20250120-具体工作

在学习的第三个周期里，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structs和enums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，归纳总结第三周期知识点和练习题目如下：

struct（结构体）是用于创建自定义数据类型的核心工具之一。它允许你将多个值组合在一起，形成一个逻辑上的单元。以下是关于Rust中structs部分的知识点总结：

### **1. 定义结构体**

结构体通过struct关键字定义，可以包含多个字段，每个字段都有自己的名称和类型。

struct User {

username: String,

email: String,

sign\_in\_count: u64,

active: bool,

}

**字段类型：**

结构体的字段可以是任意类型，包括基本类型（如i32、String）、其他结构体或枚举等。

字段的类型决定了结构体的大小和内存布局。

结构体实例可以通过字段初始化语法来创建。

let user1 = User {

username: String::from("Alice"),

email: String::from("alice@example.com"),

sign\_in\_count: 1,

active: true,

};

使用字段初始化简写

如果字段名与变量名相同，可以使用字段初始化简写语法：

let username = String::from("Bob");

let email = String::from("bob@example.com");

let user2 = User {

username,

email,

sign\_in\_count: 0,

active: true,

};

访问结构体字段

可以通过点操作符（.）访问结构体的字段。

println!("User: {}", user1.username);

println!("Email: {}", user1.email);

结构体的不可变性

默认情况下，结构体实例是不可变的。如果需要修改字段，必须将结构体声明为mut。

let mut user3 = User {

username: String::from("Charlie"),

email: String::from("charlie@example.com"),

sign\_in\_count: 0,

active: true,

};

user3.email = String::from("new\_charlie@example.com");

结构体方法

可以通过在结构体定义中实现impl块来为结构体添加方法。

impl User {

// 关联函数（类似于构造函数）

fn new(username: String, email: String) -> User {

User {

username,

email,

sign\_in\_count: 0,

active: true,

}

}

// 实例方法

fn change\_email(&mut self, new\_email: String) {

self.email = new\_email;

}

fn print\_info(&self) {

println!("User: {}, Email: {}", self.username, self.email);

}

}

let mut user4 = User::new(String::from("David"), String::from("david@example.com"));

user4.change\_email(String::from("new\_david@example.com"));

user4.print\_info();

self关键字：在实例方法中，self表示结构体实例本身。

可以使用&self（不可变引用）、&mut self（可变引用）或self（消费实例）。

元组结构体（Tuple Structs）：元组结构体是一种特殊的结构体，字段没有名称，只有类型。

struct Color(i32, i32, i32); // RGB

struct Point(i32, i32);

let black = Color(0, 0, 0);

let origin = Point(0, 0);

println!("Color: ({}, {}, {})", black.0, black.1, black.2);

println!("Point: ({}, {})", origin.0, origin.1);

单元结构体（Unit Structs）

单元结构体没有字段，通常用于实现trait或作为标记类型。

struct AlwaysEqual;

impl AlwaysEqual {

fn print\_message() {

println!("This is always equal!");

}

}

AlwaysEqual::print\_message();

结构体的可见性

默认情况下，结构体的字段是私有的（private）。

可以通过pub关键字将结构体或字段声明为公共的（public）。

pub struct User {

pub username: String,

email: String, // 私有字段

pub sign\_in\_count: u64,

pub active: bool,

}

let user = User {

username: String::from("Alice"),

email: String::from("alice@example.com"),

sign\_in\_count: 1,

active: true,

};

println!("Username: {}", user.username); // 可访问

// println!("Email: {}", user.email); // 错误：字段是私有的使用结构体实现封装

结构体可以通过方法和私有字段实现封装，隐藏内部实现细节。

struct User {

username: String,

email: String,

sign\_in\_count: u64,

active: bool,

}

impl User {

fn new(username: String, email: String) -> User {

User {

username,

email,

sign\_in\_count: 0,

active: true,

}

}

fn change\_email(&mut self, new\_email: String) {

self.email = new\_email;

}

fn print\_info(&self) {

println!("User: {}, Email: {}", self.username, self.email);

}

}

let mut user = User::new(String::from("Alice"), String::from("alice@example.com"));

user.change\_email(String::from("new\_alice@example.com"));

user.print\_info();

使用derive自动生成方法

Rust允许通过derive属性为结构体自动生成一些标准方法，如Debug、PartialEq、Clone等。

#[derive(Debug, PartialEq, Clone)]

struct User {

username: String,

email: String,

}

let user1 = User {

username: String::from("Alice"),

email: String::from("alice@example.com"),

};

let user2 = user1.clone();

println!("{:?}", user1); // 使用Debug打印

println!("{:?}", user2);

assert\_eq!(user1, user2); // 使用PartialEq比较

结构体可以通过字段嵌套其他结构体，实现组合关系。

struct Address {

street: String,

city: String,

}

struct User {

username: String,

email: String,

address: Address,

}

let address = Address {

street: String::from("123 Main St"),

city: String::from("Anytown"),

};

let user = User {

username: String::from("Alice"),

email: String::from("alice@example.com"),

address,

};

println!("User: {}, Address: {}, {}", user.username, user.address.street, user.address.city);

结构体是Rust中用于组织数据的核心工具，具有以下特点：1.可以通过字段存储数据。

2.支持方法和关联函数，实现面向对象的编程风格。3.可以通过impl块为结构体添加行为。

4.支持封装、组合和继承（通过组合实现）。5.可以通过derive自动生成标准方法。

6.理解结构体的定义、使用和方法实现是掌握Rust语言的重要一步。结构体与枚举（enum）一起构成了Rust的核心数据类型。

1. 枚举的定义

枚举（enum）是一种自定义数据类型，用于表示一组可能的值（称为变体）。定义枚举时，使用enum关键字，后跟枚举名称和一组变体。

enum Direction {

North,

South,

East,

West,

}

2. 枚举的变体

枚举的变体可以是无数据的（类似于C语言中的枚举），也可以携带关联数据。

enum Message {

Text(String), // 携带一个String

Image(String, u32, u32), // 携带文件名和尺寸

Quit, // 无数据

}

3. 枚举与模式匹配

枚举与match表达式结合使用，可以实现强大的控制流。

let msg = Message::Text(String::from("Hello, Rust!"));

match msg {

Message::Text(content) => println!("Text Message: {}", content),

Message::Image(filename, width, height) => {

println!("Image File: {}, Dimensions: {}x{}", filename, width, height)

}

Message::Quit => println!("Quit Message"),

}

4. 枚举的方法

可以通过impl块为枚举实现方法。

impl Message {

fn print(&self) {

match self {

Message::Text(content) => println!("Text: {}", content),

Message::Image(filename, width, height) => {

println!("Image: {} ({}x{})", filename, width, height)

}

Message::Quit => println!("Quit"),

}

}

}

5. 枚举的类型安全性

枚举确保变量只能处于预定义的状态之一，从而避免非法值。

6. 枚举在结构体中的使用

枚举可以作为结构体的字段，用于表示状态。

struct User {

name: String,

status: Status,

}

enum Status {

Active,

Inactive,

}

let user = User {

name: String::from("Alice"),

status: Status::Active,

};

match user.status {

Status::Active => println!("{} is active", user.name),

Status::Inactive => println!("{} is inactive", user.name),

}

7. 枚举的常见应用场景

状态机：表示有限状态，如交通灯。

错误处理：定义自定义错误类型。

数据分类：对不同类型的数据进行分类。

8. 枚举的特性

紧凑表示：通过单一类型表示多种状态。

类型安全：防止非法状态或值。

灵活性：可以存储数据和行为。

9. 枚举的扩展

可以通过外部库（如strum）为枚举添加额外功能，例如迭代枚举变体。

use strum\_macros::EnumIter;

#[derive(EnumIter)]

enum Days {

Monday,

Tuesday,

Wednesday,

}

fn main() {

for day in Days::iter() {

println!("{:?}", day);

}

}

枚举是Rust语言中非常强大的工具，结合模式匹配和方法实现，可以用于清晰地建模复杂的数据结构和状态机。