10Rust中的宏

2020年3月9日 12:20

在Rust中,分为声明宏(Declarative)和三种过程宏.

(1): 声明宏
使用macro_rules!定义.

(2): 三种过程宏
    1: 自定义宏(#[derive]), 在结构体和枚举上通过derive属性添加的代码
    2: 类属性(Attribute)宏定义可用于任意项的自定义属性.
    3: 类函数宏看起来像函数,不过作用于作为参数传递的token.

在调用宏之前必须定义并将其引入作用域,而函数可以在任何地方定义和调用.

一: 声明宏,在Rust中最常用的就是声明宏.

```
#[macro_export] // 宏可以被别的crate引入作用域
macro_rules! vec {
    ($($x:expr), *) => {
        let mut temp_vec = Vec::new();
        $(
            temp_vec.push($x);
        )*
        temp_vec
    };
}
```

注意:上面的声明宏,虽然可以在模块中定义,但是use的时候,宏其实是在crate的根部.

```
hades_lib-- |
            |--src----  |
                        |---lib.rs
                        |
                        |---my_mod.rs ---    |--macro定义在了这个模块中
                                             |
```

在使用的时候是:use hades_lib::vec; 而不是use hades_lib::my_mod::vec;
也就是说声明宏无论在哪里定义,最后use的时候都是在crate根引用作用域的.

十分注意:
(1): 声明宏可以在二进制的可运行程序中,也可以在lib crate中. 可以存在模块.
(2): 过程宏的实现,必须在其自己的 crate 内。该限制最终可能被取消. 不能存在模块.

```
    error: `proc-macro` crate types currently cannot export any items other
than functions tagged with `#[proc_macro]`, `#[proc_macro_derive]`, or
`#[proc_macro_attribute]`
  --> src/lib.rs:13:1
   |
    13 | pub mod my_mod;
```

    意思就是说除了用#[proc_macro]`, `#[proc_macro_derive]`, or
`#[proc_macro_attribute]`这些标记的函数可以导出(pub), 其他的都不能导出,也就是pub出来.
    必须在lib.rs中使用pub mod my_mod.rs,报错,去掉pub是可以的.

在lib.rs中,pub trait Nihao {..},是错误的,去掉pub是可以的.
也就是说除了那三个标志修饰的函数可以导出,其他的都不能导出.
还有过程宏的定义必须在lib.rs中,也就是说在crate的根下定义.

实现一个过程宏,需要一个trait crate,需要一个过程宏的实现crate.
由于两个 crate 紧密相关，因此在 hello_macro 包的目录下创建过程式宏的 crate。如果改变在 hello_macro 中定义的 trait ，同时也必须改变在 hello_macro_derive 中实现的过程式宏。这两个包需要分别发布，编程人员如果使用这些包，则需要同时添加这两个依赖并将其引入作用域。我们也可以只用 hello_macro 包而将 hello_macro_derive 作为一个依赖，并重新导出过程式宏的代码。但我们组织项目的方式使编程人员使用 hello_macro 成为可能，即使他们无需 derive 的功能。

```
DeriveInput {
    // --snip--

    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

item: 结构体，函数，mod 之类的
block: 用大括号包起来的语句或者表达式，也就是代码块
stmt: 一段 statement
pat: 一段 pattern
ty: 一个类型
ident: 标识符
path: 类似 foo::bar 这种路径
meta: 元类型，譬如 #[...], #![...] 内的东西
tt: 一个 token tree

(1):
自定义派生宏:

```
#[proc_macro_derive(Trait)]
fn ....
```

(2):
类属性宏:
定义:

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream{
}

// attr:就是属性本身,也就是GET, "/"部分
// item就是属性标记的项, 就是fn index(){} 和剩下的函数体
```

```
#[route(GET, "/")]
fn index {}
```

(3):
类函数宏

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {}
```

使用:

```
let sql = sql!(SELECT * FROM posts where id=1);
```

注意:
#[macro_use] 和use std::my_macro的区别.

```
#[macro_use]
extern crate my;
```

会把my这个crate中的所有宏,导入到当前的crate中来.可以直接使用.

use hades::my_macro; 只是把my_macro这个宏导入到了当前的模块中了,只能在这个模块中使用,别的模块不能使用,其他的宏并未导入.