

## 接下来你要做的（Rustlings 作业流程）

**1 进入你的作业仓库目录**（如果还没 clone，用 SSH/HTTPS clone 你 fork 的仓库）：

bash

📄 复制 🖋 编辑

```
git clone git@github.com:<你的用户名>/<作业仓库名>.git
cd <作业仓库名>
```

**2 安装“裁剪版 rustlings”**

bash

📄 复制 🖋 编辑

```
cargo install --force --path .
```

这是安装仓库自带的 rustlings 工具，和官方 rustlings 不一样。

**3 开始做题**

bash

📄 复制 🖋 编辑

```
# 练某一题
rustlings run 题目名

# 自动监视修改
rustlings watch

# 查看提示
rustlings hint 题目名

# 做完全部题目检查
rustlings verify
```

**4 提交并推送到 GitHub**

bash

📄 复制 🖋 编辑

```
git add .
git commit -m "update"
git push origin main
```

推送后 GitHub 会跑 CI，你能在仓库 Actions 页或作业提供的排行榜链接上看到分数。

好！直接用 VS Code 改题+开终端是最方便的，下面是完整步骤：

## 1. 用 VS Code 打开你的作业仓库

1. 打开 VS Code
2. 点菜单 **File** → **Open Folder...**
3. 选中你的 Rustlings 作业仓库文件夹（你 `git clone` 下来的那个目录）。

快捷方法：在终端用：

```
bash
```

📄 复制 ✎ 编辑

```
code .
```

就会直接在 VS Code 打开当前目录（前提是 `code` 命令已安装到 PATH）。

为什么必须要具体类型哇

已思考 13 秒 >

因为 **Rust 是静态类型 + 零开销抽象（zero-cost）** 的语言，编译器在**编译期**就要把所有泛型实例都“落到”具体类型上，才能：

### 1. 确定内存布局

`Vec<T>` 在堆上放一段**连续内存**。要分配多少字节、`push` 时怎么搬运元素，都要用到 `size_of::()`。如果 `T` 不确定，编译器就不知道每个元素多大，自然无法生成正确的内存操作。

### 2. 生成专用机器码（单态化 monomorphization）

像 `Vec<i32>`、`Vec<&str>`、`Vec<String>` 都会各自生成一份专用代码，避免运行时开销。这要求在编译阶段把 `T` 决定下来。

### 3. 保证类型安全

`Vec<T>` 要保证“同一容器里都是同一种类型”的不变量。否则在读取、排序、比较时都无法静态保证安全。

## 那为什么我有时不写类型也能编译？

因为 **类型推断**。例如：

```
rust
let mut v = Vec::new();
v.push(1); // ← 从这里推断出 T = i32
```

编译器可从后续用法推出 **T**。

## 如果我真的想放“不同类型”的东西呢？

静态 `Vec<T>` 不支持混装，但有几种“变通”方式：

- `enum` 枚举（最常用、最高效）：

```
rust
enum Item { Text(String), Count(i32) }
let mut v: Vec<Item> = vec![Item::Text("milk".into()), Item::Count(3)];
```

- 特征对象（动态分发）：

```
rust
trait Show { fn show(&self); }
let mut v: Vec<Box<dyn Show>> = Vec::new(); // 统一成“指针大小”，允许不同实现
```

- 通用值类型（如 `serde_json::Value`、`enum Any` 等）：牺牲静态类型检查换灵活性。

## 关于Struct

好的！我帮你用中文详细讲解 Rust 官方书籍 **The Rust Programming Language (Rust Book)** 第五章第一节《定义结构体 (Defining Structs)》的内容。这一节主要介绍了 Rust 中结构体 (`struct`) 是什么、如何定义、如何使用。以下是分段讲解：

### 1. 什么是结构体 (Struct) ？

Rust 的 **结构体 (struct)** 是一种自定义数据类型，允许你把相关的数据组合在一起，形成一个更有意义的整体。它有点像 C、C++ 或 Go 里的结构体，也像 Python 的类 (class) 中只有数据属性的版本。

- 用途：表示一个概念实体，它可以包含多个字段，每个字段有自己的名字和类型。
- 好处：数据是命名的、结构化的，便于理解和使用。

举例：如果我们想表示一个“用户（User）”，而不是用一堆孤立的变量，我们可以把它们组合成一个结构体。

## 2. 定义结构体

语法格式如下：

```
struct 结构体名 {  
    字段1: 类型1,  
    字段2: 类型2,  
    ...  
}
```

示例：

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

- `struct` 关键字：表示定义一个结构体。
- 每个字段都有 名字和类型。
- 结构体字段之间用逗号分隔。

注意：结构体名字通常是大写开头的 驼峰命名法（**CamelCase**）。

## 3. 创建结构体实例

定义好结构体后，你可以创建它的实例：

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

- 使用花括号 `{}` 给每个字段赋值。
- 必须给所有字段赋值，Rust 不支持可选跳过某些字段。

---

## 4. 访问和修改结构体字段

---

访问字段：用点号 `.`

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

修改字段（变量必须是 `mut` 可变的）：

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

---

## 5. 用函数返回结构体实例

---

我们可以写一个函数来构建结构体：

```
fn build_user(email: String, username: String) -> User {  
    User {  
        email,  
        username,  
        active: true,  
        sign_in_count: 1,  
    }  
}
```

这里有一个重要特性：字段初始化简写（field init shorthand）

- 如果函数参数名和结构体字段名相同，可以直接写 `email` 而不是 `email: email`。

## 6. 结构体更新语法 (struct update syntax)

有时候我们想基于一个现有实例创建一个新实例，但改变某些字段，可以用 `..`：

```
let user2 = User {  
    email: String::from("another@example.com"),  
    username: String::from("anotherusername"),  
    ..user1  
};
```

- 意思是： `user2` 的 `active` 和 `sign_in_count` 字段直接拷贝 `user1` 的值。
- 这个语法类似于“继承现有实例的剩余字段”。

注意：被拷贝的字段如果是非 `Copy` 类型（如 `String`），旧实例（`user1`）会失效，所有权转移。

## 7. 元组结构体 (Tuple Structs)

Rust 允许定义 没有字段名 的结构体，称为元组结构体，语法和元组类似：

```
struct Color(i32, i32, i32);  
struct Point(i32, i32, i32);  
  
let black = Color(0, 0, 0);  
let origin = Point(0, 0, 0);
```

- 和普通元组类似，但有 独立的类型名称，即使字段类型相同也不能混用。

## 8. 单元结构体 (Unit-like Structs)

如果结构体不需要保存任何数据，可以定义一个 没有字段的结构体：

```
struct AlwaysEqual;  
  
let subject = AlwaysEqual;
```

- 用于实现某些 trait 或者做标记。
- 类似于 C 中的“空 struct”。

---

# 小结

---

这一节介绍了 **Rust 结构体** 的核心概念：

1. 用 `struct` 定义结构体，字段是命名的数据。
2. 如何创建、访问、修改结构体实例。
3. 字段简写和结构体更新语法。
4. 元组结构体和单元结构体。

**Rust 中的结构体是组织数据的核心工具**，后续章节会介绍如何为结构体实现方法、如何和 `impl` 配合使用。

---

👉 你要不要我 顺带翻译并讲解下一节（**ch05-02 Example: Programming a Rectangle**）？那一节是用矩形的例子深入展示结构体的使用和方法定义，能帮你更好理解这一节。

另外，我可以给你画一个 **结构体概念图**来加深印象，要不要？