

05 所有权

所有权的定义

上次课我们讲到现代化语言的自动内存管理方案无外乎就是 GC 和 ARC 两种。但是 Rust 属于另辟蹊径的搞出来一个“所有权”（Ownership）的概念。

Rust 中的所有权概念规则如下：

1. 每个值都有一个所有者
2. 在同一时刻，一个值只能有一个所有者
3. 当值的所有者离开作用域，值将被丢弃，其对应的内存也被释放

回顾一下，我们在基础概念一章的时候，讲到过 Rust 中使用 `let` 语句定义变量的过程是：

1. 创建一个值
2. 创建一个变量。这个变量是值的“所有者”
3. 将值绑定到变量。这个变量拥有某个值

之前的内容中我们提到过，在 Rust 中，要显式区分值和变量（所有者），就是为了能够方便理解所有权的概念。

变量作用域

既然前面提到了“当值的所有者离开作用域，值将被丢弃，其对应的内存也被释放”，那么，什么是变量的作用域（Variable Scope）呢？

简单来说，一对儿花括号（{}）就是一个作用域。在这组花括号中定义的变量，当超出花括号的范围之后，就表明这个变量的作用域结束了。推演一下，一个函数的参数的作用域就是整个函数体了。

Rust 中的变量作用域几乎和 JavaScript 中使用 `let` 或 `const` 定义的变量的作用域是一样的，都是块级的。

```

1 fn foo() {
2     { // 此时, s 还不可用
3         let s = "hello"; // 从这一行开始, s 可用了
4         // 使用 s
5     } // s 的作用域到此结束
6 }
```

再看一个例子：

```

1 fn foo() {
2     { // 此时, s 还不可用
3         let s = String::from("hello"); // 从这一行开始, s 可用了
4         // 使用 s
5     } // s 的作用域到此结束
6 }
```

回顾一下，上一节中我们讲到了栈和堆：编译时已知大小的值，存放到栈上；编译时未知大小的值，或者大小不固定的值，存放到堆上。在 Rust 中的 `String` 类型的数据是可变大小的，所以，上面的例子中，变量 `s` 的值存放到堆上。堆上的存储空间由程序申请，也是由程序负责释放。那么请问上面的例子中，堆上用来存储 `hello` 的内存是如何释放的？答案就在于编译器。编译器会在变量作用域结束之前，插入一些代码来释放内存。上面的例子中，你可以想象成编译器会在行 5 之前插入 `drop()` 释放堆上内存。

所有权转移

在 Rust 中，将一个值赋值给一个变量（让变量拥有这个值）、把一个值作为参数传入到函数中、或者从函数中返回一个值，都会发生所有权转移（Move）。这个设定看起来是非常违反直觉的。所以，我们再回顾一下上一节给出的 JavaScript 的代码示例：

```

1 function foo() {
2     const obj = {
3         name: "John Wick",
```

```

4     age: 40,
5     pet: "Dog"
6   };
7
8   bar(obj);
9   bar(obj); // 这里再调用一次。在 JavaScript 中没有问题
10 }
11
12 function bar(obj) {
13   console.log(JSON.stringify(obj));
14 }
15
16 foo();

```

翻译成类似的 Rust 代码：

```

1 // 以下代码无法通过编译
2 fn main() {
3     let s = String::from("hello");
4     foo(s);
5     foo(s); // s 在上一行调用中已经发生了所有权转移了，所以这一行就无法通
6     //过编译了
7
8     fn foo(s: String) {
9         println!("{}", s);
10    }

```

但是，等等，不太对啊。前面几次课程的练习中，经常将拥有数值类型的变量传入到函数调用中，为什么就不报错呢？这是因为标量类型和 `String` 类型在内存中存放的位置不同：标量类型在编译时大小已知，所以是存放到栈上的；`String` 类型在编译时大小不确定，所以是存放到堆上的。存放到栈上的值很简单，直接复制就可以了：

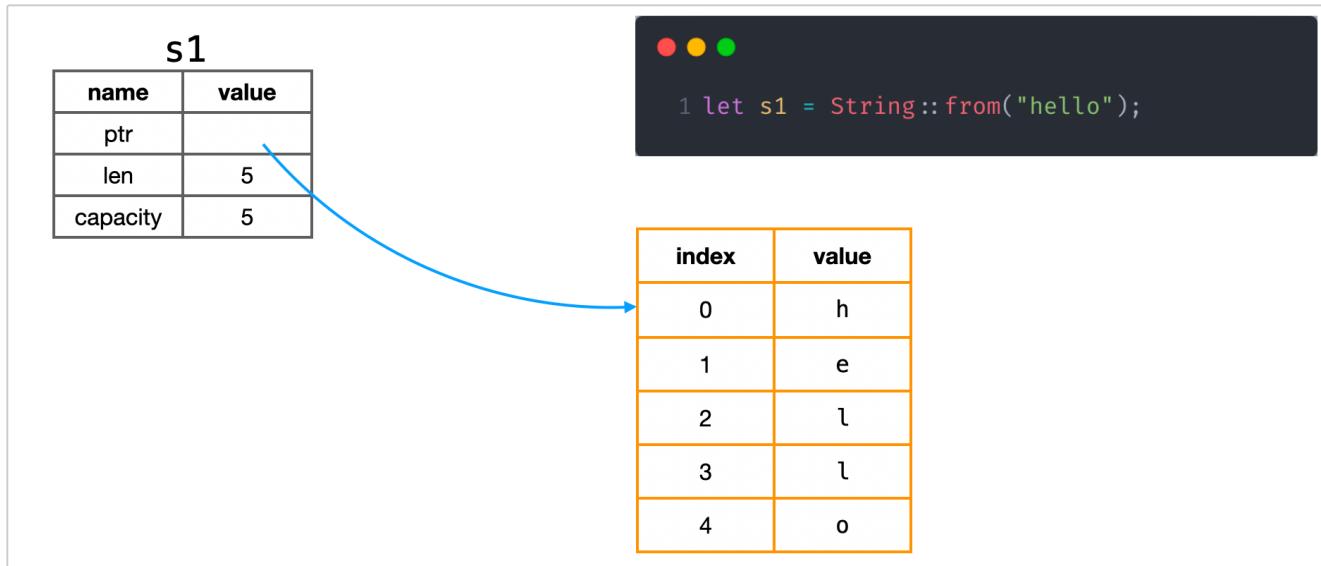
```

1 fn foo() {
2     let x = 5; // 创建值 5, 创建变量 x, 将值 5 的所有者设置为 x (将值绑定
3     let y = x; // 复制值 5, 创建变量 y, 将新复制的值 5 的所有者设置为 y
4 }
```

那么 `String` 类型呢? 我们来看下面的代码:

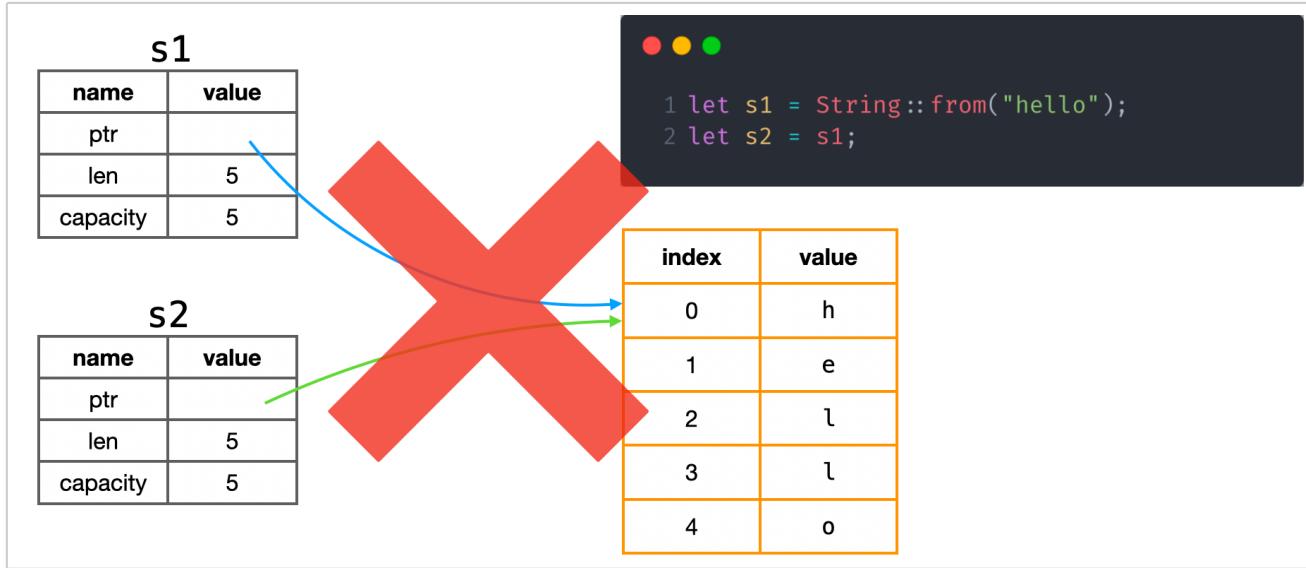
```

1 fn foo() {
2     let s1 = String::from("hello");
3     let s2 = s1;
4 }
```



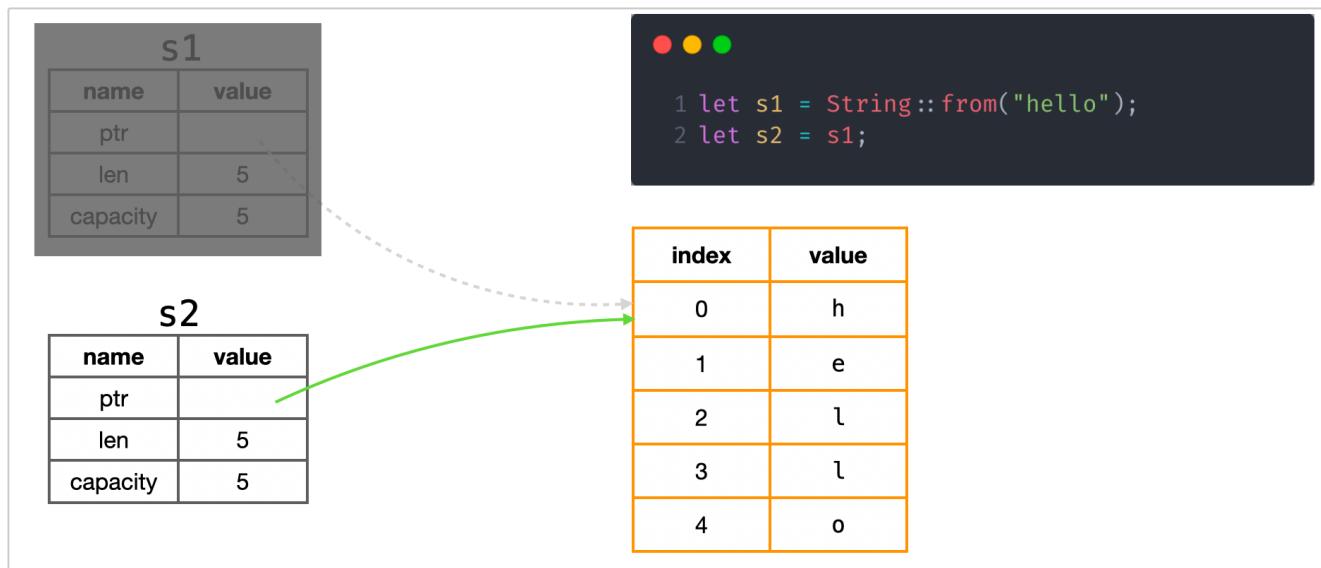
行 2 的代码运行之后, 在堆上申请了一段内存 (图中橘色部分), 存入“hello”字符, 然后在栈上创建一个变量, 这个变量指向堆上所申请的内存的起始地址, 并记录其对应的长度和容量。

我们假定行 3 的代码执行之后, 栈和堆上的示意如下:

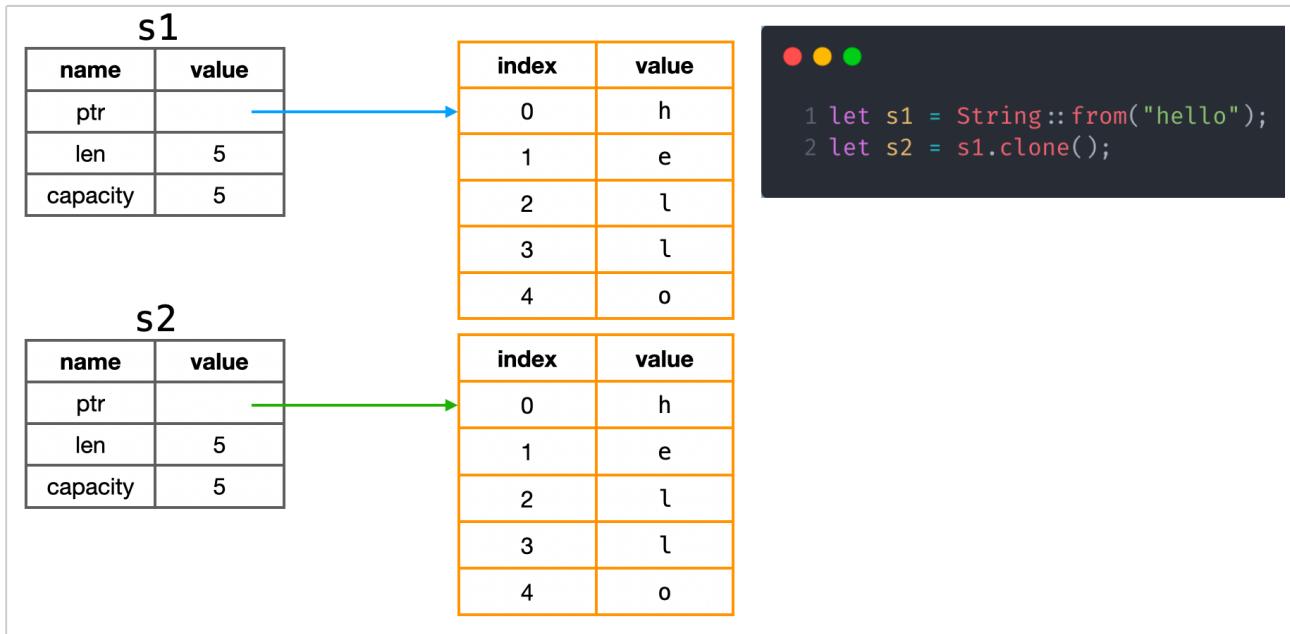


那么，这就违背了 Rust 所有权的规则：在同一时刻，一个值只能有一个所有者。按上图所示，`s1` 和 `s2` 指向了堆上的同一块地址，如果 `s1` 超出作用范围，会释放此部分内存；`s2` 超出作用范围的时候，再一次释放同样地址的内存。这就引发了双重释放的问题。

所以，在上面的代码中，Rust 是采用了所有权转移的方式：当执行 `let s2 = s1` 这行代码的时候，`s1` 就失效了，堆上值的所有者从 `s1` 变成了 `s2`。如下图所示：



如果，我真的需要两个变量拥有“同样”的值，该怎么办？可以使用 `clone()` 方法。类似于 JavaScript 中的“深度拷贝”，就是完完整整的将堆上的值克隆一份，并将其所有者设置为新创建的变量：



通常情况下，我们应该尽量避免克隆。因为毕竟，堆上经常存放较大的数据，如果频繁的克隆数据，会导致占用更多的内存，以及性能的下降。

对于标量类型，因为都实现了 `Copy` trait（trait 可以理解为“接口”，JavaScript 中没有对应的概念），所以可以放心大胆的赋值、传入到函数中等。不过，对于元组和数组类型，必须得是这个元组或数组中的元素都实现了 `Copy` trait，元组或数组本身才是可复制的。

回想一下，我们在前面课程中练习数值计算的时候，一个 `[u32; 5]` 的数组，由于 `u32` 是 `Copy` 的，所以 `[u32; 5]` 的数组也是 `Copy` 的。类似下面的代码，是可以通过编译的：

```

1 fn main() {
2     let numbers = [1u32, 2, 3, 4, 5];
3     let total = arr_add1(numbers);
4     println!("{}:{} sum => {}", numbers, total);
5 }
6
7 fn arr_add1(items: [u32; 5]) -> u32 {
8     let mut total = 0u32;
9     for n in items {
10         total += n;
11     }
12     total
13 }

```

听起来，所有权转移这个概念并不难，但是还有好多场景都会涉及到所有权转移，下面我们将再通过一些例子加深对所有权转移的理解：

赋值

```

1 fn foo() {
2     let mut s = String::from("Hello");
3     s = String::from("World");
4 }
```

在行 3 的代码执行后，值 `Hello` 被丢弃，释放堆内存，`s` 成为 `World` 这个值的所有者。

函数的返回值

```

1 fn generate_greeting_message(name: &str) -> String {
2     let s = format!("Hi {}", name);
3     s
4 }
5
6 fn main() {
7     let msg = generate_greeting_message("Mr. Wang");
8 }
```

为了更加清晰的演示这个场景，上面的示例代码特意在 `generate_greeting_message` 定义了一个变量 `s`，然后让 `s` 作为该函数的返回值。注意，行 3 代码后面，编译器不会插入 `drop()` 调用来丢弃 `s` 所持有的值，而是将这个值的所有权转移出去，对应的就是在 `main` 中，`msg` 变量成为这个值的新所有者。

流程控制

```

1 // 以下代码无法通过编译
2 fn say_hi(name: String) {
3     println!("Hi! {}", name);
4 }
5
6 fn say_hello(name: String) {
```

```

7     println!("Hello! {}", name);
8 }
9
10 fn say_hello_in_chinese(name: String) {
11     println!("你好! {}", name);
12 }
13
14 fn main() {
15     let some_condition = false;
16     let name = String::from("John Smith");
17     if some_condition {
18         say_hi(name);
19     } else {
20         say_hello(name);
21     }
22
23     say_hello_in_chinese(name);
24 }
```

上面这段代码中，`if` 表达式中，无论走到哪个分支，都会导致值 `John Smith` 的所有权从 `name` 转移到所对应的函数中，因此，在 `if` 执行之后，`say_hello_in_chinese` 就无法正确获得所有权了。以上代码无法通过编译。

再来看一个：

```

1 // 以下代码无法通过编译
2 fn main() {
3     let names = [
4         String::from("John"),
5         String::from("Tom"),
6         String::from("Penny"),
7         String::from("Sheldon"),
8     ];
9
10    for i in 0..4 {
11        let s = names[i]; // 这行代码导致 names 拥有的值的所有权被转移
12        println!("{} ", s);
```

```

13 }
14
15     println!( "names[0] = {}", names[0]);
16 }
```

另外，在调用对象方法的时候也可能产生所有权转移，例如：

```

1 // 以下代码无法通过编译
2 fn main() {
3     let s = String::from("Hello world");
4     let bytes = s.into_bytes();
5     println!("{}", s);
6 }
```

上面的代码中，`String` 类型的 `into_bytes()` 方法的定义如下：

```
1 pub fn into_bytes(self) -> Vec<u8>
```

我们可以看到，方法定义的参数是 `self`，表示会发生所有权转移。这种调用类型值方法的时候发生所有权转移的行为，在 Rust 中被称为源类型值被“消费”（Consume）了。

`String.into_bytes()` 方法会消费原本的字符串值，发生所有权转移。

回顾

1. 所有权的定义
2. 变量作用范围
3. 所有权转移