# OJ 大作业报告

BY 张扬 致理书院 2023012455

## 1 程序结构和说明

系统分为两个 crate：server 服务端和 juder 评测端。两个 crate （仅）共享一份 common.rs，其中包括两者通讯的结构体定义，以及实现 RPC 的结构体和方法。两者具体通信方式将于“独立评测进程”部分详细描述。

### 1.1 web 服务端

服务端的代码结构如下：

```
src
├── api
│   ├── jobs.rs
│   └── users.rs
├── api.rs
├── callcc.rs
├── common.rs -> ../judger/src/common.rs
├── config.rs
├── lib.rs
├── main.rs
├── response.rs
├── service.rs
└── user.rs
```

#### 1.1.1 main.rs

服务器入口，读取配置文件，绑定请求处理函数并启动服务器。

#### 1.1.2 api/*.rs

用于相应 http 请求的各个函数，其中仅判断参数合法性，不涉及评测/用户等功能的实现。

#### 1.1.3 config.rs

读取配置文件，从 RawConfig （文件中所储存的结构）解析为 Config（服务及评测所需结构）。解析过程主要包含：

- 解析题目配置，将分散的输入输出文件统一复制至 /data 目录下，以便于后续映射到 judger 上
- 解析 SPJ 配置，将 SPJ 所依赖的文件以内嵌的方式打包至命令内，实现 SPJ 的远程调用（实现详见 common.rs 中的 RemoteCommand）
- 解析打包评测配置，将数据点打包解析为（judger 接口中的）依赖项和包总分（见 judger::Case 中的 dependency 和 pack_score 字段）

#### 1.1.4 callcc.rs

基础库，将 rust 和 actix 的异步部分封装为非惰性、仅调用一次的 call-with-current-continuation 和 call-with-composable-continue 形式，用于 web 请求的异步处理。

（事实上发现不用实现“阻塞”评测，所以实际不需要这部分，所有请求均可同步返回）

#### 1.1.5 common.rs

包括 server 和 judger 通讯的结构体定义，以及实现 RPC 调用的结构体和方法。

#### 1.1.6 response.rs

server 各个 api 在收到非法请求时的错误处理。

#### 1.1.7 service.rs

server 中实现了评测分发/收集结果。在 server 启动时，新建一个 channel 用于接受评测请求，单独的评测分发线程轮询 channel，收到评测任务时，启动评测容器进行评测。

#### 1.1.8 user.rs

用户相关功能的实现。维护了从 id 查找用户和从 name 查找用户的两个结构。

### 1.2 judger 评测端

代码结构如下：

```
src
├── bin
│   ├── sandbox.rs
│   └── test_config.rs
├── common.rs
├── fs.rs
├── lib.rs
└── main.rs
```

#### 1.2.1 main.rs

实现了评测的主要部分，包括编译，运行各个测试点，运行 spj 等。从标准输入获取数据，通过标准输出将评测信息传递给 server。

通过 wait4 获得子进程的内存使用信息，通过启动前后 Instant 测量子进程的 real time，通过定时 kill 的方式限制子进程的 real time。关于资源限制的具体方式与提高要求的“评测安全性”部分中详细描述。

#### 1.2.2 bin/sandbox.rs

用于运行待评测程序的环境。通过 setuid 限制访问权限，通过 setrlimit 限制资源使用。当启用 sandbox 选项时通过 libseccomp 限制系统调用（白名单模式）。

#### 1.2.3 bin/test_config.rs

生成测试用的输入，用于对 judger 单独测试。

#### 1.2.4 common.rs

同 server 的 common.rs

#### 1.2.5 fs.rs

对 judger 中的所有文件访问做了封装，便于统一调整文件结构，减少 typo，并提供了 Remmote Command 所需的 filelist 生成器。

# 2 OJ 主要功能说明

实现了基础要求中除了排行榜支持的部分。

实现了以下 api：

- POST /jobs
- GET /jobs
- GET /jobs/{jobid}
- PUT /jobs/{jobid}
- DELETE /jobs/{jobid}
- POST /users
- GET /users

# 3 提高要求实现方法

实现了提高要求中的：非阻塞评测、独立评测进程、资源限制进阶、评测安全性、打包测试、Special Judge。

## 3.1 非阻塞评测

service.rs 中有专门线程负责分发/收集评测任务。接收到评测请求后，将请求推入队列（channel），立即返回结果。

## 3.2 独立评测进程

对于每个评测请求，service 中的分发线程会启动独立的 docker 评测进程。

评测机应视为较为不稳定的系统，因此所有的提交记录/评测队列应储存于 server 端上（而不是评测机上），即 service 中的 chanel 属于 server 端的一部分，而 judger 容器则是独立的进程。

server 与 judger 之间通过文件夹映射传递输入输出文件，通过标准输入输出完成其余通信。具体的，对于测试点的输入输出文件，这一般是较大的文件，所以在处理 config 时将其集中到 data 目录，（如果需要分布式评测时）将文件分发到到评测机环境上，再通过 docker 的文件映射（只读地）映射到容器内；对于代码段、测试点信息、spj 信息等较小的文件，直接打包为 json 格式的字符串，通过标准输入传递给评测容器；容器通过标准输出将更新信息传递给 server。

docker 评测进程与 server 分离，可单独运行。judger/src/test_config.rs 可以生成一份测试 judger 的输入，例如：

```
{
  "code":{
    "language":{
      "name":"Rust",
      "file_name":"main.rs",
      "command":[
        "rustc",
        "-o",
        "%OUTPUT%",
        "%INPUT%"
```

```
      ]
    },
    "source":"fn main() {\n  println!(\"Hello, Wolrd!\");\n}\n"
  },
  "sandbox":true,
  "cases":[
    {
      "uid":0,
      "score":50.0,
      "time_limit":1000000,
      "memory_limit":67108864,
      "dependency":[],
      "pack_score":50.0
    },
    {
      "uid":1,
      "score":50.0,
      "time_limit":11000000,
      "memory_limit":67108864,
      "dependency":[],
      "pack_score":50.0
    }
  ],
  "checker":{
    "command":[
      {"String":"python3"},
      {"File":"#!/usr/bin/env python3\nimport sys\n\noutput = open(sys.argv[1],
\"r\").read().split('\\n')\nanswer = open(sys.argv[2], \"r\").read().split('\\
n')\n\ndef trim(s):\n    for i in range(len(s)):\n        s[i] = s[i].rstrip()\
n    while len(s) > 0 and s[-1] == \"\":\n        s.pop()\n\ntrim(output)\
ntrim(answer)\n\nif \"\\n\".join(output) == \"\\n\".join(answer):\n    print(\
"Accepted\")\nelse:\n    print(\"Wrong Answer\")\n"},
      {"String":"%OUTPUT%"},
      {"String":"%ANSWER%"}
    ]
  }
}
```

在 data 文件夹中，输入输出数据储存于 in<case.uid>, ans<case.uid> 中，例如：

```
data
├── ans0
├── ans1
├── in0
└── in1
```

可以通过如下命令单独运行评测：

```
cargo run --bin test_config | docker run --rm -i --network=none --cpuset-cpus=0 -
m=2G -v=./data:/work/a/data oj-judger:latest
```

### 3.3 评测安全性

评测安全性由多层次保障：

- 评测容器：在 docker 选项中关闭网络，限制使用某个固定 cpu 核心，限制内存不能超过某较大值（默认 2G，防止评测相互影响或影响 server）

- judger/main.rs：以 docker 中的 root 权限运行。通过 judger/sandbox 运行待评测代码。若子进程在启动后超过时间限制仍未退出，就用 SIGKILL 信号杀死子进程，避免子进程 real time 超时。

- judger/sandbox.rs：先以 docker 中的 root 权限运行。通过 setrlimit 限制（自身）的内存、system tim、user time，通过 seccomp 限制系统调用（默认关闭，通过在题目配置中添加 sandbox: true 或在 server 的环境变量中添加 OJ_SANDBOX=true 手动开启），随后通过 setgid 和 setuid 切换为权限较低的 test 用户，该用户无权访问评测相关数据，然后 exec 启动待评测的程序。

## 3.4 打包测试

通过 case 中的 dependency 和 pack_score 实现。即每个测试点依赖包中前一个（如有）测试点，每个包中最后一个测试点的 pack_score 为包 score 的总和，其余测试点 pack_score 为 0。总分为所有 pack_score 之和。

## 3.5 Special Judge

读取配置中的 special judge 命令，通过 RemoteCommand 的 pack 和 unpack 方法传递到 judger 上。

对于 spj 错误的处理，若 spj 命令没有被成功执行（如 python3 环境不存在），则认为是评测机环境配置问题，评测机返回 SystemError，并立即结束评测，该提交的 state 为 SystemError。若执行后结果为错误（返回值非 0，或超时/资源超限），则该测试点结果为 SPJ Error。对于 SPJ，其只应判断答案的正确性，即只应给出 Accepted 或非 Accepted（即 Wrong Answer）的结果，因此若其第一行输出为 Accepted 则结果为 Accepted 并获得全部分数，否则结果为 Wrong Answer 并不获得分数。