程序设计训练之 Rust 编程语言

第一次习题课

陈嘉杰

清华大学计算机科学与技术系

2022 年 8 月 29 日

1

小作业

第一次小作业

- 从标准输入读取：
  - 课上讲的：`std::io::stdin().read_line(&mut line)` 和 `line.trim().parse()`
  - OJ 上引入的第三方库 `text_io` 的 `text_io::read!()`
  - 类似 C++ 的 `std::cin`：根据类型来判断要如何解析输入数据
  - 对比 C 的 `scanf`：解析格式化字符串来判断数据格式
- 返回多个参数的函数：
  - C/C++: 传指针 `int exgcd(int a, int b, int *x, int *y)`
  - Rust: 利用元组 `fn exgcd(a: i64, b: i64) -> (i64, i64, i64)`

第二次小作业

使用迭代器遍历 Vec 的时候，如何修改 Vec 的内容?

```rust
// Task 5
let mut data = vec![100, 200, 300];
// region: Task 5
for i in &data {
    if i == 200 {
        data[0] += 200;
    }
}
// endregion: Task 5

// Expected: "Task 5: [300, 200, 300]"
println!("Task 5: {:?}", data);
```

可变迭代器

第一种场景：需要修改的元素就是正在遍历的元素，此时可以用可变迭代器：

```
for i in &mut data {
    *i += 200;
}
```

此时迭代器一定是有效的，因为 Vec 的元素个数并不会变。

第二种场景：需要修改的元素不是正在遍历的元素：

```
for i in &mut data {
    // cannot borrow `data` as mutable more than once at a time
    data[0] += 200;
}
```

怎么办?

解决方法

第一种办法，先记录下来要完成的修改，在循环结束后再修改：

```
let mut count = 0;
for i in &data {
    count += 1;
}
data[0] += 200 * count;
```

第二种办法，用下标访问数组：

```
for i in 0..data.len() {
    data[0] += 200;
}
```

枚举和模式匹配

- 使用枚举来表示等级

```
enum Grade {
    APlus,
    // ...
    F
}
```

- 使用模式匹配来转换 `&str` 为 `Grade`，再计算各个等级的绩点

```
match self {
    Self::APlus | Self::AMinus => 4.0,
    Self::BMinus => 3.0,
    // ...
}
```

第三次小作业

- 使用 BTreeMap/HashMap 统计出现次数 `*map.entry(key).or_insert(0) += 1`
  - 类比 Python 的 defaultdict，C++ 的 map
  - Wordle 中统计单词出现次数时可以用到
  - 也可以用数组来统计：`let count = [0; N]`
- 迭代器的高级用法
  - 课上讲的：`a.iter().map(|x| x * 2).collect()`，等价于

```
let ret = vec![];
for x in &a {
    ret.push(x * 2);
}
ret
```

  - 迭代器“拉链”：`a.iter().zip(b.iter()).map(|(a, b)| a + b).collect()`
  - 求和：`a.iter().sum()` 或 `a.iter().fold(0, |acc, x| acc + x)`

二叉搜索树

实现一个二叉搜索树：

Rust

```rust
struct Node {
    value: i32,
    left: Option<Box<Node>>,
    right: Option<Box<Node>>,
}

struct Tree {
    root: Option<Box<Node>>,
}
```

C/C++

```cpp
struct Node {
    int value;
    Node *left;
    Node *right;
}

struct Tree {
    Node *root;
}
```

二叉搜索树 - 插入

插入算法的思路：

1. 从根结点开始，按照要插入的值与结点的值的大小关系，进入左子树或右子树
2. 发现空结点，说明找到了要插入的位置，新建一个结点然后更新父结点的左儿子或右儿子

如何用 Rust 实现?

1. 记录当前结点：不希望把结点的所有权移走，所以要用借用；但是最终需要修改结点的内容，还需要是可变借用。
   - 类比 C++ 的 `T *`，既可以修改指向的 `T` 内容，也可以修改指针以指向另一个 `T`
   - Rust 中则是 `let mut current_node: &mut T`
2. 模式匹配：遇到 Option 的时候，需要判断它是 Some 还是 None

二叉搜索树 - 插入

常见错误：

```
// cannot move out of `self.root.0` which is behind a mutable reference
match self.root {
    None => {}
    Some(node) => {
    }
}
```

回忆一下，上课讲的如何在模式匹配时借用：

1. 直接对借用模式匹配：`match &self.root`
2. 模式匹配时使用 `ref`：`Some(ref node)`

还有一种方法：`Option::as_ref` 把 `&Option<T>` 转换为 `Option<&T>`

二叉搜索树 - 插入

常见错误：

```
match &mut self.root {
    None => {}
    Some(node) => {
        if node.value < value {
            // cannot assign twice to immutable variable `node`
            node = node.left.as_mut().unwrap();
        }
    }
}
```

这里的 node 是一个可变借用，但是 node 本身不可变，也就是不能指向新的结点。

解决方法：let mut current = node; 就是可变的可变借用了。

二叉搜索树 - 遍历、搜索和 Left 操作

中序遍历已经给出了提示，递归编写，通过传递一个 `&mut Vec<i32>` 参数来写入结果。

查询和插入的过程是类似的，只不过因为查询不需要修改结点内容，所以所有的可变借用都改成不可变借用即可。同时用一个 `Vec<Path>` 记录下查询的路径。

`Left` 操作：舍弃根结点以及右儿子结点，仅保留左儿子结点。核心矛盾在于，如果借用了左儿子，就无法更新根结点。因此可以先用 `Option::take` 转移所有权（原来的变成了 `None`），然后再修改根结点。

2

Wordle 大作业

基础要求

- 颜色判断逻辑：给出的答案一定是不能误导玩家的，也就是说不能把正确答案排除出去
  - 位置和字母都正确的，一定是 G（绿色）
  - 剩下的字母正确但是位置不正确的，先出现的是 Y（黄色），后出现的是 R（红色），根据数量判断
  - 其余字母不正确的是 R（红色）
  - 实现：利用小作业里学到的计算出现次数方法
- 随机打乱：
  - 真随机数是不可预测的，伪随机数是从种子中计算出来的
  - 每次随机一个数，可能还会出现重复
  - 随机打乱再顺序取，保证了随机性和不重复性
- 多关键字排序：
  - 如何实现按次数降序和字典序升序?
  - 第一种方法：自定义比较函数，当次数相等时，按字典序比较
  - 第二种方法：使用稳定排序算法，先按字典序排序，再按次数排序

基础要求

- 命令行参数匹配：
  - 使用 `clap` 或者 `structopt` 进行解析为结构体
  - 可选的参数可以用 `Option`
  - 指定默认值等等
- JSON
  - 类型：字符串，字典，数组，数字
  - 使用 `serde_json` 解析为结构体
  - 类似地，可能不出现的键值也用 `Option`
  - 需要注意的是，不同的库的标注是分别工作的，也就是说 `clap` 的默认值不会作用于 `serde_json` 的默认值
- 合并配置文件和命令行参数：第三方库 `merge`

提高要求

- 信息熵计算：
  - 信息熵：做出这个决策可以带来的信息量的期望
  - 算法思路是，对于每个词，计算后续可能出现的 3^5 种状态下（SSSSS），分别可能出现多少个词语，然后计算信息熵
  - 朴素办法：枚举猜测词，枚举 3^5 种状态，枚举答案，然后判断猜测单词与答案是否对应状态，然后计数
  - 优化：猜测词和答案唯一确定状态，因此只需要枚举猜测词，枚举答案，然后计算出状态，计数
  - 在 Release 模式下，初始情况下的猜测大概需要几秒
  - 按信息熵排序：`f64` 没有实现 `Ord`，是因为 `NaN` 不满足全序关系。因为信息熵不会出现 `NaN`，所以可以自定义比较函数来绕过这个问题。

3

OJ 大作业

相关知识

- HTTP：请求和响应，JSON
- 子进程：运行，等待，输入输出重定向
- 全局变量：Arc，Mutex

HTTP

- HTTP 分为请求和响应
  - 客户端发送 HTTP 请求
  - 服务端回复 HTTP 响应
- 例子：浏览器访问清华首页
  - 浏览器发送 HTTP 请求 `GET http://www.tsinghua.edu.cn`
  - 服务器回复 HTTP 响应，正文是 HTML
  - 浏览器根据 HTML 内容进行渲染，同时发送 HTTP 请求去获取图片、CSS、JavaScript 等文件

HTTP 请求

HTTP 请求有如下几个部分（其中所有的换行均为 CRLF，即 `\r\n`）：

1. 方法：如 `GET`、`POST`、`PUT`、`DELETE` 等
2. 地址：如 `http://www.baidu.com/s?wd=%E9%82%B1%E5%8B%87`，这里域名是 `www.baidu.com`，路径是 `/s`，参数是 `wd=%E9%82%B1%E5%8B%87`
3. 头部：一些键值对，记录了域名，正文类型，正文长度，Cookie 等信息
4. 正文：用于传输额外的数据

HTTP 请求例子

下面是一些实际的例子：

```
GET /s?wd=%E9%82%B1%E5%8B%87
Host: www.baidu.com
Accept: text/html
```

上面的请求向 `http://www.baidu.com/s?wd=%E9%82%B1%E5%8B%87` 发送了一个 GET 请求，没有请求的正文。需要注意的是，实际的 HTTP 请求中 `GET` 后面的路径不包括域名，域名会在头部的 `Host` 处出现。

HTTP 请求例子

```
POST /users
Host: www.example.com
Content-Type: application/json
Content-Length: 16

{"name": "user"}
```

上面的请求向 `http://www.example.com/users` 发送了一个 POST 请求，请求的正文是一个 JSON。

HTTP 请求方法

HTTP 请求的方法，通常有如下两类：

1. GET：表示获取数据
2. POST/PUT/DELETE 等：表示提交/更新/删除数据等操作

一般来说，GET 表示的是获取数据，比如 HTML，CSS，用户数据等等。GET 请求一般不附带正文，而是在 URL 上附带参数，例如：

```
GET /s?wd=%E9%82%B1%E5%8B%87
```

表示参数 wd= 邱勇。常用于浏览器获取 HTML 等静态资源，获取用户信息等动态资源。

而 POST/PUT/DELETE 等请求一般在正文附带数。一般用于有副作用的请求，例如登录，注册，创建等等。由于 URL 通常会记录在日志当中，所以密码等私密的参数都会通过正文传输。

HTTP 响应

服务器收到请求以后，经过一些处理，会发送响应给客户端。一个 HTTP 响应有如下几个部分：

1. 状态码：常见的有 200 OK，404 Not Found，500 Internal Server Error 等
2. 头部：一些键值对
3. 正文：用于传输额外的数据

HTTP 响应例子

```
HTTP 200 OK
Content-Type: application/json
Content-Length: 25

{"id": 0, "name": "user"}
```

上面的响应的状态码是 200 OK，正文是一个 JSON。

HTTP 小结

简单总结一下，HTTP 实际上就是客户端向服务端发送一个请求，服务端向客户端返回一个响应；请求包括方法、地址、头部和正文；响应包括状态码、头部和正文。

你要实现的 OJ 系统会运行一个 HTTP 服务端，监听在本机的一个端口上。它会处理来自客户端的 HTTP 请求，并提供响应。

本次的 OJ 系统不涉及跨主机访问，因此所有的监听地址都只需要设置为 127.0.0.1（仅本机可访问）。同一时间，通常只能有一个进程监听在同一个地址的同一个端口上。因此，如果你运行程序的时候出现 `Address already in use` 的报错，那就说明之前运行的程序还没有退出。

HTTP 服务器开发

在模板仓库中，已经用 actix-web 启动了一个简单的 HTTP 服务器：

```rust
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .wrap(Logger::default())
            .route("/hello", web::get().to(|| async { "Hello World!" }))
            .service(greet)
    })
    .bind(("127.0.0.1", 12345))?
    .run()
    .await
}
```

HttpServer

```rust
HttpServer::new(|| {
    // ...
})
.bind(("127.0.0.1", 12345))?
.run()
.await
```

启动了一个 HTTP 服务器，监听在 127.0.0.1 地址的 12345 端口上。

App

```
App::new()
    .wrap(Logger::default())
    .route("/hello", web::get().to(|| async { "Hello World!" }))
    .service(greet)
```

启用了请求日志，然后实现了 API `GET /hello`，它的行为是直接返回一个 `Hello World!` 的响应。同时引入了 `greet`，它的定义在前面：

```
#[get("/hello/{name}")]
async fn greet(name: web::Path<String>) -> impl Responder {
    log::info!(target: "greet_handler", "Greeting {}", name);
    format!("Hello {name}!")
}
```

GET

```rust
#[get("/hello/{name}")]
async fn greet(name: web::Path<String>) -> impl Responder {
    log::info!(target: "greet_handler", "Greeting {}", name);
    format!("Hello {name}!")
}
```

`#[get("/hello/{name}")]` 表示它实现了一个 `GET /hello/{name}` 的 API，其中 `{name}` 部分会解析到 `name: web::Path<String>` 参数上，也就是说，如果访问了 `/hello/abcde`，那么这个函数会被调用，并且 `name` 会等于 `abcde`。

接着，代码打印了日志，然后返回了 `format!("Hello {name}")` 作为响应的正文。

POST

如果想要实现一个 POST 方法的 API，只需要修改 `#[get()]` 为 `#[post()]`。请求正文用 JSON 格式传输参数是很常见的，因此框架可以自动进行 JSON 解析，只需要结构体标注了 `#[derive(Deserialize)]`：

```
#[derive(Deserialize)]
struct PostJob {
    // ...
}

#[post("/jobs")]
async fn post_jobs(body: web::Json<PostJob>) -> impl Responder {
    // ...
}
```

POST

同时不要忘记在 `main` 函数中添加：

```
App::new()
    .service(greet)
    // Add this
    .service(post_jobs)
```

否则访问 API 的时候会出现 `404 Not Found` 状态码。

JSON 响应

类似地，响应中也经常会使用 JSON 格式，可以让框架自动帮你把结构体转换为 JSON，只需要结构体标注了 `#[derive(Serialize)]`：

```
#[derive(Serialize)]
struct Job {
    // ...
}

#[post("/jobs")]
async fn post_jobs(body: web::Json<PostJob>) -> impl Responder {
    // ...
    HttpResponse::Ok().json(Job {
        // ...
    })
}
```

响应状态码

有时候，如果出现了错误，你会希望设置响应的状态码：

```
#[post("/jobs")]
async fn post_jobs(body: web::Json<PostJob>) -> impl Responder {
    // ...
    HttpResponse::BadRequest().json(Error {
        // ...
    })
}
```

这次大作业中，就针对各种错误情况所需要汇报的错误信息和响应状态码进行了规定。

更进一步，可以进行一些处理，自动把 `Result` 通过 `?` 操作符转换为 API 文档中所需要的格式，可以大大简化错误处理的代码。请同学们参考 `actix-web` 的文档学习。

访问配置

有时候，你在实现 API 的时候，可能会想要访问配置，此时可以利用 `web::Data` 来传递：

```
#[derive(Clone)]
struct Config {
    // ...
}

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

访问配置

这样，只需要在 `main` 函数中设定 `web::Data`：

```
fn main() {
    // ...
    App::new()
        .app_data(web::Data::new(config.clone()))
}
```

这样，API 处理函数就会自动得到一份配置。这个方法也经常用来传递数据库的连接对象。需要注意的是，使用这样的方法，同样类型的参数同时只能有一个。

调试工具

下面是一些工具，能帮助你调试 HTTP 服务器：

- Postman
- VSCode REST Client
- httpie

使用这些工具，可以很方便地发起 HTTP 请求，并观察 HTTP 响应，看是否符合 API 文档的要求。

VSCode REST Client

下面讲解 VSCode REST Client 插件的基本使用方法。首先在 VSCode 中安装插件，然后新建一个文件 post_jobs.http，填入如下内容：

```
POST http://127.0.0.1:12345/jobs HTTP/1.1
content-type: application/json

{
  "source_code": "fn main() { println!(\"Hello, world!\"); }"
}
```

上面的内容表示要向 http://127.0.0.1:12345/jobs 发送一个 POST 请求，正文是一个 JSON。此时 POST 上方会显示 Send Request 提示，按下它，VSCode REST Client 插件就会发送请求，并显示出响应的内容。

VSCode REST Client

可以得到类似下面的输出：

```
POST http://127.0.0.1:12345/jobs HTTP/1.1
content-type: application/json

{
  "source_code": "fn main() { }",
  "language": "Rust",
  "user_id": 0,
  "contest_id": 0,
  "problem_id": 0
}
```

```
HTTP/1.1 200 OK
content-length: 123
connection: close
content-type: application/json

{
    "id": 0,
    // ...
}
```

更详细的用法，可以阅读插件的文档。

全局变量

在 OJ 系统，需要维护当前的评测任务列表，如果没有实现数据库支持，可以采用全局变量来保存评测任务列表。但是，全局变量引入了新的问题，在多线程的场景下，如何保证它同时只有一个可变借用呢?

后续课程上会讲到这个问题，这里提前预告一下，解决方法是使用 `Arc<Mutex<T>>`，其中 `Mutex` 是互斥锁，提供了线程安全的内部可变性，`Arc` 是线程安全的引用计数器。例如要用 `Vec<Job>` 保存所有评测任务的话，可以用 `Arc<Mutex<Vec<Job>>>` 类型来定义一个全局变量，比如:

```
use std::sync::{Arc, Mutex};

struct Job;
pub static JOB_LIST: Arc<Mutex<Vec<Job>>> =
    Arc::new(Mutex::new(Vec::new()));
```

但是这样的代码不能通过编译。

全局变量 + lazy_static

但是，上面的代码不能通过编译，因为 `Arc::new()` 函数无法用于全局变量的初始化。

因此，我们需要引入第三方库 `lazy_static` 来实现这个事情，比如：

```
use std::sync::{Arc, Mutex};
use lazy_static::lazy_static;

struct Job;

lazy_static! {
    static ref JOB_LIST: Arc<Mutex<Vec<Job>>> =
        Arc::new(Mutex::new(Vec::new()));
}
```

这样就可以通过编译了。

互斥锁

那么，在读取或者修改的时候，需要首先获取互斥锁，然后就可以对内部的 `Vec<Job>` 进行操作了：

```
let mut lock = JOB_LIST.lock().unwrap();
lock.push(Job);
```

为了保证多线程安全，互斥锁保证同时只能有一个线程获取，保证了数据在多线程场景下同时只有一个地方拥有可变借用。当 `lock` 结束生命周期时，锁会自动释放。编写代码的时候，需要小心出现死锁的情况。

死锁

为了防止死锁，这里介绍几种常见的死锁情况：

1. 获取锁顺序

```rust
let lock_a = A.lock().unwrap();
let lock_b = B.lock().unwrap();
```

```rust
let lock_b = B.lock().unwrap();
let lock_a = A.lock().unwrap();
```

获取锁的顺序颠倒，如果上面两侧代码同时运行，可能会出现左侧代码获取了 A 的互斥锁，右侧代码获取了 B 的互斥锁的情况，此时两侧代码都会阻塞在第二次 `lock` 上。

死锁

❷ 恐慌

如果获取了锁，但是在释放之前出现了恐慌，会导致锁进入 Poisoned 状态：

```
let lock = LOCK.lock().unwrap();
panic!();
```

之后如果其他线程尝试获取这个锁，都会失败。

建议使用互斥锁的时候，尽量缩小持有锁的时间，并且使用的时候尽量不要用 `unwrap()` 等可能出现恐慌的错误处理方法。

死锁

❸ 模式匹配时，注意不要在匹配内部再次获取锁：

```
match LOCK.lock().unwrap() {
    XX(yy) => {
        // dead lock
        let lock = LOCK.lock().unwrap();
    }
}

if let XX(yy) = LOCK.lock().unwrap() { } else {
    // dead lock
    let lock = LOCK.lock().unwrap();
}
```

如何避免死锁

保证获取锁的代码区域（临界区，Critical Section）尽量小，并且不会出现恐慌：

```
let mut lock = JOB_LIST.lock().unwrap();
lock.push(Job);
drop(lock);

// later
let mut lock = JOB_LIST.lock().unwrap();
lock.pop();
drop(lock);
```

竞争条件

但是，临界区不一定是越小越好：

```
let mut lock = JOB_LIST.lock().unwrap();
// find maximum job id
let max_job_id = ???;
drop(lock);

// later
let mut lock = JOB_LIST.lock().unwrap();
lock.push(Job {
    id: max_job_id + 1
});
drop(lock);
```

可能会导致任务 ID 冲突。

JSON 序列化与反序列化

在两个大作业中都出现了 JSON 的序列化和反序列化。你可以使用 `serde_json` 来实现 JSON 的序列化（Serialize）与反序列化（Deserialize），实现代码中的值与 JSON 值的双向转换：

```rust
#[derive(Serialize, Deserialize)]
struct Config {
    random: Option<bool>,
    difficult: Option<bool>,
    stats: Option<bool>,
    day: Option<usize>,
    seed: Option<u64>,
    // ...
}
```

```json
{
  "random": true,
  "difficult": false,
  "stats": true,
  "day": 5,
  "seed": 20220123,
  // ...
}
```

在 OJ 大作业中，涉及到 JSON 的地方会更多，因此这里介绍一下 `serde_json` 的进阶用法。

枚举与 JSON 的转换

在 OJ 大作业中，为了表示评测任务的状态，会出现枚举：

```
{ "state": "Queueing" }
```

此时，我们可以用枚举类型来表示 `state`：

```
#[derive(Serialize, Deserialize)]
enum State { Queueing, Running, Finished }

#[derive(Serialize, Deserialize)]
struct Job {
    // ...
    state: State,
}
```

枚举与 JSON 的转换

但有时候，并不能直接对应到 Rust 代码，因为出现了空格：

```
{ "result": "Compilation Error" }
```

此时，可以用 `#[serde(rename = "")]`：

```rust
#[derive(Serialize, Deserialize)]
enum JudgeResult {
    Waiting,
    #[serde(rename = "Compilation Error")]
    CompilationError,
}
```

枚举与 JSON 的转换

有时候，也会出现命名方式的不兼容：CamelCase vs snake_case：

```
{ "type": "standard" }
```

此时，可以用 `#[serde(rename_all = "snake_case")]` 批量重命名：

```
#[derive(Serialize, Deserialize)]
#[serde(rename_all = "snake_case")]
enum ProblemType {
    Standard,
    Strict,
    DynamicRanking
}
```

JSON 与 Rust 关键字冲突

还有一种情况，JSON 的键正好对应了 Rust 中的关键字：

```
{ "type": "standard" }
```

```
#[derive(Serialize, Deserialize)]
struct Problem {
    // expected identifier, found keyword `type`
    type: ProblemType,
}
```

此时有两种方法可以解决：

1. 把 `type` 改为 `r#type`，强制定义一个名为 `type` 的成员
2. 把 `type` 改为 `ty`，然后加上标注 `#[serde(rename = "type")]`

如何理解 JSON 比较结果

在 OJ 大作业中，自动测试程序会把 OJ 响应中的 JSON 与预期结果进行比对。于是，你需要学会从错误信息中判断错误出在了什么地方：

例如：

```
json atoms at path ".result" are not equal:
    expected:
        "Wrong Answer"
    actual:
        "Time Limit Exceeded"
```

如何理解 JSON 比较结果

错误信息：

```
json atoms at path ".result" are not equal:
    expected:
        "Wrong Answer"
    actual:
        "Time Limit Exceeded"
```

表示的是：

实际：

```
{ "result": "Time Limit Exceeded" }
```

期望：

```
{ "result": "Wrong Answer" }
```

如何理解 JSON 比较结果

错误信息：

```
json atoms at path ".cases[1].result" are not equal:
    expected:
        "Wrong Answer"
    actual:
        "Time Limit Exceeded"
```

表示的是：

实际：

```
{ "cases": [
    {},
    {"result": "Time Limit Exceeded"}
] }
```

期望：

```
{ "cases": [
    {},
    {"result": "Wrong Answer"}
] }
```

如何理解 JSON 比较结果

错误信息：

```
json atom at path ".result" is missing from actual
```

表示的是：

实际：

```
{ }
```

期望：

```
{ "result": "Accepted" }
```

子进程

在 OJ 系统中，编译可执行文件以及评测的时候，都需要运行程序，并且按照需求传递命令行参数，或进行输入输出的重定向。Rust 标准库提供了 `std::process::Command` 来运行程序。

首先最简单的用法是，直接运行一个程序，等待其运行结束，并获得它的结束状态：

```rust
// Taken from rust docs
use std::process::Command;
let status = Command::new("/bin/cat")
                      .arg("file.txt")
                      .status()?;

println!("process finished with: {status}");
```

上面的代码运行了 `/bin/cat file.txt` 命令，等待其运行完成并获取它的结束状态。

传递参数 + 输入输出重定向

进一步，我们可能想要传递多个参数，或者对标准输入输出进行重定向：

```
let in_file = File::open("wordle.in")?;
let out_file = File::create("wordle.out")?;
let status = Command::new("wordle")
                     .args(["--random", "-t"])
                     .stdin(Stdio::from(in_file))
                     .stdout(Stdio::from(out_file))
                     .stderr(Stdio::null())
                     .status()?;
```

上面的代码运行了 `wordle --random -t` 命令，将标准输入重定向为 `wordle.in`，标准输出重定向为 `wordle.out`，丢弃它的标准错误输出，等待其运行完成并获取它的结束状态。

上面的代码会一直等待程序运行完成，如果想要设置超时，可以使用第三方库 `wait_timeout`。

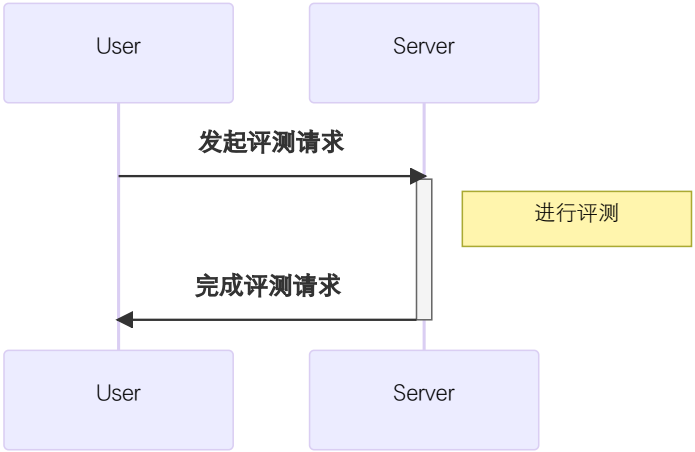
OJ 评测流程

以 A+B 题目为例，OJ 系统会做下面几件事情来进行评测：

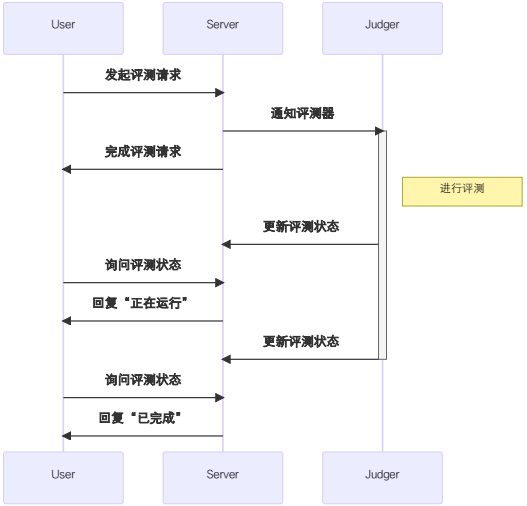
1. 创建一个评测临时目录，用于保存评测时使用的源代码、可执行文件和输出文件，下面用 TMPDIR 来指代这一步创建的临时目录；
2. 将源代码保存到临时目录中，如 `TMPDIR/main.rs`；
3. 将源代码编译成可执行文件（Windows 下运行还需要保证后缀名为 `.exe`），如 `rustc -C opt-level=2 -o TMPDIR/test.exe TMPDIR/main.rs`；
4. 编译成功后，按照顺序对数据点进行评测：运行 `TMPDIR/test.exe < ./data/aplusb/1.in > TMPDIR/test.out`，然后再比对 `TMPDIR/test.out` 与 `./data/aplusb/1.ans` 的内容；
5. 完成评测后，删除评测临时目录中的所有内容。

为了保证多个评测可以同时进行，需要保证临时目录不会冲突。也请注意不要把临时目录中的文件提交到仓库中。

OJ 阻塞评测流程图



OJ 非阻塞评测流程图


User
Server
Judger
发起评测请求
通知评测器
完成评测请求
进行评测
更新评测状态
询问评测状态
回复“正在运行”
更新评测状态
询问评测状态
回复“已完成”
User
Server
Judger

自动测试

基础要求：

```
cargo test --test basic_requirements -- --test-threads=1
```

提高要求（部分）：

```
cargo test --test advanced_requirements -- --test-threads=1
```

自动测试配置

每个自动测试都有两个文件：

1. `[case_name].config.json`：OJ 的配置文件，通过参数 `-c [case_name].config.json` 传递给 OJ
2. `[case_name].data.json`：评测流程，包括自动测试会发起的 HTTP 请求以及预期的响应。

自动测试流程

自动测试的每个测例会进行如下的操作：

1. 结束当前正在运行的 OJ
2. 运行 OJ：`oj -c [case_name].config.json --flush-data`
3. 按照 `[case_name].data.json` 的顺序发送 HTTP 请求，将 HTTP 响应与答案进行比较（仅比较答案中出现的字段）
4. 结束 OJ

`[case_name].data.json` 中可能出现的键：

1. `request`：发送的 HTTP 请求
2. `response`：预期的 HTTP 响应
3. `poll_for_job`：在非阻塞评测时，轮询任务状态
4. `restart_server`：重启 OJ，用于测试持久化功能

自动测试日志

自动测试运行每个测试点后，会生成以下的文件：

- `[case_name].stdout/stderr`：OJ 程序的标准输出和标准错误。你可以在代码中添加打印语句，然后结合输出内容来调试代码。
- `[case_name].http`：测试过程中发送的 HTTP 请求和收到的响应。调试时，你可以先自己启动一个 OJ 服务端（`cargo run`），然后用 VSCode REST Client 来手动发送这些 HTTP 请求，并观察响应。