从C++到Rust，references，dereference，.operator

原创 Ajonbin AJonbin的杂货铺 2024年03月23日 20:41 美国

上次讲了references，已经最近本的用法和一些限制。

今天继续讲讲怎么derefence一个reference。

在Rust中，显式的创建一个引用reference使用 **&** 操作符，显式的解引用deference是用 ***** 操作符。

这和c++不同，C++中创建引用和解引用是隐式的。&和*操作符是取地址和解指针的。

Rust中，refences的标准用法是这样的

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

fn main(){
    let mut p: Point = Point{x: 10, y: 20};
    let r_p: &mut Point = &mut p;
    (*r_p).x = 0;
    println!("{:?}", *r_p);
}
```


上面的3个红框演示了如何创建和解引用r_p。

等等，之前的例子里也有使用引用，似乎没有用*操作符来解引用。可以直接用

```
r_p.x = 0
println!("{:?}", r_p);
```

是的，没错，你说的对。

这两个行代码分别对应了两种自动解引用的情况。

先看第一种，. operator 点操作符。

由于 点操作符实在是使用的太频繁了，如果每次使用都要加上*操作符，那对写代码实在太不友好了， 估计没人受得了。

因此，在Rust中，在使用点操作符的时候，会自动解引用，auto-dereference。

这也就是说 r_p.x 和 (*r_p).x 是一样的。

另外，当使用点操作符调用对象函数的时候，点操作符会自动创建引用。

来看段代码

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

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

    fn get(&self) -> (i32, i32){
        (self.x, self.y)
    }
}
fn main(){
    let mut p: Point = Point{x: 10, y: 20};

    p.set(0,0);
    let (x, y) = p.get();
    println!("{}, {}", x, y);

    (&mut p).set(100,100);
    let (x1, y1) = (&p).get();
    println!("{}, {}", x1, y1);
}
```


首先，我们定义了个结构Point。

然后我们通过 impl Point{ } 来为Point 类型增加两个函数接口，set()和get()。

在标记1处，我们定义了set函数。注意第一个参数，&mut self。

在标记2处，我们定义了get函数。注意第一个参数，&self。

在C++中，类函数的会被隐式的加上一个参数，this指针，用来指向对象本身。

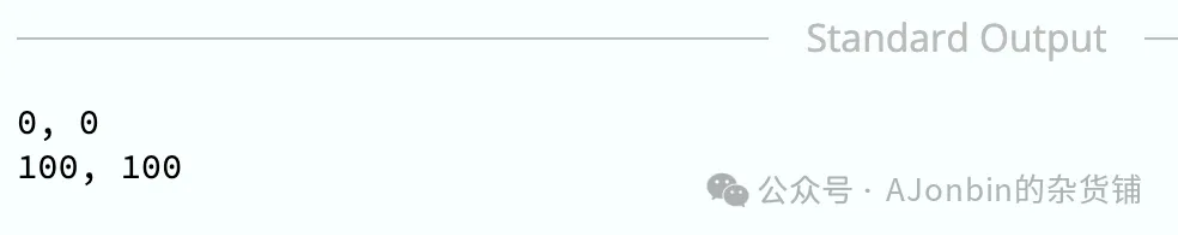
在Rust中，类型的接口中第一个参数是现式的&self，是对本身的一个引用，和Python类似。如果函数会改变本身的值，则self是可读写的，&mut self。

可见，类型的函数接口是通过self引用来调用的。

在标记3和4中，我们通过p直接调用了Point的函数，而p并不是引用reference。

这是因为Rust中，当通过变量调用其函数时，Rust会自动创建变量的引用。因为，我们可以直接用变量p来调用Point的函数。

标记5和6才是原本该有的样子。



今天讲了点操作符和reference的关系，下次讲讲 println!("{:?}", *r_p) 时reference都发生了什么。

上一篇：从C++到Rust，出借所有权，references