从C++到Rust，不要Move，要Copy和Clone

原创 Ajonbin AJonbin的杂货铺 2024年03月07日 15:24 美国

之前讲了在Rust中，Move是默认语义。赋值，传参和返回值默认都是Move，会交出所有权Ownership。

那么我们不希望Move改怎么办？

其实在之前的编译错误中，Rust已经告诉我们该怎么做了。

```
error[E0382]: borrow of moved value: `v1`
 --> tmp.rs:6:22
  |
2 |     let v1: Vec::<i32> = vec![1,2,3,4];
  |         -- move occurs because `v1` has type `Vec<i32>`, which does not implement the `Copy` trait
3 |
4 |     let _v2 = v1;
  |               -- value moved here
5 |
6 |     println!("{:?}", v1);
  |                      ^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of
the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
  |
4 |     let _v2 = v1.clone();
  |               ++++++++

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
```


来看看上一篇中的编译错误。

标记1：编译器告诉我们，之所以会发生Move，是因为v1的类型是Vec::<i32>，而Vec::<i32>这个类型没有实现"Copy trait"。

标记2：编译器给了我们一个提示，可是考虑使用克隆clone。

标记3：然后编译器直接给出了参考答案，let _v2 = v1.clone();

这里逻辑挺简单的，因为Vec::<i32>不能"Copy"，所以被"Move"，但是可以用"Clone"来解决。

也就是说"Copy"和"Clone"可以让我们打破"Move"的禁锢。

Copy和Clone是Rust标准库中定义的两个Trait。

Trait又是什么？就是接口定义。

> 在Rust中，Trait用来定义一种特定的行为，其他类型可以同时拥有和实现这种行为。
> 类似Java中的interface或是C++中的抽象类。

还是通过代码来讲解。

```rust
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p1 = Point{x:0, y:0};

    let _p2 = p1;

    println!("p=({},{})", p1.x, p1.y);
}
```



```
error[E0382]: borrow of moved value: `p1`
  --> tmp.rs:11:33
   |
7  |     let p1 = Point{x:0, y:0};
   |         -- move occurs because `p1` has type `Point`, which does not implement the `Copy` trait
8  |
9  |     let _p2 = p1;
   |               -- value moved here
10 |
11 |     println!("p=({},{})", p1.x, p1.y);
   |                            ^^^^ value borrowed here after move
   |
```


这还是一段有问题的代码。我们定义了一个结构Point。当所有权从p1转移到_p2后，p1就不能再使用了。

来看看Copy和Clone怎么帮我们解决Move的问题。

```
1 #[derive(Copy, Clone)]
2 struct Point {
3     x: i32,
4     y: i32,
5 }
6
7 fn main() {
8     let p1 = Point{x:0, y:0};
9
10    let _p2 = p1;
11
12    println!("p=({},{})", p1.x, p1.y);
13 }
```


我们在第一行，Point定义的前面加上

> #[derive(Copy, Clone)]

然后神奇的事发生了，编译通过，得到我们想要的输出。

Standard Output

```
p=(0,0)
```


来解释下。

首先，Copy和Clone都是Trait。Trait定义行为。那么显而易见Copy和Clone的行为都是用来拷贝值的，那么它们有什么区别呢。

Copy是继承于Clone的。

> pub trait Copy: Clone { }

Copy Trait

Copy的逻辑很简单，就是**按位拷贝内存**。它是用来告诉编译器复制这个类型是可以按比特位来拷贝（bit-wise）的，并由编译器产生按位拷贝内存的的代码。因此，在Rust中，你是不能自己实现Copy或是修改Copy的代码逻辑的。

如果你要为你的类型实现Copy Trait，只要在结构定义前加上#[derive(Copy)]就可以了。

#[derive]是Rust的一个属性，用来“获得”某个Trait的行为。

Copy Trait是隐式调用的。一旦获得了Copy行为，默认的Move语义就会被改变，当赋值或是传参时，Rust就会调用Copy来复制值，避免发生所有权转移，这样就产生了一个新的值，而原先的值可以继续使用。

但是，由于Copy Trait是由按位拷贝的，所以并不是所有的类型都可以实现Copy Trait。

如果一个类型的成员变量都是可以Copy的，那么这个类型就可以实现Copy；

如果一个类型的某个成员变量不能Copy，那么这个类型就不能实现Copy。

从C++的角度来看，Copy实现的是浅拷贝。如果类型需要深拷贝，那就不能用Copy。需要用Clone来代替。

比如我们之前定义的 Point{x:i32, y:i32}。两个成员变量x，y都是i32，都是简单类型，可以按位拷贝，那么Point就可以实现Copy Trait。

但如果Point里有一个变量z:Vec::<i32>，我们知道Vec::<i32>是包含一个实际数组的堆地址，那么Vec就不能通过比特位来拷贝，也就是做Vec不能实现Copy。由于Point包含了Vec，所以Point也是不可以Copy的。

```rust
#[derive(Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
    z: Vec::<i32>,
}
```



```
error[E0204]: the trait `Copy` cannot be implemented for this type
 --> tmp.rs:1:10
  |
1 | #[derive(Copy, Clone)]
  |          ^^^^
...
5 |     z: Vec::<i32>,
  |     ------------- this field does not implement `Copy`
  |
```


通过上面的代码和编译错误，就可以看出有了Vec，就不能Copy。

Clone Trait

和Copy不同，你可以自己实现Clone来进行复杂的操作，特别是当需要深拷贝时，就必须使用Clone了。

另外，Clone需要显式的调用，不像Copy会在赋值时被隐式调用。

```
fn main(){
    let v1 = vec![1,2,3,4];
    let v2 = v1.clone();
    println!("{:?}", v1);
}
```


上面的例子就是通过调用Vec::<i32>的clone()函数来新建一个变量v2，之后v1还是继续存在的，可以通过println!来打印它的值。

对于一些简单的类型，只需要通过#[derive(Copy, Clone)]，就可以获得Copy和Clone的默认实现，从而改变默认的移动语义，实现拷贝语义。

单对于一些复杂的类型，特别是包含指针的结构，就需要自己实现Clone来实现拷贝了。

这个下次聊。

上一篇：从C++到Rust，内存所有权管理Ownership，Move，转移所有权