TinTin

第二课：Rust的所有权

Mike Tang

daogangtang@gmail.com

2023-5-18

Rust中“奇怪”的行为

下面例子打印了什么?

```
fn main() {
    let a = 10u32;
    let b = a;
    println!("{a}");
    println!("{b}");
}
```

然后我们来看字符串的行为，大家猜一下下面的程序会输出什么：

```
fn main() {
    let s1 = String::from("I am a superman.");
    let s2 = s1;
    println!("{s1}");
    println!("{s2}");
}
```

是两行 `"I am a superman"` 吗？其它语言中都是这样的。

先不要被吓倒。这片出错信息非常清晰明了，我们一起来读一下。

```
-- move occurs because `s1` has type `String`, which does not implement the
`Copy` trait
```

移动发生了，因为 `s1` 的类型是 `String`，而这种类型并没有实现 `Copy` trait.

然后，编译器指出了移动发生的地方（上述信息中的第16行）。然后又给出了s1被移动后仍然被使用的地方（上述信息中的第17行）。然后在后面还给出怎么改的建议：

```
help: consider cloning the value if the performance cost is acceptable
```

“在性能可以接收的情况下，可以考虑克隆这个值。”

代码建议都给出了：

```
let s2 = s1.clone();
```

于是，我们直接照着代码建议改一下试试。

```
fn main() {
    let s1 = String::from("I am a superman.");
    let s2 = s1.clone();
    println!("{s1}");
    println!("{s2}");
}
```

我们现在来总结一下这个例子，解析一下Rust中的字符串为何有如此奇怪的行为。

首先，我们可以看到在Rust中，字符串的行为好像与u32这种数字类型不一样。前面我们说过，u32这种类型是占用内存字节数固定的类型，而String这种类型，是占用字节数不固定的（动态的）类型。一般来说，对于固定字节数的类型，会默认放在栈上（栈帧中）；而不固定字节数的类型，会默认创建在堆上（成为堆上的一个资源），然后在栈上用一个局部变量来指向它（如上述代码中的s1）。

其它语言对这个底层结节的处理是这样的（当然不同语言细节也不一样），我们拿Java举例。前面我们说过，局部变量都是定义在栈帧中的，Java也是一样。Java语言对于int这类固定字节数类型，在复制给另一个变量的时候，会直接复制它的值。在面对Object这种复杂对象的时候，默认只会复制这个Object的引用给另一个变量（也就是所谓的浅拷贝。浅拷贝也可以看成值复制，只不过复制的不是对象的实际内容，而是对那个实际内容的一个引用，而这个引用的地址值，就存在栈上的局部变量里面）。

为什么会这样设计，因为如果那个Object占用的内存很大，每一个重新赋值，就把那个对象重新拷贝一次（也就是所谓的深拷贝），是非常低效的。一个正常的语言都不会那样干。

所以Java实际上是隐藏了对象的引用和浅拷贝这个细节。

回到Rust。我们看到，对于u32这种固定字节数的类型来说，与Java也是同样的处理，直接在栈上进制值的拷贝。而对于字符串这种动态字节数的类型来说，在变量的再赋值上，Rust除了拷贝字符串的引用外，实际还做了更多事情。

```
fn main() {
    let s1 = String::from("I am a superman.");
    let s2 = s1;
    //println!("{s1}");
    println!("{s2}");
}
```

也就是说，s1把内容“复制”给s2后，s2可用，s1不能用了！

我们也可以说，s1把值（资源）“移动”给了s2。既然是移动了，那原来的变量就没有那个值了。请仔细体会这里与Java的不同之处。Java默认做了引用的拷贝，并且新旧两个变量同时指向那个对象。而Rust没有这么做，Rust虽然也是把字符串的引用由s1拷贝到了s2，但是只保留了最新的s2到字符串的指向，同时却把s1到字符串的指向给“抹去”了。这就是Rust编译器做的那个“更多”的部分。

好奇怪呀！Rust怎么会这样设计。

这正是Rust从头开始梳理整个软件体系的地方，剑指一个目标：内存安全。

所有权

为什么要这样设计。长久以来，计算机领域最聪明的大脑都在探索如何写出更安全的程序，而Rust就走出了一条全新的思路。接下来我们就一起来好好品鉴这种独特的思想角度，不需要带着之前的固有思维来学习。

Rust明确了所有权的概念。值也可以叫资源。所有权就是对资源拥有的权利。Rust基于所有权定义出发，推导了整个世界。所有权的基础是三条定义：

- Rust中，每一个值（资源）都有一个所有者；
- 任何一个时刻，一个值只有一个所有者；
- 当所有者所在作用域结束的时候，值会被释放掉。

三个规则，涉及两个概念，所有者，作用域。

所谓所有者，在代码中，就表示为变量。也就是说所有者会用变量名来表示。

变量的作用域，就是变量有效（valid）的那个区间。在Rust中，简单来说就是在一对花括号括起的里面部分中，从变量创建时开始，到花括号结束的地方。比如：


```
fn main() {
    let s = String::from("hello");
    // do stuff with s
}
```

```
fn main() {
    let a = 1u32;
    {
        let s = String::from("hello");
    }
    // other stuff
}
```

我们现在尝试用所有权规则去翻新一下对前面例子的理解。

```
fn main() {

    let a = 10u32;

    let b = a;

    println!("{a}");

    println!("{b}");

}
```

这个例子中，a具有对值 10u32的所有权。执行 let b = a的时候，把值 10u32 复制了一份，b具有对这个新的 10u32的所有权。当main函数结束的时候，a,b两个变量就离开了作用域，其对应的两个10u32，就都被回收了。这里是栈帧直接结束，栈帧内存被回收，局部变量所占用的内存就一起被回收。

对字符串的例子来说：

```
fn main() {

    let s1 = String::from("I am a superman.");

    println!("{s1}");

}
```

变量s1具有对这个字符串的所有权。s1的作用域从定义到开始，直到花括号结束。s1（栈帧上的局部变量）离开作用域时，变量s1上绑定的内存资源（字符串），就被回收掉了。注意，这里发生的事情是，栈帧中的局部变量离开作用域了，顺带要求堆内存中的字符串资源被回收了，能够做到这一点，是因为这个堆中的字符串资源被栈帧中的局部变量指向了的。

而从Rust的语法层面来看，就是s1对那个字符串拥有所有权，所以s1离开作用域的时候，那个资源就自动被回收了。

RAII

这种堆内存资源随着关联的栈上局部变量一起被回收的内存管理特性，叫作 RAII（Resource Acquisition Is Initialization）。它实际不是Rust的原创，是C++创造的。学过C的同学可以对比一下C中的malloc()方式，C语言里面必须由程序员手动在后面的代码中使用free()来释放堆内存资源，而RAII不需要手动写free()。可以看到RAII是一个相当大的进步。

我们再来分析这个例子：

```
fn main() {
    let s1 = String::from("I am a superman.");
    let s2 = s1;
    //println!("{s1}");
    println!("{s2}");
}
```

变量s1具有对这个字符串的所有权。s1对字符串的所有权从定义到开始，到 let s2 = s1 执行后结束。这句执行后，s2具有对那个字符串的所有权，而此时s1处于什么状态呢？就是处于一种不可用的状态，或者叫无效状态（invalid），Rust编译器把这些给咱们打理得明明白白的。

然后，直到花括号结束，s2及s2所拥有的字符串内存，就被回收掉了。栈帧结束，s1原来所对应的那个局部变量的内存空间也一并被回收掉了。

所有权就是我们Rust的出发点！

使用所有权书写函数

下面我们来看一下，基于所有权规则，函数的写法会变成怎样。

```
fn foo(s: String) {
    println!("{s}");
}

fn main() {
    let s1 = String::from("I am a superman.");
    foo(s1);
}
```

稍微改动一下例子：

```
fn foo(s: String) {
    println!("{s}");
}
fn main() {
    let s1 = String::from("I am a superman.");
    foo(s1);
    println!("{s1}");     // 这里加了一行
}
```

这个例子在其它语言中，一般是不会有问题的。在foo函数中，也许会修改字符串的值，重新打印的时候，会打印出新的值。但是，这其实是一种相当隐晦的模式，而Rust阻止了这种模式。

函数的参数s获取了这个值的所有权。函数参数是这个函数的一个局部变量，其在此函数栈帧结束的时候会被回收，因此这个字符串在这个函数调用结束后，就已经被回收了。

那我们后面的代码还想用s1，该怎么办。可以这样，既然能把所有权移动进函数里面，也当然能把所有权转移出来。

```
fn foo(s: String) -> String {
    println!("{s}");
    s
}
fn main() {
    let s1 = String::from("I am a superman.");
    let s1 = foo(s1);
    println!("{s1}");
}
```

移动还是复制

对于变量的绑定来说，哪些类型默认是做移动（所有权）操作，哪些基本（primitive）类型默认是做复制（而产生新的资源及所有权）操作呢？

默认做复制操作的有：

- 所有的整数类型，比如u32,
- 布尔类型，bool
- 浮点数类型：f32, f64
- 字符类型 char
- 由以上类型组成的元组类型 Tuple，如（i32, i32, char）

其它类型，默认都是做值的移动操作。

借用与引用

但是，正如，前面例子所说，如果要在调用完函数后，继续用这个变量，怎么办呢？我们采取的办法就是把那个值返回回来——把所有权转移回来。

但是这样，相当麻烦。于是，Rust又引入了借用的概念。

借用概念是很自然的，也是跟现实的思维一致的。你有一样东西，别人想用一下，可以从你这里借，你可以出借。那“引用”概念，又是什么呢？其实借用和引用是一体两面，对同一个事情的两个面的描述。你把东西借给别人，也就是别人持有了对你这个东西的引用。借用是站在资源拥有者角度来说的，引用是站在想借这个资源的变量角度来说的。

在Rust中，引用具体就在变量前用“&”符号来表示，比如 &x。其实，引用也是一种值，并且是固定长度的值。既然是值，当然可以赋给另一个变量。既然是固定长度的值，那其实做的就是引用的复制操作。

```
let b = &a;

let c = b;      // 复制操作
```

TinTin

```
fn main() {
    let a = 10u32;
    let b = &a;
    let c = &&&&&a;
    let d = &b;
    let e = b;
    println!("{a}");
    println!("{b}");
    println!("{c}");
    println!("{d}");
    println!("{e}");
}
```

从上面示例可以看出，Rust识别了我们一般情况下的意图，不会打印出引用的地址什么的。上面示例中的c,实际是a的5次引用，但是仍然正确获取到了a的值。d是到a的间接引用，但是仍然正确获取到了a的值。这里可以看出Rust与C这种纯底层语言有显著的区别，Rust还是面向人类（业务）更多一些。因为人们普遍还是关注最终那个值的部分，而不是中间的内存地址。

上面示例中，`let e = b;` b和e都是对a的引用。由于引用是固定长度的值，做的就是引用的复制操作，而并没有对a的值10u32再复制一份。

TinTin

那对字符串来讲会怎样呢?

```
fn main() {
    let s1 = String::from("I am a superman.");
    let s2 = &s1;
    let s3 = &&&&&s1;
    let s4 = &s2;
    let s5 = s2;
    println!("{s1}");
    println!("{s2}");
    println!("{s3}");
    println!("{s4}");
    println!("{s5}");
}
```

符合我们的期望。同样，这些引用，都没有导致堆中的字符串资源被复制一份或多份。字符串的所有权仍然在s1那里，其它s2, s3, s4, s5都是对这个所有权的引用。

同时还可以看出，在Rust中，一个所有权型变量（上述示例中的s1）带有值和类型的信息，一个引用型变量（上述示例中的s2, s3, s4, s5）也带有值和类型的信息，不然它没法正确回溯到最终的值。这些信息是由Rust编译器内部维护的。

不可变引用、可变引用

- 引用包含不可变引用和可变引用
- &x 对变量x的不可变引用
- &mut x 对变量x的可变引用

为什么会有可变引用的存在呢？这个事情是这样的。到目前为止，如果要对一个变量绑定的值进行修改，我们只有拥有那个值的所有权才行。而很多时候，我们没法拥有那个值的所有权（要不然就需要把所有权在整个代码中传来传去，不是一种很好的设计，很多库不会把底层的资源的所有权交出来）。因此需要一种方法，又是一种引用，又能够修改指向的那个值。这样就引入了可变引用。

引用的scope特性

我们来看下面的例子：

```
fn main() {
    let a = 10u32;
    let b = &mut a;
    *b = 20;

    println!("{b}");
}
```

前面我们说到过，要修改一个变量的值，变量名前要加mut修饰符，我们忘加了（是非常正常的事情），Rust编译器给我们指出来了。于是我们加上：

```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;

    println!("{b}");
}
```

把例子变一下：

```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;

    println!("{b}");
    println!("{a}");
}
```

我们再变一下两个打印语句的位置试试。

```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;

    println!("{a}");
    println!("{b}");     // 这里会输出什么?
}
```

我们看下面的例子：

```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;
    let c = &a;
}
```

而下面的代码不能编译：

```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;
    let c = &a;

    println!("{b}");
}
```

而下面的代码又能编译：

```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;
    let c = &a;

    println!("{c}");  // 加了一行这个
}
```

下面的代码又不能编译：

```
fn main() {
    let mut a = 10u32;
    let c = &a;          // 这句位置放在了这里
    let b = &mut a;
    *b = 20;

    println!("{c}");
}
```

下面的代码可以编译：

```
fn main() {
    let mut a = 10u32;
    let c = &a;
    let b = &mut a;
    *b = 20;

    println!("{b}");     // 这里换成了打印b
}
```

有没有发现什么规律？引用的“调用“时机很关键。

前面我们讲过，一个拥有所有权的变量的作用域是从它定义时到花括号结束。而引用的作用域不是这样！**引用的作用域是从它定义到它最后一次使用时结束**！如果它定义了，但并没有被使用，那它的作用域就只有它定义的那一行，即，出生即死亡。

同时，还存在一条规则：一个资源的可变引用与不可变引用的作用域不能交叠（overlap）！也可以说不能同时存在。

然后我们再看一个例子：

```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;
    let d = &mut a;

    println!("{b}");
}
```

总结前面的示例，我们可以得出关于引用（借用）的规则。

- 引用（不可变引用和可变引用都是）变量的作用域不会长于所有权变量的作用域。肯定的，不然就会出现悬锤引用了，这是典型的内存安全问题。Rust中的引用必定是有效的；
- 一个资源的可变引用与不可变引用的作用域不能交叠（overlap），也可以说不能同时存在；
- 某个时刻对某个资源只能存在一个可变引用，不能有超过一个可变引用同时存在；
- 一个资源的不可变引用，可以同时存在多个；

可变引用的排它性

我们再来试试可变引用能否被复制。

```
fn main() {
    let mut a = 10u32;
    let r1 = &mut a;
    let r2 = r1;

    println!("{r1}")
}
```

TinTin

改下例子，

```
fn main() {
    let mut a = 10u32;
    let r1 = &mut a;
    let r2 = r1;

    println!("{r2}")
}
```

- 不可变引用可以被复制
- 可变引用不能被复制，只能被move

用&和&mut来改进函数的定义

TinTin

下面将引用与函数参数结合起来来看，我们要改进前面将字符串所有权传进函数然后又传出来的例子。

第一个例子是将字符串的不可变引用传进函数参数。

```
fn foo(s: &String) {
    println!("in fn foo: {s}");
}
fn main() {
    let s1 = String::from("I am a superman.");
    foo(&s1);
    println!("{s1}");
}
```

然后我们试试将字符串的可变引用传进函数，并修改字符串的内容。

```
fn foo(s: &mut String) {
    s.push_str(" You are batman.");
}
fn main() {
    let mut s1 = String::from("I am a superman.");
    println!("{s1}");
    foo(&mut s1);
    println!("{s1}");
}
```

其实函数的形参接受实参的过程，就是变量绑定值的过程，跟前面那些普通的变量绑定是一个道理。不过函数的参数是这个新函数的局部变量。

从代码可以看到，Rust的代码非常清晰。如果一个函数参数**接受的是可变引用**，或所有权参数，那么它里面的逻辑一般都会对引用的资源进行修改。如果一个函数参数只接受不可变引用，那么它里面的逻辑，就一定不会修改被引用的资源。就是这么**清晰**，太利于代码的阅读了。

Q&A

TinTin