程序设计训练之 Rust 编程语言

2026 年第一次习题课

Rust 教学团队

清华大学计算机科学与技术系

2026 年 8 月 26 日

编译错误通常已经告诉了你修改方向

```
let x = 1;
x += 2;
```

```
cannot assign twice to immutable variable `x`
help: consider making this binding mutable: `mut x`
```

- 先看第一条错误，不要一次处理整页输出
- 注意错误指向的是变量、值，还是借用
- 修改后立即重新运行 `cargo check`

Rustlings 1：值和绑定要分清

```rust
let mut x = 1;                  // 可变绑定
let cat = ("Mimi", 3.5);        // 元组
let (name, age) = cat;          // 模式匹配
let numbers = (1, 2, 3);
let second = numbers.1;         // 元组下标

let array = [0; 5];
let slice = &array[..3];        // 借用其中三个元素
let mut data = Vec::new();
data.push(1);
```

数组长度固定；`Vec` 可以增长；切片只是对连续元素的借用。

Poly：先把输入读成正确的类型

```rust
use text_io::read;

let n: usize = read!();
let x: i64 = read!();
let mut coefficients: Vec<i64> = Vec::new();
for _ in 0..=n {
    coefficients.push(read!());
}
```

- 数组长度和下标通常使用 `usize`
- 题目指定 64 位有符号整数时使用 `i64`
- `0..=n` 包含右端点，一共有 `n + 1` 项

Poly: 霍纳法只需要一次遍历

系数按 $a_0, a_1, \ldots, a_n$ 给出：

$$f(x) = a_0 + x(a_1 + x(a_2 + \cdots + xa_n))$$

```rust
let mut result = 0_i64;
for &coefficient in coefficients.iter().rev() {
    result = result * x + coefficient;
}
```

每一步都保持相同的含义；不必重复计算 `x.pow(i)`。

Inverse：元组让函数自然地返回多个值

```rust
fn exgcd(a: i64, b: i64) -> (i64, i64, i64) {
    if b == 0 {
        (1, 0, a)
    } else {
        let (x0, y0, gcd) = exgcd(b, a % b);
        (y0, x0 - a / b * y0, gcd)
    }
}
```

由 $ax + Ny = 1$ 可知 x 就是逆元所在的同余类：

```rust
let answer = x.rem_euclid(n); // 保证结果落在 0..n
```

文本统计首先考查的是需求拆分

程序从标准输入读入全部内容，输出四项结果：

结果	需要回答的问题
`lines`	空输入、空行、末尾换行怎样计算？
`words`	连续空白会不会产生空词？
`chars`	计算字节、Unicode 标量值还是屏幕字符？
`longest`	换行符是否属于一行？

在让 Agent 写代码前，先把这些边界写进提示或测试。

一次读完输入更符合这道题的结构

```rust
use std::io::{self, Read};

fn read_input() -> io::Result<String> {
    let mut input = String::new();
    io::stdin().read_to_string(&mut input)?;
    Ok(input)
}
```

逐行调用 `read_line` 也能实现，但必须自己处理：

- 本次读入是否包含换行符
- 文件末尾是否还有一行
- 空输入和空行的区别

`len()` 和 `chars().count()` 回答不同的问题

文本	`len()`	`chars().count()`
abc	3	3
你好	6	2
U+1F600（一个 emoji）	4	1
两个区域指示符组成的旗帜	8	2

```rust
let n = line.chars().count();
```

本题按 `chars().count()` 计算；它不是 UTF-8 字节数，也不保证等于屏幕上看到的图形数量。

四类输入可以快速暴露边界错误

`< 空输入 >`

```
one  two

  three
```

`你好 + 一个 emoji`

还要分别测试最后一行**有换行**和**没有换行**的情况。

不要只检查程序能运行；要逐项写出预期的 `lines`、`words`、`chars` 和 `longest`。

所有权决定一个值之后还能不能使用

```rust
let name = String::from("Ferris");
let moved = name;
// println!("{name}"); // name 的所有权已经移走
```

只需要读取时，优先传借用：

```rust
fn print_name(name: &str) {
    println!("{name}");
}
```

需要修改时传 `&mut T`；确实需要接管值时才传 `T`。

遍历时修改当前元素，用可变迭代器

```rust
let mut data = vec![1, 2, 3];
for value in &mut data {
    *value += 1;
}
```

但遍历期间不能再次借用整个 `Vec`：

```rust
for value in &mut data {
    // data.push(*value); // 同时出现两个可变借用
}
```

如果需要改变长度，先记录修改，等遍历结束后统一执行。

容器表达数据的组织方式

需求	容器
按键查找，并保持键有序	`BTreeMap<K, V>`
去重，并保持元素有序	`BTreeSet<T>`
随时取得最大值	`BinaryHeap<T>`
在两端插入和删除	`VecDeque<T>`

计数时常用 `entry`:

```
for ch in text.chars() {
    *count.entry(ch).or_insert(0) += 1;
}
```

迭代器把“逐个处理”连成一条流程

```rust
let doubled: Vec<_> = data.iter()
    .map(|x| x * 2)
    .collect();

let sums: Vec<_> = a.iter()
    .zip(b.iter())
    .map(|(x, y)| x + y)
    .collect();

let total: i32 = data.iter().sum();
```

`iter()` 只借用元素；`into_iter()` 会消费容器。是否还要使用原容器，决定选哪一个。

Tree: Option<Box<Node>> 表示可空的子树

```rust
struct Node {
    value: i32,
    left: Option<Box<Node>>,
    right: Option<Box<Node>>,
}

struct Tree {
    root: Option<Box<Node>>,
}
```

- None：这里没有结点
- Some(Box<Node>)：结点存放在堆上
- Box 让递归类型具有确定的大小

插入时让可变借用沿着树向下移动

```rust
let mut current = &mut self.root;

while let Some(node) = current {
    if value < node.value {
        current = &mut node.left;
    } else if value > node.value {
        current = &mut node.right;
    } else {
        return Err(Error::DuplicateValue);
    }
}

*current = Some(Box::new(Node::new(value)));
```

更新根结点前，先用 take() 取出旧值

错误思路：借用 root.right 的同时修改 root。

```
let root = self.root.take().ok_or(Error::EmptyTree)?;
self.root = root.right;
Ok(())
```

take() 完成两件事：

1. 把 Option 中的值移出
2. 在原位置留下 None

因此旧根结点不再被 self 借用，可以安全地把右子树放回根结点。

把编译器、测试和 Agent 分工用好

- **编译器**：检查类型、所有权和借用是否成立
- **测试**：检查程序是否满足题目要求和边界情况
- **Agent**：生成与修改代码，但不能替你判断结果是否正确

遇到问题时依次确认：

1. 代码能否通过 `cargo check`
2. 最小样例是否得到预期结果
3. 边界输入是否仍然正确