20250127-具体工作

在学习的第四个周期里，通过阅读《Rust程序设计》和《Rust圣经》，我学习了Rust语言的基础语法知识，并完成了strings、modules和hashmaps部分的练习题。在学习的过程中，我发现Rust语言有很多抽象的，离应用较远的概念。学习速度缓慢，归纳总结第四周期知识点和练习题目如下：

1. 字符串类型

Rust中有两种主要的字符串类型：

String：一个可增长、可变的字符串类型，存储在堆上。

&str：字符串切片，通常是不可变的，通常作为字符串的引用。

1.1 字符串字面量

字符串字面量是不可变的&str类型，存储在程序的只读内存中。

let s: &str = "Hello, Rust!";

1.2 创建String

String可以通过String::new()或String::from()创建。

let s1 = String::new(); // 创建一个空的String

let s2 = String::from("Hello, Rust!"); // 从字符串字面量创建

let s3 = "Hello, Rust!".to\_string(); // 使用to\_string方法

2. 字符串的不可变性

默认情况下，String是不可变的。如果需要修改字符串，必须将其声明为mut。

let mut s = String::from("Hello");

s.push\_str(", Rust!"); // 在字符串后追加内容

println!("{}", s); // 输出：Hello, Rust!

3. 字符串的所有权和借用

字符串的所有权和借用规则遵循Rust的一般规则。

3.1 所有权转移（Move）

当一个String被赋值给另一个变量时，所有权会转移。

let s1 = String::from("Hello");

let s2 = s1; // s1的所有权被move到s2

// println!("{}", s1); // 错误：s1的值已经被move

println!("{}", s2); // 正确

3.2 借用（Borrowing）

可以通过引用访问字符串，而不转移所有权。

let s = String::from("Hello");

let len = calculate\_length(&s); // 借用s

fn calculate\_length(s: &String) -> usize {

s.len()

}

4. 字符串切片（Slices）

字符串切片&str是字符串的一部分，通常用于表示字符串的子段。

let s = String::from("Hello, Rust!");

let hello = &s[0..5]; // "Hello"

let rust = &s[7..12]; // "Rust!"

println!("{} {}", hello, rust);

4.1 字符串字面量作为切片

字符串字面量实际上是字符串切片&str。

let s: &str = "Hello, Rust!";

4.2 字符串切片的生命周期

字符串切片的生命周期不能超过其原始字符串。

fn first\_word(s: &str) -> &str {

let bytes = s.as\_bytes();

for (i, &item) in bytes.iter().enumerate() {

if item == b' ' {

return &s[0..i];

}

}

&s[..]

}

5. 字符串的迭代

Rust提供了多种方式来迭代字符串。

5.1 字符迭代

字符串可以通过.chars()方法按字符迭代。

let s = String::from("Hello, Rust!");

for c in s.chars() {

println!("{}", c);

}

5.2 字节迭代

字符串可以通过.bytes()方法按字节迭代。

let s = String::from("Hello, Rust!");

for b in s.bytes() {

println!("{}", b);

}

6. 字符串的修改

String是可变的，可以通过多种方法修改。

6.1 追加内容

可以通过push\_str或push方法追加内容。

let mut s = String::from("Hello");

s.push\_str(", Rust!"); // 追加字符串

s.push('!'); // 追加单个字符

println!("{}", s); // 输出：Hello, Rust!!

6.2 替换内容

可以通过replace方法替换内容。

let s = String::from("Hello, Rust!");

let new\_s = s.replace("Rust", "World");

println!("{}", new\_s); // 输出：Hello, World!

7. 字符串的比较

字符串可以通过==和!=进行比较，比较的是内容。

let s1 = String::from("Hello");

let s2 = String::from("Hello");

let s3 = String::from("World");

println!("{}", s1 == s2); // true

println!("{}", s1 == s3); // false

8. 字符串的性能

String是可增长的，底层通过动态内存分配实现。

字符串的大小可能会随着内容的增加而变化，因此需要合理管理内存分配。

9. Unicode支持

Rust的字符串支持Unicode，因此可以存储多种语言的字符。

let s = String::from("你好，世界！");

println!("{}", s); // 输出：你好，世界！

9.1 字符与字节

由于Unicode字符可能占用多个字节，因此Rust中的字符（char）和字节（u8）是不同的概念。

let s = "你好";

println!("Length in bytes: {}", s.len()); // 输出：6

println!("Number of chars: {}", s.chars().count()); // 输出：2

10. 总结

Rust的字符串设计注重安全性和灵活性：

String是可变的、可增长的字符串类型，适合动态修改。

&str是不可变的字符串切片，适合表示字符串的一部分。

字符串的所有权和借用规则遵循Rust的一般规则。

Rust支持Unicode，可以处理多种语言的文本。

掌握字符串的使用是编写Rust程序的重要基础，尤其是在处理文本数据时。

### **1. 什么是**HashMap**？**

HashMap是Rust标准库std::collections模块中的一个数据结构，用于存储键值对（Key-Value Pair）。每个键必须是唯一的，而值可以通过对应的键访问。

### **2. 使用**HashMap**的原因**

**键值对存储**：适合表示数据之间的映射关系，例如存储学生的姓名和成绩。

**快速访问**：底层通过哈希函数实现，查找和更新操作效率高。

**灵活的键和值类型**：支持自定义类型作为键和值。

**3. 创建**HashMap

#### **3.1 使用**HashMap::new

可以通过HashMap::new创建一个空的HashMap，然后使用insert方法添加键值对。

use std::collections::HashMap;

fn main() {

let mut scores = HashMap::new();

scores.insert("Alice", 90);

scores.insert("Bob", 80);

println!("{:?}", scores); // 输出：{"Alice": 90, "Bob": 80}

}

3.2 使用from方法

可以通过from方法和数组初始化一个HashMap。

use std::collections::HashMap;

fn main() {

let scores: HashMap<&str, i32> = [("Alice", 90), ("Bob", 80)].into\_iter().collect();

println!("{:?}", scores); // 输出：{"Alice": 90, "Bob": 80}

}

4. 访问HashMap中的值

4.1 使用get方法

get方法通过键返回一个Option<&V>类型，如果键不存在则返回None。

use std::collections::HashMap;

fn main() {

let mut scores = HashMap::new();

scores.insert("Alice", 90);

match scores.get("Alice") {

Some(&score) => println!("Alice's score is {}", score), // 输出：Alice's score is 90

None => println!("No score found for Alice"),

}

}

4.2 直接索引

可以直接使用[]访问值，但如果键不存在，程序会panic。

fn main() {

let mut scores = HashMap::new();

scores.insert("Alice", 90);

println!("{}", scores["Alice"]); // 输出：90

}

5. 修改HashMap

5.1 添加或更新键值对

使用insert方法可以添加新的键值对，或者更新已存在的键的值。

use std::collections::HashMap;

fn main() {

let mut scores = HashMap::new();

scores.insert("Alice", 90);

scores.insert("Alice", 95); // 更新值

println!("{:?}", scores); // 输出：{"Alice": 95}

}

#### ****5.2 条件更新****

使用entry和or\_insert方法可以条件性地更新值。

use std::collections::HashMap;

fn main() {

let mut scores = HashMap::new();

scores.insert("Alice", 90);

scores.entry("Alice").or\_insert(95); // 不会更新，因为"Alice"已存在

scores.entry("Bob").or\_insert(85); // 添加"Bob"

println!("{:?}", scores); // 输出：{"Alice": 90, "Bob": 85}

}

5.3 删除键值对

使用remove方法删除键值对。

use std::collections::HashMap;

fn main() {

let mut scores = HashMap::new();

scores.insert("Alice", 90);

scores.remove("Alice");

println!("{:?}", scores); // 输出：{}

}

6. 遍历HashMap

可以使用iter方法遍历HashMap中的键值对。

use std::collections::HashMap;

fn main() {

let map = HashMap::from([

("a", 1),

("b", 2),

("c", 3),

]);

for (key, value) in &map {

println!("Key: {}, Value: {}", key, value);

}

}

7. 性能注意事项

在当前实现中，遍历HashMap的时间复杂度为O(capacity)，而不是O(len)，因为内部会访问空桶。

HashMap的容量可能大于实际存储的元素数量，以避免重新分配。

8. 总结

HashMap是Rust中用于存储键值对的强大工具，具有以下特点：

键必须唯一，值可以通过键快速访问。

支持灵活的键和值类型。

提供丰富的操作方法，包括插入、更新、删除和遍历。

掌握HashMap的使用可以极大地提高处理键值对数据的效率。