3impl以及impl trait和自动解引用的关系

2020年3月9日 11:05

我们在给结构体或者枚举用impl实现的方法的时候，有一些重要的细节需要注意到：

(1) : impl类型
我们定义好一个结构体或者枚举的时候，想实现一些方法，例如：
```
struct Nihao(String);
impl Nihao {    // 这里必须是base,也就是说必须是原类型,如果使用&Nihao, &&Nihao等等,都是错误的
    ...
}
```

如果我们确实想使用&Nihao,可以用一个新的类型包装一下.

(2): impl一个trait的时候,就不同了,for 后面的类型就比较灵活了.
```
trait Women {
    fn hello(&self);
}

impl Women for Nihao {
    fn hello(&self){
        println!("{:p}", self);
    }
}
```
这里是为这个Nihao实现这个trait.
```
let n = Nihao(String::from("Hades"));
n.hello();  // 编译器通过添加一次&找到了匹配的hello的函数签名.编译器只能添加一次&,但是可以多次自动解引用,参照2rust自动解引用;

let c = &n;
c.hello(); //正确,打印的依然是n的地址,c就是&Nihao类型,直接找到了匹配的hello函数的签名.

let d = &c;
c.hello(); //正确,打印依然是n的地址,通过添加一次&不能,那么编译器采取自动解引用,使用了标准库中的通过实现&&Nihao => &Nihao => 找到了正确的函数签名.
```
一次类推,(&&&&&&&&&&n).print(); 打印的都是n的地址,编译器会多次解引用找到合适的函数签名. 参照2rust自动解引用.

// 通过看fn hell(&self)这个方法,self代表的就是类型Nihao这个变量n的地址,最终的多次解引用和一次的自动添加&, 都会找到这个变量n的地址.

接下来的会更加的有意思了:

```
1: impl Women for Nihao {
    fn print(&self) {
        println!("Nihao: print: {:p}", self);
    }
}

2: impl Women for &Nihao {
    fn print(&self) {
        println!("&Nihao: print: {:p}", self);
    }
}

3: impl Women for &&Nihao {
    fn print(&self) {
        println!("&&ihao: print: {:p}", self);
    }
}

fn main() {
    let n = Nihao(String::from("hades"));
    println!("n");
    n.print();  // 匹配1
    println!("&n: {:p}", &n);

    let c = &n;
    println!("&n");
    c.print(); // 匹配1
    println!("&c: {:p}", &c);

    //(&&n).print();
    //(&&&&&&n).print();
    let d  = &c;
    println!("&&n");
    d.print();    // 匹配2
    println!("&d: {:p}", &d);
    (&&&&n).print(); //匹配3
    //d.print1();
}
```

上面看着很是凌乱,其实是有诀窍的,那就是一句话:编译器是根据trait里面的方法的签名进行选择的,不合适的进行多次解引用或者进行一次的添加&, 只需要添加一次能找到合适的方法签名的,不会选择多次解引用的.所以编译器自动添加一次&应该是优先选择,找不到才会选择进行多次的解引用.参照2rust自动解引用.

对于n.print(); print方法接收的是一个&n的参数,那么编译器添加一次&,就匹配了.

对于(&n).print(); impl 1中的方法是一个print(&Nihao),匹配直接调用.

对于(&&n).print();impl 2中的方法是一个print(&&Nihao),匹配直接调用.

如果把impl 2注释掉:
对于n.print(); print方法接收的是一个&n的参数,那么编译器添加一次&,就匹配了.
对于(&n).print(); impl 1中的方法是一个print(&Nihao),匹配直接调用.
对于(&&n).print();找不到合适和的方法,优先添加一次&,匹配到了impl 3中的方法是一个print(&&&Nihao),匹配直接调用.

如果把impl 3修改成:

```
impl Women for &&Nihao {
    fn print(&self) {
        println!("&&ihao: print: {:p}", self);
    }
}
```

对于n.print(); 添加一次&,找到了impl 1.
对于(&n).print(); 直接匹配impl 1中的方法
对于(&&n).print();添加一次&找不到,解引用一次,直接匹配到了impl 1中的方法.

如果删除impl 1:
对于n.print(); 添加一次&找不到,找不到(其实编译器添加两次&就会找到impl 2的,但是编译器只是会添加一次.)报错了
对于(&n).print(); 直接匹配impl 2中的方法
对于(&&n).print();添加一次&找不到,解引用一次,直接匹配到了impl 2中的方法.

上面的都是通过匹配函数签名的.但是在泛型的trait bound中,就是精确匹配了.

```
fn nihao<T: Women>(x: T) {
    ...
}
```

```
struct Nihao;
```

(1): Impl Women for Nihao{}
(2): Impl Women for &Nihao{}

```
Let n = Nihao;
Nihao(n); // 正确,匹配了1
Nihao(&n); //正确,匹配了2
Nihao(&&n); //错误, &&Nihao类型并未实现Women trait.
```