Rust的ownership是什么?

2019-12-21

Rust是内存安全的。Facobook的Libra使用Rust开发，并推出了新的编程语言Move。Move最大的特性是将数字资产作为资源（Resource）进行管理，资源的含义是只能够移动，无法复制，就像纸币一样，以此来保证数字资产的安全。其实Move的这种思想并不是独创的，Rust早已使用这样的方式来管理内存，因此Rust是内存安全的。Rust中的内存由ownership系统进行管理。

Java的引用计数

垃圾回收有很多种方式，ownership是其中之一。Java使用的是引用计数，引用计数法有一个广为人知的缺陷，无法回收循环引用涉及到的内存空间。引用计数的基本规则是，每次对内存的引用都会触发计数加一，比如实例化对象，将对象赋值给另一个变量，等。当变量引用被取消，对应的计数就减一，直到引用计数为0，才释放空间。

```
class Test {
    Test ref = null;
}

Test a = new Test(); // a的计数加一
Test b = new Test(); // b的计数加一
// 此时a的计数是1，b的计数是1

a.ref = b;           // a的计数加一，因为ref是a的类变量
b.ref = a;           // b的计数加一，因为ref是b的类变量
// 此时a的计数是2，b的计数是2

a = null;            // a的计数减一，因为a的引用被释放
b = null;            // b的计数减一，因为b的引用被释放
// 此时a的计数是1，b的计数是1
```

因此，在a和b的引用被释放时，它们的计数仍然为1。想要a.ref的计数减一，就要将a.ref指向nulll，需要手动操作指定为null吗？当然不需要，Java从来没有手动释放内存空间的说法。一般情况下，a.ref执行的对象也就是b的空间被释放（计数为0）时，a.ref的计数也会自动减一，变成0，但此时因为发生了循环引用，b需要a的计数变为0，b的计数才能变成0，可a要想变成0，需要b先变成0。相当于死锁。

这和Rust的ownership有关系吗？当然，没有关系……

ownership

ownership有三条基本规则：

- 每个值都拥有一个变量`owner`
- 同一时间只能有一个`owner`存在
- 当`owner`离开作用域，值的内存空间会被释放

作用域多数情况由`{}`界定，和常规的作用域是一样的概念。

```
{                       // s还没有声明
    let s = "hello";    // s是可用的
}                       // s已经离开作用域
```

Rust的变量类型分简单类型和复杂类型，相当于普通变量和引用变量，因为ownership的存在，简单类型发生赋值操作是，值是被复制了一份的，但复杂类型是将引用直接重置到新的引用变量上，原先的变量将不可用。

```
let x = 5;
let y = x;                          // y是5，x还是5
```

```
let s1 = String::from("smallyu");
let s2 = s1;                        // s2是"smallyu", s1已经不可用
```

赋值过程中，s2的指针先指向string，然后s1的指针被置空，这也就是移动（Move）的理念。如果想要s1仍然可用，需要使用clone复制一份数据到s2，而不是改变指针的指向。

```
let s1 = String::from("smallyu");
let s2 = s1.clone();                // s1仍然可用
```

函数

目前提到的有两个概念，一是ownership在离开作用域后会释放内存空间，二是复杂类型的变量以移动的方式在程序中传递。结合这两个特点，会发生这样的情况：

```
fn main() {
    let s = String::from("smallyu");
    takes(s);               // s被传递到takes函数
                            // takes执行结束后，s已经被释放
    println!("{}", s);      // s不可用，程序报错
}
fn takes(s: String) {       // s进入作用域
    println!("{}", s);      // s正常输出
}                           // s离开作用域，内存空间被释放
```

如果把s赋值为简单类型，比如5，就不会发生这种情况。对于复杂类型的变量，一旦离开作用域空间就会释放，这一点是强制的，因此目前可以使用函数的返回值来处理这种情况：

```
fn main() {
    let s = String::from("smallyu");
    let s2 = takes(s);
    println!("{}", s2);
}
fn takes(s: String) -> String {
    println!("{}", s);
    s
}
```

takes把变量原封不动的返回了，但是需要一个变量接住takes返回的值，这里重新声明一个变量s2的原因是，s是不可变变量。

引用变量

引用变量不会触发ownership的drop方法，也就是引用变量在离开作用域后，内存空间不会被回收：

```
fn main() {
    let s = String::from("smallyu");
    takes(&s);

    println!("{}", s);
}

fn takes(s: &String) {
    println!("{}", s);
}
```

可变变量

引用变量仅属于可读的状态，在takes中，s可以被访问，但无法修改，比如重新赋值。可变变量可以解决这样的问题：

```
fn main() {
```

```
    let mut s = String::from("smallyu");
    takes(&mut s);

    println!("{}", s);
}

fn takes(s: &mut String) {
    s.push_str(", aha!");
}
```

可变变量也存在限制，同一个可变变量同一时间只能被一个其他变量引用：

```
let mut s = String::from("smallyu");
let r1 = &mut s;
let r2 = &mut s;
println!("{}, {}", r1, r2);
```

程序会报错，这是容易理解的，为了保证内存安全，一个变量只能存在一个可变的入口。如果r1和r2同时有权力更改s的值，将引起混乱。也因此，如果是`r1 = &s`而不是`r1 = &mut s`，程序会没有问题，只能存在一个引用针对的是可变变量的引用变量。

返回值

函数的返回值类型不可以是引用类型，这同样和ownership的规则有关，返回普通变量相当于把函数里面的东西扔了出来，如果返回引用变量，引用变量指向的是函数里面的东西，但函数一旦执行结束就会销毁内部的一切，所以引用变量已经无法引用到函数。

```
fn dangle() -> &String {
    let s = String::from("smallyu");
    &s;
} // 到这里s的内容空间已经释放，返回值无法引用到这里
```

?

没有更多内容了。

最近看了一部能够让人振奋的美剧《硅谷》，编剧给主角挖了很多坑，感觉他们倒霉都是自己作的，编剧也给观众留了很多坑，剧情跌宕起伏到想给编剧寄刀片。抛开那些情节，剧中渲染的geek真的很帅，很帅！当然，神仙打架，凡人也参与不了。