Rust 语法课一

河南科技大学-徐堃元

2024/4/10

课程内容与安排

- Rust 课程共三周，九节课，三节语法课，三节答疑课，三节算法课
- 配套实验：训练营110题版 Rustlings
- 晋级要求：完成训练营110题版 Rustlings 的所有题目

学习建议与时间安排

- 完成全部 Rustlings 题目一般需要五到十个小时，基础较好的同学一般三四个小时可全部完成
- 建议配合官方教材进行学习，课程时间有限无法面面俱到，部分内容可能需要自行拓展
- 学习过程中可随时在微信群内提问，讲师和助教会积极解答大家的问题
- 每节语法课对应三十道左右的 Rustlings 练习题，大家可以参考课程安排规划实验进度

本节课程内容

- Rust 与 Rustlings 的配置与使用
- Rust 基本编程元素
- Rust 所有权机制与 slice 类型
- 结构体与常用集合

《Rust 程序设计语言》：

https://kaisery.github.io/trpl-zh-cn/

《通过例子学 Rust》：

https://rustwiki.org/zh-CN/rust-by-example/

Rust 语言中文社区：

https://rustcc.cn/

Rust 与 Rustlings 的配置与使用

Rust 语言官网：

https://www.rust-lang.org/zh-CN/

rCore-Tutorial-Book 的环境配置教程：

https://rcore-os.cn/rCore-Tutorial-Book-v3/chapter0/5setup-devel-env.html

Rustlings 仓库：

https://classroom.github.com/a/-WftLmvV

Rust 自动安装脚本指令：

```
curl https://sh.rustup.rs -sSf | sh
```

Rust 基本编程元素

从一个 Hello world！ 开始：

```
fn main(){

    println!("Hello, world!");

}
```

运行：

```
$  cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
     Running `target/debug/first`
Hello, world!
```

变量与可变性

Rust 中，使用关键字 let 声明变量。

```
let x = 5;
```

在 Rust 中，**变量默认是不可改变的（immutable）**，当变量不可变时，一旦值被绑定到一个名称上，这个值就不能被改变，这样的设计有助于防止意外的数据修改和并发问题。若要使得变量可变，需要在声明时使用 mut 关键字。

```
let mut x = 5;
```

Rust 中常量总是不可变，声明常量使用 const 关键字而不是 let。

常量只能被声明为常量表达式，而不可以是其他任何只能在程序运行时计算出的值。

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

基本数据类型-原生类型（primitives）

标量类型

- 有符号整数（signed integers）：`i8`、`i16`、`i32`、`i64`、`i128` 和 `isize`（指针宽度）
- 无符号整数（unsigned integers）：`u8`、`u16`、`u32`、`u64`、`u128` 和 `usize`（指针宽度）
- 浮点数（floating point）：`f32`、`f64`
- `char`（字符）：单个 Unicode 字符，如 `'a'`，`'α'` 和 `'∞'`（每个都是 4 字节）
- `bool`（布尔型）：只能是 `true` 或 `false`
- 单元类型（unit type）：`()`。其唯一可能的值就是 `()` 这个空元组

复合类型

- 数组（array）：如 `[1, 2, 3]`
- 元组（tuple）：如 `(1, true)`

元组（tuple）

元组是一个将多个其他类型的值组合进一个复合类型的主要方式。元组长度固定：一旦声明，其长度不会增大或缩小。

我们使用包含在圆括号中的逗号分隔的值列表来创建一个元组。元组中的每一个位置都有一个类型，而且这些不同值的类型也不必是相同的。

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

`tup` 变量绑定到整个元组上，因为元组是一个单独的复合元素。为了从元组中获取单个值，可以使用点号（`.`）后跟值的索引来直接访问它们，也可以使用模式匹配（pattern matching）来解构（destructure）元组值

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {y}");
}
```

数组（array）

与元组不同，数组中的每个元素的类型必须相同。Rust 中的数组与一些其他语言中的数组不同，Rust 中的数组长度是固定的。

我们将数组的值写成在方括号内，用逗号分隔：

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

可以像这样编写数组的类型：在方括号中包含每个元素的类型，后跟分号，再后跟数组元素的数量。

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

还可以通过在方括号中指定初始值加分号再加元素个数的方式来创建一个每个元素都为相同值的数组：

```
let a = [3; 5]; // a = [3, 3, 3, 3, 3]
```

数组是可以在栈 (stack) 上分配的已知固定大小的单个内存块。可以使用索引来访问数组的元素

函数

函数在 Rust 代码中非常普遍。你已经见过语言中最重要的函数之一：`main` 函数，它是很多程序的入口点。你也见过 `fn` 关键字，它用来声明新函数。

Rust 代码中的函数和变量名使用 *snake case* 规范风格。在 snake case 中，所有字母都是小写并使用下划线分隔单词。这是一个包含函数定义示例的程序：

```
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

if 表达式

`if` 表达式允许根据条件执行不同的代码分支。你提供一个条件并表示 “如果条件满足，运行这段代码；如果条件不满足，不运行这段代码。"

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

因为 `if` 是一个表达式，我们可以在 `let` 语句的右侧使用它

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {number}");
}
```

loop 循环

`loop` 关键字告诉 Rust 一遍又一遍地执行一段代码直到你明确要求停止。

```
fn main() {
    loop {
        println!("again!");
    }
}
```

Rust 提供了一种从代码中跳出循环的方法。可以使用 `break` 关键字来告诉程序何时停止循环。

循环中的 `continue` 关键字告诉程序跳过这个循环迭代中的任何剩余代码，并转到下一个迭代。

从循环返回值

```
fn main() {
    let mut counter = 0;
    let result = loop {
        counter += 1;
        if counter == 10 {
            break counter * 2;
        }
    };
    println!("The result is {result}");
```

循环标签

如果存在嵌套循环，`break` 和 `continue` 应用于此时最内层的循环。你可以选择在一个循环上指定一个**循环标签**（*loop label*），然后将标签与 `break` 或 `continue` 一起使用，使这些关键字应用于已标记的循环而不是最内层的循环。

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }
        count += 1;
    }
    println!("End count = {count}");
}
```

while 条件循环与 for 循环


```rust
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number}!");

        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

```rust
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }
}
```

所有权概念

所有权规则

- Rust 中的每一个值都有一个所有者（owner）
- 值在任一时刻有且只有一个所有者
- 当所有者（变量）离开作用域，这个值将被丢弃

```
let x = 123;
let y = x;
```

变量作用域与变量隐藏

变量 `s` 绑定到了一个字符串字面值，这个字符串值是硬编码进程序代码中的。这个变量从声明的点开始直到当前 **作用域** 结束时都是有效的。

```
 {                      // s 在这里无效，它尚未声明
        let s = "hello";   // 从此处起，s 是有效的

        // 使用 s
    }                      // 此作用域已结束，s 不再有效
```

我们可以定义一个与之前变量同名的新变量。Rustacean 们称之为第一个变量被第二个 **隐藏（Shadowing）** 了

```
fn main() {
    let x = 5;
    let x = x + 1;
    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }
    println!("The value of x is: {x}");
}
```

String 类型

String literal（字符串字面值）

```
let string_literal = "hello world";
```

String 类型的基本操作

创建：

- 直接创建

```
let mut s = String::new();
```

- 从字符串字面值创建

```
let mut s = string_literal.to_string();
let mut s = String::from("hello world");
```

扩展

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // 注意 s1 被移动了，不能继续使用
```

函数参数的所有权

直接将值传递给函数，会将所有权转移

```
fn main() {
    let s = String::from("hello");  // s 进入作用域
    takes_ownership(s);             // s 的值移动到函数里 ...
                                    // ... 所以到这里不再有效
    let x = 5;                      // x 进入作用域
    makes_copy(x);                  // x 应该移动函数里，
                                    // 但 i32 是 Copy 的，
                                    // 所以在后面可继续使用 x
} // 这里，x 先移出了作用域，然后是 s。但因为 s 的值已被移走，
  // 没有特殊之处

fn takes_ownership(some_string: String) { // some_string 进入作用域
    println!("{}", some_string);
} // 这里，some_string 移出作用域并调用 `drop` 方法。
  // 占用的内存被释放

fn makes_copy(some_integer: i32) { // some_integer 进入作用域
    println!("{}", some_integer);
} // 这里，some_integer 移出作用域。没有特殊之处
```

函数返回值

返回值也可以转移所有权

```
fn main() {
    let s1 = gives_ownership();         // gives_ownership 将返回值
                                        // 转移给 s1
    let s2 = String::from("hello");     // s2 进入作用域
    let s3 = takes_and_gives_back(s2);  // s2 被移动到
                                        // takes_and_gives_back 中,
                                        // 它也将返回值移给 s3
} // 这里, s3 移出作用域并被丢弃。s2 也移出作用域, 但已被移走,
  // 所以什么也不会发生。s1 离开作用域并被丢弃
fn gives_ownership() -> String {             // gives_ownership 会将
                                             // 返回值移动给
                                             // 调用它的函数
    let some_string = String::from("yours"); // some_string 进入作用域。

    some_string                              // 返回 some_string
                                             // 并移出给调用的函数
}
// takes_and_gives_back 将传入字符串并返回该值
fn takes_and_gives_back(a_string: String) -> String { // a_string 进入作用域
    a_string  // 返回 a_string 并移出给调用的函数
}
```

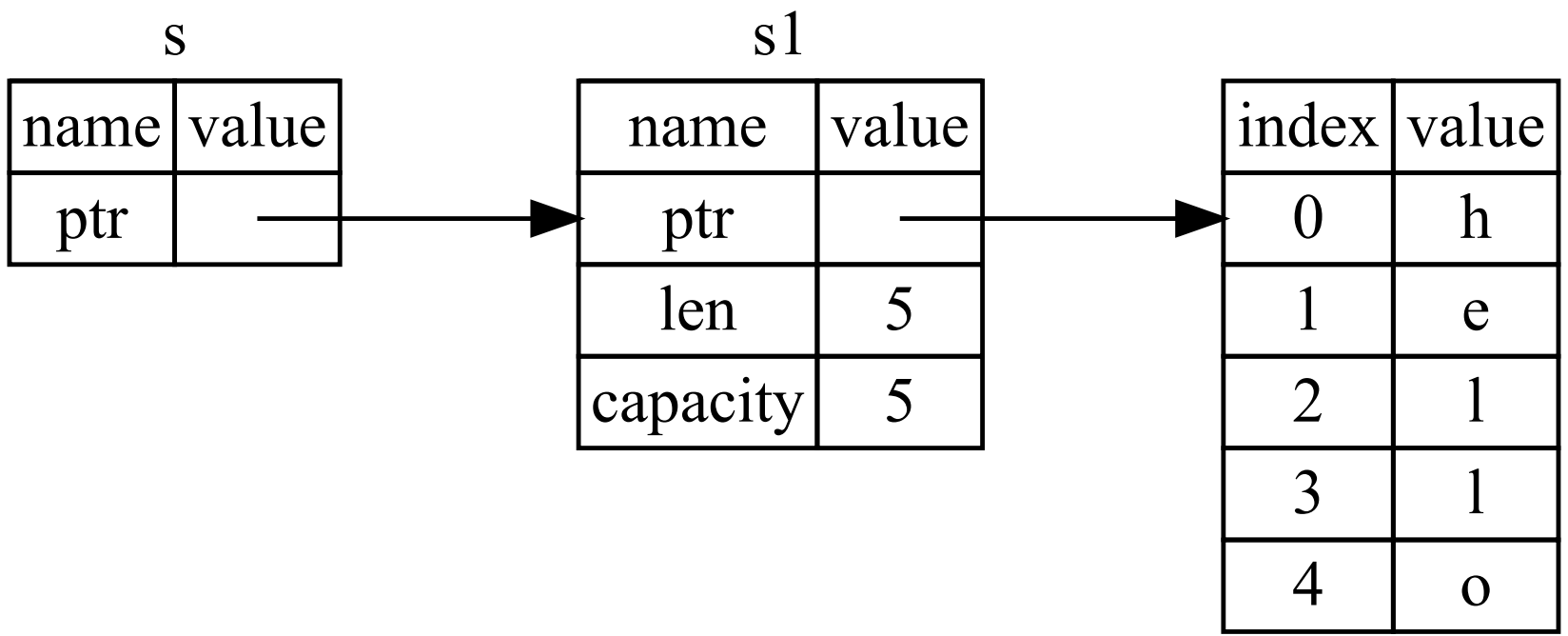
引用与借用

引用（*reference*）像一个指针，因为它是一个地址，我们可以由此访问储存于该地址的属于其他变量的数据。与指针不同，引用确保指向某个特定类型的有效值

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize { // s 是 String 的引用
    s.len()
} // 这里，s 离开了作用域。但因为它并不拥有引用值的所有权，
  // 所以什么也不会发生
```

我们将创建一个引用的行为称为 **借用**（*borrowing*）。

引用与借用


s
name
value
ptr
s1
name
value
ptr
len
5
capacity
5
index
value
0
h
1
e
2
l
3
l
4
o

可变引用

默认的，引用也同样不能改变对象的值，因此需要可变引用

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

不变引用可以同时有多个（可以有多个读者）

不变引用存在的同时不能有可变引用（读者写者不能共存）

不能同时有多个可变引用（多个写者也不能共存）

String slice 类型

slice 允许你引用集合中一段连续的元素序列，而不用引用整个集合。slice 是一类引用，所以它没有所有权。

```
let a = [1, 2, 3, 4, 5];
let slice = &a[1..3];
```

该类型是 &i32

String Slice类型的写法：&str

```
let s = String::from("hello world");
let hello = &s[0..5];
let world = &s[6..11];
```

自定义类型-结构体

定义结构体，需要使用 `struct` 关键字并为整个结构体提供一个名字。结构体的名字需要描述它所组合的数据的意义。接着，在大括号中，定义每一部分数据的名字和类型，我们称为 **字段**（*field*）。

一旦定义了结构体后，为了使用它，通过为每个字段指定具体值来创建这个结构体的 **实例**。创建一个实例需要以结构体的名字开头，接着在大括号中使用 `key: value` 键 - 值对的形式提供字段，其中 key 是字段的名字，value 是需要存储在字段中的数据值。实例中字段的顺序不需要和它们在结构体中声明的顺序一致。

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}

fn main() {
    let user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };
}
```

自定义类型-结构体

为了从结构体中获取某个特定的值，可以使用点号。举个例子，想要用户的邮箱地址，可以用 `user1.email` 。如果结构体的实例是可变的，我们可以使用点号并为对应的字段赋值。

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}

fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

自定义类型-元组结构体

元组结构体有着结构体名称提供的含义，但没有具体的字段名，只有字段的类型。当你想给整个元组取一个名字，并使元组成为与其他元组不同的类型时，元组结构体是很有用的，这时像常规结构体那样为每个字段命名就显得多余和形式化了。

要定义元组结构体，以 `struct` 关键字和结构体名开头并后跟元组中的类型。例如，下面是两个分别叫做 `Color` 和 `Point` 元组结构体的定义和用法：

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

注意 `black` 和 `origin` 值的类型不同，因为它们是不同的元组结构体的实例。你定义的每一个结构体有其自己的类型，即使结构体中的字段可能有着相同的类型。例如，一个获取 `Color` 类型参数的函数不能接受 `Point` 作为参数，即便这两个类型都由三个 `i32` 值组成。

其它类型-Vector

`Vec<T>`，也被称为 *vector*。vector 允许我们在一个单独的数据结构中储存多于一个的值，它在内存中彼此相邻地排列所有的值。vector 只能储存相同类型的值。

为了创建一个新的空 vector，可以调用 `Vec::new` 函数

```
let v: Vec<i32> = Vec::new();
```

这里我们增加了一个类型注解。因为没有向这个 vector 中插入任何值，Rust 并不知道我们想要储存什么类型的元素。

Rust 提供了 `vec!` 宏，这个宏会根据我们提供的值来创建一个新的 vector

```
let v = vec![1, 2, 3];
```

对于新建一个 vector 并向其增加元素，可以使用 `push` 方法

```
let mut v = Vec::new();

    v.push(5);
```

其它类型-hashmap

`HashMap<K, V>` 类型储存了一个键类型 `K` 对应一个值类型 `V` 的映射。它通过一个 **哈希函数**（*hashing function*）来实现映射，决定如何将键和值放入内存中。

可以使用 `new` 创建一个空的 `HashMap`，并使用 `insert` 增加元素。

```
use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);
```

可以通过 `get` 方法并提供对应的键来从哈希 map 中获取值

```
use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    let team_name = String::from("Blue");
    let score = scores.get(&team_name).copied().unwrap_or(0);
```

课后练习

Exercise	Book Chapter
variables	§3.1
functions	§3.3
if	§3.5
primitive_types	§3.2, §4.3
vecs	§8.1
move_semantics	§4.1, §4.2
structs	§5.1, §5.3
strings	§8.2