从C++到Rust，出借所有权，references

原创 Ajonbin AJonbin的杂货铺 2024年03月19日 22:43 美国

之前讲了所有权Ownership。当某个值的所有者Owner的生命周期结束时，这个值也将被释放。

我们也知道默认情况下，Rust的所有权是移动语义的。

实际上，有很多时候，我们并不希望所有权发生转移，特别是在调用函数的时候。

这个时候，我们就可以使用reference，引用。

Rust的引用和C++的引用很类似，但又有些很关键的区别。

引用reference从本质上讲就是对象的指针。这点Rust和C++一样。

但是使用起来，你会感觉Rust引用和C++很不一样，这也是我觉得对一个C++程序员来说最不适应的地方，会觉得用起来很不顺手，很别扭。

这主要是由Rust引用的两个特点引起的。

第一个是reference的使用限制。 简单来说，Rust为了保证数据的安全性，允许同时有多个只读引用，但只能有一个可读写的引用。

第二个是refernce的生命周期lifetime。为了确保引用是有效的，Rust引入了新的参数lifetime，用来在编译期保证引用不会指向一个已经被释放的对象。

引用reference在Rust中一种所有权的出借行为。reference并不拥有它所指向的值，只拥有值使用权。reference变量的被销毁后，它所指向的值并不会被销毁。

先来看看refence基本的用法。

```rust
fn main(){
    let v: Vec<i32> = vec![1,2,3,4];
    let r_v_1: &Vec<i32> = &v;
    let r_v_2: &Vec<i32> = &v;

    println!("v = {:?}", v);
    println!("r_v_1 = {:?}", r_v_1);
    println!("r_v_2 = {:?}", *r_v_2);
}
```


这是一个引用最基本的例子。r_v_1和r_v_2是两个指向同一个Vec<i32>的两个只读引用。

Standard Output

```
v = 2
r_v_1 = [1, 2, 3, 4]
r_v_2 = [1, 2, 3, 4]
```


可见，你可以同时拥有多个只读引用指向同一个值。

再来看看可读写引用。我们把上面的例子稍加改动。

```rust
fn main(){
    let mut v: Vec<i32> = vec![1,2,3,4];
    let r_v: &Vec<i32> = &v;
    let r_v_mut: &mut Vec<i32> = &mut v;

    println!("{:?}", r_v);
}
```


首先为了使用可读写引用，我们必须先将变量v申明成可读写mutable。

然后创建了一个只读引用变量r_v和一个可读写引用r_v_mut。

最后我们通过只读引用r_v来打印Vec的值。

来看看编译结果。出错了。

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:5:34
  |
4 |     let r_v: &Vec<i32> = &v;
  |                          -- immutable borrow occurs here
5 |     let r_v_mut: &mut Vec<i32> = &mut v;
  |                                  ^^^^^^ mutable borrow occurs here
6 |
7 |     println!("{:?}", r_v);
  |                      --- immutable borrow later used here
```


首先编译器告诉我们错误发生在第5行，由于v已经有了一个只读的引用，就不能再创建一个可读写的引用。之前我们说过引用是一种所有权出借的方式，也就是这里说的borrow。

标记1处，编译器告诉我们在创建r_v的时候，发生了一次只读的引用。

标记2处，编译器告诉我们错误在此处发生了。我们在创建r_v_mut这个可读写的引用的时候，由于已经有了只读引用r_v， 所以编译器不再允许创建一个可读写的引用。

这就是Rust中对引用使用的一个限制，你不能同时有一个可读写引用和只读引用指向同一个值。

这就是Rust中使用引用的一条限制，

注意下标记3处，编译器给了一个额外的信息，也就是当我们调用println!的时候，之前的只读引用在这里使用了。编译器认为这里也是和错误有关的。

那么如果我们不去使用这个只读引用r_v会怎么样呢？那我们再改下代码来试试。

我们只修改最后一行，println!()的时候不再打印r_v，而是打印r_v_mut。

```rust
fn main(){
    let mut v: Vec<i32> = vec![1,2,3,4];
    let r_v: &Vec<i32> = &v;
    let r_v_mut: &mut Vec<i32> = &mut v;

    println!("{:?}", r_v_mut);
}
```


这次并没有出错，而是顺利地打出了Vec的值。

```
Standard Output

[1, 2, 3, 4]
```


从代码看，我们同时有了只读引用r_v和可读写引用r_v_mut。这似乎违背了刚刚的结论。

但细想一下也不难明白，虽然我们创建了r_v，但是由于在后面的代码中我们并没有用到r_v，所以编译器认为在r_v_mut之前，r_v的生命就已经结束了。因此，实际上并没有发生只读引用和可读写引用共存的情况。这也就是之前的例子的标记3处，编译期要强调只读引用被使用了。

再看看同时两个可读写引用的情况。

```
fn main(){
    let mut v: Vec<i32> = vec![1,2,3,4];
    let r_v_mut: &mut Vec<i32> = &mut v;
    let r_v_mut_1: &mut Vec<i32> = &mut v;

    println!("{:?}", r_v_mut);
}
```


这个例子里，我们创建了两个可读写的引用，r_v_mut和r_v_mut_1。然后在打印r_v_mut。注意它们的创建顺序。

```
error[E0499]: cannot borrow `v` as mutable more than once at a time
 --> src/main.rs:5:36
  |
4 |     let r_v_mut: &mut Vec<i32> = &mut v;
  |                                   ------ first mutable borrow occurs here
5 |     let r_v_mut_1: &mut Vec<i32> = &mut v;
  |                                    ^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{:?}", r_v_mut);
  |                      ------- first borrow later used here
```


这次的错误不同，同一时间可读写的引用个数不能大于一个。

但本质和之前只读+可读写是一样的，都是为了保证数据的正确性。

同样，如果println!()的时候，我们打印r_v_mut_1的话，由于没有用到r_v_mut，它的生命周期在r_v_mut_1之前就结束了， 这样就可以通过编译了。

今天演示了使用引用最基本的情况--引用本地变量，之后再讨论其他的情况。

上一篇：从C++到Rust，不要Move，要自己的Clone