20生命周期

2020年3月9日 12:30

我们知道生命周期的注解是一下这个样子的:
&'a str:
&'a mut str:

其实'a就可以看做是一种泛型,在赋值的时候,编译器会计算出被赋值参数的生命周期.
```
let s = String::from("hades");
let b = &s;
```
其实编译器内部是这样展开的:`let b: &'a String = &'a s;`
编译器会把变量的b的生命周期计算出来赋值给泛型'a,
&'a s,这里的'a就有了具体的值了,就是变量b的生命周期.
&'a s的意义就是说:变量s的生命周期一定要大于等于'a的生命周期.变量s至少要活得和b一样久.

&'a String:的意思就是说类型是String的变量的生命周期一定要大于'a的生命周期.

明白了这个意思,我们来看函数:
```
fn nihao<'a>(x: &'a String) -> &'a String {}
```
我们在调用这个函数的时候,比如:
```
let s = String::from("hades");
let ret = nihao(&s);
```

这个函数有一个泛型,可以接受具体的值的.
因为s的引用赋值给了形参x,所以可以看做是`let x = &'a String`,所以这里的泛型'a的生命周期
其实被赋值为形参x的生命周期.因为x是函数nihao内部的作用域的,String的生命周期肯定是大于x的生命
周期的,编译通过. 再看返回值:`let ret = nihao(&s);`可以看做是`let ret: &'a String = nihao(&s);`
ret的生命周期赋值给了'a,然后比较String的生命周期是否和ret的生命周期大或者相同.

完整的例子:
```
fn nihao<'a>(x: &'a String) -> &'a String {
    x
}

let name  = String::from("hades");
let ret = nihao(&name);
```
因为nihao返回的是x,是name的引用,那么name的生命周期是比ret长的.

如果是:
```
fn nihao<'a>(x: &'a String) -> &'a String {
```

```
    &String::from("wawa");
}
let name = String::from("hades");
let ret = nihao(&name);
```

这里就会发生错误,传递参数地方正确,但是返回的时候,String的生命周期是nihao的内部,这里的'a的生命周期是ret的,
String的生命周期小于'a了,错误.

下面解释一下生命周期bound拥有泛型的引用:

```
struct Inner<'a, T> {
    data: &'a T,
}
```

这里会发生编译错误! 因为T可以是任意类型,T自身也可能是一个引用类型,或者是一个存放了一个或多个引用的类型,他们可能
有着不用的生命周期.Rust编译器不能确认T会与'a存活一样久.

前面我们不是说过&'a String,就是表示String类型的变量的生命周期比'a的生命周期长的表示吗?
怎么这里错误了呢?
因为T可能是一个引用.比如:
假如T为&String.

```
struct Inner<'a, &'b String>
where 'b: 'a
{
    data: &'a &'b String,
}

let name = String::from("wawa");
let name_ref = &name;
let s = Inner{ data: &name_ref };
```

name_ref的生命周期给了'b,data的生命周期给了'a,可以看到name的生命周期大于'b,
'b的生命周期大于'a,正确.

如果用泛型T表示T里面的所有生命周期和'a的关系,这样写:

```
struct Inner<'a, T:'a> {
    data: &'a T,
}
```

如果T是一个引用的话,那么'b:'a.记住一个引用的引用也是具有生命周期的.

生命周期的省略规则:(准确的说是生命周期注解的生成方式)
(1):每一个引用的参数都有自己的生命周期
(2):如果只有一个输入生命周期参数,那么他被赋予所有生命周期参数.
(3):如果方法有多个输入生命周期参数,如果存在&self或者&mut self,那么self的生命周期赋值给
    所有的输出生命周期参数.

注意:这里说的赋值其实是一种生命周期注解的生成方式,并不是上面说的真实的赋值.

'static: 这里的生命周期不是泛型了,是一个具体的值,这真是一个特例啊,
因为
static N: &'static str = "nihao"; // 'static是一个确切的值 给N的,代表N的生命周期就是'static N会作为一个静态变量一切存在
static N: &str = "nihao"; // 这样也是正确的,因为编译器会自动添加'static.就像在main函数中,编译器会自动生成一些'a,'b等进行区分生命周期
static M: Nihao = Nihao {}; //结构体的形式,M会一直存在
fn main() {
    let b = &M; // 对M的引用会产生生命周期的赋值.
    // 按照一般的main函数会自动生成一下'a,'b的生命周期标识进行区分,但是这里,是这样的,let b: &'static = &M;
    // 也就是说,b引用了一个静态变量,那么编译器判断b的生命周期为'static. 按照道理来说,b的生命周期应该在main函数里面才对,
    // 这个static还真是一个特例般的存在.

    // 我们来看&M取引用这个动作,在编译器中,会获取M变量的生命周期,然后和b的进行对比.
}

变性:

一个特别有意思的语法糖是，每一个 let 表达式都隐式引入了一个作用域。大多数情况下，这一点并不重要。但是当变量之间互相引用的时候，这就很重要了。

我们所说的子类型是这样的,
'big:'small
big的生命周期比small的大,就说big就是small的子类型.

将生命周期类型化是一个过于自由的设计,当然你不能写一个接收'a类型的值的函数!
生命周期只是别的类型的一部分,所以我们需要一些办法来处理它,这就是变性.

变性是类型构造函数和它的参数相关的一个属性.Rust中的类型构造函数是一个带有
无界参数的通用类型.这里说的构造函数,是构造类型的,并不是我们一般说的构造特定类型值的函数.这里
是编译器自动构造出一个类型的函数.
比如: Vec是一个构造函数,它的参数是一个无界的通用类型T,参数就是T,返回值是类型vec<T>,

&和&mut也是类型构造函数,他们有两个参数,一个是生命周期泛型,一个是引用指向的通用类型泛型.

下文说的T和U是类型泛型,T和U是一种所有权类型,也可能是一种引用类型.
(1): 如果当T是U的子类型是, F<T>也是F<U>的子类型, 函数F对于T是协变的.
如果T和U是所有权类型,所有权类型也有一个生命周期,如果T比U的大,F<T>比F<U>的也大,那么
函数F就是对于T协变的.
如果T和U是引用类型,比如:T是&'static str, U是&'small str, T的生命周期是'static,
U的生命周期是'small, 'static比'small大,同时F<T>的生命周期也比F<U>的生命周期大,
那么函数F就是对T协变的.
(2): 如果当T是U的子类型时,F<U>是F<T>的子类型,那么函数F对于T是逆变的.
反过来了,T的生命周期比U要大,用类型构造函数构造出来的类型的生命周期,F<U>的却比F<T>的大,
我们就叫做函数F对于T是逆变的.
(3): 其他情况,子类型之间没有关系,则F对于T是不变的.

<协变就是大的生命周期可以变成较小的生命周期.这里说的协变,主要关注类型互相引用的情况,
下面说的泛型T,其实大部分情况是一个引用类型的情况,拥有所有权的类型的情况,可以不用关注.>

一些重要的变性:
一: &'a T对于'a和T是协变的. &是一个类型构造函数,'a和T是两个类型参数(一般是泛型),那为什么&这个类型
构造函数为什么对于'a和T是协变的呢? 如果'a:'b, 'a是'b的子类型,F<'a>也是F<'b>的子类型,F对于'a
是协变的. 同理得出对于T也就是协变的.因为'a比'b的大,可以协变成'b的生命周期,类型T的作用域比U的大,
在&函数构造类型的时候,T的作用域协变成U的作用域.协变成'b的生命周期,当'b作用域结束的时候,可以释放这个
借用.

二: &'a mut T对'a是协变的,对于T是不变的.这很重要. 'a: 'b, 'a比'b的大,当构造的时候,
```
fn overwrite<T: Copy>(intput: &mut T, new: &mut T) {
    *input = *new;
}
```
注意,这里的泛型T可以是一个引用类型.
比如overrite(&mut &'static str, &mut &'a str);因为编译器上面的代码可以生成为:
```
fn overwrite<T: Copy>(intput: &mut &'a str, new: &mut &'a str) {
```

```
            *input = *new;
        }
```
'static是'a的子类型,如果&mut T对于T是协变的话,'static要降级和new的生命周期一样:就变成了:

```
overrite(&mut &'a str, &mut &'a str);
```

接下来: *intput = *new;就可以正常赋值了,如果不降级的话,一个短的生命周期'a是不能赋值给一个大的生命周期的'static的.

但是在*input可以长久存在,*new可能已经释放了.造成了错误.

三: UnsafeCell<T>, Cell<T>,RefCell<T>,Mutex<T> 和其他的内部可变类型对于 T 都是不变的

四: Box, Vec 以及所有的集合类对于它们保存的类型都是协变的

五: fn(T) -> U 对于 T 是逆变的, 对于 U 是协变的. 函数类型,函数也是一种类型.let a: fn(x:&'static str) -> &'a str = add;

这是可以传递的.

变量a的期待的类型是fn(x: T) -> U,

那么我们定义一个函数fn add(x: &'a str) -> &'static str {}; 'static是'a的子类型,协变就是大的生命周期类型可以给小的生命周期类型

赋值,我们定义的函数'a小于'static, 如果参数是协变的话,这里不能调用, 但是这里是逆变,小的生命周期赋值给大的生命周期.返回值是协变的.

六: *const 和 & 有着完全一样的语义, 所以变性也是一样的。*mut 正相反, 它可以解引用出一个 &mut, 所以和 cell 一样, 它也是不变的。

结构体会继承它的成员的变性。如果结构体 Foo 有一个成员 a, 它使用了结构体的泛型参数 A, 那么 Foo 对于 A 的变性就等于 a 对于 A 的变性。可如果 A 被用在了多个成员中:

如果所有用到 A 的成员都是协变的, 那么 Foo 对于 A 就是协变的
如果所有用到 A 的成员都是逆变的, 那么 Foo 对于 A 也是逆变的
其他的情况, Foo 对于 A 是不变的

```
use std::cell::Cell;
```

结构体A继承了自己成员的变性.

```
struct Foo<'a, 'b, A: 'a, B: 'b, C, D, E, F, G, H, In, Out, Mixed> {
    a: &'a A,      // 对于'a和A协变
    b: &'b mut B, // 对于'b协变, 对于B不变

    c: *const C,  // 对于C协变
    d: *mut D,    // 对于D不变

    e: E,         // 对于E协变
```

```
    f: Vec<F>,    // 对于F协变
    g: Cell<G>,   // 对于G不变

    h1: H,        // 对于H本该是可变的，但是……
    h2: Cell<H>,  // 其实对H是不变的，发生变性冲突的都是不变的

    i: fn(In) -> Out,       // 对于In逆变，对于Out协变

    k1: fn(Mixed) -> usize, // 对于Mix本该是逆变的，但是……
    k2: Mixed,              // 其实对Mixed是不变的，发生变性冲
突的都是不变的
}
```

下表展示了各种牛 X 闪闪的 PhantomData 用法：

```
Phantom 类型    'a   'T
PhantomData<T> -    协变 (可触发 drop 检查)
PhantomData<&'a T>  协变  协变
PhantomData<&'a mut T>   协变  不变
PhantomData<*const T>    -    协变
PhantomData<*mut T> -    不变
PhantomData<fn(T)>  -    逆变 (*)
PhantomData<fn() -> T>   -    协变
PhantomData<fn(T) -> T>  -    不变
PhantomData<Cell<&'a ()>>     不变  -
```

(*) 如果发生变性的冲突，这个是不变的