# 2rust自动解引用

2020年3月9日 10:29

Rust并没有一个与->等效的运算符，相反，Rust有一个叫自动引用和解引用的功能。
方法调用是Rust中少数几个拥有这种行为的地方。
他是这样工作的，当使用object.something()调用方法时，Rust会自动为object添加&、&mut或*以便使object与方法签名匹配,但是只能添加一次.

自动解引用的诀窍：
优先找到匹配的reref(),优先选择自动添加一次&符号能找到函数签名的方法,如果找不到再进行一次或者多次的解引用操作.

deref可以手动调用，编译器也会自动的调用，自动调用发生在一下几种情形：

先看标准库中的通用泛型实现:

```
#[stable(feature="rust1",since="1.0.0")]
impl<T:?Sized>Deref for &T {
    typeTarget=T;

    fn deref(&self) -> &T {
        *self
    }
}

#[stable(feature="rust1",since="1.0.0")]
impl<T:?Sized>Deref for &mut T {
    typeTarget=T;

    fn deref(&self) -> &T {
        *self
    }
}
```

1、函数参数不匹配的时候，但是这个参数必须是一个引用的类型。

```
[derive(Debug)]
struct Nihao {
    age: i32,
}
use std::ops::Deref;
impl Deref for Nihao {
    type Target = i32;
```

```
    fn deref(&self) -> &Self::Target {
        &self.age
    }
}

// 函数的定义参数必须是引用，才会自动解引用。如果是i32那就是精确的匹配了，不会自动解引用。
fn print_age(age: &i32) {
    println!("age: {}", age);
}

fn print_age_mut(age: &mut i32) {
    println!("age: {}", age);
}
use std::ops::DerefMut;
impl DerefMut for Nihao {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.age
    }
}
```

注意：*y的操作在Rust编译器的底层是这样执行的：*(y.deref())，
注意，每次当我们在代码中使用 * 时， * 运算符都被替换成了先调用 deref 方法再接着使用 * 解引用的操作，且只会发生一次，不会对 * 操作符无限递归替换.

```
let n = Nihao{ age: 10 };
print_age(&n);
```

这里是Nihao类型的引用，必须是引用，编译器会自动解引用,过程是*((&n).deref()),
这里需要注意的是,这个deref匹配的是我们自己的,而不是标准库中的通用实现,因为&n刚好满足我们定义的deref的函数签名.参照上面说的解引用的诀窍.
解引用后是一个i32类型,为了满足函数print_age(age: &i32)的签名,编译器自动添加一次&符号.

让我们来看下面自动解引用的过程:

```
print_age(&&&n);
*(&(&&n)) => *((&&&n).deref()) => *&&&n => &&n;
*(&(&n)) => &n; // 这个步骤之前,都是匹配的标准库中的通用实现deref函数签名.
```

&n => (&n).deref() -> i32 => 为了满足函数标签生命,自动添加一次&变成&i32类型传入参数;

```
let mut n = Nihao {age: 100};
print_age_mut(&mut n); // 这里是Nihao类型的可变引用，编译器会自动的调用deref_mut的函数变成&mut i32的类型。
```

2、通过点操作操作符号

```
#[derive(Debug)]
struct Nihao1 {
    age: i32,
}

#[derive(Debug)]
struct Nihao2 {
    n: Nihao1,
}

use std::ops::Deref;
impl Deref for Nihao2 {
    type Target = Nihao1;
    fn deref(&self) -> &Nihao1 {
        &self.n
    }
}
let n = Nihao2{n: Nihao1{ age: 10}};
let m = &*n; // *n为Nihao1类型，&*n就是&Nihao的引用
let m  = n.deref(); // 这样也是可以的。
```

*n返回的是Target类型。过程是*(n.deref()).

注意此时的：let m = *n;是错误的，deref是不能移动的。
那为什么Box<T>就可以移动的呢？那是因为Box<T>的定义有一个#[lang = "owned_box"]属性，
这是一个被编译器照顾的类型，所以可以移动T，甚至是T的成员。

通过点操作符调用方法也会被编译器自动的deref,以此找到合适的函数签名.

```
use std::ops::Deref;

#[derive(Debug)]
struct Nihao1{
    age: String,
}

#[derive(Debug)]
struct Nihao2 {
    n: Nihao1,
}

impl Nihao1 {
    fn print_nihao_1(&self) {
       println!("print_nihao_1: {}", self.age);
    }
}

impl Deref for Nihao2 {
```

```
    type Target = Nihao1;
    fn deref(&self) -> &Nihao1 {
        &self.n
    }
}

let n = Nihao2{
        n: Nihao1 {
            age: "hades".to_string()
        }
    };

n.print_nihao_1();
```

虽然Nihao2的类型并没有实现print_nihao_1这个函数，但是编译器把&n自动的调用deref变成了&Nihao1然后调用了print_nihao_1函数。

重点：编译器的自动解引用是什么样的步骤呢?
比如：(&&&&&&&n).print_nihao_1();
编译器在&&&&&&&n中找不到print_nihao_1()这个函数，&(&&&&&&n) 括号里面的看做是一个类型，找到了标准库中的通用实现deref
所以*(&(&&&&&&n)) => &(&&&&&n);
*(&(&&&&&n)) => &&&&&n;
*(&(&&&&n)) => &&&&n;
*(&(&&&n)) => &&&n;
*(&(&&n)) => &&n;
*(&(&n)) => &n;
因为n是一个Nihao2的类型，编译器发现Nihao2没有实现print_nihao_1函数，编译器尝试查找Nihao2的Deref实现。
然后采取自动解引用 *((&n).deref()) => Nihao1 类型，为了匹配Nihao1的类型中print_nihao_1的函数签名,自动添加&.

print_nihao_1()这个方法是不能这样定义的：fn print_nihao_1(self),因为rust不允许解引用移动所有权，Box除外.

3、在match匹配的时候，编译器不会帮我们自动解引用的，需要手动处理的。因为编译器要给我找到一个最佳的匹配。

4、在let定义的时候，不能自动调用解引用，例如let a = &&&&&&&&n;这个不会自动解引用的。
引用的引用,我们又不是这样定义的,let n = 10; let b = &n; let c: &&i32 = &b;

那为什么上面的a还有地址的值呢?这个值到底是多少呢?
原因就是Rust会生成中间变量的形式. let a = &&&n;其实在Rust的底层是这样的:let tmp1 = &n; let tmp2 = &tmp1; let a = &tmp2;的形式生成的.