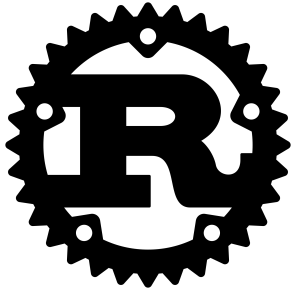

TinTin

第四课：类型、Trait

Mike Tang

daogangtang@gmail.com

2023-5-31

同一个二进制值，由类型赋予其意义

比如，01100001 这个二进制数字，同样的内存表示，如果是整数，就表示97这个整数。如果是字符，就表示 'a' 这个 char. 如果没有类型去赋予它额外的信息，你看到这串二进制编码，是不知道他的意义是什么的。

类型化的好处

类型化有5大好处：正确性，不可变性，封装性，组合性，可读性。

这5大好处都是一个良好的软件工程所推崇的。

类型是一种约束

a: u8

a: u64

s: String

Rust中的 冒号:统一表示一种约束。

Rust的类型体系

Rust有严密的类型体系。从底层到上层构建了一套完整严密的类型大厦。

几大武器：

- 结构体
- 枚举
- 泛型-洋葱结构 A<B<C<D<E<T>>>>>
- type 关键字

泛型(参数化类型、类型参数)

泛型是复用代码的方式。可以让写出的代码更紧凑。

需求是这样的:很多不同的类型，其实它们实现某个逻辑时，需求是通的。因此如果没有泛型，就得对每个具体的类型，重新实现一次，这样就显得代码很臃肿，重复的代码也不好维护，容易出错。

而有了泛型，这样的需求代码只需要写一份，让编译器来帮我们分析具体到时要用到多少种不同的类型上。最典型的就是排序操作。

函数参数中的泛型

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=f85304a2d0c8120397590506d260a466

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=a7b9b2177630ce8dd2b91f6a63c706ad

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=7bcd8b30dab3b2a693d5298ddfa09429

结构体中的泛型

结构体由一些字段组成。字段是有类型的，这个类型当然适用于泛型。因此结构体中，是可以出现泛型的。

在结构体类型名后定义泛型名称，在字段里面使用。

```
struct Point<T> {
    x: T,
    y: T,
}
fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```


```
struct Point<T> {
    x: T,
    y: T,
}
fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=430c9f8318c92e14869a9ee4cb4f8e28

不止一个泛型参数

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=246d56e6e46b561ea74862f7553bc0c7

PhantomData（了解）

```
struct A<T> {
    _x: PhantomData<T>,
}
```

https://doc.rust-lang.org/std/marker/struct.PhantomData.html

枚举中的泛型

枚举是各种带类型参数的变体的和类型，每个类型参数位置当然可以出现泛型。

最常见的两个泛型

比如最常见的两个，就是泛型。

```
enum Option<T> {
    Some(T),
    None,
}
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

方法中的泛型

函数中可以有泛型，结构体中可以有泛型，它们组合起来，当然可以有泛型。

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=0c03708b2e421e28c2b282e89087149f

实现单态化方法

单态是相对于多态来说的。可以在实现方法的时候，给具体某一个态实现方法（所以叫单态化方法）

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

TinTin

方法中的泛型参数与结构体中的可以不同

方法中的泛型参数与结构体中的可以不同

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=bdddc2f3d83e1509f0f1945af401e586

Trait

特征。

Trait实际是对类型的约束

trait实际是对类型的约束，或者说是代表了一类(满足条件的)类型。它与Haskell中的typeclass类似，实际对类型高一层次的抽象。

T: Animal

T: VegeAnimal

类型的约束依赖

trait VegeAnimal: Animal

这种表示，怎么理解。是不是有点像 OOP中的类的继承？

前面我们讲过，Rust中的 :统一表示约束关系。

T: VegeAnimal

T: Animal

trait VegeAnimal: Animal

前面两个好理解：

- 第一行表示，所有素食 动物，或指代其中的一种。将 T的可能范围约束到了VegeAnimal
- 第二行表示，所有 动物，或指代其中的一种。将 T的可能范围约束到了Animal
- 第三行表示，VegeAnimal所约束的类型范围必然属于 Animal所约束的类型范围。比如一个 类型属于 Animal 约束的范围，但它不一定在 VegeAnimal所约束的范围中。所以，冒号前面的 约束所约束的范围是冒号后面的 约束所约束的范围的子集

其实还有这种表示：

```
T: Animal + Tame

trait VegeAnimal: Animal + Tame

T: VegeAnimal
```

这个表示：

- 第一行，T必须同时实现 Animal和 Tame. T的类型范围为动物 并且 能够被驯化。因为有的动物无法被驯化。所以这时这个+号，表示两个集合的交集关系。
- 第二行，表示，实现了 VegeAnimal 这个trait的类型，必须同时也实现了 Animal 和 Tame 两个 trait. 也就是说，在Animal和 Tame约束的类型范围里面，又加了一层 VegeAnimal 约束，这样。符合 VegeAnimal 约束的类型的范围，一定属于 Animal 和 Tame 所约束的类型范围的交集的子集。
- 第三行，就是第二行的使用。表示 T被VegeAnimal约束。

Where

Where关键字可用来把约束关系统一放在后面，这样更清晰一点。特别是当泛型和约束有多个的时候。

```
fn some_function<T: VegeAnimal + Tame, U: Clone + Debug>(t: &T, u: &U) ->
i32 {}
```

```
fn some_function<T, U>(t: &T, u: &U) -> i32

where

    T: VegeAnimal + Tame,

    U: Clone + Debug,

{}
```

trait 可以用 Self 这个符号

```
trait Animal {

  fn myself(self) -> Self;

  fn eat(&self);

  fn eat_mut(&mut self);

}
```

增强约束实现

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=4652a773ab1be7f1c225d332e1af8081

Blanket Implementation

```
impl<T: Display> ToString for T {
    // --snip--
}
```

Blanket Implementation 与单个实现的冲突

统一实现后，就不要再对某个具体的参数再实现一次了。因为同一个trait只能实现一次到某个类型上。这个不像 impl 直接到类型上，可以实现多次（函数名需要不冲突）。

trait的函数

函数中第一个参数不是Self类型（self, &self, &mut self）的方法就是只属于trait本身的方法。

```
trait Default {

    // function

    fn default() -> Self;

}
```

关联类型

```
pub trait Iterator {
    type Item;

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

Self::Item 实际是 <Self as Iterator>::Item.

trait使用中指定关联类型

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=93a318b25b4a86e5c3ea8bc2475cf362

对关联类型的约束

```
trait StreamingIterator {

    type Item: Debug + Display;

}
```

在类型上直接带关联类型

```
trait AAA {
  type Mytype;
}

T: AAA

T::Mytype
```

Scope机制

不引入对应的trait,你就得不到相应的能力。所以Rust 的trait可以看作是能力配置型机制。

Trait Object

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=43bc69d95b51d9b68829a18dd0edc3e2

对比

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=1bd651de6c322e092e086b764528a120

&dyn and Box<dyn>

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=b9fa7c6fd45c53cc953974e1e5ee8493

trait Object 安全

traint的方法中不能使用带所有权的self. 只能用引用模式，不然会报错。因为定义的时候不知道Self(具体被实现的那个类型)的尺寸。

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=7056abd97771d574f3358d9fe89111ab

trait object作为返回值类型

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=6d01425445cade5a66734b1eadb3a861

Vec中放置 trait object

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=ecd34c1144d8d5f553951f8168368706

TinTin

HashMap中可以吗？

当然可以，可以自己去试验。一般放在value的位置。

与使用Enum承载不同类型的区别

它们都是消除泛型参数的方法。(泛型参数具有传染性)。

Vec中

函数参数中

函数返回值中

enum是已知类型范围，有明确封闭边界 closed set

trait object相当于定义了协议，是开放的 open set

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=324919d6d4c89c76e49787a1f4b5642b

https://users.rust-lang.org/t/performance-implications-of-box-trait-vs-enum-delegation/11957

https://www.mattkennedy.io/blog/rust_polymorphism/

https://www.possiblerust.com/guide/enum-or-trait-object

Q&A

TinTin