第二三讲课后练习讲解

游宇凡 2026.8.25

第二讲第 2 题

```rust
pub struct Job {
    pub name: String,
    pub payload: Vec<u8>,
}

fn process(p: Vec<u8>) -> usize {
    p.len()
}

fn audit(j: &Job) {
    println!("audited {}", j.name);
}

pub fn run(job: Job) -> usize {
    let payload = job.payload;
    let n = process(payload);
    audit(&job);
    n
}
```

- A: `let payload = job.payload` 只移动了 `payload` 字段，其他字段依然可以访问
- B: 正确
- C: `#[derive(Clone, Copy)]` 需要各个字段都 `Copy`，而 `payload` 是 `Vec`，无法 `Copy`

  一般来说，在堆上分配空间的容器无法 `Copy`
- D: 正确，把参数从 `Vec<u8>` 改成 `&[u8]` 是好的做法，在设计函数参数类型时，应当避免不必要地取得所有权

第二讲第 4 题 (2)

```rust
pub struct UserName(String);

impl UserName {
    pub fn new(raw: String) -> UserName {
        UserName(raw.trim().to_ascii_lowercase())
    }

    pub fn as_str(&self) -> &str { &self.0 }
}

pub fn login(roster: &[UserName], name: &UserName) -> bool {
    for u in roster {
        if u.as_str() == name.as_str() {
            return true;
        }
    }
    false
}
```

- 这里 `UserName::new` 的参数类型为 `String`，但其实不需要取得所有权，使用 `&str` 就可以了，避免不必要的 `clone()`
- 也不应使用 `&String`，对 `String` 取引用可以作为 `&str` 使用，而 `&str` 要分配内存得到一个新的 `String` 才能取到 `&String`

第二讲第 4 题 (3)

```rust
pub struct UserName(pub String);

impl UserName {
    pub fn new(raw: &str) -> UserName {
        UserName(raw.trim().to_ascii_lowercase())
    }

    pub fn as_str(&self) -> &str { &self.0 }
}

pub fn login(roster: &[UserName], name: &UserName) -> bool {
    for u in roster {
        if u.0 == name.0 {
            return true;
        }
    }
    false
}
```

这里 `UserName` 内部使用了 `pub`，因此外部代码可以绕过 `UserName::new`，直接使用 `UserName(...)` 构造，就无法保证经过规范化

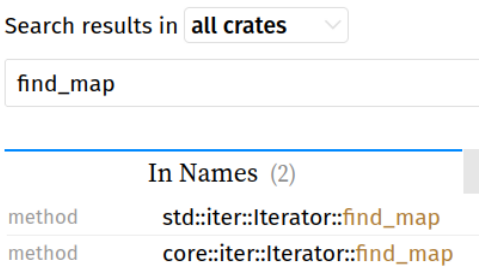
第二讲第 4 题 (4)

```rust
pub struct UserName(String);

impl UserName {
    pub fn new(raw: &str) -> UserName {
        assert!(
            !raw.starts_with(char::is_whitespace)
                && !raw.ends_with(char::is_whitespace)
                && !raw.chars().any(|c| c.is_ascii_uppercase()),
            "用户名首尾不能有空格，而且不能含大写英文字母"
        );
        UserName(raw.to_string())
    }

    pub fn as_str(&self) -> &str {
        &self.0
    }
}
```

- 这里把规范化改成了 `assert!`，可以保证结果是符合要求的，但对不合法的输入会 panic 而不是规范化，改变了语义
- 在实际代码中，检查而非规范化也是很常见的写法，但如果要这样，一般应该返回 `Result` 而非 panic

第三讲第 1 题

这题涉及到 `map`、`collect`、`flatten`、`filter_map`、`find_map`、`fold`、`Option::or` 等大量库函数，如果不清楚函数的用法，当然可以问 AI，但 Rust 的文档其实是写得非常好的，查阅起来很方便，鼓励大家如果明确地想要知道一个函数的用法可以多查阅文档




```rust
fn find_map<B, F>(&mut self, f: F) -> Option<B>
where
    Self: Sized,
    F: FnMut(Self::Item) -> Option<B>,
```

Applies function to the elements of iterator and returns the first non-none result.

`iter.find_map(f)` is equivalent to `iter.filter_map(f).next()`.

Examples

```rust
let a = ["lol", "NaN", "2", "5"];

let first_number = a.iter().find_map(|s| s.parse().ok());

assert_eq!(first_number, Some(2));
```

第三讲第 2 题

这题主要考察高效使用 `HashMap` / `BTreeMap` 的 `entry` 方法，在同时需要查询修改时，一般就是 B/C 的这种写法：

- `*m.entry(w.clone()).or_insert(0) += 1;`
- `m.entry(w.clone()).and_modify(|n| *n += 1).or_insert(1);`

注意 `D` 选项首先会无条件查询一次，然后在 `None` 分支会查询第二次

```
match m.get_mut(w) {
    Some(n) => *n += 1,
    None => { m.insert(w.clone(), 1); }
}
```

第三讲第 3 题

这题主要考察的是，参数会在调用时取值，如果直接将 `x.to_string()` 作为参数就总会执行，如果传函数或者闭包作为参数，则是在接收的函数内部调用

A 选项虽然总会 `to_string`，但这是根据分支分别对取到的 `&String` 或者 `"anonymous"` 进行 `to_string`

```
cfg.get("name").map(|s| s.as_str()).unwrap_or("anonymous").to_string()
```

第三讲第 5 题

这题考察的是 Rust 中 `String` 和 `&str` 是 UTF-8 变长编码，每个字符的字节数不确定，因此需要逐字符遍历才能确定每个字符的位置，不能通过下标直接随机访问

- B: 下标通过 `char_indices` 得到，在正确的位置就不会 panic
- C: `char_indices` 和原本的 `chars` 都要逐字符遍历，不是额外代价，且不需要整串走一遍，通过 `nth(n)` 取时只用遍历 n 个字符
- D: 主要问题是 `nth(n)` 其实取的是第 n+1 个元素，第一个元素是 `nth(0)`，这个类似数组下标，所以是对的没有差