Field, Enum, Variant

```rust
#[derive(Default)]
3 implementations
pub struct Adjust {
    boxed: bool,
    attrs: Vec<FastStr>,
    pub(crate) nested_items: Vec<FastStr>,
}
```

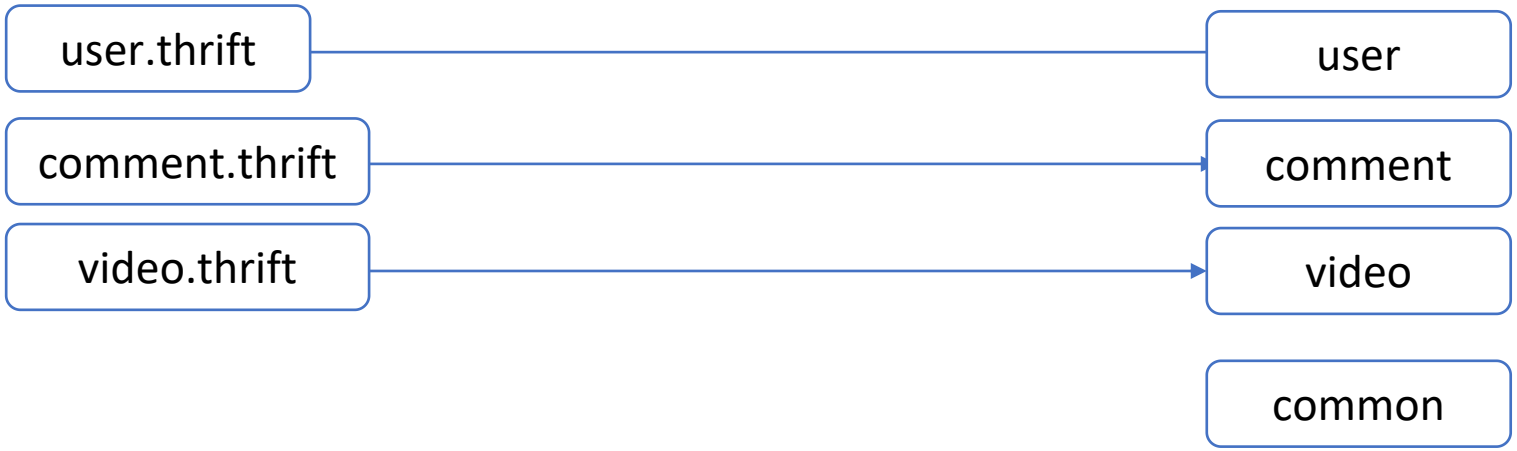
attrs: 调整生成的 attributes
nested_items: 在相邻位置插入代码

04 Pilota 的未来

用户还是有问题

1. 全部生成到一个文件里，代码量还是过大？
2. 希望用户完全不感知idl的存在？

Crate 生成



充分利用编译器缓存

THANKS

CloudWeGo