oj 系统

项目介绍

OJ = Online Judge

正常刷题：用户可以选择题目，然后可以在其中编写代码，可以判断题目的正确性；
竞赛栏目：可以创建一个竞赛栏目，里面的题目可以是题库里面的题，也可以是出题人创建的私有题目，出题人具有将题设置公共还是私有的权限；竞赛具有封榜、滚榜的功能

难点：判题逻辑

用于在线评测编程题目代码的系统，能够根据用户提交的代码、出题人预先设置题目输入和输出用例，进行编译代码、运行代码、判断代码运行结果是否正确

判题系统将作为一个开放API，便于开发者开发自己的OJ

OJ 基本概念

AC 表示题目通过：结果正确
题目限制：时间限制，空间限制
题目介绍，题目输入，题目输出，题目输入用例，题目输出样例

关于代码的限制：不能随便让用户引入包，随便遍历、暴力破解，需要使用合适的算法。（安全性）
提交之后，会生成一个提交记录，有运行的结果以及运行信息（时间限制、运行限制）

关于题目的类型：基本题，spj题目，交互题
基本题：判断用户算法输出与给出输出是否一致
spj题目：输出不是唯一的，
交互题：一问一答式的题目

项目流程

1. 项目介绍、项目调研、需求分析
2. 确定核心业务流程
3. 项目要做的功能（功能模块）
4. 技术选型（技术预研）
5. 项目初始化
6. 项目开发
7. 测试
8. 优化
9. 代码提交、代码审核
10. 产品验收
11. 上线

写文档、持续调研、持续记录总结

现有系统调研

1. https://github.com/HimitZH/HOJ
2. https://github.com/QingdaoU/OnlineJudge?tab=readme-ov-file
3. https://github.com/hzxie/voj
4. https://github.com/vfleaking/uoj
5. https://github.com/zhblue/hustoj

实现核心

1. 权限校验
   谁能提交代码，谁不能提交代码
2. 代码沙箱（安全沙箱）
   用户代码藏毒：写个木马文件
   沙箱：隔离的、安全的环境，用户的代码不会影响到沙箱之外的系统的运行
   资源分配：系统的内存 2G，用户疯狂占用资源占满内存不行。要限制用户程序的占用资源
3. 判题规则
   题目用例的比对，结果的验证
4. 任务调度
   服务器资源优先，用户要排队，按照顺序去依次执行判题，而不是直接拒绝

判题服务：需要获取题目信息、预计的输入输出结果，返回给主业务后端：用户的答案是否正确
代码沙箱：只负责运行代码，给出结果，不管什么结果是正确的

功能

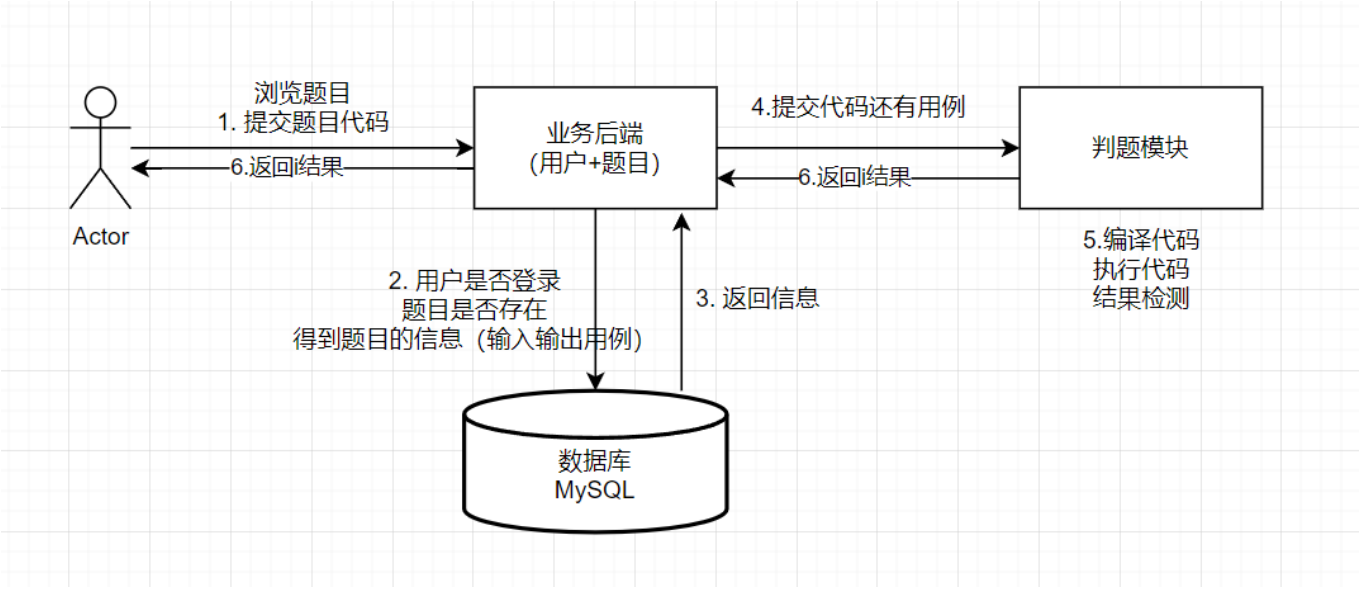
1. 题目模块
   a. 创建题目（管理员）
   b. 删除题目（管理员）
   c. 修改题目（管理员）
   d. 搜索题目（用户）
   e. 在线做题
   f. 提交题目代码
2. 用户模块
   a. 注册
   b. 登录

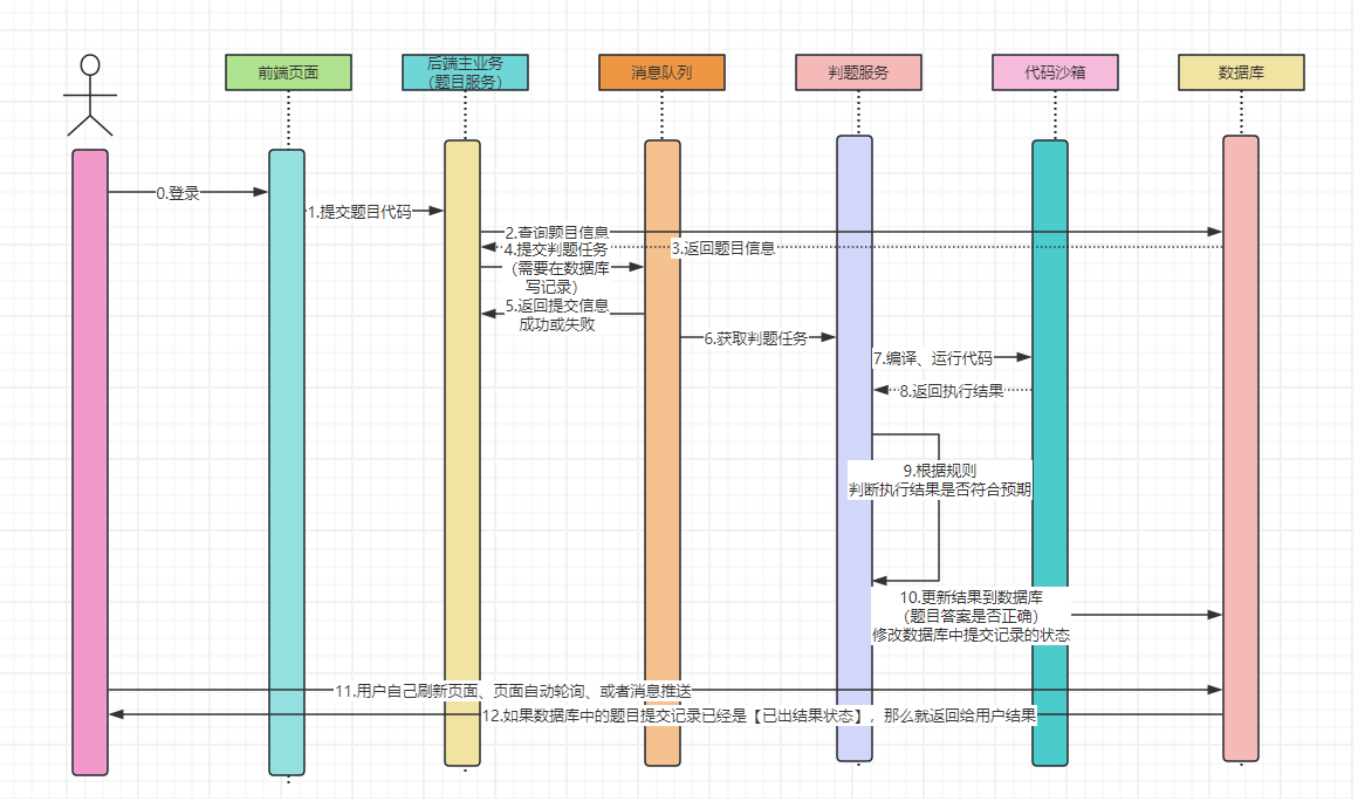
3. 判题模块
   a. 提交判题（结果是否正确与错误）
   b. 错误处理（内存溢出、安全性、超时）
   c. 代码沙箱（安全沙箱）
   d. 开放接口（提供一个独立的新服务）

项目扩展思路

1. 多种语言评测
2. remote judge
3. 完善的评测功能：普通测评、特殊测评、交互测评、在线自测、子任务分组评测、文件IO、
4. 统计分析用户判题记录
5. 权限校验

核心业务流程

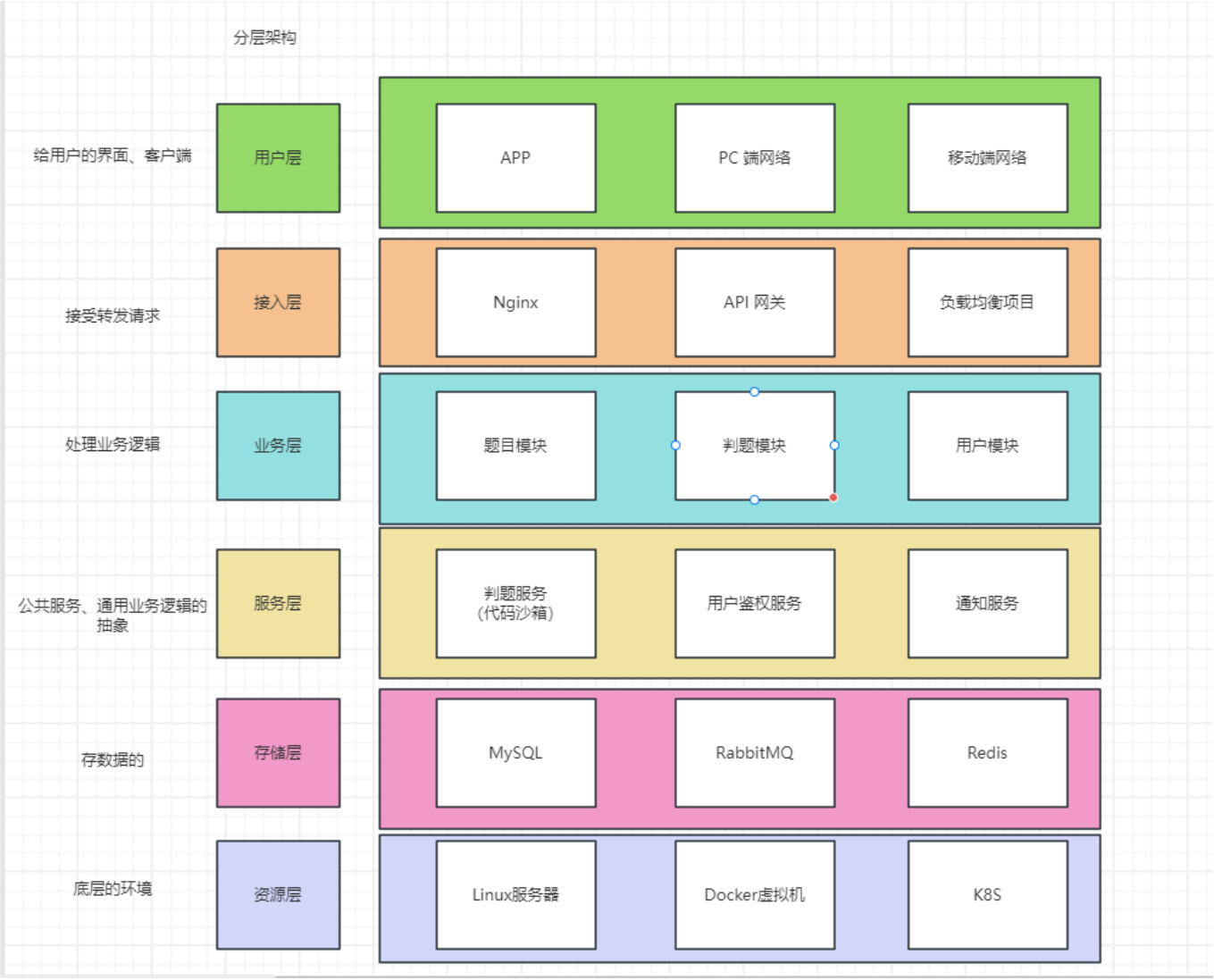




技术选型

前端：Vue3，Arco Design 组件库、手写项目模板、在线代码编辑器、在线文档浏览
Java 进程控制、Java 安全管理器、部分JVM 知识
虚拟机（云服务器）、Docker（代码沙箱实现）
Spring Cloud 微服务、消息队列

架构设计



主流的 OJ 系统实现方案

开发原则：能用别人现成的，就不要自己写

1）用现成的 OJ 系统，比如 judge0
https://github.com/judge0/judge0
自己用源码来部属、公有云、私有云

2）用现成的判题API（judge0）
https://ce.judge0.com/ https://rapidapi.com/hub

API 的作用：接受代码、返回结果

先注册
再开通订阅
测试language接口
测试执行代码接口 submissions

3）自主开发

4）把 AI 当作代码沙箱

5）移花接木。可以通过模拟浏览器的方式，用别人的OJ来帮你判题。比如无头浏览器，像人一样去再别人的项目中提交代码，并获取结果

开始项目

前端项目初始化

首先 NodeJS 版本：v18.16.0 或者 16
这里我暂时使用v20，如果后续有问题再换
命令 `node -v`
npm 版本: 9.5.1
这里我使用 10.2.4

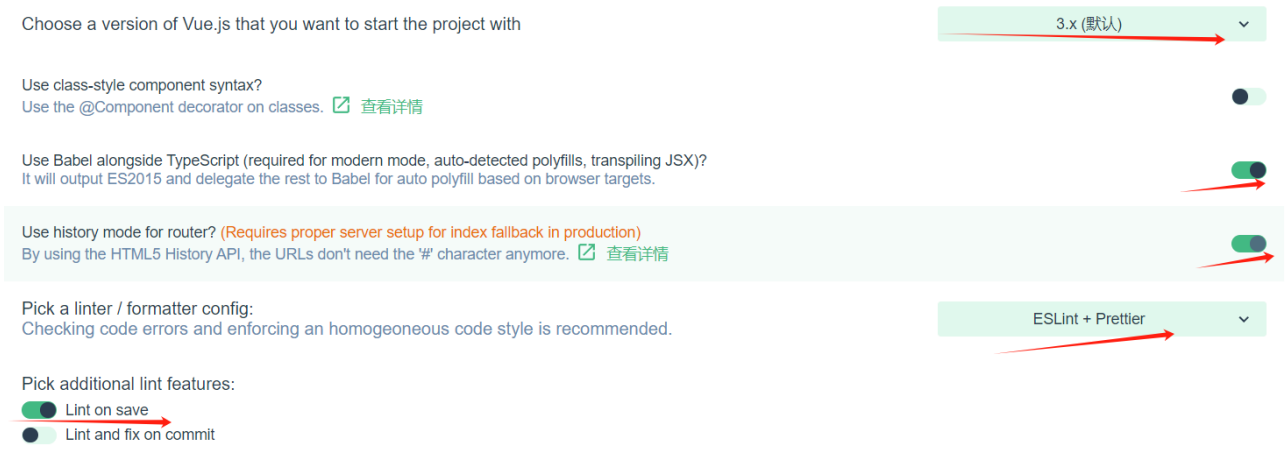
切换和管理node版本的工具：https://github.com/nvm-sh/nvm

初始化
vue-cli 脚手架

版本：`vue -V` 5.0.8

首先：创建项目手动配置；选择：
Babel,TS,Router,Vuex,Linter/Formatter；

配置：



其次：下载安装WebStorm

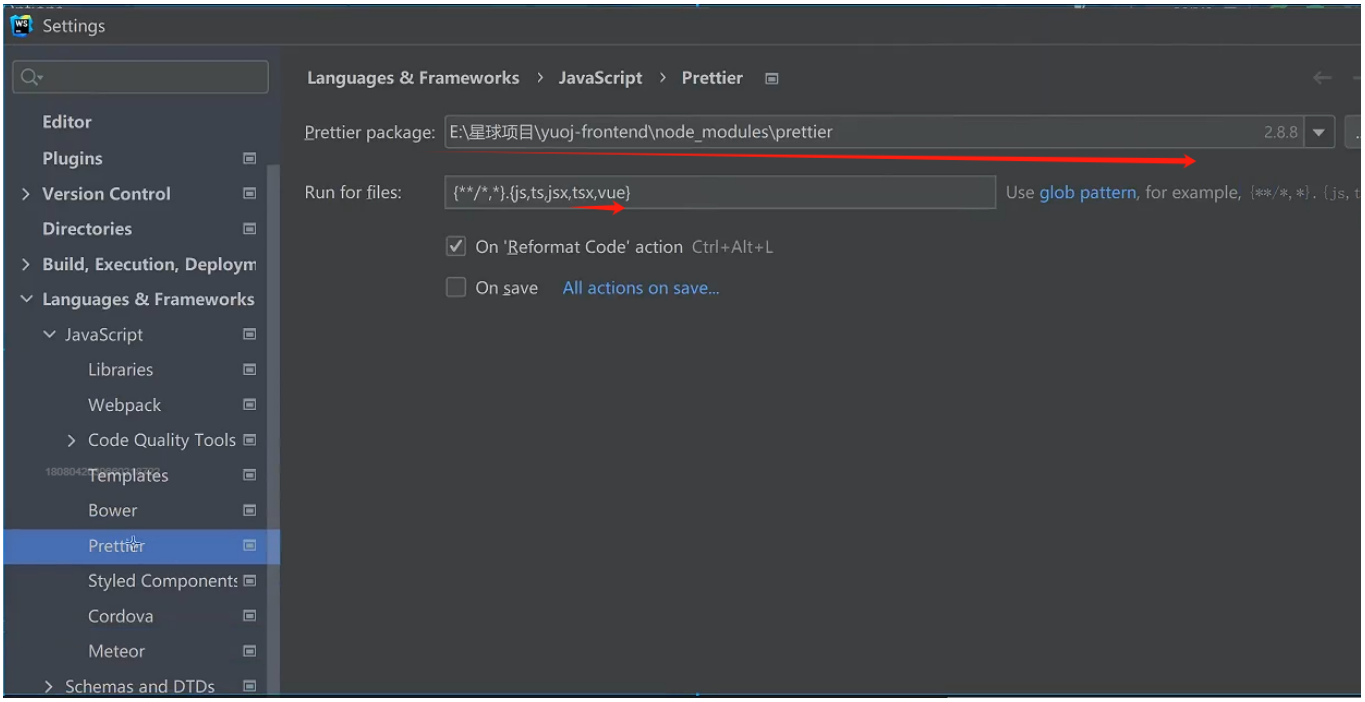
破解：

https://blog.csdn.net/weixin_50670076/article/details/136444408

前端工程化配置

脚手架已经帮我们配置了代码美化、自动校验、格式化插件等，无需自行配置

需要再webstorm 里开启代码美化插件 另外还需要搜eslint



在vue文件中执行格式化快捷键（ctrl+alt+L），不报错，表示配置工程化成功
脚手架自动整合了 vue-router

引入组件

组件库：arco.design
快速上手：https://arco.design/vue/docs/start
执行安装：在vue ui 上搜索依赖 arco.design/web-vue

改变 main.ts

```
import { createApp } from "vue";
import App from "./App.vue";
import ArcoVue from "@arco-design/web-vue";
import "@arco-design/web-vue/dist/arco.css";
import router from "./router";
import store from "./store";

createApp(App).use(store).use(ArcoVue).use(router).
mount("#app");
```

引入一个组件，如果显示，则表示引入成功
比如

```
  <a-calendar v-model="value" />
```

项目通用布局

新建一个布局，在app.vue中引用布局

问题：scoped 用处？ setup 的作用?
[写layout] 特点：边看别调整（需要前端css样例知识）渐变样例

首先上中下编排好，然后再填充内容
Navbar 编写
这里写 layout 要注意首先把layout抽象出来作为一个组件，这个组件共同的部分是 header 和 footer。然后就是 Narbar 也要抽象出来作为一个公共组件（使用acro.design的menu导航栏）。最后抽象Narbar里面的内容，就是动态地改变，利用router机制

实现通用菜单

把菜单上路由改成读取路由文件，实现更通用的动态配置

1）提取通用路由文件
2）菜单组件读取路由，动态渲染菜单项
3）绑定跳转事件
4）同步路由到菜单项

首先点击菜单项=》跳转更新路由=》更新路由后，同步去更新菜单栏的高亮状态

如何实现通用菜单?

1. 首先修改 a-menu-item 将其设置为 v-for 结构，然后其 `:key` 设置为路由的路径`path`，然后导航栏的名字就设置为路由的名字 `name`
2. 设置点击事件跳转，利用router.push
3. 设置刷新高亮处不变，利用 afterEach

全局状态管理

https://vuex.vuejs.org/
所有页面全局共享的变量，而不是局限在某一个页面中
适合作为全局状态的数据
本质上：提供了一套增删改查全局变量的API，不过多了些功能

state:存储状态信息，比如用户信息
mutation：定义了对变量进行增删改查的方法
action：提交的是 mutation，而不是直接变更状态。-Action 可以包含任意异步操作。

使用组件然后显示user 信息；注意username处在右边的配置；
这里使用grid布局

然后定义user模块，首先mutation里面有更新user，action里面是调用mutation的更新 一定要注意写法！！！

```
    as StoreOptions<any>; // 没有设置类型的时候要加
```

获取状态变量使用 useStore 即可
setTimeout 适合用来测试

全局权限管理

能够以一套通用的机制，去定义哪些页面需要哪些权限
`meta` \

思路：

1. 再路由配置文件，定义某个路由的访问权限
2. 再全局页面组件中，绑定一个全局路由监听。每次访问页面时，根据用户要访问页面的路由信息，先判断用户是否有对应权限
3. 如果有，跳转；否则拦截到 404

1）定义权限

```
const ACCESS_ENUM = {
  NOT_LOGIN: "notLogin",
  USER: "user",
  ADMIN: "admin",
};

export default ACCESS_ENUM;
```

2）定义一个公用的权限校验方法
为什么？因为菜单组件中要判断权限、权限拦截也要用到权限判断功能，所以将其抽离成公共方法

```
/**
 * 检查权限（判断当前某个用户具有某些权限）
 * @param loginUser 当前登录用户
 * @param needAccess 需要有的权限
 */
import ACCESS_ENUM from "@/access/accessEnum";

const checkAccess = (loginUser: any, needAccess =
ACCESS_ENUM.NOT_LOGIN) => {
  // 获取当前用户具有的权限（如果没有 loginUser，则表示
未登录）
  const loginUserAccess = loginUser?.userRole ??
ACCESS_ENUM.NOT_LOGIN;
  if (needAccess === ACCESS_ENUM.NOT_LOGIN) {
    return true;
  }

  // 如果用户登录才能访问
  if (needAccess === ACCESS_ENUM.USER) {
    // 如果用户没登录，那么表示无权限
    if (loginUserAccess === ACCESS_ENUM.NOT_LOGIN)
{
      return false;
    }
  }

  // 如果需要管理原权限
  if (needAccess === ACCESS_ENUM.ADMIN) {
    if (loginUserAccess !== ACCESS_ENUM.ADMIN) {
```

```
        return false;
      }
    }

    return true;
  };
```

单独定义一个文件，控制路由的显隐
后端项目初始化
前端代码自动生成

2024.7.8

优化布局

1. 修改footer的bug
2. 后端项目初始化（万用模板）
3. 前端接口调用代码的自动生成
4. 前后端联调
5. 登录界面登录

修改bug
对footer的position使用 sticky；然后让整个a-layout 填充到 100vh
navbar 缩小名字不换行
要学会利用调试工具修改前端样式

通用组件

根据配置控制菜单的显隐

1）给路由新增一个标志位，用于判断路由是否显隐

2）不要用 v-for + v-if 去条件渲染元素，这样会先循环所有的元素，导致性能的浪费

推荐：只显示过滤后的数组

```
const visibleRoutes = routes.filter((item) => {
  if (item.meta?.access !== "canAdmin") {
    return true;
  }
  if (store.state.user?.role !== "admin") {
    return false;
  }
  return true;
});
```

根据权限隐藏菜单

不仅是隐藏的菜单，更需要设置权限

需求：只有具有权限的菜单，才对用户可见

原理：

全局项目入口

app.vue 预留全局初始化逻辑的代码

后端项目初始化

先把通用的后端项目跑起来

1）下载springboot-init万用模板
2）全部替换 springboot-init为项目名
3）全局替换springbootinit包名为新的包名
4）修改springbootinit文件夹的名称为新的包名对应的名称
5）修改端口号 8121
6）本地新建数据库，直接执行sql/create_table.sql脚本，修改库名

初始化模板阅读

1. README.md
2. sql
3. es
4. aop:用于全局校验、全局日志记录
5. common：万用的类，比如通用响应类
6. config：用于接受application.yml中的参数，初始化一些客户端的配置类（比如对象存储客户端）
7. constant:定义常量
8. controller:接受请求
9. esdao：类似mybatis的mapper，用于操作es的
10. exception：异常处理相关
11. job：任务相关（定时任务、单词任务）
12. manager：服务层（一般定义一些公用的服务，对接第三方API）
13. mapper：mybatis的数据访问层，用于操作数据库
14. model：数据模型、实体类、包装类、枚举值
15. service：业务逻辑
16. utils：工具类，各种各样的方法
17. wxmp: 微信公众号相关

18. test：测试
19. MainApplication.java：入口
20. Dockerfile：构建docker镜像

前后端联调

前端和后端怎么连接？接口 / 请求
前端发送请求调用后端接口

axios 对原生的进行封装
https://axios-http.com/docs/intro

编写调用后端的代码
传统情况下，每个请求都要单独编写代码。至少得写一个请求路径、
完全不用！！
自动生成即可：
https://github.com/ferdikoomen/openapi-typescript-codegen

运行即可得到生成的ts代码

```
openapi --input http://localhost:8121/api/v2/api-docs
--output ./generated --client axios
```

非常重要的bug！！！！
在openAPI.ts 中 BASE 改为 BASE: "http://localhost:8121"
否则端口路径会有两个api

如果想要自定义请求参数，how?
1）openAPI.ts 中去修改

2）直接定义axios请求库的全局参数 （不太懂）

用户登录功能

自动登录

1. 在 store/user.ts 编写获取远程登录用户信息的代码
2. 在某处触发getloginUser函数的执行？应当在全局的位置
   有许多选择
   1. 路由拦截
   2. 全局页面入口 APP.vue
   3. 全局通用布局（所有页面都共享的组件）、

这里选择在权限管理模块中实现
全局权限管理优化
1）新建 index.ts 文件在access中，相当于路由的索引文件。把原来的路由拦截、权限校验逻辑放在独立的文件中
优势：只要不引入，就不会执行，不会对项目有影响
加await是为了等用户登录成功之后，再执行后续的代码
没有权限分为两种情况：未登录

2）编写权限管理和自动登录逻辑
如果没有登录过，先自动登录依次
如果用户访问的页面不需要登录，是否需要强制跳转到登录页?
不需要，而且需要重定向回到原来的位置
如果无权限，则跳转到无权限页面

3）登录页面

支持多套布局

就是路由的子路由 children
修改app.vue ， 设置 v-if

1. 在 routes 路由文件中新建一套用户路由，使用 vue-router 自带的子路机制，实现布局和嵌套路由
2. 新建 UserLayout,UserloginView,UserRegisterView
3. 在 app.vue 根页面文件，根据路由区分多套布局
   当前这种 app.vue 中通过 if else 区分布局的方式，不是最优雅的，理想状态下是直接读取 routes.ts，在这个文件中定义多套布局，并且根据嵌套路由去区分多套布局
   就是那种下拉菜单

- `useRoute`：用于访问当前路由对象，适合需要读取路由信息的组件。
- `useRouter`：用于访问路由实例，适合需要程序化导航或操作路由栈的组件。

登录页面

ref 用于处理普通普通数据
reactive用于处理类
menu.hide还是需要实现
开cookies
a-space

1. 在 acro-design中找 form，然后修改登录页面
2. acro-design：message-error
3. 加logo进入UserBasicLayout 优化布局

后端设计

设计库表

用户表
用户id

用户账号
用户密码（存密文）
用户简介
用户昵称
用户头像
用户权限
创建时间
更新时间
是否逻辑删除
（关于微信公众号，暂时不管）

用户表有一些需要注意：userRole 用来权限管理
题目表
题目编号 // 暂时
题目标题
题目内容
题目标签
题目答案？？？需要思考
题目通过数量
题目提交数量
判题用例：JSON 存储
判题配置：JSON存储
（点赞收藏？）
创建者id
创建时间
更新时间
是否被逻辑删除

题目提交表
题目提交编号
题目编号

提交用户编号
题目提交语言
题目提交代码
题目判断状态（判题中、等待判题、成功、失败）
判题信息（Accept这些 JSON）
提交时间
更新时间（也许有必要，因此可能题目出错，所有题目需要 rejudge）
是否逻辑删除提交

解决网页禁止F12的方法
https://ednovas.xyz/2021/07/31/f12/

数据库索引
首先从业务出发，无论是单个索引、还是联合索引，都要从实际的查询语句、字段枚举值的区分度、字段的类型考虑
原则上：能不用索引就不用索引；能用单个索引就别联合多个索引；不要给没区分度的字段加索引。因为索引也是要占用空间

后端接口开发

后端开发流程
1）根据功能设计库表
2）自动生成对数据库基本的增删改查
3）编写Controller层，实现基本的增删改查和权限校验
4）根据业务定制开发新的功能/编写新的代码

代码生成方法
1）mybatis X 插件
2）生成代码
3）将生成代码移动到相应位置
4）编写controller 找相似的代码去复制
写 QuestionController，QuestionSubmitController

5）复制实体类的 DTO、VO、枚举值字段（用于接受前端请求、或者业务间传递信息）
updateRequest 和 editRequest 的区别：前者给管理员，后者给普通用户
1）DTO：QuestionADDRequest，QuestionUpdateRequest，QuestionEditRequest,QuestionQueryRequest
QuestionSubmitAddRequest.java
DTO 中本质上都是一些可序列化的类，然后用来过滤掉前端给的信息，为了安全！！！
复制之后，然后复制生成的entity中的question，然后留下需要的字段,这四个都要
注意：edit是给用户用的，update是给管理员用的
2）给相关JSON写类或者枚举值
judgeCase 用例，JudgeConfig 内存限制之类的
questionsubmit 下的JudgeInfo
3）写 QuestionVO 这个是专门给前端用的
4) 注意阅读 QuestionController
5) 查看整个代码，判断有没有问题，其他报错之类的
6) 编写枚举类 language，judgeinfo，status

6)编写Service层代码，QuestionService复制PostService，然后QuestionServiceImpl复制PostServiceImpl
下载generator all setter and getter 插件，对象.allget 即可
修改：validQuestion：检查是否合法，getQueryWrapper：查询
！！注意，entity中的Question的tags不能由String改为 List，因为存入数据库的就是要String，所以后面QuestionVO中需要转换List成为StringJSON或者StringJSON转换为List
getQuestionVO，getQuestionVOPage 这两个稍微复杂点
另外有QuestionSubmitService 和 QuestionSubmitServiceImpl

string manipulation 有利于转化枚举值成大写
然后doQuestionSubmit 提交多加一个判断是否语言正确?

！！！注意测试
另外 id 要 assign_id，防止爬虫
// submit还需要一些查询信息

1. 根据题目id查询题目信息并且提交给前端
2. 根据userid查询所有提交信息（展示前端）
   // todo
   注意事项：仅本人（看自己）和管理员（看所有人）能看见提交代码（暂时如此设置）

如何实现：先查再脱敏
// to do
！！！！ (这里一定要学会如何去写这些东西)
getQuestionSubmitVOPage 本质上的功能就是一次从数据库全部查询，然后再分页从本地

精修前端

现在就是开发各种模块的页面
在线代码编辑器（markdown）、在线文档浏览
不过的代码编辑器

1. 题目模块
   a. 创建题目（管理员）
   b. 删除题目（管理员）
   c. 修改题目（管理员）
   d. 搜索题目（用户）
   e. 在线做题
   f. 提交题目代码

    g. 题目列表页
    h. 题目详情页
    i. 题目提交列表页
    j. 题目状态的查看
2. 用户模块
    a. 注册
    b. 登录
3. 判题模块
    a. 提交判题（结果是否正确与错误）
    b. 错误处理（内存溢出、安全性、超时）
    c. 代码沙箱（安全沙箱）
    d. 开放接口（提供一个独立的新服务）

接入组件

先接入可能用到的组件，再去写页面，避免因为后面依赖冲突、整合组件失败带来的返工

Markdown 编辑器

通用的文本编辑语法，可以在各大网站上统一标准，渲染出统一的样式，比较简单易学

推荐有：bytemd
地址：https://github.com/bytedance/bytemd
下载本体 `npm i @bytemd/vue-next`
然后下载插件：这里下载了： `npm i @bytemd/plugin-math` `npm i @bytemd/plugin-highlight` `npm i @bytemd/plugin-gfm`
高光数学公式表格之类的

前端引入项目：
把这个编辑器写入到一个组件当中

引入组件之后还有一个重要的问题就是需要获取编辑的值，不能让值只在组件内部

隐藏不需要的图标
利用F12 去找样式，然后 `display:none`

要把MdEditor 当前输入的值暴露给父组件，便于父组件去使用，同时也是提高组件通用性，需要定义属性，把 value 和 handleChange时间交给父组件去管理

注意：如果组件参数定义的驼峰式，父组件引用参数要用 handle-change

！！！！ 引入 math需要 `import 'katex/dist/katex.css'` 否则会有重复bug

代码编辑器

monaco-editior ： 微软官方编辑器
https://microsoft.github.io/monaco-editor/

项目扩展 用 diff-eiditor 对比答案

注意：vue-monaco-editor
项目整合这些东西需要自己查询

vue-cli 封装了WebPack，因此需要把 monaco-editor 整合进来需要弄配置
先安装 monaco-editor
然后安装插件：npm install monaco-editor-webpack-plugin

然后就是如何使用monaco-editor，如何配置vue.config.js 的问题，关于配置可以在谷歌上找，关于如何使用可以去官方的playground

https://microsoft.github.io/monaco-editor/playground.html?source=v0.50.0#example-creating-the-editor-hello-world

// to do
！！！ 注意，code-editor还有很多配置可以调，日后有很多东西可以设置

monaco 必须自己手动创建结点，然后才能挂到这个节点上，如果结点还没有加载出来就挂不上
有的组件就是提前帮你做好了，不用挂载实例，然后再执行初始化

注意一下：
我的似乎要安装

```
npm install @babel/plugin-transform-class-static-block
```

另外还需要配置 babel

```
plugins: [ '@babel/plugin-transform-class-static-block' ]
```

实例代码

```
<template>
  <div id="code-editor" ref="codeEditorRef"
style="min-height: 400px" />
  {{ value }}
  <a-button @click="fillValue">填充值</a-button>
</template>

<script setup lang="ts">
import * as monaco from "monaco-editor";
import { onMounted, ref, toRaw } from "vue";

const codeEditorRef = ref();
const codeEditor = ref();
const value = ref("hello world");

const fillValue = () => {
  if (!codeEditor.value) {
    return;
  }
  // 改变值
  toRaw(codeEditor.value).setValue("新的值");
};

onMounted(() => {
  if (!codeEditorRef.value) {
    return;
  }
  // Hover on each property to see its docs!
  codeEditor.value =
```

```
  monaco.editor.create(codeEditorRef.value, {
      value: value.value,
      language: "java",
      automaticLayout: true,
      colorDecorators: true,
      minimap: {
        enabled: true,
      },
      readOnly: false,
      theme: "vs-dark",
      // lineNumbers: "off",
      // roundedSelection: false,
      // scrollBeyondLastLine: false,
    });

    // 编辑 监听内容变化
    codeEditor.value.onDidChangeModelContent(() => {
      console.log("目前内容为: ",
  toRaw(codeEditor.value).getValue());
    });
  });
  </script>

  <style scoped></style>
```

编写页面

首先还是生成前端代码，以便调用API；然后开始编写各种页面

！！！！注意自定义代码模板 live templates
！！！！注意生成的代码之后要改openAPI 里面的东西，不然地址不对，也不携带cookie

创建题目页面

嵌套表单和动态增减表单
这里的话就是各种查看acro-design的样式
// to do
然后现在还有一些bug需要解决：

1. 不知道为什么，我同时弄两个MD editor 一致有那个组件错误
2. 然后就是传输的问题，现在指定了 form 类型导致一些错误的出现
3. 然后就是刷新如果是管理员则会去登录界面
4. mdEditor 放大之后居然有些样式没有被覆盖

关于 1) 已经解决，就是把那个 mode 属性改成 split 就行了

关于 2) 传输，可能是编辑器的错误，另外 mdEditor 的值穿不进来还是本质上是要 form.content 和 form.answer ，不能设置 ref 之类的

管理题目页面

1） 使用表格，需要找到自定义操作的实例
2）查询数据
3）定义表格列
4）加载数据
5）调整格式
JSON 格式不好看怎么办？？

1. 直接组件库自带的语法，自动格式化

2. 完全自定义渲染，想展示什么就展示什么（更灵活）
   6）删除后更新操作
   使用table自带的methods？？？
   使用loadData刷新数据

关于 5）调整格式非常困难 还是需要看视频

bug修复
！！！ 修复QuestionController的bug，注意判断judgeCase 和 judgeConfig

更新页面开发

由于更新和策略都是相同的表单，所以完全没必要开发/复制2遍，可以直接复用创建页面
关键实现：如何区分两个页面?

1. 路由（/add/question 和 /update/question）
2. 请求参数（id=1）

更新页面相比于创建页面，多了两个改动：

1. 加载页面时，更新页面需要加载出之前的数据
2. 在提交时，请求的地址不同

修改controller，新增一个未脱敏的API，但是要注意权限校验，因为我们要根据 id 获取所有信息填充到前端中
注意更新跳转的函数编写，query 能带id参数

```
const doUpdate = (question: Question) => {
  router.push({
    path: "/update/question",
    query: {
      id: question.id,
    },
  });
};
```

这块也挺难的，主要是要给 form 传递值
注意query的用法，然后根据页面是 update 还是 add 来判断是创建题目还是更新题目

BUG修复
另外，解决了一个BUG，就是在更新的时候，标题，标签和题目内容可以为空，这显然不合理，这是因为在 Controller 里面的 `questionService.validQuestion(question, true);` 要是 true 才会验证这个合理性

一个问题
代码上传到 git ，然后回滚的后再进入 ws，会有代码格式的问题，这个时候要设置 git 在保存的时候不要更改格式

注意：也有地方可以直接格式化所有代码，然后node_modules 不能格式化

优化界面

1）先处理菜单项的权限控制和显示隐藏
他是让普通用户也可以添加题目，但是我的思路是不然
只有管理员可以增加题目，而且只能管理自己创建的题目
超级管理员可以管理所有的题目
// to do
超级管理员的设置等项目完结再思考
目前有两种方案：第一种就是细分管理员；第二种就是添加一种做题权限
如果是第一种的话，个人认为需要重新进行权限设计，然后需要修改很多代码
第二种的话可能稍微简单一点：就是user多加一列，然后修改稍许代码
目前来看 不可能让普通用户来创建题目，不然无限的题目被创建，质量得不到保证。
个人认为超级管理员权限设置更加合理
2）分页功能的实现

watchEffect 可以监听函数里面的所有变量，改变的时候可以改变

pageNumer 要改成 current，因为后端API 就是这样的

关于 2）稍微有点难，要绑定事件，使用 event： page-change

3）之前的一个bug，刷新进入登录界面
因为我们判断是的局部变量 loginUser ，而不是 store 里面的，虽然 store 里面的已经更新了
一定要仔细思考 index.ts

题目浏览页面

使用表格组件
表格很复杂，注意学习插槽

1）留下合适的列 √
2）自定义表格列的渲染 √
首先需要把要自定义的列使用 slotName
标签：使用tag组件
注意，这里要用脱敏后的API
通过率：自行计算
！！ 注意除法要判 0
创建时间：使用 moment 库进行格式化
https://momentjs.cn/#google_vignette
安装 `npm install moment --save`
然后改成这样 `moment(record.updateTime).format("YYYY-MM-DD")`
操作按钮：补充跳转到做题页的按钮

另外 注意修改按钮点击事件，跳转到做题页面
思考
为什么 update 那里跳转需要 query，而这里需要使用
path: `/view/question/${question.id}`,

3）添加题目搜索，写搜索表单，使用 form 的 layout = inline 布局，让用户的输入和 searchParams 同步，并且给体骄傲按钮绑定修改 searchParachs，从而被 watchEffect 监听到

因此，searchParams 就可以设置成 QuestionQueryRequest 的对象
注意，每次点击submit搜索的时候要设置 current 为 1

小细节
由于watchEffect监听了searchParams 的变量，因此这里可以不用调用 loadData
注意不能直接 searchParams.value.current = 1
这样不能算改变

题目列表页面

1）定义动态路由，开启 props为true，这样可以在页面的 props 种直接获取到动态参数

```
interface Props {
  id: string;
}

const props = withDefaults(defineProps<Props>(), {
  id: () => "",
});
```

注意使用 gird 的布局

制作 markdown 浏览组件，注意这里使用的 Viewer

这里的制作也是不简单的
注意要展示的信息：判题配置

解决bug
minimap 的 absolute 是的一直粘贴在那里，因此直接取消掉minimap即可，直接干掉提出问题的东西

做题页面

可以参考poj 之类的

select 编程语言怎么改
关于语言切换，需要特殊的写法

// to do !
了解这个函数的机制

```
watch(
  () => props.language,
  () => {
    if (codeEditor.value) {
      monaco.editor.setModelLanguage(
        toRaw(codeEditor.value).getModel(),
        props.language
      );
    }
  }
);
```

一定要记得写 codeEditor 自定义的函数参数

// to do!
有一个小细节：就是在某一页点击做题页面的时候，我点击返回它会回到第一页，这个可能使得用户体验并不好

后端航行

后端判题机模块的预开发

这里会使用到 （2-4种设计模式），跑通整个项目的前后端流程

> 目的：跑通完整的业务流程

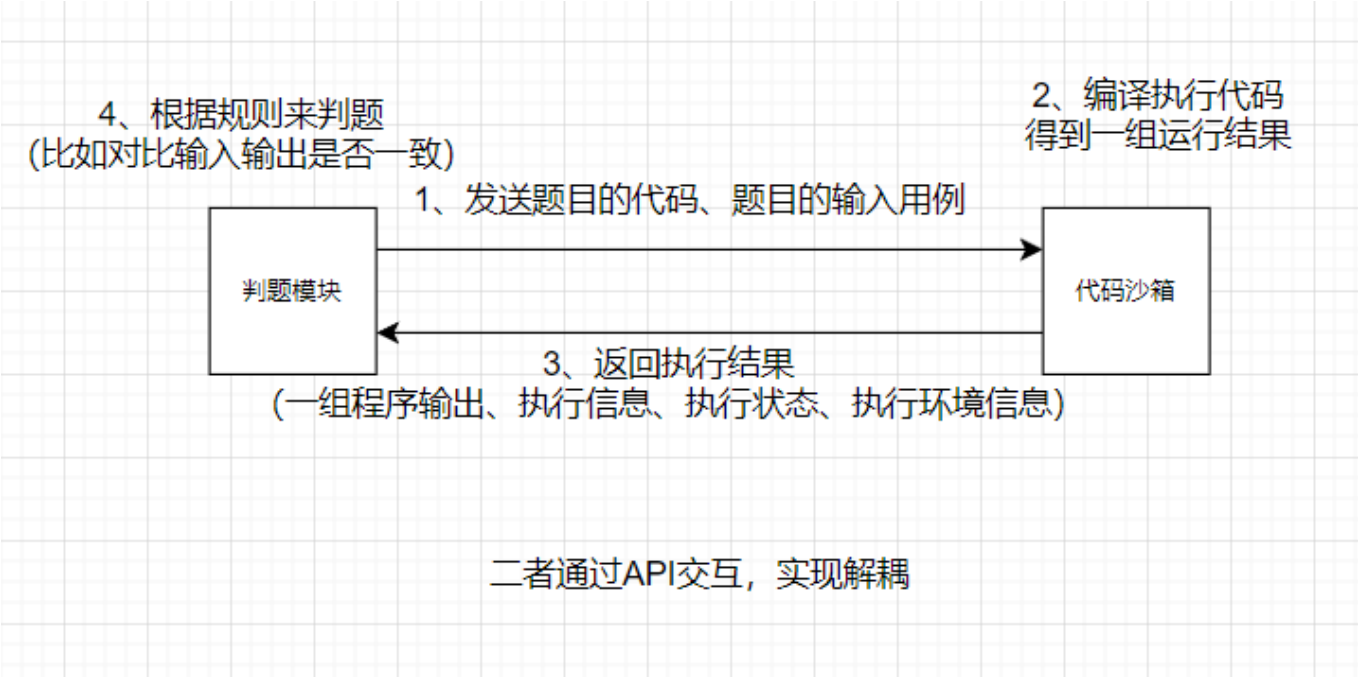
一定要明确！！！！！

梳理判题模块和代码沙箱的关系

判题模块：调用代码沙箱，把代码和输入交给代码沙箱去执行

代码沙箱：只负责接受代码和输入，返回编译运行的结果，不负责判题（可以作为独立的项目/服务，提供给其他的需要执行代码的项目去使用）

这两个模块完全解耦



为什么代码沙箱要接受和输出一组运行用例
前提：题目有多组测试用例
如果是每个用例单独调用一次代码沙箱，会调用多次接口、需要多次网络传输、程序要多次编译、记录程序的执行状态（重复的代码不重复编译）

常见性能优化！！！

为什么不用消费者生产者模式？
减少耦合，如果是内部使用无所谓

！！！ 设置 git，保证跨平台的时候文件格式保持一致
注意
如果日后要将其弄到 linux 服务器上，要注意修改
git config --global core.autocrlf false

沙箱接口

judge.codesandbox 文件夹下

1）定义代码沙箱的接口，提高通用性
judge.codesandbox 下新建一个接口 CodeSandbox

理由：之后的项目代码只调用接口，不调用具体的实现类，这样在使用其他的代码沙箱实现类时，就不用去修改名称，便于扩展

> 代码沙箱的请求接口中，`timeLimit`完全可以不加，可以自行扩展，即使中断程序

言

2）
新建model 一个类 ExecuteCodeRequest

!!! 注意 @builder 注解

需要 测试用例，代码，编程语言
想想需要沙箱执行代码需要做什么：代码，测试用例，语
需要**timeLimit**？ 暂时不需要，日后也许可以扩展

ExecuteCodeResponse
返回结果，接口信息，执行信息，状态

还是需要思考返回结果：输出结果，最终执行（内存，时间），如果执行错误需要返回的信息（接口信息）

扩展思路：增加一个查看代码沙箱状态的接口（就是查看目前执行的情况）

3）
新建 impl
实现三种代码沙箱
ExampleCodeSandbox ： 示例代码沙箱：为了跑通项目
RemoteCodeSandbox：远程代码沙箱：实际调用的接口
ThridPartyCodeSandbox 接入第三方沙箱（用来接入其他语言） go-judge

架构，暂时

4）编写测试类
@SpringBootTest 注解

什么是**Builder**注解？ （Lombok中）

```
ExecuteCodeRequest executeCodeRequest =
ExecuteCodeRequest.builder()
        .code(code)
        .inputList(inputList)
        .language(language)
        .build();
```

工厂模式：根据用户传入的字符串参数，来生成对应的代码沙箱实现类
此处使用静态工厂模式，实现比较简单
为什么要使用工厂模式？因为我要 new 一个代码沙箱的时候需要修改大

量的带啊吗，因此，我可以使用工厂模式，我只需要输入字符串就可以给我创建一个相应的沙箱实例

5）参数配置化，把项目中的一些可以交给用户去自定义的选项或者字符串，写到配置文件中，这样开发者只需要修改配置文件，而不需要去看你的项目代码，就能够自定义使用你的项目的更多功能。

application.yml 配置文件中添加 codesandbox
然后使用 @value 注解

```
codesandbox:
  type: example # 代码沙箱类型

// 使用
@Value("${codesandbox.type:example}")
private String type;
```

6）代码沙箱能力增强
比如：需要在调用代码沙箱签，输出请求参数日志；在代码沙箱调用后，输出响应结果日志，便于管理员去分析。

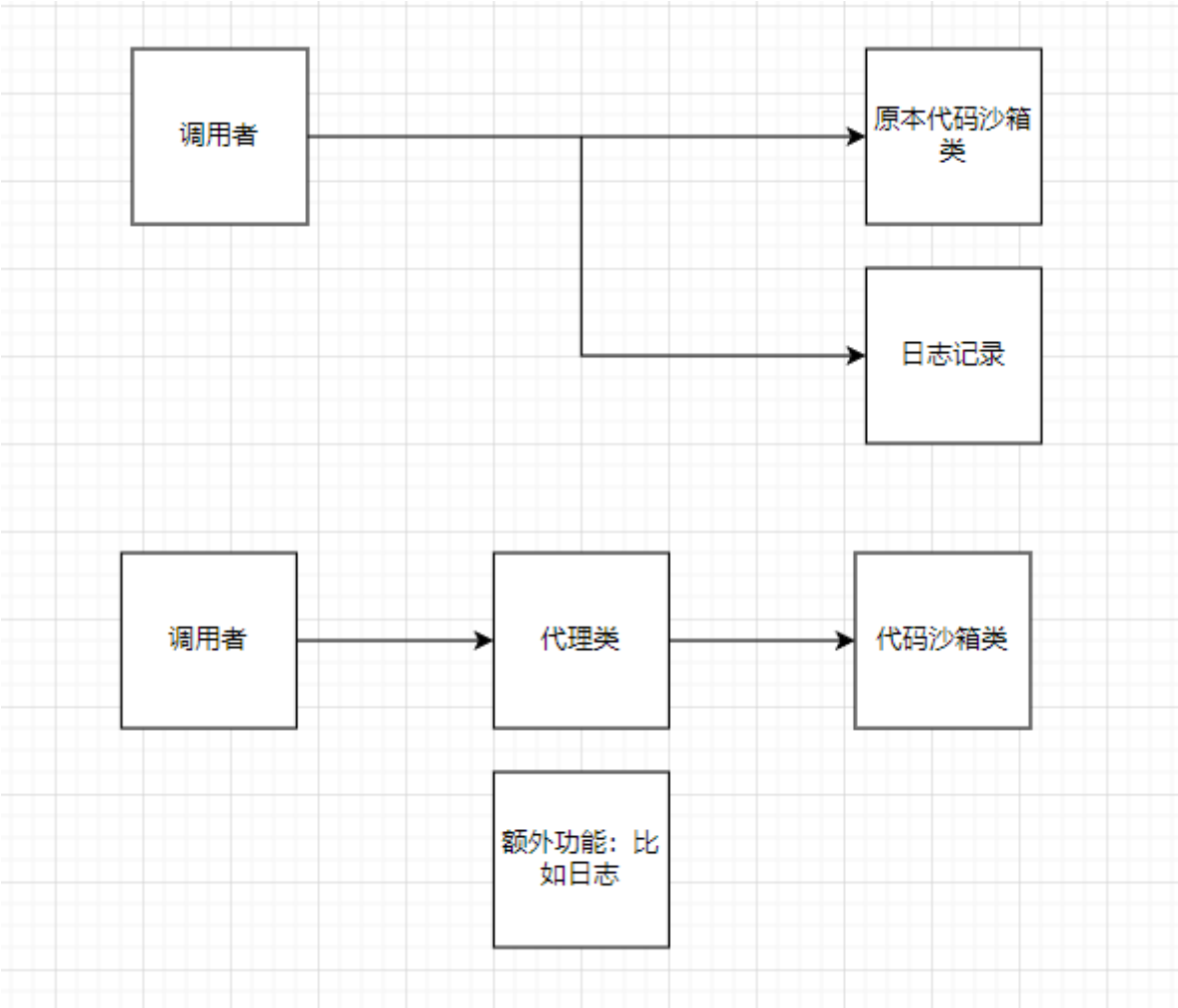
难道每个代码沙箱都要info？每个代码沙箱执行后也要 info ？
info需要注解（@Slf4j）

接下来就需要考虑日志的问题，因为如果不写这些东西，很有可能日后错哪里都不知道
因为日后调用codeSandbox 是调用接口，因此其前后应该需要设置日志功能

但是每个沙箱都写日志有点多，因此可以使用代理模式

！！！ 使用代理模式

codeSandboxProxy 实现 codeSandbox



注意变量 codeSandbox 的变量要用 final 修饰，因为只会被改变一次
可以看到：使用代理后，不仅不改变原有的代码沙箱实现类，而且对调用者来说，调用方式几乎没有改变，也不需要在每个调用沙箱的地方去统计代码

7）简单实现下 exampleSandbox
事实上，测试的时候已经写了很多了

工厂模式

代码沙箱工厂（根据字符串参数创建指定的代码沙箱示例）
codeSandboxFactory

里面的函数根据 string 输出一个 codeSandbox 实例

> 扩展思路：如果确定代码沙箱实例不会出现线程安全问题、可复用，那么可以使用单例工厂模式

代理模式

给代码弄日志！！

使用代理模式，提供一个Proxy，来增强代码沙箱的能力（代码沙箱的作用就是增强能力）

为什么代理要实现 codesanbox
因为代理本质上是增强代码沙箱的功能，所以是代码沙箱的一份子

代理模式实现：

1. 实现被代理的接口
2. 通过构造函数接受一个被代理的接口实现类
3. 调用被代理的接口实现类，在调用前后增加对应的操作

判题服务完整业务流程实现

judge 下

创建接口 JudgeService
创建这个东西就是不让很多代码堆砌在 questionSubmitService 中
而且就是要把这个东西抽象出来，后面好做微服务

在 QuestionSubmitController 中加一个 // to do
因为 代码提交还是需要判题的

判题服务需要做什么？？
1）传入题目的提交 id，获取到对应的题目、提交信息（包含代码、编程语言等）
2）如果题目的提交状态不为等待中，就不用重复执行
（考虑以下情景：题目提交了，因为交给代码沙箱肯定需要时间，因此我们会提交给出提交**id**告诉用户已经提交了，但是如果这个时候有小人根据这个提交**id**一致调用后端**api**，会导致大量重复提交或者已经提交过了再次执行，因此，只有等待中的提交**id**才需要被执行）
3）只有状态为 0（等待中） 的才需要将其变为 1 （执行中）
2）调用沙箱，获取执行结果
// to do!
学习stream 流的知识
3）根据沙箱的执行结果，设置题目的判题状态和信息 （返回 questionVO），然后参数为 submitId
4）调用沙箱
5）根据结果判断是否正确
6）修改数据库中的结果

首先就是异常之类判断之类的
id 存在？ 题目存在？ 判题状态？？ 用户能看到题目在什么状态

每次对于一个题目的提交只能是一次
判断逻辑：

1. 输出数量和预期数量是否相等
2. 判断每一项输出和预期输出是否相等
3. 判断题目的限制是否符合要求
4. 其余异常情况（沙箱执行异常）

思考：
如果我们的代码沙箱本身执行程序需要消耗事件，这个时间可能不同的编程语言是不同的，比如沙箱执行 java 要额外花 10 秒

因此，针对不同的情况，定义独立的策略，而不是把所有的判题策略都混合在一起

judge.strategy 下 策略接口 JudgeStrategy

定义 judgeContext，
定义 DefaultJudgeStrategy： JudgeInfo，inputList, outputList，question
strategy

在这个过程中，需要思考，判断题目需要哪些东西

同样地，面对不同的情况，会选择不同的判题策略，如果这个时候再判题服务里面写，那么就又要变得复杂很多，因此可以使用一个类来管理
尽量简化对判题功能的调用

策略模式

每个策略写一个类
如何切换策略？？
写 if - else ？？ 岂不是要写很多?

定义 judgeManager，目的是尽量简化对判题功能
判题管理

实现代码沙箱

1. Java 原生代码沙箱
2. Docker 实现代码沙箱

代码沙箱：只负责接受代码和输入，返回编译运行的结果，不负责判题

以 java 编程语言为主，实现代码沙箱

> 扩展：可自行实现c++语言的代码沙箱

新建spring boot web 项目
loj-code-sandbox
如果这里不能选择JDK为8，则在配置中更改配置，springboot的版本和jdk版本都可
然后修改application.yml，添加服务器端口

```
server
        port: 8090
```

然后新增controller，编写一个测试controller代码，目的跑通该项目
小细节：复制接口一起需要的参数和相应参数需要 JudgeInfo，这个时候将其复制到 judge.model 下

如果要执行的文件是在某个 package 下，那么使用命令行是执行不了的，因为java的类的类名是代码 package 的地址的，因此想要不加package，其本身就得把 package 去掉（不对，暂时是发现有包名就不行）

Java 原生实现代码沙箱

原生：尽可能不借助第三方库和依赖，用最干净、最原始的方式实现代码沙箱
代码沙箱需要：接受代码=》编译代码（javac）=》执行代码

编码的问题，为什么编译后的程序中文乱码，因为命令行默认的GBK编码，而idea是utf-8编码，

可以使用 chcp 命令查看编码，GBK 是936，
javac 可以指定编码编译
`javac -encoding utf-8 [文件名]`

实际 OJ 系统中，对用户输入的代码会有一定的要求，便于系统统一地处理。所以此处，把用户输入代码的类名限制为 `Main` ，可以减少类名不一致的风险

实际执行：

```
javac -encoding utf-8 .\Main.java
java -cp . Main 1 2
```

核心流程实现

核心实现思路：用程序代替人工，用程序来操作命令行，去编译执行代码
java 进程执行管理类：Process ，帮助执行其他程序

1. 把用户的代码保存为文件
2. 编译代码，得到class文件
3. 执行代码，得到输出结果
4. 收集整理输出结果
5. 文件清理
6. 错误处理，提升程序健壮性

.gitignore 中加入 tmpCode ，不把临时文件加入到 git 中

1、把用户的代码保存为文件
那如何保存为文件呢?
1）考虑要保存的地址：根目录下的 tmpCode，这里有两个问题，根目录如何获取？tmpCode判断是否存在

2）思考是否每一个接受的java文件是否全部保存在 tmpCode 下？经过测试，最好是在 tmpCode 下新建一个随机文件夹，然后存到这个文件夹里面，因为这样可以保证线程读写不冲突
小细节：在连接 **tmpCode** 路径名字的时候，**tmpCode** 可以设置为一个常量

to do
// 学习 hutool 工具类

加入 hutool 工具类
注意文件分隔符使用 File.separator

那么接下来就是需要写一个 Main 函数进行测试
然后 code 的获取是利用库 直接获取 Resource 下的java文件的代码即可

2、编译java代码，得到class文件
接下来就是编译代码：思考一下编译代码需要什么？
1）指定要编译的文件
2）获取错误信息，那么关键是什么信息的获得，如果是成功还是，但是如果是失败了，就得报编译错误，并且哪里错误得返回给用户（如果有必要的话）
另外编译成功了也不代表一定可以，因为还有运行错误和段错误。这个实现方式是获取流

```java
BufferedReader bufferedReader = new
BufferedReader(new
InputStreamReader(compileProcess.getInputStream()))
;
String compileOutputLine;
while ((compileOutputLine =
bufferedReader.readLine()) != null) {
    System.out.println(compileOutputLine);
}
```

！！！注意，如果编译错误，不仅要读取输入流，而且要读取错误流
//to do
注意：读取的信息是乱码的，还需要继续思考

3、java执行程序
1）同样地，使用命令行运行
2）需要获取控制台的内容，将其封装起来
2）这里运行可能就会出现运行错误了，但是要区分运行错误和段错误

！！！封装错误信息

在写命令行运行的代码的时候，发现运行的代码和编译的代码差不多，有很多东西，因此需要将其封装成一个信息输出类
然后思考，既然输出肯定是需要一些信息的：就拿之前写的，退出代码，正常信息和错误信息都要输出
然后就是正常信息

```java
String runCmd = String.format("java -
Dfile.encoding=UTF-8 -cp %s Main %s",
userCodePathParentName, inputArgs);
```

可以看到，现在程序的输入是从 String[] args 获得，如何从标准输入流中获得
那么，对于ACM制的话，是从标准输入输出流里面读取，因此需要我们需要源源不断写测试用例到输出流中

```java
InputStream inputStream =
runProcess.getInputStream();
            OutputStream outputStream =
runProcess.getOutputStream();
            OutputStreamWriter outputStreamWriter =
new OutputStreamWriter(outputStream);
//            String[] s = args.split(" ");
//
outputStreamWriter.write(StrUtil.join("\n", s) +
"\n");
            outputStreamWriter.write(args + "\n");
```

4、整理输出
每个用例都有一个输出
但是如果有错误信息呢?
如果存在编译错误？运行错误？段错误？怎么办？是在这个里面判断，还是判题系统

这里统一判题系统！！！
获取运行空间很麻烦，因此原生的不实现

这里使用最大值来统计

> 扩展：每个测试用例都有一个独立的内存、时间占用的统计

5、文件清理
使用FileUtil 清除
建议还是先判断文件是否存在

6、更多异常处理，提升健壮性
另外，如果编译错误，需要不同的返回
就是多判断一下异常的问题

异常情况

到目前为止，核心流程已经实现，但是要上线的话，是否安全？
用户如果提交恶意代码如何？

1）执行阻塞，占用资源不释放
使用休眠程序

2）占用内存、不释放
无限占用空间，占用内存

实际运行中，JVM 有保护机制，一旦内存占用到了限制，就是自动报错

JvisualVM 或 JConsole 工具，可以看到JVM虚拟机上来可视化查看运行状态（一般会有监测工具）
可以写到简历上

3）读文件（文件泄露）

4）写文件，越权写木马文件

5）运行其他程序
直接通过Process 执行危险程序，或者电脑上的其他程序

6）执行高危命令
甚至都不用写木马文件，直接执行系统自带的危险命令
比如删除服务器的所有文件
执行dir（windows）、ls（linux）获取系统上的所有文件信息

怎么解决这些问题？？

1）超时控制
2）限制给用户分配的资源
3）限制代码-黑白名单
4）限制用户的操作权限（文件、网络、执行）
5）运行环境隔离

1）超时控制
开线程判断时间，超时直接 kill thread
有个问题：运行结果如果是对的会咋办
！！！ 所有注意这个代码还是有问题，因为正常运行结束按道理是尽量不能销毁 runProcess 的，因为这个时候有可能刚好在处理Process的东西

```java
// 利用线程进行超时控制
new Thread(() -> {
    try {
        Thread.sleep(TIME_OUT);
        runProcess.destroy();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}).start();
```

2）限制资源的分配
不能让每个java进程的执行占用的JVM 最大堆内存空间和系统的一致，实际上应该小一点，比如 256MB

JVM参数：-Xmx256m （最大堆空间大小）-Xms（初始堆空间大小）

```java
String runCmd = String.format("java -Xmx256m -
Dfile.encoding=UTF-8 -cp %s Main %s",
userCodePathParentName, inputArgs);
```

有可能会超过，这个不代表实际占用的最大资源
如果需要更严格是限制看需要在系统层面去限制，而不是JVM层面

如果是 Linux 系统，可以使用 cgroup 来实现对某个进程的 CPU、内存等资源的分配

3）限制代码 - 黑白名单
首先定义黑白名单，
然后建立静态字典树，将黑白名单加入进去
最后在执行前判断

但是黑白名单会有问题！！

1. 如果用户的代码是合法，但是其中变量有这些东西，岂不是过滤掉了？
2. 也可能想到所有的黑白名单
3. 不同的编程语言，对应的领域、关键词都不一样，限制人工成本很大

4）限制用户的操作权限
限制用户对文件、内存、CPU、网络等资源的操作和访问

Java 安全管理器（Security Manager）来实现更加严格的限制

编写安全管理器，只需要继承Security Manager
重写 checkPermission 方法

事实上，Security Manager 方法也是需要加入大量白名单的，因此也不适用

限制删除文件异常：

实际情况下，只需要限制子程序的权限即可，而不用限制程序员的代码，因此需要在命令行中直接编写即可
使用命令指定安全管理器

优点
权限控制很灵活，实现简单

缺点

1. 如果要做比较严格是权限限制，需要自己去判断哪些文件、报名需要允许读写，粒度太细了；难以精细化控制
2. 安全管理器本身也是Java代码，也有可能存在漏洞（还是程序上的限制，没到系统的层面）

5）运行环境隔离
系统层面上，把用户程序封装到沙箱里，和宿主机（我们的电脑/服务器）隔离开

Docker 容器奇数能够实现（底层是用 cgroup、namespace 等方式实现）

JAVA 安全管理器可以写在简历上

// to do
46 集的内容最后的限制用户程序的代码编译和执行部分没有实现了

Docker

1. 什么是Docker
2. Docker 的基本用法
   a. 命令行的用法
   b. Java 操作 Docker
3. 用Docker实现代码沙箱
4. 怎么提升Docker的安全性

Docker 容器技术

为什么要用 Docker 容器技术?
为了提升系统的安全性，把不同的程序和宿主机进行隔离，使得某个程序的执行不会影响到系统本身
宿主机：启动的东西都是依托在宿主机上

Docker 技术可以实现程序和宿主机的隔离。

什么是容器?
理解为对一系列应用程序、服务和环境的封装，从而把程序运行在一个隔离的、密闭的、隐私的空间内，对外整体提供服务。
可以把一个容器理解为一个新的电脑（定制化的操作系统）。

Docker 基本概念

镜像：用来创建容器的安装包，可以理解为给电脑安装操作系统的系统镜像
容器：通过镜像来创建的一套运行环境，一个容器里可以运行多个程序，可以理解为一个电脑实例
Dockerfile：制作镜像的文件，类似 pom.xml ，可以理解为制作镜像的一个清单
镜像仓库：存放镜像的仓库，用户可以从仓库下载现成的镜像，也可以把做好的镜像放到里面

Docker 实现原理

1）Docker 运行在 Linux 内核上
2）CGroups：实现了容器的资源隔离，底层是 Linux CGroup 命令，能够控制进程使用的资源
3）Network 网络：实现容器的网络隔离，docker 容器内部的网络互不影响
4）namespaces 命令空间：可以把进程隔离在不同的命名空间下，每个容器它都可以有自己的命令空间，不同的命令空间下的进程互不影响
5）Storage 存储空间：容器内的文件是相互隔离的，也可以去使用宿主机的文件

`docker compose`：是一种同时启动多个容器的集群操作工具（容器管理工具）

安装Docker

一般情况下，不建议在 windows 系统上安装

Windows 本身就自带了一个虚拟机叫 WSL，但是不推荐，肯定不如专业的、隔离的虚拟机软件方便

推荐虚拟机，免费的 VMWare workstation player 软件

虚拟机安装：
https://blog.csdn.net/weixin_74195551/article/details/127288338
ubuntu的ios 源在清华镜像下载
虚拟机防止不卡：
https://blog.csdn.net/davidhzq/article/details/102461957

apt更换源：https://midoq.github.io/2022/05/30/Ubuntu20-04%E6%9B%B4%E6%8D%A2%E5%9B%BD%E5%86%85%E9%95%9C%E5%83%8F%E6%BA%90/

docker更换源：

1. 获取阿里云镜像加速器地址

首先，你需要登录阿里云并获取你的镜像加速器地址：

1. 登录到阿里云控制台。
2. 搜索“容器镜像服务”并进入。
3. 在左侧菜单中找到“镜像加速器”。
4. 复制给你的镜像加速器地址。

2. 配置 Docker 使用镜像加速器

Linux 系统

1. 打开或创建 `/etc/docker/daemon.json` 文件：

```bash
sudo nano /etc/docker/daemon.json
```

2. 在文件中添加以下内容，将 `<your-mirror-address>` 替换为你从阿里云获取的镜像加速器地址：

```json
{
  "registry-mirrors": ["https://<your-mirror-address>"]
}
```

3. 保存文件并退出，然后重新启动 Docker 服务：

```bash
sudo systemctl daemon-reload
sudo systemctl restart docker
```

Docker 常用操作 - 命令行

1）查看命令用法 `docker --help`
查看具体子命令的用法 `ducker run --help`

2）从远程仓库拉取镜像

```shell
docker pull hello-world
```

3）根据镜像创建容器实例
创建之后返回一个容器实例 containerId

```shell
docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

4）查看容器状态

```shell
sudo docker ps -a
```

5）启动实例

```shell
 docker start [OPTIONS] CONTAINER [CONTAINER...]
```

例子

```shell
sudo docker start loving_rubin
```

执行完就退出

6）查看日志

```shell
sudo docker logs loving_rubin
```

7）删除实例

```shell
sudo docker rm loving_rubin
```

9）推送镜像
docker push

Docker-java

安装 3.3.0 java-core ; java-tranport apache 3.3.0
https://github.com/docker-java/docker-java/blob/main/docs/getting_started.md

DockerClientConfig:用于初始化 DockerClient的配置（类比MySQL的连接，线程数配置）

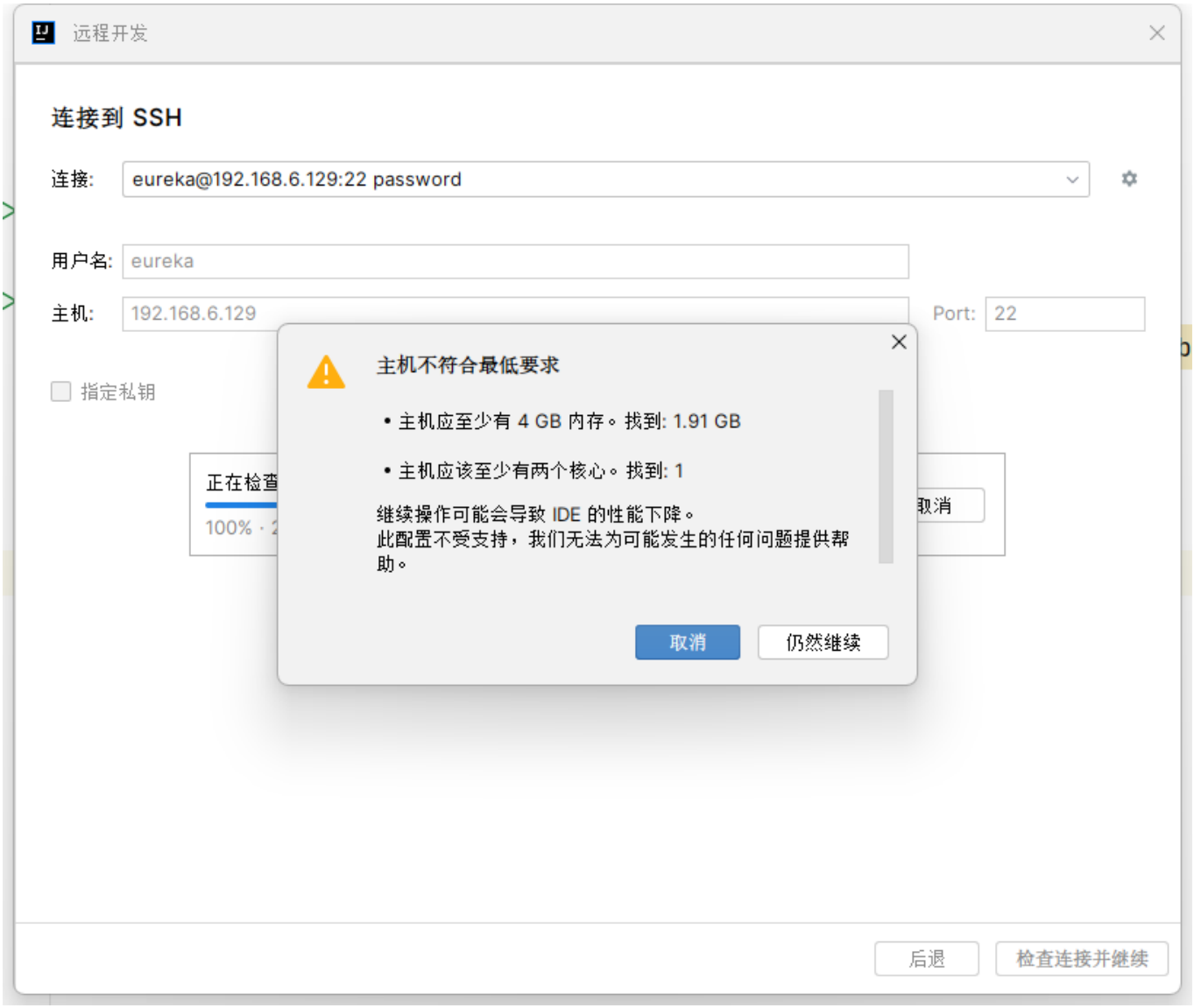
DockerHttpClient：用于向Docker守护进程（操作Docker的接口）发送请求的客户段，低层封装，但是要自己构建请求参数

DockerClient：才是真正和Docker守护进程交互的、最方便的SDK，高层封装，对DockerHttpClinet 再进行了一层封装，提供了现成的增删改查

Linux docker

第一种方式
工具的部署的配置里面，根据虚拟机的ip地址进行连接，然后注意写映射
最后根据根目录同步到 linux 上

第二种方式
新建项目远程连接，但是配置得稍微高一点点



这种开发方式，虚拟机就是屏幕，但是要求虚拟机性能要高
相当于windows就是个屏幕

利用代码拉取镜像

记得要给 Docker 加用户权限
然后加完权限能重启的都重启一下

拉取镜像代码

JAVA

```java
package com.yupi.lojcodesandbox.docker;

import com.github.dockerjava.api.DockerClient;
import com.github.dockerjava.api.command.PingCmd;
import
com.github.dockerjava.api.command.PullImageCmd;
import
com.github.dockerjava.api.command.PullImageResultCa
llback;
import
com.github.dockerjava.api.model.PullResponseItem;
import
com.github.dockerjava.core.DockerClientBuilder;

public class DockerDemo {
    public static void main(String[] args) throws
InterruptedException {
        DockerClient dockerClient =
DockerClientBuilder.getInstance().build();
//        PingCmd pingCmd = dockerClient.pingCmd();
//        pingCmd.exec();
        String image = "nginx:latest";
        PullImageCmd pullImageCmd =
dockerClient.pullImageCmd(image);
        PullImageResultCallback
pullImageResultCallback = new
PullImageResultCallback() {
            @Override
            public void onNext(PullResponseItem
```

```
item) {
                System.out.println("下载镜像: " +
item.getStatus());
                super.onNext(item);
            }
        };

pullImageCmd.exec(pullImageResultCallback).awaitCom
pletion();
        System.out.println("下载完成");
    }
}
```

创建容器:

JAVA

```
        CreateContainerCmd containerCmd =
dockerClient.createContainerCmd(image);
        CreateContainerResponse
createContainerResponse = containerCmd
                .withCmd("echo", "hello Docker")
                .exec();

System.out.println(createContainerResponse);
```

返回一个containerId

查看容器状态

```java
        ListContainersCmd listContainersCmd =
dockerClient.listContainersCmd();
        List<Container> containerList =
listContainersCmd.withShowAll(true).exec();
        for (Container container: containerList)
            System.out.println(container);
```

启动容器：

```java
        // 启动容器

dockerClient.startContainerCmd(containerId).exec();
```

查看容器日志

JAVA

```
 // 如果容器启动太慢，得睡个几秒
        Thread.sleep(5000L);

        // 查看容器id

        LogContainerResultCallback
logContainerResultCallback = new
LogContainerResultCallback() {
            @Override
            public void onNext(Frame item) {
                System.out.println("日志: " + new
String(item.getPayload()));
                super.onNext(item);
            }
        };
        // 一定要加 await 阻塞，因为是多线程
        dockerClient.logContainerCmd(containerId)
                .withStdErr(true)
                .withStdOut(true)
                .exec(logContainerResultCallback)
                .awaitCompletion();
```

删除容器
dockerClient.removeContainerCmd(containerId).exec();

删除镜像
dockerClient.removeImageCmd(image).exec();

Docker 实现代码沙箱

实现流程：docker 复杂运行 java 程序、并且得到结果

1. 把用户的代码保存为文件
2. 编译代码，得到class文件
3. 要把编译好的文件上传到容器环境内
4. 执行代码，得到输出结果
5. 收集整理输出结果
6. 文件清理
7. 错误处理，提升程序健壮性

   模板方法设计模式，定义同一套实现流程，让不同的子类去负责不同流程中的具体实现。执行步骤一样，每个步骤的实现方式不一样。

创建容器，上传编译文件

自定义容器的两种方式：
1）在已有镜像的基础上扩充：比如拉取现成的java环境（包含jdk），再把编译后的文件复制到容器里
2）完全自定义容器：适合比较成熟的项目，比如封装多个语言的环境和实现

也可以不同语言不同镜像

思考：每个测试用例都单点创建一个容器，每个容器只执行一次java命令？

浪费性能，所以要创建一个 可交互 的容器，能接受多次输入并且输出

创建容器时，可以指定文件路径（Volumn）映射，作用是把本地文件同步到容器中，可以让容器访问。

```java
HostConfig hostConfig = new HostConfig();
hostConfig.setBinds(new
Bind(userCodePathParentName, new Volume("/app")));
```

启动容器执行代码

Docker 执行容器命令（操作已启动容器）

带参数的命令执行

```shell
docker exec silly_chatelet java -cp /app Main 3 4
```

注意：要把命令按照空格拆分，作为一个数组传递，否则可能会被识别为一个字符串，而不是作为参数

不带参数的命令，ACM 制度

```shell
sh -c 'echo -e "1\n3 2 6\n" | java -cp /app Main'
```

存到 cmdArray 就是

```java
String[] cmdArray = {"sh", "-c", "echo -e \"1\\n3 2
6\\n\" | java -cp /app Main"};
```

上面这点非常重要，最重要的是后面转成文件流命令

```shell
sh -c 'java -cp /app Main <
/data/problem/input.txt'
```

创建命令：

```java
            String[] cmdArray = getCmd(inputArgs);
//              String[] cmdArray = {"sh", "-c",
"echo -e \"1\\n3 2 6\\n\" | java -cp /app Main"};
            System.out.println("最终cmd: " +
Arrays.toString(cmdArray));
            ExecCreateCmdResponse
execCreateCmdResponse =
dockerClient.execCreateCmd(containerId)
                    .withCmd(cmdArray)
                    .withTty(true)
                    .withAttachStderr(true)
                    .withAttachStdin(true)
                    .withAttachStdout(true)
                    .exec();
```

获取输出值：

尽量复用之前的ExcuteMessage模式，在异步接口中填充正常和异常信息。
也就是说 message 和 errormessage 以及 time 都要再这里面

获取程序执行时间：和Java原生一样，使用StopWatch在执行前后统计时间

获取程序占用内存：
程序占用内存每个时刻都在变化，所以你不可能获取到所有时间点的内存

因此需要定义一个周期，每个时刻获取内存，使用docker提供的方法，也就是 statCmd，然后就是使用 stopWatch 来检测运行时间

docker 容器安全性

超时控制

代码中加上 `.awaitCompletion(TIME_OUT, TimeUnit.MICROSECONDS);`

后面判断逻辑，如何区分是正常超时还是没超时，重写onComplete，因为超时的话是不会到这个函数的

内存资源

通过 Hostconfig来限制

```java
HostConfig hostConfig = new HostConfig();
hostConfig.withMemory(100 * 1024 * 1024L);
hostConfig.withMemorySwap(0L);
hostConfig.withCpuCount(1L);
```

网络资源
创建容器的时候直接关掉
`.withNetworkDisabled(true)`

权限管理
Docker 已经做了系统方面的隔离了，比较安全，但不能保证绝对安全
事实上不太好做，只能暂时如此了
1）结合java安全管理器和其他策略去使用
2）限制用户不能向 root 根目录写文件：

简历可以写 **java** 安全管理器

3）Linux 自带的一些安全管理措施 （Security Compute Mode），Linux 安全机制

代码沙箱docker 实现，模板方法改造

模板方法：定义一套通用的执行流程，让子类负责每个执行步骤的具体实现
模板方法的使用场景：适用于有规范的流程，且执行流程可以复用
作用：大幅节省重复代码量，便于项目扩展、更好维护

1、抽象具体的流程

设置为一个抽象类。
先复制具体的类，把代码从完整的方法抽离成一个一个子写法

定义模板流程：

JAVA

```java
    @Override
    public ExecuteCodeResponse executeCode(ExecuteCodeRequest executeCodeRequest)
{
        System.setSecurityManager(new DefaultSecurityManager());

        String language = executeCodeRequest.getLanguage();
        String code = executeCodeRequest.getCode();
        List<String> inputList = executeCodeRequest.getInputList();


        // 1. 把用户的文件放在某个确定的文件夹下
        File userCodeFile = saveCodeToFile(code);

        // 2. 编译代码
        ExecuteMessage compileMessage = compileFile(userCodeFile);
        System.out.println(compileMessage);

        // 注意，如果编译失败就不能往下来，日后还需要改这部分代码
        // 3. 运行代码
        List<ExecuteMessage> executeMessageList = runFile(userCodeFile, inputList);
        // 4. 获取输出
        ExecuteCodeResponse outputResponse =
```

```
        getOutputResponse(executeMessageList);

        // 5. 文件清理
        boolean b = deleteFile(userCodeFile);
        if (!b) {
            log.error("deleteFile error,
userCodeFilePath = {}",
userCodeFile.getAbsolutePath());
        }
        return outputResponse;
    }
```

2、定义子类的实现

对于原生代码沙箱：直接复用模板方法定义好的方法实现

对于 Docker 代码沙箱，需要重写 runFile 函数和 getOutputList 函数

3、给代码沙箱提供开放 API

就是要写一个 controller 暴露沙箱

调用安全性
如果将服务不做任何的权限校验，直接发到公网，是不安全

1）调用方与服务提供方之间约定一个字符串（最好加密 MD5）
优点：简单，比较适合内部系统之间相互调用（相对可信的环境内部调用）
缺点：不够灵活，如果key泄露或变更，需要重启代码

```java
        // 基本认证
        String authHeader =
request.getHeader(AUTH_REQUEST_HEADER);
        if
(!AUTH_REQUEST_SECRET.equals(authHeader)) {
            response.setStatus(403);
            return null;
        }
```

2）API 签名认证，给允许调用的人员分配 accessKey，secretKey，然后校验这两组 key 是否匹配

跑通项目流程

1）移动 questionSubmitController 代码到 questionController
2）由于后端改了接口地址，前端需要重新生成接口
3）开发题目提交页面

单体项目改造为微服务（该部分尚未完成）

新建一个项目

什么是微服务？

服务：提供某类功能的代码
微服务：专注于提供某类特定功能的代码，而不是把所有的代码全部放到同一个项目里，会把整个大的项目按照一定功能、逻辑进行拆分，拆分为

多个子模块，每个子模块可以独立运行、独立负责一类功能，子模块之间相互调用，互不影响

微服务的几个重要的实现因素：服务管理、服务调用、服务拆分

微服务实现技术?

Spring Cloud
Spring Cloud Alibaba（本项目使用）
Dubbo（DubboX）
RPC（GRPC）
本质上是通过 HTTP、或者其他的网络协议进行通讯来实现的。

Spring Cloud Alibaba

中文文档：https://sca.aliyun.com/

本质：是在 Spring Cloud的基础上进行了增强

注意：一定要使用合适的版本
2021.0.5.0 1.8.6 2.2.0 4.9.4

改造前思考

从业务需求出发，思考单机和分布式的区别。
用户登录功能：需要改造为分布式登录

Nacos

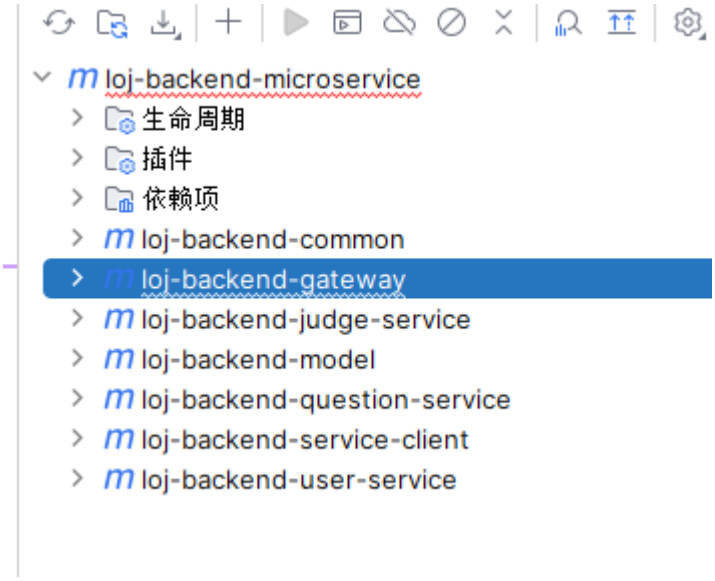
官网下载 Nacos https://github.com/alibaba/nacos/releases/tag/2.2.0
启动

```
startup.cmd -m standalone
```

云原生脚手架：https://start.aliyun.com/

```
版本 21.0.5
```

子父关系



open Feign 组件实现跨服务的远程调用

配置 gateway

Knife4j

分布式 session 登录

跨域和权限校验