

4-2 Iterators

题意回顾：使用迭代器来表示一个无穷自然数列 $0, 1, 2, \dots$ ，然后对迭代器进行操作，支持的操作类型：

- Advance：遍历迭代器
- Multiply：每个元素做乘法
- Subtract：每个元素做减法
- ZipAdd：每两个元素做加法
- Odd：只保留奇数
- Repeat：每个数重复若干次

用迭代器实现，不需要事先把无穷多个数放到一个数组里再计算，而是在遍历的时候懒惰地进行计算。

5-2 Date

关于日期的一些有趣的小知识:

- 由于夏令时等原因, 一些时间是不存在的:

```
$ date -d "1919-04-13"
date: invalid date '1919-04-13'
$ TZ=UTC date -d "1919-04-13"
Sun Apr 13 00:00:00 UTC 1919
```

- 历史上出现过历法的变迁, 从儒略历到格勒哥里历, 直接从 1582-10-04 跳到了 1582-10-15, 在编程的时候可能会带来麻烦。同时还有一个问题: 1300-02-29 是合法的日期吗?

```
$ date -d "1300-02-29"
date: invalid date '1300-02-29'
```

6-2 Map Reduce

题意回顾：实现大整数的批量 $a^b \bmod m$ 运算，最后进行求和。

解决方法：本题的标题叫做 **Map Reduce**，意思是对于一个操作，可以分为两个步骤：**Map**（映射）和 **Reduce**（规约）。在本题中，要执行的操作是 N 次 $a^b \bmod m$ 运算，然后再对结果进行 $N - 1$ 次加法运算。我们可以使用并行计算来加快这个操作，首先把操作分成 **Map** 和 **Reduce**：

- **Map**：第一步并行计算 N 次 $a^b \bmod m$ ，可以看到这 N 次计算之间互相没有数据依赖，可以很好地并行计算出 N 个结果。
- **Reduce**：第二步，利用加法运算的结合律，我们不需要按顺序求和，可以采用分治的方法，把 N 个数分为两部分，分别并行求和，再把两部分的和加起来。

可以用类似下面的代码来实现 **Map Reduce** 运算：

```
data.iter().map(mapper).fold(identity, reducer);
```


Git 使用

- 用 `.gitignore` 忽略文件：
 - 生成的文件（如 `target` 目录中的编译结果）
 - 临时文件（如 OJ 评测结果，持久化的 JSON 文件）
 - 第三方依赖（如 `node_modules`）
 - Sqlite 数据库文件
 - 操作系统生成的文件（如 `.DS_Store`）

Git 使用

- commit 相关
 - commit 不要太少，体现出增量开发的过程
 - commit 信息简练清晰
 - 注意 commit 用户名，可以用 `git config` 命令进行设置

错误处理

在 `actix-web` 中，如何优雅地处理错误？

文档告诉我们，可以用 `Result<Responder, ResponseError>` 作为函数的返回值，这样就方便了？操作符的使用。

错误处理

```
#[derive(Debug, Display, Error)]
#[display(fmt = "my error: {}", name)]
struct MyError {
    name: &'static str,
}

// Use default implementation for `error_response()` method
impl error::ResponseError for MyError {}

async fn index() -> Result<&'static str, MyError> {
    Err(MyError { name: "test" })
}
```

错误处理

而 OJ 大作业需要一个 JSON 错误响应，因此可以自定义错误类型：

```
#[derive(Serialize, Deserialize, Debug, Clone)]  
struct JSONError {  
    code: u64,  
    reason: String,  
    message: String,  
}
```

错误处理

然后定义如何从 `Error` 类型生成对应的 HTTP 响应:

```
impl ResponseError for Error {  
    fn status_code(&self) -> request::StatusCode {  
        self.status_code  
    }  
  
    fn error_response(&self) -> HttpResponse<actix_web::body::BoxBody> {  
        HttpResponse::build(self.status_code()).json(&self.json)  
    }  
}
```

错误处理

对于常见的错误类型，可以实现：

```
fn not_found(message: String) -> Error {  
    Error {  
        status_code: StatusCode::NOT_FOUND,  
        json: JSONError {  
            code: 3,  
            reason: "ERR_NOT_FOUND".to_string(),  
            message,  
        },  
    }  
}
```

错误处理

使用 `Option::ok_or()` 或 `Option::ok_or_else()` 把 `Option<T>` 转换为 `Result<T, Error>`:

```
let language = config
    .languages
    .iter()
    .find(|l| l.name == job.submission.language)
    .ok_or_else(|| {
        Error::not_found("...")
    })?;
```

错误处理

指定如何从实现了 `std::error::Error` 特型的其他错误转换到自定义的错误类型:

```
impl<T: std::error::Error> From<T> for Error {
    fn from(err: T) -> Self {
        Error {
            status_code: StatusCode::INTERNAL_SERVER_ERROR,
            json: JSONError {
                code: 6,
                reason: "ERROR_INTERNAL".to_string(),
                message: format!("Internal error: {}", err.to_string()),
            },
        }
    }
}
```

非阻塞评测

`actix-web` 处理 HTTP 请求的方式:

- 启动若干个线程，每个线程里运行一个单线程的异步运行时，在每个线程中启动 `Worker` 异步任务，默认数量是 CPU 的核心数；
- 在主线程中启动 `Accept` 异步任务，负责处理客户端新的 TCP 连接；
- `Worker` 异步任务从 `Accept` 异步任务获取 TCP 连接；
- 每个 `Worker` 接收到 TCP 连接后，在当前线程启动一个新的异步任务，负责该 TCP 连接上的 HTTP 请求。

非阻塞评测

阻塞评测为什么会导导致自动测试中后续的请求超时：

- 自动测试采用了一个 TCP 连接来发送多个请求；
- 因此该 TCP 连接会被分配到某一个线程的 Worker；
- 当该线程执行了阻塞的系统调用的时候，Worker 无法处理 TCP 连接上的新 HTTP 请求；
- 但是 Accept 异步任务还在继续运行，所以可以接受新的 TCP 连接，所以 VSCode REST Client 依然可以请求；
- 当所有工作线程都在阻塞时，就无法处理新请求了。

非阻塞评测

- 如何实现非阻塞评测？
- 两个思路：
 - ① 化阻塞调用为非阻塞，用 `tokio::process::Command` 替代 `std::process::Command`
 - ② 把阻塞调用放在单独的线程池中跑，用 `actix_web::web::block`
- 如何在 `async` 函数中启动一个异步任务：`actix_web::rt::spawn`

Web 安全

如何实现登录功能？

- 登录是需要更新状态的，所以一般是使用 **POST** 方法的 **HTTP** 请求；
- 用户名和密码通过 **URL** 传给后端？不行，因为 **URL** 一般会打印在日志中，所以一般是放在请求的正文中；
- 密码可以明文写在 **POST** 请求的正文吗？只要是用 **HTTPS** 加密就没问题；
- 有没有必要在前端对密码做哈希？一般没有必要，因为哈希不能抵抗重放攻击，如果没有用 **HTTPS**，那么攻击者只要窃听了登录请求，就可以登录，不需要知道密码；
- 如果登录时不想传输密码，可以用 **Challenge-Response** 方法来进行认证；
- 后端可否明文保存密码？不能，如果采用了明文密码，攻击者就可以利用明文去攻击同一个用户在其他网站上的用户；
- 如何处理用户名或密码错误？不能告诉用户是哪个错了，应该说二者都错，否则可能会降低攻击难度。

Web 安全

如何维护登录状态?

- **Cookie**: 服务端在 HTTP 响应的头部加入 **Set-Cookie**, 要求浏览器保存 **Cookie**, 那么之后一定时间内浏览器发起请求的时候, 都会在请求的头部中添加 **Cookie** 字段;
 - 更进一步, 需要设置 **Cookie** 的安全属性, 如限制路径, 限制不允许 JS 获取, 限制仅通过 **HTTPS** 发送, 限制跨域等等;
 - **Cookie** 内容可能是加密后的用户信息, 或者只是一个随机数, 由后端在数据库中查询
 - 常用于浏览器。
- **Bearer Token**: 服务端通过 **JSON** 响应等方式告诉客户端一个 **Token**, 之后客户端发请求的时候, 需要自行添加 **Authorization: Bearer xxx** 头部来认证;
 - 由于浏览器不会自动发送 **Authorization: Bearer xxx** 头部, 这一步通常是由 **JS** 完成的;
 - **Token** 的格式比较自由, 常见的格式有 **JWT**, 是服务器将一段 **JSON** 签名并编码后的结果。

Web 安全

浏览器阻拦了跨域请求怎么办？

- 同学在为 OJ 开发 Web 前端的时候，如果前端部署在 `http://localhost:8000`，而后端部署在 `http://localhost:12345`，此时前端的 JS 向后端发起请求，可能会被浏览器拦截；
- 原因是 CORS 安全机制：默认情况下不允许 JS 访问其他域名的资源，否则可能会导致安全性问题，例如访问 A 网站时，JS 偷偷爬取了在 B 网站上登录的用户信息；
- 如何解决：浏览器会先发送 OPTIONS 请求询问后端是否允许来自前端 JS 的跨域请求，后端可以在响应中的头部告诉浏览器，允许前端 JS 做些什么事情；
- 在代码中，可以用 `actix-cors` 库来帮助配置 CORS。

数据库使用

数据库里存储数据的方式是表：

```
CREATE TABLE users (  
    id INT,  
    name VARCHAR(255)  
);
```

每一行表示一个用户，它的属性就是 CREATE TABLE 时指定的各个列。对于 OJ 大作业，应该把各个字段对应到数据库的表的列中，而不是序列化以后以字符串的形式放在数据库中。

把数据按列排放以后，可以方便搜索：

```
SELECT * FROM users WHERE name = "abc"
```

数据库也可以创建索引来进一步优化查询。如果序列化保存到了表格中，这些就没法实现了。

数据库使用

- 每个提交信息有多个数据点，如何表示？表的列是需要实现确定的，创建足够多的列不够优雅；
- 为了实现一对多的关系，通常的办法是根据 ID 来关联：创建一个提交信息的表 `jobs` 和一个数据点的表 `cases`，二者通过 ID 进行关联，例如从一个 `job id` 可以查询到 `cases` 表中数据这个提交的若干个数据点；
- 查询的时候，使用 `JOIN` 语句就可以把提交信息和数据点信息都对应起来；
- 还可以使用 `FOREIGN KEY` 来保证 A 表中记录的 B 表的 ID 一定是合法的；
- ID 常用 `AUTO INCREMENT` 来实现；
- 为了方便多线程场景下使用，需要使用连接池。

数据库安全

- 创建数据库连接的时候，需要指定数据库的地址，用户名和密码；
- 连接配置是比较敏感的，因为获取了连接配置就可以直接访问数据库，并且拥有很大的权限；
- 所以真正的连接配置一定不可以写在源代码中，而是在部署的时候配置，并且只有服务端自己可以知道；
- 更进一步，还需要限制数据库用户的权限，仅保留需要的权限，把攻击面缩到尽量小。

