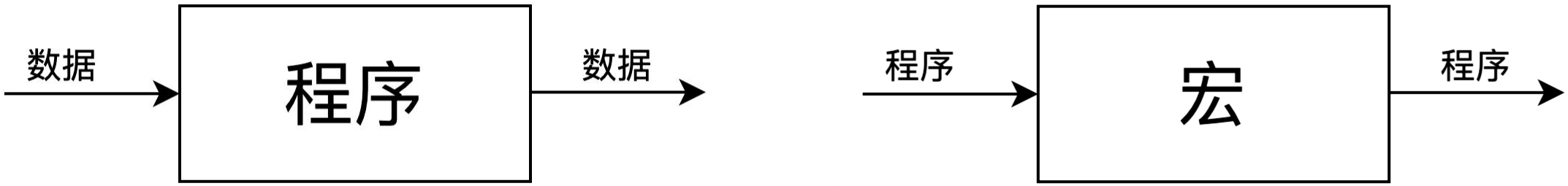
Rust 过程宏入门

尹思维 2021-04-18

什么是宏 (Macro) ?

- (过程) 宏本身也是一段程序，用来产生程序的程序
- 宏可以用来创造语法 (DSL)



实现基于宏的 HTTP Router

```rust
#[routes(get "hello$")]
async fn hello() ->
anyhow::Result<Response<Body>> {
    Ok(Response::builder()
        .status(StatusCode::OK)
        .body(Body::from("hello"))?)
}

#[routes(get "world$")]
async fn world() ->
anyhow::Result<Response<Body>> {
    Ok(Response::builder()
        .status(StatusCode::OK)
        .body(Body::from("world"))?)
}
```

```rust
routes_group!(pub root "" {
    "this" => world,
    "another" => {
        "thing" => {
            hello,world
        }
    }
});
```

```rust
#[tokio::main]
async fn main() {
    let builder = hyper::Server::bind(&"127.0.0.1:4000".parse().unwrap());
    serve(&root::endpoint, builder).await.unwrap();
}
```

实现基于宏的 HTTP Router

```
$ curl http://127.0.0.1:4000/this/world
world

$ curl http://127.0.0.1:4000/another/thing/hello
hello

$ curl http://127.0.0.1:4000/another/thing/world
world
```

开始整起来!

```
$ cargo init routes-macro --lib
```

```toml
[package]
name = "routes-macro"
version = "0.1.0"
authors = ["heymind <heymind@outlook.com>"]
edition = "2018"

[lib]
proc-macro = true

[dependencies]
quote = "1.0" # 用于更方便的生成代码
proc-macro2 = "1.0" # 用于更好的处理 Token 流
syn = { version = "1.0", features = ["full", "extra-traits"]} # 用于语法树解析
either = "1.6"
```

开始整起来!

```rust
#[routes(get "hello$")]
async fn hello() -> anyhow::Result<Response<Body>> {
    Ok(Response::builder()
        .status(StatusCode::OK)
        .body(Body::from("hello"))?)
}
```

在这里，**程序**被当作**数据**作为属性宏函数的输入。

syn 提供的 **`parse_macro_input!`** 完成语法解析的过程。

```rust
#[proc_macro_attribute]
pub fn routes(
    attr: proc_macro::TokenStream,
    input: proc_macro::TokenStream,
) -> proc_macro::TokenStream {
    let attr = parse_macro_input!(attr as RoutesAttr);
    let item_fn = parse_macro_input!(input as ItemFn);
    match transform(attr, item_fn) {
        Ok(ts) => ts.into(),
        Err(err) => err.into_compile_error().into(),
    }
}
```

开始整起来!

```rust
use proc_macro2::TokenStream;
use syn::{parse_macro_input, ItemFn, Result};
```

```rust
fn transform(attr: RoutesAttr, mut input: ItemFn) -> Result<TokenStream> {
 todo!()
}
```

proc_macro 这个 crate 只能在宏 crate 内使用。

proc_macro2 与 proc_macro 一样但没有上述限制。

两种 **TokenStream** 可以互相转换。（ Into ）

```rust
#[proc_macro_attribute]
pub fn routes(
    attr: proc_macro::TokenStream,
    input: proc_macro::TokenStream,
) -> proc_macro::TokenStream {
    let attr = parse_macro_input!(attr as RoutesAttr);
    let item_fn = parse_macro_input!(input as ItemFn);
    match transform(attr, item_fn) {
        Ok(ts) => ts.into(),
        Err(err) => err.into_compile_error().into(),
    }
}
```

```rust
#[routes(get "hello$")]
async fn hello() -> anyhow::Result<Response<Body>> {
    Ok(Response::builder()
        .status(StatusCode::OK)
        .body(Body::from("hello"))?)
}
```

Struct syn::ItemFn

Fields

```
attrs: Vec<Attribute>
vis: Visibility
sig: Signature
block: Box<Block>
```

Struct syn::Signature

Fields

```
constness: Option<Const>
asyncness: Option<Async>
unsafety: Option<Unsafe>
abi: Option<Abi>
fn_token: Fn
ident: Ident
generics: Generics
paren_token: Paren
inputs: Punctuated<FnArg, Comma>
variadic: Option<Variadic>
output: ReturnType
```

- 每一种 Rust 语法在 syn 都有与之对应的结构
  - ItemFn 代表整个函数
    - vis: 可见性 （pub, pub(crate))
    - sig: 函数签名
    - block: 函数体
- Signature
  - asyncness: 是否被 async 修饰
  - ident: 函数名
  - inputs: 参数
  - output: 返回类型

如何创造语法？[简易]

```
#[routes(get "hello$")]

#[routes(get,post,patch "hello$")]
```

```
#[derive(Debug)]
struct RoutesAttr {
    methods: Punctuated<Ident, Token![,]>,
    path: LitStr,
}
```

Ident： 名字（不包含命名空间），如 var_a

LitStr： 即字符串字面量，如 "1234"

Punctuated<A,B>: 即许多以 B 为间隔的 A，如 Punctuated<Ident,Token![,]> 一个合法的表示为 var_a, var_b

```
impl Parse for RoutesAttr {
    fn parse(input: ParseStream) -> Result<Self> {
        let parser = Punctuated::<Ident, Token![,]>::parse_separated_nonempty;
        Ok(Self {
            methods: parser(input)?,
            path: input.parse()?,
        })
    }
}
```

如何创造语法？[复杂]

```
routes_group!(pub root "" {
    "this" => world,
    "another" => {
        "thing" => {
            hello,world
        }
    }
});
```

```
routes_group!(pub <var-name> "<base>" {
    "<sub-path>" => <endpoint>,
    "<sub-path-group>" => {
        "<sub-path-1>" => {
            <endpoint-1>,<endpoint-2>
        },
        <endpoint-3>
    }
});
```

```
{
    <endpoint>,
    "<sub-path-1>" => <endpoint>,
    "<sub-path-2>" => { .. }
}
```

如何创造语法？[复杂]

```
{
    <endpoint>,
    "<sub-path-1>" => <endpoint>,
    "<sub-path-2>" => { .. }
}
```

```rust
#[derive(Debug)]
struct RoutesGroupItem(
    Punctuated<(Option<(LitStr, Token![=>])>, Either<Path, (Brace, Self)>), Token![,]>,
);
```

Path ： 即包含命名空间的名字，如 std::fs::read
Self ： 指代结构体本身，这是一个递归嵌套结构

如何创造语法？[复杂]

```
routes_group!(pub root "" {
    "this" => world,
    "another" => {
        "thing" => {
            hello,world
        }
    }
});
```

```
#[derive(Debug)]
struct RoutesGroup {
    vis: Visibility,
    ident: Ident,
    prefix: Option<LitStr>,
    brace: Brace,
    item: RoutesGroupItem,
}
```

```
impl Parse for RoutesGroup {
    fn parse(input: ParseStream) -> Result<Self> {
        let content;
        Ok(Self {
            vis: input.parse()?, ident: input.parse()?,
            prefix: input.parse()?,
            brace: braced!(content in input),
            item: content.parse()?,
        })
    }
}
```

- 调用 input.parse() 的顺序很重要
- braced! 用于处理花括号内的内容
- *对于RoutesGroup 内的成员要求实现 Parse trait 才可以用这种形式解析

如何创造语法？[复杂]

```rust
#[derive(Debug)]
struct RoutesGroupItem(
    Punctuated<(Option<(LitStr, Token![=>])>, Either<Path, (Brace, Self)>), Token![,]>,
);
```

```rust
impl Parse for RoutesGroupItem {
    fn parse(input: ParseStream) -> Result<Self> {
        fn parse_inner(
            input: ParseStream,
        ) -> Result<(
            Option<(LitStr, Token![=>])>,
            Either<Path, (Brace, RoutesGroupItem)>,
        )> {
            // ...
        }
        Ok(Self(Punctuated::parse_terminated_with(input, parse_inner)?))
    }
}
```

如何创造语法？[复杂]

```
{
    <endpoint>,
    "<sub-path-1>" => <endpoint>,
    "<sub-path-2>" => { .. }
}
```

```rust
fn parse_inner(
    input: ParseStream,
) -> Result<(
    Option<(LitStr, Token![=>])>,
    Either<Path, (Brace, RoutesGroupItem)>,
)> {
    let mut prefix = None;
    let endpoint;
    if input.lookahead1().peek(LitStr) {
        prefix = Some((input.parse()?, input.parse()?));
    }
    let ahead = input.lookahead1();
    if ahead.peek(Brace) {
        let content;
        endpoint = Either::Right((braced!(content in input), content.parse()?));
    } else {
        endpoint = Either::Left(input.parse()?);
    }
    Ok((prefix, endpoint))
}
```

代码生成
来不及了，下次再聊

谢谢