从C++到Rust，不要Move，要自己的Clone

原创 Ajonbin AJonbin的杂货铺 2024年03月14日 21:23 美国

之前讲到默认的Clone实现可以满足大多数情况。

但如果我们想要定制自己的Clone该怎么做呢。

先来看看Clone Trait的定义。

Trait std::clone::Clone

```
pub trait Clone: Sized {
    // Required method
    fn clone(&self) -> Self;

    // Provided method
    fn clone_from(&mut self, source: &Self) { ... }
}
```


Clone Trait定义了两个函数。

一个函数是你需要实现的clone函数 fn clone(&self) -> Self。

这里有两个self。

一个是小写的self，作为参数传入clone()函数。

另一个是首字大写的Self，它是clone()函数的返回值。

小self是一个Rust的关键字，指的就是实现Trait的对象本身。这个和C++中的this，Python中的self的含义一样。

Self表示一个类型，即当前对象的类型，也就是实现这个trait的类型。

fn clone(&self) -> Self这个函数签名也就是传入某个对象的实例，然后返回一个同类型的实例。

Clone trait里定义的另一个函数是 fn clone_from(&mut self, source: &Self)。

它有两个参数，一个self，也是实现Clone trait的对象本身，另一个是拷贝的源头source，它是Self类型，也就是实现Clone trait的类型。

clone_from(&mut self, source: &Self)是将source拷贝到自己。

这是一个由Clone trait提供的函数。 也就是说你可以不用去实现clone_from()。

从功能上讲，**a.clone_from(&b)** 和 **a = b.clone()** 是等价的， 都是从b拷贝到a。

```
fn clone_from(&mut self, source: &Self) {
    *self = source.clone()
}
```


这就是Clone tarit中clone_from()的实现代码，其实很简单，就是调用了clone()函数。

那么为什么还要设计这么一个clone_from()函数？

clone_from的官方文档中有这样一句话

> can be overridden to reuse the resources of `a` to avoid unnecessary allocations.
> 可以重写clone_from()来重用已有资源，已避免不必要的资源分配

怎么能避免资源分配呢？假设你定义的类型T中一块大内存。现在你已经有两个类型为T的变量，a和b。那么你可以重现clone_from，当a.clone_from(&b)的时候，你不用再为a重新分配这块大内存，只需要直接拷贝内存就可以。这样也就避免了一次内存分配。

从C++的角度来看：

clone(&self)，就像**拷贝构造函数**，用来创建一个新的对象。

clone_from(&mut self, source: &Self)，就像是**赋值构造函数**，复制到一个已经存在的对象。

理论讲完了，还是写段实际的例子吧。

```rust
#[derive(Debug)]
struct Point{
    x:i32,
    y:i32,
}

impl Clone for Point{
    fn clone(&self) -> Point{ 1
        println!("Point::clone() called");
        Point{
            x: self.x.clone(),
            y: self.y.clone(),
        }
    }

    fn clone_from(&mut self, source: &Self){ 2
        println!("Point::clone_from() called");
        *self=source.clone()
    }
}

fn main(){
    let p1 = Point{x:10,y:10};
    let mut p2 = Point{x:0,y:0};
    p2.clone_from(&p1); 3
    println!("p1 -- {:?}", p1);
    println!("p3 -- {:?}", p2);
}
```


还是以一个简单的Point结构为例。

标记1，实现了clone()，就是依次调用成员变量的clone()函数。

标记2，重写了clone_from()，然后再调用clone()函数。

标记3，通过clone_from()将p1拷贝到p2。

Standard Output

```
Point::clone_from() called
Point::clone() called
p1 -- Point { x: 10, y: 10 }
p3 -- Point { x: 10, y: 10 }
```


转移所有权Move差不多讲完了，下次讲讲出借所有权--reference。

上一篇：从C++到Rust，不要Move，要默认的Clone