

07 结构体

Rust 中的结构体（Struct）是把一组值绑定在一起，让你可以以一个整体单位操作一组相互关联的数据。大致类似于 JavaScript 中的对象，但是细节上又有所区别。

Rust 中的结构体分为三种：

1. 名值对结构体，Name-Field Structs
2. 元组式结构体，Tuple-Like Structs
3. 单元式结构体，Unit-Like Structs

名值对结构体

名值对结构体和 JavaScript 中的对象很相似，一个结构体拥有多个属性。按照 Rust 命名规范，结构体的名字采用大驼峰命名法，属性名采用蛇形命名法。

```
1 struct Student {  
2     active: bool,  
3     username: String,  
4     email: String,  
5     sign_in_count: u64,  
6 }  
7  
8 fn main() {  
9     let student = Student {  
10         active: true,  
11         username: "John Smith".to_string(),  
12         email: "j.smith@gmail.com".to_string(),  
13         sign_in_count: 0,  
14     };  
15 }
```

上面的示例代码中，我们定义了一个名为 `Student` 的结构体，它拥有 4 个属性，分别是：`active`，`username`，`email` 和 `sign_in_count`。定义结构体的时候，多个属性之间使用逗号（，）分隔。最后一个属性后面的逗号可以有，也可以省略。然后创建了一个 `Student` 的示例，并将其值绑定到 `student` 变量。这里我们第一次见到使用 `"".to_string()` 方法来生成 `String` 类型的值，和 `String::from(&str)` 是同等效果。

上面的代码用 JavaScript 的类来表示，大致如下：

JavaScript 代码

```

1 // JavaScript 代码
2 class Student {
3     constructor() {
4         this.active = false;
5         this.username = null;
6         this.email = null;
7         this.signInCount = 0;
8     }
9 }
10
11 const student = {
12     active: true,
13     username: "John Smith".to_string(),
14     email: "j.smith@gmail.com".to_string(),
15     signInCount: 0,
16 };

```

其实，单纯讲上面的 Rust 代码，更接近于 TypeScript 的 `interface`。因为我们在 Rust 代码中，仅仅定义了 `Student` 结构体，但是未给他增加任何的方法，也没有任何属性的默认值。而上面的 JavaScript 代码实际上是通过 `constructor` 方法在定义属性的同时也给属性赋值了。

TypeScript

```

1 // TypeScript 代码
2 interface Student {

```

```
3     active: boolean,
4     username: string,
5     email: string,
6     signInCount: number,
7 }
8
9 const student: Student = {
10     active: false,
11     username: "",
12     email: "",
13     signInCount: 0
14 };
```

在创建名值对结构体的时候，有一个语法和 JavaScript 一样，就是如果变量名和属性名一致，可以简写，例如：

```
1 struct Student {
2     active: bool,
3     username: String,
4     email: String,
5     sign_in_count: u64,
6 }
7
8 fn main() {
9     let active = true;
10    let username = "John Smith".to_string();
11    let student = Student {
12        active,
13        username,
14        email: "j.smith@gmail.com".to_string(),
15        sign_in_count: 0,
16    };
17 }
```

但是需要注意的是，这里依然遵循所有权的原则，行 13 的代码将 `String` 类型的数据 `John Smith` 的所有权从变量 `username` 转移到 `student` 的 `username` 属性了。但是 `bool` 类型的 `active` 则不存在所有权转移的问题，因为它是 `Copy` 的。

元组式结构体

有一些时候，我们并不一定需要给结构体的每个属性都起名字，尤其是这个结构体的属性很少的时候。此时，可以使用元组式结构体。

```

1 // 平面上一个点的坐标: (x, y)
2 struct Point(f32, f32);
3 let p = Point(0.0, 0.0);
4
5 // RGB 颜色，元组的三个元素分别对应: (R, G, B)
6 struct RgbColor(u8, u8, u8);
7 let c = RgbColor(128, 0, 255);
8
9 // RGBA 颜色，元组的三个元素分别对应: (R, G, B, A)
10 struct RgbaColor(u8, u8, u8, f32);
11 let c1 = RgbaColor(128, 0, 255, 0.4);

```

单元式结构体

单元式结构体没有任何属性，`struct` 关键字后面跟随结构体的名字即可，无需花括号。

```
1 struct AlwaysEqual;
```

访问结构体的属性

很容易想到，访问名值对结构体的属性就是使用 `.` 操作符，和 JavaScript 是一样的。但是，需要注意的是，还是可能存在所有权转移的场景的，例如：

```

1 // 以下代码无法通过编译
2 struct Student {
3     active: bool,
4     username: String,

```

```
5     email: String,
6     sign_in_count: u64,
7 }
8
9 fn main() {
10    let active = true;
11    let username = "John Smith".to_string();
12    let student = Student {
13        active,
14        username,
15        email: "j.smith@gmail.com".to_string(),
16        sign_in_count: 0,
17    };
18
19    let email = student.email;
20    println!("email = {}", student.email);
21 }
```

在行 19 的时候，`String` 类型的值 `j.smith@gmail.com` 的所有权从 `student.email` 转移到了变量 `email`，所以行 20 将无法成功编译。所以，如果你不是真的要把结构体的属性转移出来的话，应该使用 `&:` `let email = &student.email;` 来创建一个指向 `student.email` 的只读引用。

很容易想到，元组式结构体的属性访问采用 `.index` 的形式。如果元组内的元素数量较少的时候，还比较方便，如果元素数量较多的时候，采用下标方式访问元素，容易在编写代码的时候记不住哪个下标对应哪个含义的数据，所以我们还可以使用“解构”方式访问其属性：

```

1 // 平面上一个点的坐标: (x, y)
2 #[derive(Debug)]
3 struct Point(f32, f32);
4
5 fn main() {
6     let p = Point(1.0, 2.0);
7     // 使用下标访问元组内的数据
8     println!("point: ({}, {})", p.0, p.1);
9
10    // 或者, 直接将数据解构到一组变量
11    let Point(x, y) = p;
12    println!("point: ({}, {})", x, y);
13}

```

不仅仅元组式结构体可以使用解构的方式访问属性，对于名值对结构体也是可以采用解构方式访问其属性的。

结构体属性的可变性

在 JavaScript 中，我们可以自由的修改对象的属性值，例如：

```

1 // JavaScript 代码
2 const student = {
3     active: true,
4     username: "John Smith",
5     email: "j.smith@gmail.com",
6     signInCount: 10
7 };
8
9 student.active = false;

```

在上面的代码中，虽然 `student` 是使用 `const` 关键字声明的，但是它的含义是 `student` 变量指向了这个对象之后，就不允许再指向其他的变量了。但是 `student` 对象中的属性还是可以自由修改的。

在 Rust 中，情况就不一样了。如果一个结构体实例不是可变的，那么结构体的属性值也是不可变的。

```

1 // 以下代码无法通过编译
2 fn main() {
3     let student = Student {
4         active: true,
5         username: "John Smith".to_string(),
6         email: "j.smith@gmail.com".to_string(),
7         sign_in_count: 0,
8     };
9
10    student.sign_in_count = 1;
11 }
```

所以，如果我们需要修改结构体属性的值，那么该结构体的所有者必须是使用 `let mut` 语句定义。

给结构体增加方法

只有数据的结构体是没有灵魂的，所以我们得给结构体增加一些方法。

“Algorithm + Data Structure = Programs”

-尼古拉斯·沃斯，瑞士计算机科学家，Euler 语言的发明者之一，1984 年获得图灵奖时的发言

“Properties + Methods = Objects”

- 佚名

假定我们要编写一个 `Rectangle` 类，这个类有宽高属性，可以计算面积和周长，可以缩放自己的宽高。如果用 JavaScript 来编写，类似下面的代码：

```

1 // JavaScript 代码
2 class Rectangle {
```

```
3     #width;
4     #height;
5
6     /**
7      * Construct a new square
8      * @param {number} size
9      */
10    static square(size) {
11        return new Rectangle(size);
12    }
13
14    /**
15     * Construct a new rectangle
16     * @param {number} width
17     * @param {number} height
18     */
19    constructor(width, height) {
20        this.#width = width;
21        this.#height = height;
22    }
23
24    get width() {
25        return this.#width;
26    }
27
28    set width(val) {
29        this.#width = val;
30    }
31
32    get height() {
33        return this.#height;
34    }
35
36    set height(val) {
37        this.#height = val;
38    }
39
40    /**
41     * Calculate the rectangle perimeter
```

```
42     * @returns {number} The perimeter
43     */
44     perimeter() {
45         return (this.#width + this.#height) * 2;
46     }
47
48     /**
49      * Calculate the rectangle area
50      * @returns {number} The area
51      */
52     area() {
53         return this.#width * this.#height;
54     }
55
56     /**
57      * Scale the rectangle
58      * @param {number} widthScale
59      * @param {number} heightScale
60      * @returns {void}
61      */
62     scale(widthScale, heightScale) {
63         this.#width = this.#width * widthScale;
64         this.#height = this.#height * heightScale;
65     }
66 }
67
68 const square = Rectangle.square(100);
69
70 const rect = new Rectangle(100, 100);
71 console.log(`rect perimeter is: ${rect.perimeter()}`);
72 console.log(`rect area is: ${rect.area()}`);
73 rect.scale(2, 2);
74 console.log(`after scale 2: rect perimeter is:
75 ${rect.perimeter()}`);
76 console.log(`after scale 2: rect area is: ${rect.area()}`);
```

对应的 Rust 代码如下：

```
1 struct Rectangle {
2     width: f32,
3     height: f32,
4 }
5
6 impl Rectangle {
7     fn square(size: f32) -> Self {
8         Self {
9             width: size,
10            height: size
11        }
12    }
13
14    fn new(width: f32, height: f32) -> Self {
15        Self {
16            width,
17            height
18        }
19    }
20
21    fn perimeter(&self) -> f32 {
22        (self.width + self.height) * 2.0f32
23    }
24
25    fn area(&self) -> f32 {
26        self.width * self.height
27    }
28
29    fn scale(&mut self, width_scale: f32, height_scale: f32) {
30        self.width = self.width * width_scale;
31        self.height = self.height * height_scale;
32    }
33 }
34
35 fn main() {
36     let square = Rectangle::square(100.0f32);
37
38     let mut rect = Rectangle::new(100.0f32, 100.0f32);
39     println!("rect perimeter is: {}", rect.perimeter());
```

```

40     println!("rect area is: {}", rect.area());
41     rect.scale(2.0, 2.0);
42     println!("after scale 2: rect perimeter is: {}", rect.perimeter());
43     println!("after scale 2: rect area is: {}", rect.area());
44 }
```

在 Rust 中，给结构体增加方法是使用 `impl` 块（Implementation），并且，一个结构体可以有多个 `impl` 块：

```

1 impl StructName {
2     // 这里有一些函数
3 }
4
5 impl StructName {
6     // 这里可以再增加一些函数
7 }
```

静态方法与实例方法

我们看到 `Rectangle::square` 和 `Rectangle::new` 这两个函数都属于关联函数（Associated Function），对应在 JavaScript 中为静态函数，这样的函数只和类型本身有关，和类型具体的实例无关。在 JavaScript 中，我们通过 `static` 关键字表明一个函数是静态函数，但是在 Rust 的代码中并未发现类似的标记，Rust 是如何区分静态函数和实例函数的呢？

其答案就在于函数参数。在 `impl` 块中给结构体增加方法的时候，如果方法函数的参数列表第一个参数是 `self`, `&self` 或者 `&mut self`，那么这个函数就将被视为是实例方法，在实例方法中，由于 `self/&self/&mut self` 被当作参数传入了，所以我们可以使用实例上的属性。这一点和 JavaScript 的类定义不同，在 JavaScript 的类定义中，我们在实例方法中可以使用 `this` 关键字，无需通过函数参数传入。

虽然我们看到 `area(&self)` 这样的语法感觉很奇怪，因为 `&self` 没有像其他参数一样表明类型。其实这个可以称为一个语法糖。在 `impl` 块中，一个方法的第一个参数：

- `self` 是 `self: Self` 的简写
- `&self` 是 `self: &Self` 的简写
- `&mut self` 是 `self: &mut Self` 的简写

同样的，在实例方法中的 `self/&self/&mut self` 参数，也要遵循 Rust 所有权与引用的规则。例如上面实例中的 `scale` 方法，由于这个方法需要修改实例自己的 `width` 和 `height` 属性，所以第一个参数是 `&mut self`。那么在调用这个方法的时候，发生调用的实例自身也得是可变的，否则就无法获取可变引用。这就是为什么我们在行 38 使用 `mut` 关键字：`let mut rect = Rectangle::new(100.0f32, 100.0f32);`。

`Self` 关键字

我们看到，在上面的例子中，有一个静态方法用来生成正方形：`square`，一个静态方法 `new` 用来生成四边形。这两个静态方法返回的都是 `Rectangle` 类型的值，但是方法声明上却使用了 `Self` 关键字，并且用来生成实例的代码，也是用的 `Self` 关键字，而不是具体的类型 `Rectangle`。

当你在一个结构体、枚举或者实现块中定义方法时，可以用 `Self` 来引用当前的类型。这在实现关联函数（例如构造函数模式）或者方法时非常有用。

在方法中，`Self` 可以用作返回类型。这在实现一些链式调用的 API 时非常有用。

还有一种用法就是和 `trait` 搭配使用，后续会讲解到。

`new` 不是关键字

在 JavaScript 中，`new` 是一个关键字，用来生成某个类的实例。但是在 Rust 中，`new` 不是关键字，也不是保留字，所以我们可以把方法名字叫做 `new`，甚至也可以用 `new` 来作为变量名。另外，使用 `new` 作为静态函数的名称，用来生成当前类型的实例，是一个常见的、约定俗成的用法。

最后，Rust 中的结构体不支持继承。实际上，业内也有一个相对一致的结论：开发应用的时候，组合优先于继承。因为类的继承关系的设计是非常挑战的，所以在一些基础 SDK 中我们经常可以看到继承关系。但是在我们开发应用的过程中，继承是比较少见的，更多的时候是在组合。

回顾

1. Rust 中有三种类型的结构体：名值对结构体、元组式结构体以及单元式结构体
2. 要改变结构体属性值，那么结构体值本身也得是可变的
3. 使用 `impl` 为结构体增加方法
4. 结构体的方法参数中，第一个参数不是 `self/&self/&mut self` 的，叫做关联函数（静态方法）
5. 结构体的方法参数中，第一个参数是 `self/&self/&mut self` 的，叫做实例方法

