Rust 课程2.5

河南科技大学-徐堃元

2024/4/15

本节课程内容

- rustlings 环境配置再探与较为详细的演示
- 泛型扩展
- 生命周期扩展

《Rust 程序设计语言》：

https://kaisery.github.io/trpl-zh-cn/

《通过例子学 Rust》：

https://rustwiki.org/zh-CN/rust-by-example/

Rust 语言中文社区：

https://rustcc.cn/

泛型是一种多态


```
fn add<T>(a:T, b:T) -> T {
    a + b
}

fn main() {
    println!("add i8: {}", add(2i8, 3i8));
    println!("add i32: {}", add(20, 30));
    println!("add f64: {}", add(1.23, 1.23));
}
```

在方法中使用泛型


```rust
struct Point<T> {
    x: T,
    y: T,
}

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

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

在方法中定义额外的泛型参数


```rust
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c'};

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

为具体的泛型类型实现方法

对于 `Point<T>` 类型，你不仅能定义基于 `T` 的方法，还能针对特定的具体类型，进行方法定义：

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

这段代码意味着 `Point<f32>` 类型会有一个方法 `distance_from_origin`，而其他 `T` 不是 `f32` 类型的 `Point<T>` 实例则没有定义此方法。这个方法计算点实例与坐标 `(0.0, 0.0)` 之间的距离，并使用了只能用于浮点型的数学运算符。

这样我们就能针对特定的泛型类型实现某个特定的方法，对于其它泛型类型则没有定义该方法。

const 泛型

不同类型的数组

```
fn display_array(arr: [i32; 3]) {
    println!("{:?}", arr);
}
fn main() {
    let arr: [i32; 3] = [1, 2, 3];
    display_array(arr);

    let arr: [i32; 2] = [1, 2];
    display_array(arr);
}fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

针对值的泛型


```rust
fn display_array<T: std::fmt::Debug, const N: usize>(arr: [T; N]) {
    println!("{:?}", arr);
}
fn main() {
    let arr: [i32; 3] = [1, 2, 3];
    display_array(arr);

    let arr: [i32; 2] = [1, 2];
    display_array(arr);
}
```

生命周期

生命周期的本质是一个结构单元存活的周期

生命周期注解是一类泛型

- 一个存活的类型（变量），存活的周期是确定的。
- 生命周期注解 提供构造变量或者类型的模板参数，使得在不同代码上下文环境下，构造出的变量或者类型可以存活不同、 符合预期的范围。

无界生命周期

- 不安全代码(`unsafe`)经常会凭空产生引用或生命周期，这些生命周期被称为是 **无界(unbound)** 的。

  无界生命周期往往是在解引用一个裸指针(裸指针 raw pointer)时产生的，换句话说，它是凭空产生的，因为输入参数根本就没有这个生命周期：

```
fn f<'a, T>(x: *const T) -> &'a T {
    unsafe {
        &*x
    }
}
```

  上述代码中，参数 `x` 是一个裸指针，它并没有任何生命周期，然后通过 `unsafe` 操作后，它被进行了解引用，变成了一个 Rust 的标准引用类型，该类型必须要有生命周期，也就是 `'a`。

  可以看出 `'a` 是凭空产生的，因此它是无界生命周期。这种生命周期由于没有受到任何约束，因此它想要多大就多大，这实际上比 `'static` 要强大。

  若一个输出生命周期被消除了，那么必定因为有一个输入生命周期与之对应。

生命周期约束 HRTB

生命周期约束跟特征约束类似，都是通过形如 `` `'a: 'b` `` 的语法，来说明两个生命周期的长短关系。

'a: 'b

假设有两个引用 `` `&'a i32` `` 和 `` `&'b i32` ``，它们的生命周期分别是 `` `'a` `` 和 `` `'b` ``，若 `` `'a` `` >= `` `'b` ``，则可以定义 `` `'a:'b` ``，表示 `` `'a` `` 至少要活得跟 `` `'b` `` 一样久。

```
struct DoubleRef<'a,'b:'a, T> {
    r: &'a T,
    s: &'b T
}
```

生命周期约束 HRTB

T: 'a

表示类型 `T` 必须比 `'a` 活得要久：

```
struct Ref<'a, T: 'a> {
    r: &'a T
}
```

因为结构体字段 `r` 引用了 `T`，因此 `r` 的生命周期 `'a` 必须要比 `T` 的生命周期更短(被引用者的生命周期必须要比引用长)。

在 Rust 1.30 版本之前，该写法是必须的，但是从 1.31 版本开始，编译器可以自动推导 `T: 'a` 类型的约束，因此我们只需这样写即可：

```
struct Ref<'a, T> {
    r: &'a T
}
```

生命周期与子类型

子类型化

子类型化是指一种类型可以替代另一种类型的概念。

我们定义 `Sub` 是 `Super` 的子类型。

这表示生命周期 `Sub` 的范围要包含 `Super` 的范围，并且 `Sub` 的范围有可能更大。

生命周期与子类型

为了使生命周期子类型化，我们需要先定义一个生命周期：

> `'a` 定义了一段代码区域。

然后我们就可以定义它们之间的关系：

> 当且仅当 `'long` 是一个 **完全包含** `'short` 的代码区域时， `'long <: 'short` 。

`'long` 可能定义了一个比 `'short` 更大的区域，但这仍符合我们的定义。

**对于任意生命周期'a，有'static: 'a **

**有'a: 'b <=> 'a是'b的子类型 **

生命周期与子类型

将“ 'a 是 'b 的子类型”表示为偏序关系 <， 'a 与 'b 无关表示为 'a<>'b ， 则对于任意两个生命周期 'a 和 'b ， 仅存在3种不等关系： 'a<'b、 'a<>'b 和 'a>'b。将这3种关系统称为 R('a, 'b)

R 对于泛型的映射

- 对于单生命周期泛型 T，这个映射即定义为 T: R('a, 'b) --> R(T<'a>, T<'b>)
- 如果 'a 是 'b 的子类型，那么 T<'a> 是不是 T<'b> 的子类型呢？还是相反？还是无关？

生命周期与子类型

目前Rust生命周期的子类型关系对于泛型存在三种映射：

- **协变(covariant)** T: R(T<'a>, T<'b>) = R('a, 'b)。
- **逆变(contravariant)** T: R(T<'a>, T<'b>) = ~R('a, 'b)，也就是若 'a<'b，则 T<'a> > T<'b>。
- **不变(invariant)** T: R(T<'a>, T<'b>) = “<> 或 =”，也就是无法推导子类型关系。

变异性

变异性 是 Rust 引用通过它们的泛型参数，来定义引用之间的子类型关系。

在 Rust 中有三种变异性，假设 `Sub` 是 `Super` 的子类型：

- `F` 是 **协变的**，如果 `F<Sub>` 是 `F<Super>` 的子类型（子类型属性被传递）(译者注：这里被传递的意思是尖括号里面的子类型关系(`Sub <: Super`)被传递到尖括号外(`F<Sub> <: F<Super>`))
- `F` 是 **逆变的**，如果 `F<Super>` 是 `F<Sub>` 的子类型（子类型属性被 "反转"）(译者注：即尖括号里面的子类型关系(`Sub <: Super`)在尖括号外面被反转(`F<Super> <: F<Sub>`))
- 否则， `F` 是 **不变的** （不存在子类型关系）(译者注：即尖括号里面的子类型关系不会影响尖括号外面的子类型关系)

让我们回想上面的例子，如果 `'a` 是 `'b` 的子类型，我们可以将 `&'a T` 视作是 `&'b T` 的子类型，因而 `&'a T` 对于 `'a` 上是协变的。

此外，我们注意到不能将 `&mut &'a U` 视为 `&mut &'b U` 的子类型，因此我们可以说 `&mut T` 在 `T` 上是 *不变的*

NLL (Non-Lexical Lifetime)

引用的生命周期正常来说应该从借用开始一直持续到作用域结束 。×

引用的生命周期从借用处开始，一直持续到最后一次使用的地方。√

```
let mut u = 0i32;
let mut v = 1i32;
let mut w = 2i32;

// lifetime of `a` = α ∪ β ∪ γ
let mut a = &mut u;     // --+ α. lifetime of `&mut u`  --+ lexical "lifetime" of `&mut u`,`&mut u`, `&mut w` and `a`
use(a);                 //   |                            |
*a = 3; // <-----------------+                            |
...                     //                              |
a = &mut v;             // --+ β. lifetime of `&mut v`    |
use(a);                 //   |                            |
*a = 4; // <-----------------+                            |
...                     //                              |
a = &mut w;             // --+ γ. lifetime of `&mut w`    |
use(a);                 //   |                            |
*a = 5; // <-----------------+ <--------------------------+
```

Reborrow 再借用


```rust
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn move_to(&mut self, x: i32, y: i32) {
        self.x = x;
        self.y = y;
    }
}

fn main() {
    let mut p = Point { x: 0, y: 0 };
    let r = &mut p;
    let rr: &Point = &*r;

    println!("{:?}", rr);
    r.move_to(10, 10);
    println!("{:?}", r);
}
```

对应 rustlings 练习

无