程序设计训练 Rust 课堂大作业批改反馈

游宇凡 2025.9.10

代码风格规范

clippy 是你的老师，不是敌人

避免 for 循环枚举下标

来自 C++ 的三段式 for 语句的习惯：for 循环使用一个下标变量

```
for i in 0..26 {
    print!("{}", letter_status[i]);
}
```

```
warning: the loop variable `i` is only used to index `letter_status`
 |
 |                 for i in 0..26 {
 |                          ^^^^^
 |
 = help: for further information visit https://rust-lang.github.io/rust-clippy/master/index.htm
help: consider using an iterator
 |
 -                 for i in 0..26 {
 +                 for <item> in &letter_status {
```

避免 for 循环枚举下标

Rust 的 for 循环其实是使用迭代器，类似 C++ 的 range for，一般来说不需要枚举下标：

```
for status in &letter_status {
    print!("{}", status);
}
```

避免 for 循环枚举下标

Rust 的 for 循环其实是使用迭代器，类似 C++ 的 range for，一般来说不需要枚举下标：

```
for status in &letter_status {
    print!("{}", status);
}
```

使用迭代器而非枚举下标的好处：

避免 for 循环枚举下标

Rust 的 for 循环其实是使用迭代器，类似 C++ 的 range for，一般来说不需要枚举下标：

```
for status in &letter_status {
    print!("{}", status);
}
```

使用迭代器而非枚举下标的好处：

- 代码意图清晰：遍历数组中的各个元素，而不需要先枚举下标再访问这样绕一下。

避免 for 循环枚举下标

Rust 的 for 循环其实是使用迭代器，类似 C++ 的 range for，一般来说不需要枚举下标：

```
for status in &letter_status {
    print!("{}", status);
}
```

使用迭代器而非枚举下标的好处：

- 代码意图清晰：遍历数组中的各个元素，而不需要先枚举下标再访问这样绕一下。
- 省去越界检查：使用下标访问数组时每次会进行越界检查，而迭代器则不会，尤其是在循环体内需要多次访问元素时，可能对性能有明显的影响。

避免 for 循环枚举下标

Rust 的 for 循环其实是使用迭代器，类似 C++ 的 range for，一般来说不需要枚举下标：

```
for status in &letter_status {
    print!("{}", status);
}
```

使用迭代器而非枚举下标的好处：

- 代码意图清晰：遍历数组中的各个元素，而不需要先枚举下标再访问这样绕一下。
- 省去越界检查：使用下标访问数组时每次会进行越界检查，而迭代器则不会，尤其是在循环体内需要多次访问元素时，可能对性能有明显的影响。
- 避免写错枚举范围：如果是 `0..vec.len()` 的话相对还好，如果是一个常量，有可能写错范围，和数组长度不匹配。

如果下标不止用来访问数组

```
for i in 0..26 {
    println!("{i}: {}", letter_status[i]);
}
```

如果下标不止用来访问数组

```
for i in 0..26 {
    println!("{i}: {}", letter_status[i]);
}
```

`enumerate()` 方法可以同时获取下标和元素：

```
for (i, status) in letter_status.iter().enumerate() {
    println!("{i}: {status}");
}
```

拓展例子：迭代器的进阶使用

```
// 枚举下标
let mut ok = true;
for i in 0..guess_count {
    let result = get_guess_result(&guesses[i], answer);
    if result != status[i] {
        ok = false;
        break;
    }
}
```

拓展例子：迭代器的进阶使用

```
// 使用迭代器的 zip 操作
let mut ok = true;
for (guess, expected) in guesses.iter().zip(status) {
    let result = get_guess_result(guess, answer);
    if result != expected {
        ok = false;
        break;
    }
}
```

拓展例子：迭代器的进阶使用

```
// 完全使用迭代器
let ok = guesses.iter().zip(status)
    .all(|(guess, expected)| get_guess_result(guess, answer) == expected);

// 或者直接比较两个迭代器
let ok = guesses.iter().map(|guess| get_guess_result(guess, answer)).eq(status);
```

拓展例子：迭代器的进阶使用

```
// 完全使用迭代器
let ok = guesses.iter().zip(status)
    .all(|(guess, expected)| get_guess_result(guess, answer) == expected);

// 或者直接比较两个迭代器
let ok = guesses.iter().map(|guess| get_guess_result(guess, answer)).eq(status);
```

小结：

适当地使用迭代器能让代码更简单易读，不易出错，性能更高。

了解更多 `Iterator` 的方法有时可以进一步简化代码，但只使用基础功能就很好了，如果过分追求高级功能和代码简短，可能反而会使代码不可读。

使用切片（slice）作为函数参数

使用切片（slice）作为函数参数

- 如果有一个 `Vec<T>`，获取一个 `&[T]` 是很容易的，取引用即可。

使用切片（slice）作为函数参数

- 如果有一个 `Vec<T>`，获取一个 `&[T]` 是很容易的，取引用即可。
- 反之，如果有一个 `&[T]`，想要获取一个 `Vec<T>` 或者 `&Vec<T>`，则无法直接做到，需要分配新的内存然后将数据复制一份（`.to_vec()`）。

使用切片（slice）作为函数参数

- 如果有一个 `Vec<T>`，获取一个 `&[T]` 是很容易的，取引用即可。
- 反之，如果有一个 `&[T]`，想要获取一个 `Vec<T>` 或者 `&Vec<T>`，则无法直接做到，需要分配新的内存然后将数据复制一份（`.to_vec()`）。
- 而 `&Vec<T>` 和 `&[T]` 在获取数据方面是能力相同的，只是多了 `Vec` 特有的方法，例如 `capacity()`，绝大多数情况下这是不需要的。

使用切片（slice）作为函数参数

- 如果有一个 `Vec<T>`，获取一个 `&[T]` 是很容易的，取引用即可。
- 反之，如果有一个 `&[T]`，想要获取一个 `Vec<T>` 或者 `&Vec<T>`，则无法直接做到，需要分配新的内存然后将数据复制一份（`.to_vec()`）。
- 而 `&Vec<T>` 和 `&[T]` 在获取数据方面是能力相同的，只是多了 `Vec` 特有的方法，例如 `capacity()`，绝大多数情况下这是不需要的。

综上，如果一个函数的参数定义为 `&Vec<T>` 类型，它对调用者提出了额外的要求，但没有给函数自身带来什么额外的能力。因此，应当将函数参数定义为 `&[T]` 类型。

`&String` 和 `&str` 也是同理。

拓展：降低函数参数类型的约束要求

使用 slice 而非 owned type 作为函数参数类型是一个特例，更普遍地来说，降低函数参数类型的约束要求，是设计灵活易用的 Rust API 的一个重要原则。

拓展：降低函数参数类型的约束要求

使用 slice 而非 owned type 作为函数参数类型是一个特例，更普遍地来说，降低函数参数类型的约束要求，是设计灵活易用的 Rust API 的一个重要原则。

- 可以使用 `AsRef` 作为参数的特型约束，以 `std::fs::File::open` 为例：

```
pub fn open<P: AsRef<Path>>(path: P) -> io::Result<File> {
    OpenOptions::new().read(true).open(path.as_ref())
}
```

拓展：降低函数参数类型的约束要求

使用 slice 而非 owned type 作为函数参数类型是一个特例，更普遍地来说，降低函数参数类型的约束要求，是设计灵活易用的 Rust API 的一个重要原则。

- 可以使用 `AsRef` 作为参数的特型约束，以 `std::fs::File::open` 为例：

```rust
pub fn open<P: AsRef<Path>>(path: P) -> io::Result<File> {
    OpenOptions::new().read(true).open(path.as_ref())
}
```

- 如果需要参数是 `Iterator`，一般会使用 `IntoIterator`，而 `Iterator` 也实现了 `IntoIterator`，可以 `into_iter()` 得到自身。以 `std::iter::zip` 为例：

```rust
pub fn zip<A, B>(a: A, b: B)
    -> Zip<<A as IntoIterator>::IntoIter, <B as IntoIterator>::IntoIter>
where A: IntoIterator, B: IntoIterator { ZipImpl::new(a.into_iter(), b.into_iter()) }
```

拓展：降低函数参数类型的约束要求

- 创建 `struct` 时不作约束，只对特定方法进行约束，以 `std::collections::BTreeMap` 为例：

```
impl<K, V> BTreeMap<K, V> {
    pub const fn new() -> BTreeMap<K, V> { /* ... */ }
}

impl<K, V, A: Allocator + Clone> BTreeMap<K, V, A> {
    pub fn insert(&mut self, key: K, value: V) -> Option<V>
    where
        K: Ord,
    { /* ... */ }
}
```

减少 mut 变量

- Rust 变量默认不可变，不是为了让你多写 `mut`，而是提醒你尽量不要 `mut`
- 通过函数 / 代码块将完成修改后的结果放在一个不可变变量中
- 特别地，使用迭代器的方法有时能省去 `mut`
- 变量定义时不必初始化，只要在访问前有赋值即可；如果在第一次赋值前不会访问变量，则不应该初始化，因为初始化的值会直接被覆盖而不会被访问

模块结构组织

模块树的结构由 `mod` 指定，默认和代码文件位置相关，但并不直接取决于代码文件的目录结构（和 Python 不同）

- 可以直接将模块写在同一个文件的代码块中（一般只有 `tests` 会这样写）
- 可以使用 `#[path = "path/to/mod/code.rs"] mod foo;` 来指定一个模块的代码文件位置，而不使用默认的 `foo.rs`（一般不要这么干）
- 如果在多个不同的地方（例如多个 binary crate 中）同时使用 `mod` 链接到同一个文件，则这个文件中的代码会在模块树中多次出现，可能导致一些奇怪的问题，例如 dead/unused code 警告

多个 crate 共享代码

正确的在多个 binary crate 之间共享代码的方式：

- 编写一个 library crate，然后在 binary crate 中通过 package name 来访问它，例如 `Cargo.toml` 中设置 `name = wordle`，则使用 `use wordle::...` 来访问。
- 也可以分成多个 package，通过 `{ path ="../foo" }` 来指定本地依赖，可以使用 Cargo workspace 统一管理多个 package。这样的话可以有多个 library crate，也可以分别发布给他人使用。

即使只有一个 binary crate，将核心代码放在 library crate 中往往也能让代码的组织结构更好，且为未来的扩展做好准备。

多定义 struct

Rust 项目通常由很多 struct 以及相应的 method 构成，而非只是一堆函数。

从代码风格的角度，定义 struct 可以：

- 拆分代码到各个 struct，将相关代码放在一起，提升可读性
- 方便传参，避免使用大量变量和参数
- 相比于 tuple，struct 的各个字段有名字，能看出分别是什么意思，且使用字段名访问比起数字下标更不易出错

进一步地，使用 newtype pattern 定义现有类型的 wrapper，还能实现一些功能：

- 可以为非自己实现的类型定义新的 method 和 trait
- 可以在构造函数处进行校验，从而确保一个类型的值具有某种性质

Git 使用规范

使用 Git 管理版本，记录历史，而不只是提交代码

.gitignore

.gitignore

- Git 仓库中应该提交源文件，而不是临时文件（例如 OJ 评测产生的文件）、生成的文件（例如 Rust 编译产生的 `target` 目录）、存储的数据（例如 Wordle 游戏状态、OJ 持久化存储）、其他无关文件（例如操作系统产生的 `.DS_Store`）。

.gitignore

- Git 仓库中应该提交源文件，而不是临时文件（例如 OJ 评测产生的文件）、生成的文件（例如 Rust 编译产生的 `target` 目录）、存储的数据（例如 Wordle 游戏状态、OJ 持久化存储）、其他无关文件（例如操作系统产生的 `.DS_Store`）。
- 无关文件可以列入 `.gitignore` 中，避免被加入到仓库。

Git commit message

- commit message 要总结这个 commit 干了什么事，以后看提交历史能找到一个改动对应的 commit，如果和他人合作能告诉别人自己做了什么
- 常见的看不出干了什么的 commit message：
  - “fix a bug” / “debug”：请尝试描述具体修了什么 bug，或者修了哪个组件的 bug
  - “final”：commit message 描述的是一次改动，而不是标记一个版本，你的最终版相对于前一个 commit 有什么改动，依然需要写在 commit message
    - 如果要标记版本，可以使用 `git tag`；但最终版不一定真的是最终版，比起“final”，还是标记版本号比较好
    - 如果一个 commit 真的只是进行版本发布，例如修改代码中的版本号，编写更新日志，那 commit message 可以写 “Release version xxx”
- 可以参考 Conventional Commits 的格式与规范要求

如何 commit

- 拆分 commit：有时一个 commit 做了多件事，就很难用一句话概括，不好写 commit message。实际上，这并不是写 message 的问题，而是 commit 内容的问题。如果做了多处不太相关的修改，应该拆分成多个 commit 提交，分别描述做了什么。为了拆分 commit，可以每做一件事就 commit 而不攒起来，也可以使用命令 `git add -p` 或者在 VS Code 的图形化界面中选择性添加部分代码进行 commit。
- 暂存代码：作为半成品的暂存代码不应该出现在版本历史中，如果需要暂存代码，可以先只在本地 commit，不 push 到远程（GitLab），之后再通过 `git commit --amend` 或者 `git reset` / `git rebase -i` 来修改 commit，整理好后再 push
- CI 测试：CI 测试主要起到保险检查的作用，调试还是应该在本地进行，不应该完全依赖于 CI 进行检查。如果本地运行测试遇到问题，应该想办法解决本地运行的问题。

其他问题

IDE 配置

参考课程文档中的 开发辅助工具 一节。在 IDE 中进行配置可以省去手动运行命令：

- `cargo fmt`：可以在 IDE 中设置保存代码时自动格式化，或使用快捷键格式化代码。
- `cargo clippy`：可以在 IDE 中配置显示 clippy 警告，也可以在 IDE 中自动修复警告（注意检查自动修复是否正确合理）。

尤其不要只依赖于 CI 进行测试。

寻求帮助

参考课程文档中 寻求帮助 一节。

- 共性问题（例如要求不明确）尽量公开提问，个人问题可以私聊。
- 推荐使用 GitLab issue、网络学堂答疑区、邮件，可以将对话组织在一起，方便检索，避免微信消息被刷下去。
- 测试出错时，注意查看完整输出日志（尤其是 OJ 的 `.stdout`、`.stderr`、`.http` 文件，在 CI 上可以在 test job 的详情页下载 artifact）。
- 调试时，首先通过跟踪代码、输出中间值、单步调试等方法定位问题出在哪一步，获取详细的问题信息，不能对着不完整的问题信息和不精确的问题位置瞎猜。
- 提问时，描述你获取到的尽量详细的问题信息，为了解决问题进行了什么尝试。