OJ 大作业报告

计 15 宋驰 2021010797

一、 简单的程序结构和说明

整体框架如下图：

（1）定义了若干结构体

```rust
use actix_web::{get, middleware::Logger, post, web, App, HttpServer, Responder, HttpResponse, put};…

// 配置文件config
#[derive(Serialize, Deserialize, Debug, Clone)]
4 implementations
struct Config {…

#[derive(Serialize, Deserialize, Debug, Clone)]
4 implementations
struct Server {…

#[derive(Serialize, Deserialize, Debug, Clone)]
4 implementations
struct OneProblem {…

#[derive(Serialize, Deserialize, Debug, Clone)]
4 implementations
struct Mis {…

#[derive(Serialize, Deserialize, Debug, Clone)]
4 implementations
struct Question {…

#[derive(Serialize, Deserialize, Debug, Clone)]
4 implementations
struct Code {…

#[derive(Serialize, Deserialize, Debug, Clone, PartialEq)]
5 implementations
enum State {…

#[derive(Serialize, Deserialize, Debug, Clone, PartialEq)]
5 implementations
enum JudgeResult {…

#[derive(Serialize, Deserialize, Debug, Clone)]
4 implementations
enum CompilationResult {…
```

（2）定义一系列评测任务的 API 的函数

```rust
// 保存全局变量至json文件
fn write_to_longlasting() -> std::io::Result<()> {…

#[get("/hello/{name}")]
async fn greet(name: web::Path<String>) -> impl Responder {…

#[get("/jobs")]
async fn get_jobs(web::Query(info): web::Query<GetJob>) -> impl Responder {…

#[get("/jobs/{jobid}")]
async fn get_jobid(jobid: web::Path<i32>) -> impl Responder {…

#[get("/users")]
async fn get_users() -> impl Responder {…

#[get("/contests")]
async fn get_contests() -> impl Responder {…

#[get("/contests/{contestid}")]
async fn get_contests_contestid(contestid: web::Path<i32>) -> impl Responder {…

#[put("/jobs/{jobid}")]
async fn put_jobid(jobid: web::Path<i32>, config: web::Data<Config>) -> impl Responder {…

fn compile_and_run(body: PostJob, config: web::Data<Config>) -> Job {…

#[post("/jobs")]
async fn post_jobs(body: web::Json<PostJob>,
config: web::Data<Config>) -> impl Responder {…

#[post("/users")]
async fn post_users(body: web::Json<Users>) -> impl Responder {…

#[post("/contests")]
async fn post_contests(body: web::Json<Contest>, config: web::Data<Config>) -> impl Responder {…
```

除了定义了 API 相关的函数外，程序中还有用于执行编译和运行指令的函数，用于 post job 和 put job。

compile_and_run 函数的逻辑是先定位 problem_id 和 language，然后创建临时目录和临时文件。借助 std::command 进行编译，首先检查编译是否成功，编译成功后对每一个测试点借助 std::command 对可执行文件运行，并判断是否超时，超时后切断进程，不超时则判断是否存在 runtime error，最后与标准答案进行比较。

```rust
fn compile_and_run(body: PostJob, config: web::Data<Config>) -> Job {
    let mut tmp: i32 = -1;
    for i: usize in 0..config.languages.len() {
        if body.language == config.languages[i].name {
            tmp = i as i32;
        }
    }
    let mut p_tmp: i32 = -1;
    for i: usize in 0..config.problems.len() {
        if body.problem_id == config.problems[i].id {
            p_tmp = i as i32;
        }
    }
    let mut _code: &String = &body.source_code;
    let job_count: usize = {
        let lock: MutexGuard<Vec<Job>> = JOB_LIST.lock().unwrap();
        lock.len()
    };
    let mut _job_return: Job = Job {
        id: job_count as i32,
        creat_time: Utc::now().format(fmt: "%Y-%m-%dT%H:%M:%S%.3fZ").to_string(),
        updated_time: Utc::now().format(fmt: "%Y-%m-%dT%H:%M:%S%.3fZ").to_string(),
        submission: body.clone(),
        state: State::Finished,
        result: JudgeResult::Accepted,
        score: 0.0,
        cases: Vec::new()
    };
    let mut _bool_judge_result: bool = false;
    let path_dir: String = "test1".to_string() + &format!("{}", _job_return.id);
    if Path::new(&path_dir).exists() == true {
        fs::remove_dir_all(&path_dir).unwrap();
    }
```

post job 中，在检查了合法性之后便调用以上函数并返回 Job 结构体，put job 中，在定位了 job id 后调用以上函数重新评测。

write_to_longlasting 函数用于将全局变量写入 json 文件用于持久化存储。

```rust
// 保存全局变量至json文件
fn write_to_longlasting() -> std::io::Result<()> {
    let job_list: MutexGuard<Vec<Job>> = JOB_LIST.lock().unwrap();
    let users_list: MutexGuard<Vec<Users>> = USERS_LIST.lock().unwrap();
    let contest_list: MutexGuard<Vec<Contest>> = CONTESTS_LIST.lock().unwrap();
    let time: MutexGuard<Vec<Time>> = PROBLEM_SHORTEST.lock().unwrap();
    let user_submit: MutexGuard<Vec<Vec<UserSubmit>>> = USER_SUBMIT.lock().unwrap();
    let contest_ranklist: MutexGuard<Vec<ContestsUser>> = CONTESTS_RANKLIST.lock().unwrap();
    let longlasting: LongLasting = LongLasting {
        vec_job: job_list.clone(),
        vec_users: users_list.clone(),
        vec_contest: contest_list.clone(),
        vec_contest_user: contest_ranklist.clone(),
        vec_user_submit: user_submit.clone(),
        vec_time: time.clone()
    };
    let path: &Path = Path::new("./src/longlasting.json");
    fs::write(path, contents: serde_json::to_string_pretty(&longlasting).unwrap())
}
```

全局中，JOB_LIST、USERS_LIST、CONTESTS_LIST 分别用于保存每次评测的 Job 结构体、所有用户信息、所有比赛信息。

CONTESTS_RANKLIST 按照用户分类，保存了每个用户的所有提交信息，用于全局排行榜的统计。

USER_SUBMIT 按照比赛分类，保存了每个比赛的每道题目的提交次数限制，用于之后判断用户是否还有提交次数。

PROBLEM_SHORTEST 按照题目分类，保存了每道题目的最短运行时间，用于竞争得分。

二、 OJ 主要功能说明和截图

在我的大作业 OJ 中，实现了基础功能的 API，以及提高功能中的评分方式和评测技术的一些内容，下面我将针对我的基础功能的实现举例。

例：POST 指令。POST 指令中包括 POST /jobs、POST/users，由相应的函数实现。

POST /jobs 的实现在上面已有说明，POST/users 则是根据 id 和 name 进行判断，返回错误或者加入新用户。

```
981   #[post("/jobs")]
982   async fn post_jobs(body: web::Json<PostJob>,
983 > config: web::Data<Config>) -> impl Responder {…
1116
1117  #[post("/users")]
1118 > async fn post_users(body: web::Json<Users>) -> impl Responder {…
1198
```

响应效果：


```
{
  "id": 0,
  "creat_time": "2022-09-11T06:29:19.306Z",
  "updated_time": "2022-09-11T06:29:19.306Z",
  "submission": {
    "source_code": "fn main() { println!(\"Hello World!\"); }",
    "language": "Rust",
    "user_id": 0,
    "contest_id": 0,
    "problem_id": 0
  },
  "state": "Finished",
  "result": "Accepted",
  "score": 100.0,
  "cases": [
```

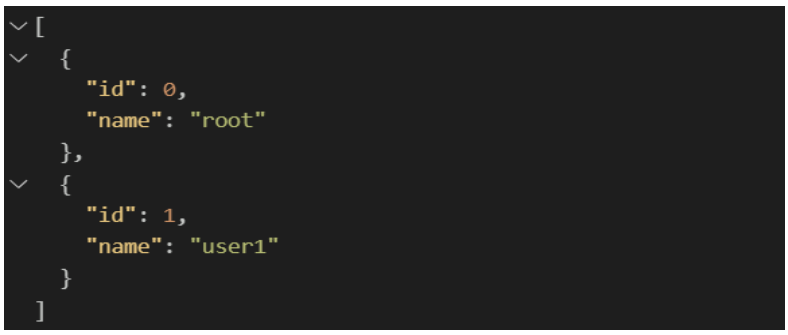
```
{
  "id": 1,
  "name": "user1"
}
```

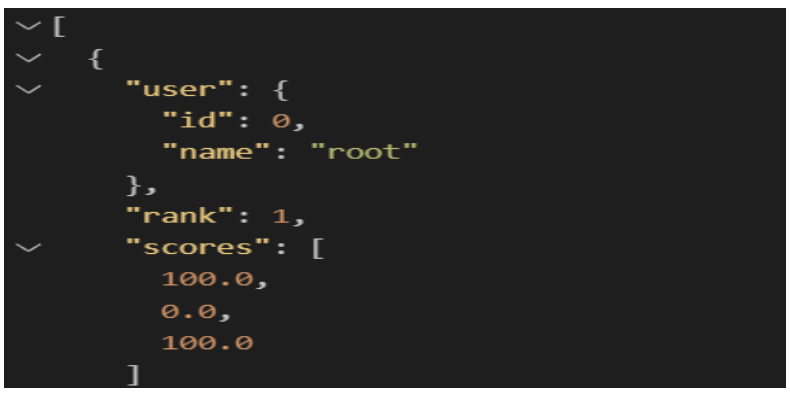
例：GET 指令。GET 指令中包括 GET /jobs、GET /jobs/{jobId}、GET /users，GET /contests/{contestId}/ranklist，由相应的函数实现，借助于全局变量。其中 GET /contests/{contestId}/ranklist 函数中的逻辑根据要求由 scoring_rule 和 tie_breaker 来采用相应的得分和排名方式。

```
314   #[get("/jobs")]
315 > async fn get_jobs(web::Query(info): web::Query<GetJob>) -> impl Responder {···
426
427   #[get("/jobs/{jobid}")]
428 > async fn get_jobid(jobid: web::Path<i32>) -> impl Responder {···
442
443   #[get("/users")]
444 > async fn get_users() -> impl Responder {···
453
```

```
1312  #[get("/contests/{contestid}/ranklist")]
1313  async fn get_contests_ranklists(contestid: web::Path<i32>,
1314 >     web::Query(contests_config): web::Query<ContestsConfig>, config: web::Data<Config>) -> impl Responder {···
```

响应效果：





三、 提高要求的实现方式

提高要求：评测技术之多比赛支持

通过实现 POST /contests、GET /contests、GET /contests/{contestId}、GET /contests/{contestId}/ranklist，前三个 API 的实现主要通过 CONTESTS_LIST 全局变量，并将 POST /contests 中的提交次数限制存入全局变量 USER_SUBMIT 中。

```
1199   #[post("/contests")]
1200 > async fn post_contests(body: web::Json<Contest>, config: web::Data<Config>) -> impl Responder {···
```

```
454   #[get("/contests")]
455 > async fn get_contests() -> impl Responder {···
464
465   #[get("/contests/{contestid}")]
466 > async fn get_contests_contestid(contestid: web::Path<i32>) -> impl Responder {···
480
```

GET /contests/{contestId}/ranklist 的实现基于基础功能中此函数 id = 0 时的实现方式，对于 id 不为 0 时也采用了类似的方式来判断和实现。

提高要求：评测技术之持久化存储

通过上面提到过的 write_to_longlasting 函数，在所有 API 中，当全局变量发生改变的时候调用此函数及时存入 json 文件中。

在 main 函数中，调用 read_from_json 函数，读取 json 文件至全局变量中。

```rust
// 保存全局变量至json文件
fn write_to_longlasting() -> std::io::Result<()> {
    let job_list: MutexGuard<Vec<Job>> = JOB_LIST.lock().unwrap();
    let users_list: MutexGuard<Vec<Users>> = USERS_LIST.lock().unwrap();
    let contest_list: MutexGuard<Vec<Contest>> = CONTESTS_LIST.lock().unwrap();
    let time: MutexGuard<Vec<Time>> = PROBLEM_SHORTEST.lock().unwrap();
    let user_submit: MutexGuard<Vec<Vec<UserSubmit>>> = USER_SUBMIT.lock().unwrap();
    let contest_ranklist: MutexGuard<Vec<ContestsUser>> = CONTESTS_RANKLIST.lock().unwrap();
    let longlasting: LongLasting = LongLasting {
        vec_job: job_list.clone(),
        vec_users: users_list.clone(),
        vec_contest: contest_list.clone(),
        vec_contest_user: contest_ranklist.clone(),
        vec_user_submit: user_submit.clone(),
        vec_time: time.clone()
    };
    let path: &Path = Path::new("./src/longlasting.json");
    fs::write(path, contents: serde_json::to_string_pretty(&longlasting).unwrap())
}
```

提高要求：评测方式之打包测试

打包测试的实现主要在函数 compile_and_run 中，每个测试点由 pack_check 记录是否应该 skip 这个点的运行，pack_add 记录每个点的分数是否计入了总分，保证打包测试时每个组需要都正确才能获得相应的分数，否则该组为 0 分。

```
646     // misc: packing
647     let mut pack_bool: bool = false;
648     let mut pack: Vec<Vec<i32>> = Vec::new();
649     let mut pack_check: Vec<bool> = vec![true; xun];
650     let mut pack_add: Vec<bool> = vec![false; xun];
651     if config.problems[p_tmp as usize].misc.packing.is_some() {
652         pack_bool = true;
653         pack = config.problems[p_tmp as usize].misc.packing.clone().unwrap();
654     }
```

提高要求：评测方式之 Special Judge

spj 的实现同样是在函数 compile_and_run 中，执行 spj 中的相应指令，按照要求返回 AC 和 WA，同时若输出的内容不符合要求以及 spj 过程出现问题，则返回 SPJ Error。

```
838             // 与标准答案进行special judge
839             else if config.problems[p_tmp as usize].ty == "spj".to_string() {
840                 let mut vec: Vec<String> = config.problems[p_tmp as usize].misc.special_judge.clone().unwrap();
841                 for k: usize in 0..vec.len() {
842                     if vec[k] == "%OUTPUT%".to_string() {
843                         vec[k] = out_location.clone();
844                     }
845                     if vec[k] == "%ANSWER%".to_string() {
846                         vec[k] = config.problems[p_tmp as usize].cases[i].answer_file.clone();
847                     }
848                 }
849                 let _out_file_location: String = path_dir.clone() + "/specialjudge";
850                 let _out_file: File = File::create(path: &_out_file_location).unwrap();
851                 let spj_result: ExitStatus = Command::new(program: &vec[0]) Command
852                         .args(&vec[1..vec.len()]) &mut Command
853                         .stdout(cfg: Stdio::from(_out_file)) &mut Command
854                         .spawn() Result<Child, Error>
855                         .unwrap() Child
856                         .wait() Result<ExitStatus, Error>
857                         .unwrap();
```

提高要求：评测方式之竞争得分

对于全局的运行最短时间，在函数 compile_and_run 中做相应的记录，并更新在全局变量 PROBLEM_SHORTEST 中，在获取排行榜的函数 GET /contests/{contestId}/ranklist 中实现。对于指定比赛的竞争得分，也在 GET /contests/{contestId}/ranklist 中实现，找到该场比赛每个题目的最优解（最短运行时间），然后计算分数。

```rust
// 竞争得分计算用户的用时
if _job_return.score == 100.0 * (1.0 - ratio) && config.problems[p_tmp as usize].ty == "dynamic_ranking".to_string() {
    let mut lock: MutexGuard<Vec<Time>> = PROBLEM_SHORTEST.lock().unwrap();
    let mut time_total: u128 = 0;
    for i: usize in 0.._job_return.cases.len() {
        time_total += _job_return.cases[i].time;
    }
    let mut find_location: usize = 0;
    for i: usize in 0..lock.len() {
        if lock[i].problem_id == config.problems[tmp as usize].id {
            find_location = i;
        }
    }
    if lock[find_location].time > time_total || lock[find_location].time == 0 {
        lock[find_location].time = time_total;
    }
    drop(lock);
}
```

```rust
                let mut ratio: f64 = 0.0;
                if config.problems[tmp as usize].misc.dynamic_ranking_ratio.is_some() {
                    ratio = config.problems[tmp as usize].misc.dynamic_ranking_ratio.unwrap();
                }
                if &score_rule == "latest" ||
                (config.problems[tmp as usize].misc.dynamic_ranking_ratio.is_some() &&
                lock[position as usize].submission[j].score == 100.0 * (1.0 - ratio)) {
                    if config.problems[tmp as usize].misc.dynamic_ranking_ratio.is_some() &&
                    lock[position as usize].submission[j].score == 100.0 * (1.0 - ratio) {
                        let _lock: MutexGuard<Vec<Time>> = PROBLEM_SHORTEST.lock().unwrap();
                        let mut lock_position: usize = 0;
                        for t in 0.._lock.len() {
                            if _lock[t].problem_id == tmp_id {
                                lock_position = t;
                            }
                        }
                        let rate: f64 = _lock[lock_position].time as f64 / lock[position as usize].submission[j].time as f64;
                        contest_return[i].scores[tmp as usize] = 100.0 * (1.0 - ratio) + 100.0 * ratio * rate;
                        drop(_lock);
                    }
                    else if config.problems[tmp as usize].misc.dynamic_ranking_ratio.is_some() {
                        continue;
```

四、 完成此作业感想

又是在一周多的时间完成这样一个大作业，与 wordle 相比，我感觉 OJ 大作业对于我的挑战更大，因为一开始面对这样一个作业的时候自己毫无头绪，刚开始前两天几乎都是在网上查资料但进展几乎没有。好在自己在后来与同学的交流中逐渐理解了整个框架以及一些逻辑上的问题，从而上手并还算顺利地完成了功能的实现。在调试代码的过程中，OJ 大作业也给予我更大的挑战，随着代码量的增加，调试也更加困难，但自己最终还是克服了这些问题，实现了 OJ 中的 API。最后，亲手实现 OJ，更加锻炼了我对于 rust 语言的使用，学习到了关于 http 请求、服务器、异步等的很多知识，收获很多。