



TinTin

Office Hour 4

老师: Mike Tang

助教: zhouzhou

Generics(泛型) + Trait(泛型)

泛型

- 类型是一种约束（类型安全）
- 泛型（参数化类型、类型参数）
 - (1) 函数参数中的泛型
 - (2) 结构体中的泛型
- PhantomData幽灵数据类型（了解）
- 枚举中的泛型（Option + Result）
- 方法中的泛型参数与结构体中的可以不同

Trait

- Trait实际是对类型的约束（Trait Bounds）
- 类型的约束依赖
- Where语句
- 默认实现（Default Implementation）
- 全局实现（Blanket Implementation）
- 关联类型
- Trait Object（dyn Trait）

一、Generics(泛型)

1、类型是一种约束（类型安全）

- 类型化的好处（5大好处）

正确性、不可变性、封装行、组合性、可读性

- Rust的类型体系-强类型

1、结构体

2、枚举

3、泛型-洋葱结构 $A < B < C < D < E < T > > > >$

4、type 关键字


类型别名（type alias），使用 type 关键字来给予现有类型另一个名字，如：

```
type Kilometers = i32;      //创建 i32 的别名 Kilometers
let x: i32 = 5;
let y: Kilometers = 5;
println!("x + y = {}", x + y);
```

2、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。

- 函数参数中的泛型

```
1  
2  /* 获取Vec中最大的元素 */
3  pub fn largest<T: std::cmp::PartialOrd> (list: &Vec<T>) -> &T {
4      let mut largest: &T = &list[0];
5      for item: &T in list {
6          if item > largest {
7              largest = item;
8          }
9      }
10     &largest
11 }
12
13 #[cfg(test)] // 标注测试模块
14     ▶ Run Tests | Debug
15 mod tests {
16     use super::*;
17
18     #[test]
19     ▶ Run Test | Debug
20     fn test() {
21         //
22         let number_list: Vec<i32> = vec![34, 50, 25, 100, 65];
23         let result: &i32 = largest(&number_list);
24         println!("The largest number is {}", result);
25         //
26         let char_list: Vec<char> = vec!['y', 'm', 'a', 'q'];
27         let result: &char = largest(&char_list);
28         println!("The largest char is {}", result);
29     }
30 }
```

I 课程关键内容回顾

2、泛型（参数化类型、类型参数）

- 结构体中的泛型。

(1) 结构体中字段是相同类型

```
#[derive(Debug)]
1 Implementation
pub struct Point<T> {
    pub x:T,
    pub y:T
}

#[cfg(test)] // 标注测试模块
▶ Run Tests | Debug
mod tests {
    use super::*;

    #[test]
    ▶ Run Test | Debug
    fn test() {
        let integer: Point<i32> = Point {x:5, y:10};
        let float: Point<f64> = Point {x:1.0, y:4.0};

        println!("integer => {:#?}", integer);
        println!("float => {:#?}", float);

        println!("integer.x => {}", integer.x);
        println!("float.y => {}", float.y);
    }
}
```

(2) 结构体中字段是不同类型

```
#[derive(Debug)]
1 Implementation
pub struct Point<T, U> {
    pub x:T,
    pub y:U,
}

#[cfg(test)] // 标注测试模块
▶ Run Tests | Debug
mod tests {
    use super::*;

    #[test]
    ▶ Run Test | Debug
    fn test() {
        let a: Point<i32, f64> = Point {x:5, y:1.0};
        let b: Point<f64, char> = Point {x:1.0, y: 'A'};

        println!("a => {:#?}", a);
        println!("{}", b);
    }
}
```

I 课程关键内容回顾

3、PhantomData (幽灵数据)

PhantomData是一个marker，当结构体中没有字段定义成T类型，但是结构体方法上用到了T类型，那么需要PhantomData引入T。

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

```
use std::marker::PhantomData;  
  
// 定义一个包含泛型参数 T 的结构体  
4 usages new *  
struct Repeater<T> {  
    phantom: PhantomData<T>,  
}  
  
new *  
impl<T> Repeater<T> {  
    new *  
    fn new() -> Self {  
        Repeater {  
            phantom: PhantomData,  
        }  
    }  
}  
  
// 复读，传入什么就传出什么  
new *  
fn repeater_value(&self, s: T) -> T {  
    s  
}  
  
new *  
#[test]  
fn test() {  
    let r1 : Repeater<String> = Repeater::<String>::new();  
    let b1 : String = r1.repeater_value(s: "hello rust.".to_string());  
    println!("{}", b1);  
  
    let r2 : Repeater<u8> = Repeater::<u8>::new();  
    let b2 : u8 = r2.repeater_value(s: 128);  
    println!("{}", b2);  
}
```

```
✓ Tests passed: 1 of 1 test - 0ms  
/Users/zhoushou/.cargo/bin/cargo test --color=always --package course04 --test test2 test  
Testing started at 16:46 ...  
warning: course04 v0.1.0 (/Users/zhoushou/rust_workspace/linInLand-Rust-Course/4_泛型与T)  
Finished test [unoptimized + debuginfo] target(s) in 9.02s  
Running tests/test2.rs (target/debug/deps/test2-67dc847871cfe428)  
  
hello rust.  
128  
  
Process finished with exit code 0
```

4、枚举中的泛型 (Option + Result)

枚举是各种带类型参数的变体和类型，每个类型参数位置当然可以出现泛型

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


I 课程关键内容回顾



5、方法中的泛型参数与结构体中的可以不同

(1) 相同的情况

```
#[derive(Debug)]
2 Implementations
struct Point<T> {
    x:T,
    y:T
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

#[test]
▶ Run Test | Debug
fn test() {
    let p: Point<i32> = Point {x:10, y:15};
    println!("p.x() => {}", p.x());
    println!("p.y => {}", p.y);
}
```

(2) 不同的情况

```
#[derive(Debug)]
2 Implementations
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1,Y1> {
    fn mixup<X2,Y2>(&self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y
        }
    }
}

#[test]
▶ Run Test | Debug
fn test6() {
    let p: Point<i32, f64> = Point{x: 5, y: 20.5};
    let o: Point<char, &str> = Point{x: 'A', y:"good"};
    let mix: Point<i32, &str> = p.mixup(o);
    println!("mix => {:?}, mix);
}
```

(3) 单态化方法

```
#[derive(Debug)]
2 Implementations
struct Point<T> {
    x:T,
    y:T
}

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

#[test]
▶ Run Test | Debug
fn test() {
    let p: Point<f32> = Point {x:10.3, y:15.1};
    println!("{}", p.distance_from_origin());
}
```

二、Trait（特征）

1、Trait实际是对类型的约束 (Trait Bounds)

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。
如：

T: Animal => T类型必须是实现了Animal trait

T: VegeAnimal => T类型必须是实现了VegeAnimal trait

I 课程关键内容回顾

2、类型的约束依赖

如: `trait VegeAnimal: Animal`

(1) 有一点点像OOP中的继承, 但并不能叫trait继承, 所有trait都是平等的, trait之间没有约束关系。

(coffee老师: 老外称之为 trait inheritance)

(2) `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范围的子集, 即

`VegeAnimal`约束的范围是`Animal`约束范围的子集。

```
1 Implementation
trait Animal {
    fn func_breathe(&self);
}

1 Implementation
trait VegeAnimal: Animal {
    fn func_eat_grass(&self);
}

2 Implementations
struct Cow {}

impl VegeAnimal for Cow {
    fn func_eat_grass(&self) {
        println!("cow eat grass.");
    }
}

impl Animal for Cow {
    fn func_breathe(&self) {
        println!("cow breathe.");
    }
}

#[test]
▶ Run Test | Debug
fn test() {
    let c: Cow = Cow{};
    c.func_eat_grass();
    c.func_breathe();
}
```

类型和trait表现方式

T: Animal + Tame

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。

trait VegeAnimal: Animal + Tame

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。

T: VegeAnimal

表示 T被VegeAnimal约束。

3、Where语句

Where关键字可以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多的时候。

```
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,
    }
    {}
```

4、默认实现 (Default Implementation)

(1) trait 可以Self 这个符号

如:

```
trait Animal {  
    fn myself (self) -> Self;  
    fn eat(&self);  
    fn eat_mut(&mut self);  
}
```

```
1 Implementation  
trait Atr {  
    |   fn foo(&self);  
}  
  
1 Implementation  
struct Foo;  
  
impl Atr for Foo {  
    |   fn foo(&self) {  
    |       |   println!("hi trait.");  
    |       }  
    }  
  
#[test]  
▶ Run Test | Debug  
fn test() {  
    |   let foo: Foo = Foo;  
    |   foo.foo();  
    }  
}
```

(2) trait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。

如：

```
trait Default {  
    // function  
    fn default() -> Self;  
}
```

调用方式：

```
Default::default();    //类似于java, c++ 类的静态方法。 :: 叫路径符
```


I 课程关键内容回顾



右侧代码中:

new方法对泛型T没有约束

cmp_display方法对泛型T约束 Display + PartialOrd

问: 什么要对不同方法做不同的约束呢?

是否我们可以做一些权限防护的能力呢? 比如在合约中只能某些角色的账户才能调用此方法查看合约中敏感数据。

```
2 Implementations
struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x:T, y:T) -> Self {
        Self { x, y }
    }
}

impl<T> Pair<T>
where
    T: std::fmt::Display + std::cmp::PartialOrd
{
    fn cmp_display(&self) {
        if self.x > self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}

#[test]
▶ Run Test | Debug
fn test() {
    let p: Pair<i32> = Pair::new(x: 55, y: 10);
    p.cmp_display();
}
```

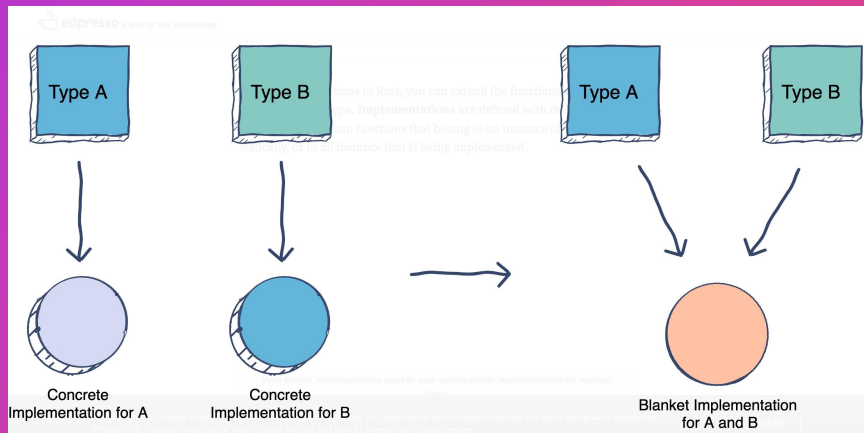
5、全局实现 (Blanket Implementation)

官方定义:

We can also conditionally implement a trait for any type that implements another trait. Implementations of a trait on any type that satisfies the trait bounds are called `_blanket implementations_` and are extensively used in the Rust standard library. For example, the standard library implements the `ToString` trait on any type that implements the `Display` trait.

我们可以有条件地为任何一个实现了另一个Trait的类型实现一个Trait。为任何一个满足 Trait bound的类型实现一个Trait，称为通用实现(`_blanket implementations_`)。且被广泛地使用于Rust标准库。举个例子，标准库为任何一个实现了`Display` Trait的类型实现了 `ToString` Trait。

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



I 课程关键内容回顾

```
1 2 usages 1 implementation new *
2 trait MyTrait {
3     1 implementation new *
4     fn my_method(&self) -> i32;
5 }
6
7 2 usages new *
8 struct MyStruct(i32);
9 new *
10 impl MyTrait for MyStruct {
11     new *
12     fn my_method(&self) -> i32 {
13         self.0
14     }
15 }
16
17 1 usage 1 implementation new *
18 trait MyOtherTrait {
19     1 implementation new *
20     fn my_other_method(&self) -> i32;
21 }
22
23 // Implement MyOtherTrait for each type that implements MyTrait
24 new *
25 impl<T: MyTrait> MyOtherTrait for T {
26     new *
27     fn my_other_method(&self) -> i32 {
28         println!("my_struct -> myTrait, MyOtherTrait");
29         self.my_method() * 2
30     }
31 }
32
33 new *
34 #[test]
35 fn test() {
36     let my_struct = MyStruct(10);
37     println!("{}", my_struct.my_other_method());
38 }
39 }
```

MyTrait

MyOtherTrait

```
1 3 usages 2 implementations new *
2 trait MyTrait {
3     2 implementations new *
4     fn my_method(&self) -> i32;
5 }
6
7 2 usages new *
8 struct MyStruct(i32);
9 new *
10 impl MyTrait for MyStruct {
11     new *
12     fn my_method(&self) -> i32 {
13         self.0
14     }
15 }
16
17 1 usage 1 implementation new *
18 trait MyOtherTrait {
19     1 implementation new *
20     fn my_other_method(&self) -> i32;
21 }
22
23 // Implement MyOtherTrait for each type that implements MyTrait
24 new *
25 impl<T: MyTrait> MyOtherTrait for T {
26     new *
27     fn my_other_method(&self) -> i32 {
28         println!("my_struct -> myTrait, MyOtherTrait");
29         self.my_method() * 2
30     }
31 }
32
33 2 usages new *
34 struct HeStruct;
35 new *
36 impl MyTrait for HeStruct {
37     new *
38     fn my_method(&self) -> i32 {
39         100
40     }
41 }
42
43 new *
44 #[test]
45 fn test() {
46     let my_struct = MyStruct(10);
47     println!("my: {}", my_struct.my_other_method());
48
49     let he_struct = HeStruct;
50     println!("he: {}", he_struct.my_other_method());
51 }
52 }
```

6、关联类型

```
pub trait Iterator {  
    type Item; //关联类型  
    fn next(&mut self) -> Option<Self::Item>;  
}
```

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

```
1 Implementation  
trait StreamingIterator {  
    type Item; //关联类型  
}  
  
#[derive(Debug)]  
2 Implementations  
struct A;  
  
impl StreamingIterator for A {  
    type Item = String;   
}  
  
#[derive(Debug)]  
1 Implementation  
struct Foo<T> where T: StreamingIterator<Item=String> {  
    x: T  
}  
  
#[test]  
▶ Run Test | Debug  
fn test() {  
    let a: Foo<A> = Foo::<A> {  
        x: A  
    };  
    println!("a => {:#?}", a);  
    println!("a.x => {:#?}", a.x);  
}
```

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  //
```

I 课程关键内容回顾

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。

通俗讲：如果你在一个struct中使用了一个trait，那么这个trait的方法和属性可能只在这个struct的作用域内可见，而不会泄露到其他struct中。这种作用范围的控制有助于确保代码的封装性和模块化。

```
6 // 定义一个简单的trait
7 2 implementations
8 trait Log {
9     fn log(&self);
10 }
11
12 // 在特定结构体的作用域中实现trait
13 1 implementation
14 struct MyStruct;
15 impl Log for MyStruct {
16     fn log(&self) {
17         println!("Logging from MyStruct");
18     }
19 }
20
21 // 在另一个结构体的作用域中实现同一个trait
22 1 implementation
23 struct AnotherStruct;
24 impl Log for AnotherStruct {
25     fn log(&self) {
26         println!("Logging from AnotherStruct");
27     }
28 }
29
30 // 一个函数，接受实现了Log trait的类型并调用log方法
31 fn log_something(loggable: &impl Log) {
32     loggable.log();
33 }
34
35 #[test]
36 ▶ Run Test | Debug
37 fn test() {
38     let my_instance: MyStruct = MyStruct;
39     let another_instance: AnotherStruct = AnotherStruct;
40
41     // 在不同的作用域中调用log_something函数
42     log_something(loggable: &my_instance);
43     log_something(loggable: &another_instance);
44 }
45
46 } mod tests
47
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

warning: unused import: `super::*`
→ src/lib.rs:4:9

```
4 use super::*;
```

= note: `[warn(unused_imports)]` on by default

warning: `course04` (lib test) generated 1 warning (run `cargo fix --lib -p course04 --tests` to apply 1 suggestion)
Finished test [unoptimized + debuginfo] target(s) in 0.02s
Running unittests src/lib.rs (target/debug/deps/course04-fb00150205675ad8)

```
running 1 test
Logging from MyStruct
Logging from AnotherStruct
test tests::test ... ok
```

I 课程关键内容回顾



7、Trait Object

泛型、&dyn、Box 之间区别：

(1) 泛型：静态分发(static dispatch)，编译后代码体较大，运行性能高。编译过程会做去泛型化处理，对Cat，Dog实现各自的talk方法。

(2) &dyn：动态分发(dynamic dispatch)，dyn Animal是trait object，编译后代码体较小，运行性能低，运行时动态调用实现Animal trait的Cat或Dog的talk方法（类似java中多态的动态绑定）。

(3) Box<dyn>：智能指针，它在堆上分配内存并允许所有权的转移。

&dyn是trait object的引用，没有所有权，有时候需要添加lifetime参数，使用起来不方便，Box<dyn>具有所有权，比较方便，所以用得更广泛。

```
2 implementations
trait Animal {
    fn talk(&self);
}

1 implementation
struct Cat {}

1 implementation
struct Dog {}

impl Animal for Cat {
    fn talk(&self) {
        println!("meow");
    }
}

impl Animal for Dog {
    fn talk(&self) {
        println!("bark");
    }
}

fn animal_talk_t<T: Animal>(a: &T) {
    a.talk();
}

fn animal_talk_dyn (a: &dyn Animal) {
    a.talk();
}

fn animal_talk_box (a: Box<dyn Animal>) {
    a.talk();
}

#[test]
▶ Run Test | Debug
fn test10() {
    let d: Dog = Dog{};
    let c: Cat = Cat{};
    animal_talk_dyn(&d);
    animal_talk_dyn(&c);

    animal_talk_t(&d);
    animal_talk_t(&c);

    animal_talk_box(Box::new(d));
    animal_talk_box(Box::new(c));
}
```


I 课程关键内容回顾

impl trait

可以用在两个地方：函数参数与返回值，静态分配，而且作为函数返回值时，数据类型只能有一种，这一点要尤为注意！

```
Run Tests | Debug | 2 Implementations
trait Greeting {
    fn greeting(&self) -> &str;
}

1 Implementation
struct Cat;
impl Greeting for Cat {
    fn greeting(&self) -> &str {
        "Meow!"
    }
}

1 Implementation
struct Dog;
impl Greeting for Dog {
    fn greeting(&self) -> &str {
        "Woof!"
    }
}

//impl 做函数参数
fn print_greeting_impl(g: impl Greeting) {
    println!("{}", g.greeting());
}

//impl 做函数返回值，下面代码会编译报错
fn return_greeting_impl(i: i32) -> impl Greeting {
    if i > 10 {
        return Cat;
    }
    Dog
}
mismatched types=expected `Cat`, found `Dog`

#[test]
Run Test | Debug
fn test() {
    let c: Cat = Cat{};
    let d: Dog = Dog{};

    print_greeting_impl(c);
    print_greeting_impl(d);
}
```



TinTin

I 课程关键内容回顾

Trait Object 安全

trait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。

```
2 implementations
trait Animal {
    fn talk(self);
    //fn eat(self) {} //T000: doesn't have a size known at compile-time
}

1 implementation
struct Cat {}

1 implementation
struct Dog {}

impl Animal for Cat {
    fn talk(self) {
        println!("meow");
    }
}

impl Animal for Dog {
    fn talk(self) {
        println!("bark");
    }
}

fn animal_talk_t<T: Animal>(a: T) {
    a.talk();
}

fn animal_talk_impl (a: impl Animal) {
    a.talk();
}

// fn animal_talk_dyn (a: dyn Animal) { //T000: doesn't have a size known at compile-time
//     a.talk();
// }

// fn animal_talk_box (a: Box<dyn Animal>) { //T000: the size of 'dyn Animal' cannot be statically determined
//     a.talk();
// }

#[test]
fn test10() {
    let d: Dog = Dog{};
    let c: Cat = Cat{};

    animal_talk_t(d);
    //animal_talk_t(c);

    //animal_talk_impl(d);
    animal_talk_impl(c);

    // animal_talk_dyn(d);
    // animal_talk_dyn(c);

    // animal_talk_box(Box::new(d));
    // animal_talk_box(Box::new(c));
}
```

I 课程关键内容回顾

Trait Object 作为返回值类型

```
1 Implementation
struct Sheep {}
1 Implementation
struct Cow {}

2 Implementations
trait Animal {
    fn noise(&self) -> &'static str;
}

impl Animal for Sheep {
    fn noise(&self) -> &'static str {
        "baaaaah!"
    }
}

impl Animal for Cow {
    fn noise(&self) -> &'static str {
        "mooooooo!"
    }
}

fn random_animal(random_number: f64) -> Box<dyn Animal> {
    if random_number < 0.5 {
        Box::new(Sheep{})
    } else {
        Box::new(Cow{})
    }
}

#[test]
▶ Run Test | Debug
fn test12() {
    let random_number: f64 = 0.234;
    let animal: Box<dyn Animal> = random_animal(random_number);
    println!("You have randomly chosen an animal, and it says {}", animal.noise());

    //Box<dyn Animal> 有所有权
    let mut array: Vec<Box<dyn Animal>> = Vec::new();
    array.push(animal); //move
    //println!("{:?}", animal); //100% animal 已经move到数组中
    let a: &Option<&Box<dyn Animal>> = &array.get(index: 0);
    println!("{}", a.unwrap().noise());
}
```

I 课程关键内容回顾



Vec中只能存储相同类型，当然包括trait object

```
2 Implementations
trait Animal {
  fn talk(&self);
}

1 Implementation
struct Cat {}
1 Implementation
struct Dog {}

impl Animal for Cat {
  fn talk(&self) {
    println!("meow");
  }
}

impl Animal for Dog {
  fn talk(&self) {
    println!("bark");
  }
}

#[test]
▶ Run Test | Debug
fn test() {
  let d: Dog = Dog{};
  let c: Cat = Cat{};
  //TODO Vec中存储trait object类型
  let ans: Vec<&dyn Animal> = vec! [&d, &c];
  print!("dog talk => ");
  ans.get(index: 0).unwrap().talk();

  print!("cat talk => ");
  ans.get(index: 1).unwrap().talk();
}
```

HashMap 中 value存储trait object

```
use std::collections::HashMap;

2 Implementations
trait Shape {
  fn area(&self) -> f32;
}

1 Implementation
struct Circle {
  radius: f32,
}

impl Shape for Circle {
  fn area(&self) -> f32 {
    3.14 * self.radius * self.radius
  }
}

1 Implementation
struct Rectangle {
  x: f32,
  y: f32,
}

impl Shape for Rectangle {
  fn area(&self) -> f32 {
    0.5 * (self.x + self.y)
  }
}

#[test]
▶ Run Test | Debug
fn test() {
  let mut shape_map: HashMap<&str, Box<dyn Shape>> = HashMap::new();

  shape_map.insert(k: "circle", v: Box::new(Circle { radius: 2.0 }));
  shape_map.insert(k: "rectangle", v: Box::new(Rectangle {x: 3.0, y: 4.0 }));

  let circle: &Box<dyn Shape> = shape_map.get("circle").unwrap();
  println!("{}", circle.area());

  let rectangle: &Box<dyn Shape> = shape_map.get("rectangle").unwrap();
  println!("{}", rectangle.area());
}

mod tests
```

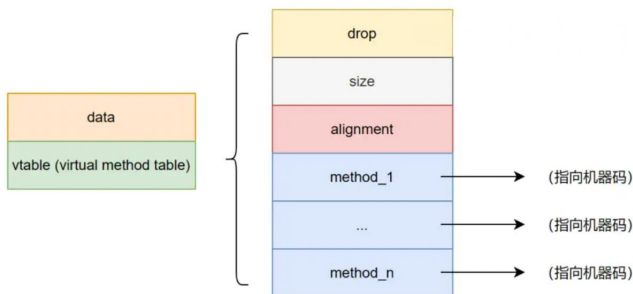
I 课程关键内容回顾

Trait Object 知识补充 – 内存布局

trait object 是一个胖指针，由两部分组成：

(1) 一个指针 ptr 指向实现了Trait的具体类型的实例（右图中实现Shape trait的Circle实例地址）

(2) 另一个指针vptr指向一个vtable（virtual method table），指向实现了Trait具体类型的实例中实现的Trait的方法（右图中实现Shape trait的Circle实例中实现了Shape trait的方法：area()、perimeter()）。



trait object的组成

```
1 trait Shape {
  1 implementation & Devin
  2 fn area(&self) -> f64;
  1 implementation & Devin
  3 fn perimeter(&self) -> f64;
  4 }
  5 usages & Devin
  6 struct Circle {
  7   radius: f64,
  8 }
  9 & Devin *
  10 impl Circle {
  11   & Devin *
  12   fn new(radius: f64) -> Self {
  13     Circle { radius }
  14   }
  15 }
  16 & Devin *
  17 impl Shape for Circle {
  18   & Devin *
  19   fn area(&self) -> f64 {
  20     3.14 * self.radius * self.radius
  21   }
  22   & Devin *
  23   fn perimeter(&self) -> f64 {
  24     2.0 * 3.14 * self.radius
  25   }
  26 }
  27 //TODO: Circle 自有方法
  28 new *
  29 impl Circle {
  30   new *
  31   fn owner_func(&self) -> &'static str {
  32     "this owner_func"
  33   }
  34 }
  35 & Devin *
  36 #[test]
  37 fn test() {
  38   let circle: Circle = Circle::new(radius: 5.0); circle: test::Circle
  39   let shape: &dyn Shape = &circle; shape: &dyn test::Shape
  40   println!("shape => {}", shape.area()); shape: &dyn test::Shape
  41   //println!("shape.owner_func() => {}", shape.owner_func()); //TODO: error!!! shape.owner_func() method not found in '&dyn Shape'
  42   println!("circle.owner_func() => {}", circle.owner_func()); circle: test::Circle
  43 }
  44 Shape for Circle : perimeter()
```

Memory View

Evaluate expression (⇧) or add a watch (⌘W)

```
> circle = (test::Circle)
> shape = (&dyn test::Shape)
> pointer = () 0x16d736640
> vtable = (&[usize; 3]) 0x10296c3d8
```

I 课程关键内容回顾



课程中需要自己补充学习的内容: Rust中的多态性: Enums vs Traits

Trait object

```
trait Shape {  
    fn area(&self) -> f64;  
}  
  
struct Rectangle {  
    height: f64,  
    width: f64,  
}  
  
impl Shape for Rectangle {  
    fn area(&self) -> f64 {  
        self.width * self.height  
    }  
}  
  
struct Circle {  
    radius: f64,  
}  
  
impl Shape for Circle {  
    fn area(&self) -> f64 {  
        std::f64::consts::PI * self.radius.powi(2)  
    }  
}  
  
struct RightAngleTriangle {  
    base: f64,  
    height: f64,  
}  
  
impl Shape for RightAngleTriangle {  
    fn area(&self) -> f64 {  
        0.5 * self.base * self.height  
    }  
}
```

Enum

```
enum Shape {  
    Rectangle { width: f64, height: f64 },  
    Circle { radius: f64 },  
    RightAngleTriangle { base: f64, height: f64 },  
}  
  
impl Shape {  
    fn area(&self) -> f64 {  
        match self {  
            Shape::Rectangle { width, height } =>  
                width * height,  
            Shape::Circle { radius } =>  
                std::f64::consts::PI * radius.powi(2),  
            Shape::RightAngleTriangle { base, height } =>  
                0.5 * base * height,  
        }  
    }  
}
```

I 作业



07题:

使用枚举包裹三个不同的类型，并放入一个Vec中，对Vec进行遍历，调用三种不同类型的各自的方法。

定义三个不同的类型，使用 `Trait Object`，将其放入一个Vec中，对Vec进行遍历，调用三种不同类型的各自的方法。同时，说明其上两种不同实现方法的区别。

```
1  #[derive(Debug)]
2  enum Wrapper {
3      Type1(i32),
4      Type2(String),
5      Type3(f64),
6  }
7
8  fn call_methods(wrapper: &Wrapper) {
9      match wrapper {
10         Wrapper::Type1(x) => println!("调用 Type1 的方法, x = {}", x),
11         Wrapper::Type2(x) => println!("调用 Type2 的方法, x = {}", x),
12         Wrapper::Type3(x) => println!("调用 Type3 的方法, x = {}", x),
13     }
14 }
15
16 pub fn homework41() {
17     let vec = vec![
18         Wrapper::Type1(10),
19         Wrapper::Type2("hello".to_string()),
20         Wrapper::Type3(3.1415926),
21     ];
22
23     for wrapper in vec {
24         call_methods(&wrapper);
25     }
26 }
```

I 作业

08题:

搜索相关文档，为你自己定义的一个类型或多个类型实现加法运算（用符号 +），并构思使用Trait Object实现类型方法的调用。



```
use std::ops::Add;

10 usages new *
#[derive(Debug, Copy, Clone, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

new *
impl Add for Point {
    new *
    type Output = Point;

    new *
    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

new *
#[test]
fn test() {
    let p :Point = Point { x: 1, y: 0 } + Point { x: 2, y: 3 };
    println!("p => {:?}", p);
    assert_eq!( Point { x: 1, y: 0 } + Point { x: 2, y: 3 }, Point { x: 3, y: 3 } );
}
```

```
✓ Tests passed: 1 of 1 test - 0ms

/Users/zhoushou/.cargo/bin/cargo test --color=always --package course04
Testing started at 17:28 ...
warning: course04 v0.1.0 (/Users/zhoushou/rust_workspace/TinTinLand-Rust)
Finished test [unoptimized + debuginfo] target(s) in 0.02s
Running tests/test3.rs (target/debug/deps/test3-66ec5d94a8d0bd3d)

p => Point {
  x: 3,
  y: 3,
}

Process finished with exit code 0
```


答疑讨论



TinTin

THANKS

–ZhouZhou

[Twitter](#)

[YouTube](#)

[Discord](#)