生命周期

jask

09/28/2024

在存在多个引用时，编译器有时会无法自动推导生命周期，此时就需要我们手动去标注，通过为参数标注合适的生命周期来帮助编译器进行借用检查的分析。

生命周期标注并不会改变任何引用的实际作用域

和泛型一样，使用生命周期参数，需要先声明 `<’ a>`

`x`、`y` 和返回值至少活得和’`a` 一样久（因为返回值要么是 `x`，要么是 `y`）

可以对生命周期进行下总结：生命周期语法用来将函数的多个引用参数和返回值的作用域关联到一起，一旦关联到一起后，Rust 就拥有充分的信息来确保我们的操作是内存安全的。

结构体中的生命周期

举例：

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

`ImportantExcerpt` 结构体中有一个引用类型的字段 `part`，因此需要为它标注上生命周期。结构体的生命周期标注语法跟泛型参数语法很像，需要对生命周期参数进行声明 `<’ a>`。该生命周期标注说明，结构体 `ImportantExcerpt` 所引用的字符串 `str` 必须比该结构体活得更久。

从 `main` 函数实现来看，`ImportantExcerpt` 的生命周期从第 `4` 行开始，到 `main` 函数末尾结束，而该结构体引用的字符串从第一行开始，也是到 `main` 函数末尾结束，可以得出结论结构体引用的字符串活得比结构体久，这符合了编译器对生命周期的要求，因此编译通过。

生命周期消除

编译器为了简化用户的使用，运用了生命周期消除大法。

举例：

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

对于 `first_word` 函数，它的返回值是一个引用类型，那么该引用只有两种情况：

从参数获取

从函数体内部新创建的变量获取

如果是后者，就会出现悬垂引用，最终被编译器拒绝，因此只剩一种情况：返回值的引用是获取自参数，这就意味着参数和返回值的生命周期是一样的。

三条消除规则

编译器使用三条消除规则来确定哪些场景不需要显式地去标注生命周期。其中第一条规则应用在输入生命周期上，第二、三条应用在输出生命周期上。若编译器发现三条规则都不适用时，就会报错，提示你需要手动标注生命周期。

1. 每一个引用参数都会获得独自的生命周期

   例如一个引用参数的函数就有一个生命周期标注： `fn foo<'a>(x: &'a i32)`，两个引用参数的有两个生命周期标注:`fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`，依此类推。

2. 若只有一个输入生命周期（函数参数中只有一个引用类型），那么该生命周期会被赋给所有的输出生命周期，也就是所有返回值的生命周期都等于该输入生命周期

   例如函数 `fn foo(x: &i32) -> &i32`，x 参数的生命周期会被自动赋给返回值 `&i32`，因此该函数等同于 `fn foo<'a>(x: &'a i32) -> &'a i32`

3. 若存在多个输入生命周期，且其中一个是 `&self` 或 `&mut self`，则 `&self` 的生命周期被赋给所有的输出生命周期

   拥有 `&self` 形式的参数，说明该函数是一个方法，该规则让方法的使用便利度大幅提升。

方法中的生命周期

首先看一下泛型的语法：

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
```

实际上，为具有生命周期的结构体实现方法时，我们使用的语法跟泛型参数语法很相似：

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

其中有几点需要注意的：

`impl` 中必须使用结构体的完整名称，包括 `<'a>`，因为生命周期标注也是结构体类型的一部分！

方法签名中，往往不需要标注生命周期，得益于生命周期消除的第一和第三规则

但是注意，如果要手动修改返回值的生命周期，就要注意到：

```
impl<'a: 'b, 'b> ImportantExcerpt<'a> {
    fn announce_and_return_part(&'a self, announcement: &'b str) -> &'b str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

需要指明输入参数的生命周期之间的关系。

就关键点稍微解释下：

’a: ’b，是生命周期约束语法，跟泛型约束非常相似，用于说明’a 必须比’b 活得久

可以把’a 和’b 都在同一个地方声明（如上），或者分开声明但通过 where ’a: ’b 约束生命周期关系

静态生命周期

在之前我们学过字符串字面量，提到过它是被硬编码进 Rust 的二进制文件中，因此这些字符串变量全部具有’static 的生命周期：

```
let s: &'static str = " 我没啥优点，就是活得久，嘿嘿";
```

在不少情况下，’static 约束确实可以解决生命周期编译不通过的问题，但是问题来了：本来该引用没有活那么久，但是你非要说它活那么久，万一引入了潜在的 BUG 怎么办?

因此，遇到因为生命周期导致的编译不通过问题，首先想的应该是：是否是我们试图创建一个悬垂引用，或者是试图匹配不一致的生命周期，而不是简单粗暴的用’static 来解决问题。

但是，话说回来，存在即合理，有时候，’static 确实可以帮助我们解决非常复杂的生命周期问题甚至是无法被手动解决的生命周期问题，那么此时就应该放心大胆的用，只要你确定：你的所有引用的生命周期都是正确的，只是编译器太笨不懂罢了。

&’static 和 T:’static

&’static

&’static 对于生命周期有着非常强的要求：一个引用必须要活得跟剩下的程序一样久，才能被标注为 &’static。

对于字符串字面量来说，它直接被打包到二进制文件中，永远不会被 drop，因此它能跟程序活得一样久，自然它的生命周期是’static。

但是，&’static 生命周期针对的仅仅是引用，而不是持有该引用的变量，对于变量来说，还是要遵循相应的作用域规则

T: ’static

相比起来，这种形式的约束就有些复杂了。

首先，在以下两种情况下，T: ’static 与 &’static 有相同的约束：T 必须活得和程序一样久

```
use std::fmt::Debug;

fn print_it<T: Debug + 'static>( input: T) {
    println!( "'static value passed in is: {:?}", input );
}

fn print_it1( input: impl Debug + 'static ) {
    println!( "'static value passed in is: {:?}", input );
}



fn main() {
    let i = 5;

    print_it(&i);
    print_it1(&i);
}
```

以上代码会报错，原因很简单： &i 的生命周期无法满足’static 的约束，如果大家将 i 修改为常量，那自然一切 OK。

稍加修改

```
use std::fmt::Debug;

fn print_it<T: Debug + 'static>( input: &T) {
    println!( "'static value passed in is: {:?}", input );
}

fn main() {
    let i = 5;
```

```
    print_it(&i);
}
```

就不会报错了

原因在于我们约束的是 `T`，但是使用的却是它的引用 `&T`，换而言之，我们根本没有直接使用 `T`，因此编译器就没有去检查 `T` 的生命周期约束！它只要确保 `&T` 的生命周期符合规则即可，在上面代码中，它自然是符合的。

Static 针对谁?

大家有没有想过，到底是 `&'static` 这个引用还是该引用指向的数据活得跟程序一样久呢?

答案是引用指向的数据，而引用本身是要遵循其作用域范围的，我们来简单验证下:

```
fn main() {
    {
        let static_string = "I'm in read-only memory";
        println!("static_string: {}", static_string);

        // 当 `static_string` 超出作用域时，该引用不能再被使用，但是数据依然会存在于 binary 所占用的内存中
    }

    println!("static_string reference remains alive: {}", static_string);
}
```

以上代码不出所料会报错，原因在于虽然字符串字面量“`I'm in read-only memory`”的生命周期是`'static`，但是持有它的引用并不是，它的作用域在内部花括号 `}` 处就结束了。