4trait细节

2020年3月9日 11:19

1.trait作为函数的参数:

```
fu nihao(x: impl Debug, y: impl Debug) -> impl Debug {

}
```

x和y可以是相同的类型,也可以是不同的类型,但是返回值必须是单一的类型.
就是说在函数内部不能返回不同的类型.

如果我们想要强制x和y必须是相同的类型的话,这样使用:

```
fn nihao<T: Debug>(x: T, y: T) -> T {}

fu nihao(x: impl Debug + Display) -> impl Debug {}

impl Debug + Display  = impl (Debug + Display)
```

2.关于trait的引入

(1)、trait定义了一个抽象化的接口，如果我们想使用这些接口的方法的话，这个trait必须在作用域中。
(2)、我们不可能凭空就知道应该use哪个trait以及该从crate中调用哪个方法，crate的使用使用位于其文档中。
(3)、实现trait的孤儿规则：只有当trait或者实现trait的类型或者实现trait的类型的泛型的类型位于crate的本地作用域时，
才能为该类型实现trait。
(4)、cargo update 更新依赖最0.x.*到最新，不会改变x的版本，如果要改变x的版本，需要修改Cargo.toml.

3.trait的定义中可以使用泛型:

trait中可以有关联类型，可以使用泛型。

(1):

```
trait Nihao<T> {} // 泛型的trait
```

这样的trait可以有多个实现

```
impl Nihao<String> for Women{}
impl Nihao<i32> for Women {}
```

... 可以多次为同一个类型多次实现Nihao,因为同一个类型实现了多次这个trait,
那么使用的时候就必须加上类型注解,表示使用的是哪种实现了.

关联类型 `trait Nihao { type Item; }`
只能有一个实现.

```
impl Nihao for Women { type Item=i32, ... }
```

(2):默认的泛型参数类型
在泛型的定义中,可以通过在<>中使用T=i32,提供默认的泛型实现.

```
fn nihao<T=String>(x: T){ };
struct Nihao<T=String> {};
enum Nihao<T=String>{};
trait Nihao<T=String> {};
```

以上都是可以的,只要是泛型定义的地方都可以使用泛型默认参数类型.

```
trait Add<RHS=Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
```

```
}

impl Add<Nihao> for i32 {

}
```
这里的i32是Self, add这个方法调用的时候,rhs参数必须是Nihao类型.

注意:trait的泛型参数类型,或者默认的泛型参数类型一般是为了方法的参数类型,
关联类型是为了方法的返回类型.

```
type Kilo = i32;
```
类型别名的定义,kilo是i32是同义词,是一样的,只是一个别名,不是新的类型.

从来返回的类型:!
```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```
允许 match 的分支以 continue 结束是因为 continue 并不真正返回一个值；相反它把控制权交回上层循环，所以在 Err 的情况，事实上并未对 guess 赋值。用在循环中.
panic!()也是!类型
loop{} 也是!类型.

动态大小类型: Sized
?Sized是编译时未知大小比如: str、 [T] 等.

break一个值只能用在loop中.
```
Loop {
    Break 10;
}
```

4.trait对象

trait对象使用方法

```
(1):
trait MyTrait {
    fn hello(&self);
}

struct Hades(i32);

impl MyTrait for Hades {
    fn hello(&self) {
        println!("hello: {}", self.0);
    }
}
let h = Hades(100);
let h_obj: &dyn MyTrait = &h as &dyn MyTrait; //这里是定义trait object
h_obj.hello();
MyTrait::hello(h_obj); // 和上面的一样

(2): 如果trait里面的某些方法不像放入到trait object中,
```
```
trait MyTrait {
    fn hello(&self) where Self: Sized;
    fn nihao(&self);
}
```

```
    struct Hades(i32);

    impl MyTrait for Hades {
        fn hello(&self) {
            println!("hello: {}", self.0);
        }

        fn nihao(&self) {}
    }
    let h = Hades(100);
    let h_obj: &dyn MyTrait = &h as &dyn MyTrait; //这里是定义trait object
    h_obj.hello(); //error, hello方法没有放入到trait object中,
    h_obj.nihao(); //right,nihao方法在trait object中.
    MyTrait::hello(h_obj); // error
    MyTrait::nihao(h_obj); // right
```

    (3): 如果整个trait都不想被trait object化的话,
```
    trait MyTrait where Self:Sized {
        fn hello(&self);
        fn nihao(&self);
    }

    struct Hades(i32);

    impl MyTrait for Hades {
        fn hello(&self) {
            println!("hello: {}", self.0);
        }

        fn nihao(&self) {}
    }
    let h = Hades(100);
    let h_obj: &dyn MyTrait = &h as &dyn MyTrait; //这里就会直接报错了,
```

    (4):trait 对象化的控制和方法是否放到trait对象中,控制语句是写在trait定义中的,和实现没有关系的.
```
    trait MyTrait where Self::Sized { // Self::Sized写在这里才有效果
        fn hello(&self);// fn hello(&self) where Self:Sized,在这里控制
        fn nihao(&self);
    }

    struct Hades(i32);

    impl MyTrait for Hades {
        fn hello(&self) { //fn hello(&self) where Self: Sized { 在impl中写没有任何效果的.
                                // 因为impl实现不能更改trait定义中的控制语句.
            println!("hello: {}", self.0);
        }

        fn nihao(&self) {}
    }
```

    (5): trait对象和&impl MyTrait的区别

&impl MyTrait:意思是一个实现了MyTrait的类型引用.这个引用不受Where: Sized的限制.
trait object:是一个抽象的类型,意思是说编译时不知道大小的类型.这里表示的是不知道大小,
是真的不知道大小,这里trait对象不能理解成指针,或者引用,我们知道指针或者引用是有固定大小的.这里仅仅是表示一个编译时不知道类型大小的表示方法.&dyn MyTrait,
Box<&dyn MyTrait>,都是表示不知道大小的一种表示.Box是表示这个类型是堆上的,
但是不知道大小.一看到Box就认为是指针是错误的.这里的Box仅仅是表示分配在堆上的
不知道大小的trait对象,一种表示方法而已.????

```
trait MyTrait where Self:Sized {
    fn hello(&self);
    fn nihao(&self);
}

struct Hades(i32);

impl MyTrait for Hades {
    fn hello(&self) {
        println!("hello: {}", self.0);
    }

    fn nihao(&self) {}
}
let h = Hades(100);
let h_obj: &dyn MyTrait = &h as &dyn MyTrait; //这里就会直接报错了,

fn my_print(x: &impl MyTrait) {
    x.hello(); // 这里完全不受限制, 因为&impl MyTrait不是trait对象类型
    x.nihao();
}

// 更一般的表示方法是用泛型的trait bound.
fn my_print<T: MyTrait>(x: &T) {
    x.hello();
    x.nihao();
}
```

(6):

```
fn main() {

      let n = Nihao { name: "hades".to_string() };
      let b = &n;
    dis(b); //  错误b是一个引用,精确查找impl实现的类型
    dis2(b);//  错误b是一个引用,精确查找bound的impl实现的类型
      dis3(b);//  正确,参数是一个trait 对象,查找的trait中的函数.自动匹配.
}

struct Nihao {
      name: String,
}

trait Women {
      fn hello(&self);
}

impl Women for Nihao {
```

```
        fn hello(&self) {
            println!("hello: {}", self.name);
        }
    }

    fn dis(n: impl Women) {
        n.hello();
    }

    fn dis2<T: Women>(n: T) {
        n.hello();
    }

    fn dis3(n: &dyn Women) {
        n.hello();
    }
```

注意:生命周期仅仅是针对引用可言的,如果我们传递一个所有权的话,生命周期的bound默认就不起作用了.

5.泛型trait bound的定义位置

泛型trait bound的定义位置,决定了代码的初始化.
满足trait bound的泛型,意思就是说可以代码实现.

比如:

```
trait MyTrait {
    fn hello(&self);
}

struct Nihao<T: MyTrait> { // 这里的意思是说要用这个结构体创建变量T必须实现Mytrait这个trait,不然报错
    name: T,
}

let a = Nihao { name: 10 }; // Error, 因为i32类型没有impl MyTrait这个trait

impl<T> MyTrait for Nihao<T> { //  如果在struct中定义了T: MyTrait,这里必须添加,不然报错

}
```

如果:

```
struct Nihao<T> {
    name: T,
}

impl<T: Trait> MyTrait for Nihao<T> {
    fn hello(&self) {...}
}

let a = Nihao { name: 10 };  //  正确,可以初始化一个实例,不会报错的.
a.hello(); // 报错,因为i32没有实现MyTrait,所以没有hello这个方法,hello这个方法只会给满足T: MyTrait的类型才会实现hello()这个方法的.
```

6.类型强转trait

类型T强转trait P的用法:

适用情况:
1、类型T实现了两个具有相同方法的trait，要具体的指明要调用哪一个。

```
trait P1 {
    fn hello(&self);
}

trait P2 {
    fn hello(&self);
}

impl P1 for T {..}
impl P2 for T {..}

let t = T;
t.hello(); // 此时不知道是哪个trait的hello方法。
T::hello(&t);//这样也是不知道的。
<T as P1>::hello(&t); // 通用的调用方法
P1::hello(&t); // 这样也是可以的。
```

2、类型T实现了trait P，想要适用P的关联类型。

```
trait P {
    type Err;
}

struct T;

impl P for T { type Err = std::err::Err;}
```

我们想要利用T类型获取trait P的关联类型Err。

```
<T as P>::Err; // 这样的调用就获取了具体的Err类型是多少了。
```

注意上述的两种情况：
都是通过类型去调用trait的函数或者获取trait的关联方法，而不是直接用trait表示的。
通过trait P调用方法也是可以的，P::hello(&t);但是关联方法不能获取。

获取关联类型好处是：

```
let r: <T as P1>::Err = String::from("hades");
```

如果直接使用 let r: P1::Err;这样是不知道具体的类型是什么的。