用 Mini-OJ 项目学 Java

判题机三代演进·完整版（考试 + 工程）

winbeau ·**Mini-OJ** 教学项目

古法手撸·教学优先·2026-06

考试向（拟合试卷，先上手）

工程向（项目深度，选学）

★★★ 考试相关度（历年真题）

目录

Mini-OJ 学习路线图 (判题机三代演进)

> 环境:Ubuntu + nvim + JDK, 古法手撸 (纯 `javac/java` + `Makefile`), 简单、教学优先。主线: 用一个 Mini-OJ 串起 Java 课程, 并贯穿一条判题机三代演进。课程调整: 第 **13** 章网络 **(Socket)** 不考, 已删除; 新增第 **15** 章泛型与集合 (并入 M5a)。

判题机三代主线

代	里程碑	判题方式	关键技术
第一代	M1–M3(已完成)	Java 对象在 JVM 内模拟	`Solution.solve()` + 多态
第二代	M4	反射工厂 + 单文件配置	`Class.forName` + `config.txt`
第三代	M5a	外部 C++ 判题机真编译真运行	`ProcessBuilder` + `setrlimit`

考试相关度（★ = 历年真题强相关）

依据新疆大学《面向对象程序设计 B / JAVA 程序设计》历年真题（2016-2017、2019-2020 A 卷 + 复习提纲、笔试模拟）。题型固定：单选 + 判断 + 读程序 + 程序填空 + 编程，各约 20 分。

模块（章）	★相关度	真题题型（原题级）	里程碑
类与对象/构造/封装/static/this(Ch4)	★★★	编程大题（Vehicle 类）、读程序、选择/判断	M2
继承/重写/多态/抽象类/接口/异常（Ch5-7）	★★★	编程大题（抽象类 Shape->Circle/Rectangle）、读程序、选择	M3
Swing GUI + 事件(Ch9)	★★★	编程（平方/按钮窗口）+ 填空（六按钮 GridLayout）双大题、选择	M5c
集合 ArrayList/LinkedList/HashMap(Ch15)	★★	读程序（Iterator）、填空（查询入 ArrayList）	M5a
JDBC 查询/更新(Ch11)	★★	填空（查询入 List）、原题（score_t / student 表）	M5b
多线程 Thread/Runnable/join/synchronized(Ch12)	★★	填空（线程类）、选择	M6a
String/StringBuffer/Random/Math(Ch8)	★★	读程序（==/equals）、选择	M4
数组/运算符/流程(Ch2-3)	★	选择、读程序	M1
异常 try-catch(Ch7)	★	选择、判断	M3

模块（章）	★相关度	真题题型（原题级）	里程碑
文件 File/流/序列化（Ch10）	★	少量选择/判断（File.mkdir、System.in.read、对象串行化）	M4/M5a
环境/编译运行（Ch1）	☆	少量选择	M0
网络 Socket(Ch13)	✗ 已删	历年仅 1 道判断	—

> 工程化加料（反射工厂、外部 C++ 判题机、ProcessBuilder、ProblemService/泛型架构、线程池/队列、MVC 解耦）不是考点，属项目深度。

两步走（每个里程碑先「拟合试卷」再「工程化」）

本项目偏工程化。为让学生先考得了、上得了手，每个里程碑拆成两步：

1. 第一步 [**preliminary** ·拟合试卷]：按真题难度做最小可运行版（教材/考试写法），快速有参与感、直接覆盖考点。
2. 第二步 [工程化]：在此基础上引入反射 / 外部 C++ 判题机 / 泛型架构 / 数据库 / 线程池 / MVC 解耦等工程方法。

> 各章文档开头已标 ★相关度与 [第一步]/[第二步] 的具体内容。考前只过第一步即可覆盖真题；想做工程深度再上第二步。

里程碑总览

里程碑	章节	主题	★	代	产物
M0	Ch1	环境 + HelloWorld	☆	—	能编译运行的 `Main.java`
M1	Ch2–3	单题判题器（硬编码）	★	一	数组 + 流程控制判 A+B
M2	Ch4	对象建模	★★★	一	`oj.core` 包 + jar + javadoc
M3	Ch5–7	接口/继承/多态/异常	★★★	一	`Judge` 接口 + `AbstractJudge` + 子类
M4	Ch8 + Ch10(单文件)	第二代：反射 + 单文件配置	★★ Ch8 / 工程反射	二	`ConfigFile/SingleFilePro`

里程碑	章节	主题	★	代	产物
M5a	Ch15 + Ch10(工业级)	第三代: 泛型集合 + C++ 判题机	★★ Ch15 / 工程 C++	三	ProblemLoader(List)/Prob
M5b	Ch11	数据库衔接 (FS/DB 分工)	★★	—	Db/ProblemDao/Submission
M5c	Ch9	Swing 大前端 (MVC, 无网络)	★★★	—	OjFrame/OjController
M6a	Ch12	多线程并发判题 (收官)	★★ 线程 / 工程队列	三	JudgeTask/JudgeQueue/Jud

最终架构 (无网络, 本地大前端)

```
        Swing 桌面客户端(Ch9)    ——MVC——
          | 选题/选语言/贴源码          |
          v                           v
   多线程判题队列(Ch12) --调用--> C++ 判题机(judge)
          |                      真编译/真运行/真 setrlimit
          v                           | 一行 JSON
     MySQL(Ch11)  <--元数据/历史---- ProblemService
          ^                           |
          +-- 文件系统:problems/<id>/{config,N.in,N.out} + submissions/ 源码
```

项目目录 (最终形态)

```
mini-oj/
+-- src/oj/
|   +-- core/       Status TestCase ProblemMeta Problem JudgeResult Submission JudgeTask
|   +-- judge/      Solution Judge AbstractJudge StandardJudge SpecialJudge
|   |   |           JudgeFactory(M4) MachineJudge(M5a)
|   |   +-- queue/  JudgeQueue JudgeWorker (M6a)
|   +-- exception/  JudgeException TimeLimitException (M3)
|   +-- io/         ConfigFile SingleFileProblemLoader (M4)
|   |               ProblemLoader ProblemRepository SubmissionStore (M5a)
|   +-- db/         Db ProblemDao SubmissionDao (M5b)
|   +-- service/    ProblemService (M5b)
|   +-- gui/        OjFrame OjController (M5c)
|   +-- Main.java
```

```
+-- problems/<id>/  config.txt + N.in/N.out
+-- submissions/    选手源码落地
+-- judge/          C++ 判题机(judge.cpp,编译型)
+-- lib/            mysql-connector-j.jar (M5b)
+-- Makefile
```

工具链演进

- **M0–M1**:nvim + javac + java + javap, 单文件。
- **M2**: 多文件 + package,Makefile,jar,javadoc。
- **M4**:File/BufferedReader 读 config.txt;g++ 预编译出 judge 二进制 (M5a 起调用)。
- **M5b**: 引入 MySQL 驱动 jar(java -cp build:lib/*)。

逐里程碑要点 (★相关度·第一步拟合试卷 / 第二步工程化)

- **M0 Ch1** ☆: 走通编写-> 编译-> 运行;Main 打印就绪。
- **M1 Ch2–3** ★: 数组 +if/switch/循环, 硬编码判 A+B 出 AC/WA(真题: 数组遍历、++i、运算符)。
- **M2 Ch4** ★★★:[第一步 · 考试向] 用考试级写法把 OJ 的 TestCase/Problem/JudgeResult 写成最简普通类;[第二步 · 工程向] 收成 ProblemMeta+static+ 包/jar。
- **M3 Ch5–7** ★★★:[第一步 · 考试向] 写一个简单判题 (判出 AC/WA)+ try-catch;[第二步 · 工程向] 抽成 Judge 接口/AbstractJudge 模板 + 多态 + 自定义异常。
- **M4 Ch8+Ch10** ★★:[第一步 · 考试向] 用 trim/equals/split+StringBuilder 做 OJ 输出比对与报告;[第二步 · 工程向] config.txt+ 反射工厂、单文件读写。
- **M5a Ch15+Ch10** ★★:[第一步 · 考试向] 用 ArrayList/HashMap 装 OJ 的多组用例/题目 (内存题库);[第二步 · 工程向] ProblemLoader/Repository + ProcessBuilder 调 C++ 判题机 + 序列化。
- **M5b Ch11** ★★:[第一步 · 考试向] 用最简 JDBC(Statement) 把 OJ 的提交/题目存取 MySQL;[第二步 · 工程向] ProblemDao/事务/PreparedStatement/ProblemService FS/DB 分工。
- **M5c Ch9** ★★★:[第一步 · 考试向] 写 OJ 提交窗口最简版 (选题 + 输入 + 提交 + 标签显示结果);[第二步 · 工程向] OjFrame/OjController MVC 解耦 + SwingWorker。
- **M6a Ch12** ★★:[第一步·考试向] 给 OJ 判题加最简并发 (Thread/Runnable+synchronized/join);[第二步 · 工程向] 泛型阻塞队列 + 线程池 + 调 C++ 判题机 (收官)。

学习方法

- 一里程碑一可运行产物, 过了验收再进下一阶段。
- 每章新知识先用在 OJ 的真实需求上。
- 每个里程碑 git 提交一次, 方便回看演进。

> 进度:M0 ☐ M1 ☐ M2 ☐ M3 ☐ M4 ☐ M5a ☐ M5b ☐ M5c ☐ M6a ☐

Mini-OJ 总设计 (全局契约)

> 这份是地基: 定下全项目的包结构、类/接口清单、核心数据模型签名。各章文档都引用本文, 不得擅自改名或改签名。原则: 简单、教学优先; 同时贯穿一条「判题机三代演进」主线。

1. 判题机三代主线 (全项目的骨架)

代	里程碑	判题方式	关键技术
第一代	M1–M3(已完成)	Java 对象在 JVM 内模拟判题	`Solution.solve()` + `Judge` 多态
第二代	M4	反射工厂 + 单文件配置	`Class.forName` + `config.txt`
第三代	M5a	外部 C++ 判题机, 真编译/真运行/真限资源	`ProcessBuilder` -> `judge` 二进制 + `setrlimit`

第三代之后 (M5b/M5c/M6a) 都是围绕” 第三代” 做工程化: 数据库供配置、Swing 做前端、多线程做并发。

> 课程调整: 第 **13** 章网络编程 **(Socket)** 不考, 已删除 (原 M6b 取消,M6a 即收官); 新增第 **15** 章泛型与集合, 并入 M5a。考试相关度 (★, 基于历年真题) 与「两步走 (第一步拟合试卷 / 第二步工程化)」见 `mini-oj-plan.md` 的「考试相关度」「两步走」两节; 各章开头也标了 ★ 与第一步/第二步。

2. 关键设计决策

决策	选择	理由
判题结果状态	`enum Status`	最干净 (enum 不在大纲但极简单)
第一代用户解法	`Solution` 接口:`String solve(String)`	纯 Java, 教接口 + 多态
第二代换判题器	`JudgeFactory.create(类名)` 反射 + `config.txt`	换题/换判法不改代码
第三代真判题	`MachineJudge` 用 `ProcessBuilder` 调外部 C++ 判题机	真限 CPU/内存、支持多语言
题目元数据来源	M4 `config.txt` -> M5b MySQL(`ProblemMeta` 作接缝)	数据源可换,`Problem` 不变

决策	选择	理由
FS/DB 分工	元数据/历史进 DB;.in/.out 与源码留文件系统	各取所长
TLE/MLE	由 C++ 判题机 setrlimit+ 墙钟掐断	Java 侧不外套超时

3. 最终包结构 (逐章长出来)

```
mini-oj/
+-- src/oj/
|   +-- core/          Status, TestCase, ProblemMeta, Problem, JudgeResult, Submission, JudgeTask
|   +-- judge/         Solution, Judge, AbstractJudge, StandardJudge, SpecialJudge,
|   |   |              JudgeFactory(M4 反射), MachineJudge(M5a 第三代)
|   |   +-- queue/     JudgeQueue, JudgeWorker                                  (M6a)
|   +-- exception/     JudgeException, TimeLimitException                       (M3)
|   +-- io/            ConfigFile, SingleFileProblemLoader (M4),
|   |                  ProblemLoader, ProblemRepository, SubmissionStore    (M5a)
|   +-- db/            Db, ProblemDao, SubmissionDao                            (M5b)
|   +-- service/       ProblemService(DB 元数据 + FS 测试点合流)                (M5b)
|   +-- gui/           OjFrame, OjController                                    (M5c)
|   +-- Main.java
+-- problems/          题库:<id>/config.txt, <id>/N.in, <id>/N.out
+-- submissions/       选手源码落地(<id>.cpp/.py)                               (M6a)
+-- judge/             C++ 判题机(单文件 judge.cpp,编译型)                  (M5a 起调用)
+-- lib/               mysql-connector-j.jar                                    (M5b)
+-- Makefile
```

> 已删除 net/(JudgeServer/JudgeClient/Socket)——第 13 章网络不考。

4. 全量类/接口清单

类/接口	包	引入	代	一句话职责
Main	(默认/oj)	M0	—	程序入口、各阶段演示
Status(enum)	core	M2	—	判题状态:AC/WA/TLE/MLE/RE/CE/

类/接口	包	引入	代	一句话职责
`TestCase`	core	M2	—	一组测试: 输入 + 期望输出
`ProblemMeta`	core	M4	—	题目元数据: 标题/判题类名/时限 (数据源可换)
`Problem`	core	M2(重构)	—	题目:id + `ProblemMeta` + 测试点
`JudgeResult`	core	M2	—	判题结果: 状态/通过数/总数/详情/耗时
`Submission`	core	M2	—	一次提交: 谁/哪题/语言/源码路径/结果
`JudgeTask`	core	M6a	三	在途工作项: 源码字节 + `complete/await`
`Solution`	judge	M2->M3	一	用户解法:`solve(input)->output`
`Judge/AbstractJudge/StandardJudge/SpecialJudge`	judge	M3/M4	一/二	Java 内判题器 (接口 + 模板 + 子类)
`JudgeFactory`	judge	M4	二	`Class.forName` 反射造判题器
`MachineJudge`	judge	M5a	三	`ProcessBuilder` 调外部 C++ 判题机
`JudgeException/TimeLimitException`	judge	M3	—	自定义异常
`ConfigFile`	io	M4	二	读 `config.txt` -> `ProblemMeta`
`SingleFileProblemLoader`	io	M4	二	单文件原型: 读 input/output/config
`ProblemLoader`	io	M5a	三	扫目录读多组用例 -> `List<TestCase>`
`ProblemRepository`	io	M5a	三	`HashMap<Integer,Problem>` 缓存题目

类/接口	包	引入	代	一句话职责
SubmissionStore	io	M5a	—	对象序列化 (M5b 后由 DAO 取代)
Db/ProblemDao/SubmissionDao	db	M5b	—	JDBC 连接/元数据/提交历史 (事务)
ProblemService	service	M5b	—	DB 元数据 + FS 测试点合流入口
OjFrame/OjController	gui	M5c	—	Swing 大前端 (MVC)
JudgeQueue/JudgeWorker	judge.queue	M6a	三	泛型阻塞队列 + 并发判题工作线程

5. 核心数据模型契约 (签名锁定, 后续章节照此实现)

```
// oj.core.Status ——全项目唯一权威定义
enum Status { AC, WA, TLE, MLE, RE, CE, PE }
// 谁产生谁: 第一代 Java 内判题 (M3/M4) 只产 AC/WA/RE(PE 由可选 FormatChecker 产);
//          第三代 C++ 判题机 (M5a) 产 AC/WA/TLE/MLE/RE/CE, 其内部错 ERR 由 MachineJudge
 ↪ 映射成 RE。

// oj.core.TestCase
class TestCase {
    private final String input, expected;
    TestCase(String input, String expected)
    String getInput();  String getExpected();
}

// oj.core.ProblemMeta ——元数据接缝:M4 来自 config.txt,M5b 来自 MySQL,Problem 不变
class ProblemMeta {
    private final String title;
    private final String judgeClass;          // 判题器全限定名, 如
     ↪ "oj.judge.StandardJudge"
    private final long timeLimitMs;
    ProblemMeta(String title, String judgeClass, long timeLimitMs)
    String getTitle(); String getJudgeClass(); long getTimeLimitMs();
}

// oj.core.Problem —— id + 元数据 + 测试点 (M4 用 TestCase[];M5a 起 Ch15 重构为
 ↪ List<TestCase>)
```

```
class Problem {
    private final int id;
    private final ProblemMeta meta;
    private final List<TestCase> cases;        // M4 阶段为 TestCase[]
    Problem(int id, ProblemMeta meta, List<TestCase> cases)
    int getId(); String getTitle(); String getJudgeClass(); long getTimeLimitMs();
    List<TestCase> getCases();
}

// oj.core.JudgeResult
class JudgeResult {
    private final Status status;
    private final int passed, total;
    private final String detail;
    private final long elapsedMs;
    JudgeResult(Status status, int passed, int total, String detail, long elapsedMs)
    Status getStatus(); int getPassed(); int getTotal(); String getDetail(); long
 ↪  getElapsedMs();
    boolean isAccepted();
    @Override String toString();               // "AC 3/3 (12ms)"
}   // implements Serializable

// oj.core.Submission  ——增加 lang + srcPath(用判题机/入库需要)
class Submission {                             // implements Serializable
    private static int counter = 0;
    private final int id;                      // counter 自增
    private final int problemId;
    private final String userName;
    private final String lang;                 // "cpp" | "python"
    private final String srcPath;              // 选手源码在文件系统的路径
    private JudgeResult result;
    Submission(int problemId, String userName, String lang, String srcPath)
    int getId(); int getProblemId(); String getUserName(); String getLang(); String
     ↪  getSrcPath();
    JudgeResult getResult(); void setResult(JudgeResult r);
    static int count();
}

// oj.judge.Solution(第一代,M3 起接口)/ Judge(M3)
interface Solution { String solve(String input); }
interface Judge { JudgeResult judge(Problem problem, Solution solution); }
```

```
// oj.judge.MachineJudge(第三代,M5a)
class MachineJudge {
    MachineJudge(String judgeBinary)
    JudgeResult judge(String problemDir, String srcFile, String lang, long timeMs, int
    ↪ memMb)
}
```

6. 命名 / 风格约定

- 包名小写 `oj.xxx`; 类名大驼峰; 方法/变量小驼峰; 常量全大写。
- 缩进 4 空格; 采用 K&R(左括号不换行)。每个 public 类/方法写一句 Javadoc。
- 一个文件一个 public 类;`main` 只放在 `Main`。

7. 文档清单 (本系列)

文件	对应	里程碑	代
`00-项目总设计.md`	全局契约	—	—
`01-Ch1-环境与HelloWorld.md`	第 1 章	M0	—
`02-Ch2-3-单题判题器.md`	第 2-3 章	M1	一
`03-Ch4-对象建模.md`	第 4 章	M2	一
`04-Ch5-7-判题器与异常.md`	第 5-7 章	M3	一
`05-M4-第二代判题机.md`	第 8 章 + 第 10 章 (单文件)	M4	二
`06-M5a-第三代判题机.md`	第 **15** 章 + 第 10 章 (工业级) + C++	M5a	三
`07-M5b-数据库衔接.md`	第 11 章	M5b	—
`08-M5c-Swing 客户端.md`	第 9 章	M5c	—
`09-M6a-多线程收官.md`	第 12 章 (收官)	M6a	三

判题机 (真实编译运行 C++/Python 提交) 是独立组件, 代码在仓库 `judge/`(单文件 `judge.cpp`)。第一代 Java 内判题 (`Judge`/`Solution`) 在 M3/M4 用于教 OOP; 第三代 (M5a 起) 真实判题走 `MachineJudge` 调 `judge/`。第 13 章网络 (`Socket`/`JudgeServer`/`JudgeClient`) 已删除, 不出现在任何章节。

01 ·第 1 章环境与 HelloWorld(里程碑 M0)

> 考试相关度 ☆ ·真题考点参考: 编译运行、字节码、.class(少量选择)。本章即「第一步·上手」, 纯命令行 javac/java/javap, 无两步拆分。

> 目标: 装好 JDK, 理解" 编写-> 编译-> 运行", 跑出第一个程序。本章只有一个文件、一个方法。

知识点 (大纲对照)

要点	处理
Java 地位/特点/James Gosling	了解即可, 不实现
JDK 安装与环境变量	动手
编写-> 编译-> 运行三步	动手 (核心)
javap 反编译	动手 (看一眼字节码即可)
编程风格 Allman/K&R、注释	了解, 本项目统一用 K&R

要写的代码

文件:Main.java(默认包, 先不分包)

类 Main

+-- public static void main(String[] args)

方法	签名	职责	关键逻辑
入口	public static void main(String[] args)	打印就绪信息	一行 System.out.println("Mini-OJ judge ready");

> 就这么简单。重点不在代码, 在于走通工具链。

动手步骤

```
# 1. 装 JDK(二选一)
sudo apt install openjdk-21-jdk
java -version && javac -version

# 2. 建目录、写代码
mkdir -p ~/mini-oj && cd ~/mini-oj
```

```
nvim Main.java          # 写上面的 Main 类

# 3. 编译 -> 运行
javac Main.java         # 生成 Main.class
java Main               # 输出: Mini-OJ judge ready

# 4. 反编译看字节码 (理解 .class)
javap -c Main
```

验收标准

- ☐ `java -version` 正常,`echo $JAVA_HOME` 有值 (如配了)。
- ☐ 终端打印 `Mini-OJ judge ready`。
- ☐ 能用自己的话说清:`Main.java` –javac–> `Main.class` –java(JVM)–> 运行。
- ☐ `javap -c Main` 能看到 `main` 方法的字节码 (认出 `getstatic`/`invokevirtual` 即可, 不求全懂)。

本章产物

一个能编译运行的 `Main.java`。下一章在它里面加判题逻辑。

02 ·第 2-3 章单题判题器 (里程碑 M1)

考试相关度 ★ ·真题以选择/读程序考数组遍历、运算符、`++i`、流程控制。本章即基础语法, 无需两步拆分。

目标: 还不用类, 只用基本类型、数组、循环、分支, 写出能判 A+B 的判题器, 跑出 `AC/WA`。全部代码仍在单文件 `Main.java`, 用 `static` 方法 + 数组组织。

知识点 (大纲对照)

要点	处理
标识符/关键字、基本数据类型、类型转换	动手 (int/double/String/类型转换)
Scanner 输入 / println 输出	动手
数组: 声明/分配/length/初始化/遍历	动手 (用例存数组)
算术/关系/逻辑/赋值运算符	动手
位运算符、instanceof	了解即可, 不实现
if / if-else / switch	动手
for / while / do-while、break/continue	动手

要写的代码

文件:Main.java(默认包, 纯 static 方法)

```
类 Main
+-- main                程序入口,依次跑三道题的判题
+-- solve               题1:模拟用户解法 a+b
+-- judgeAplusB         题1:遍历用例,统计 AC/WA 并打印
+-- grade               题2:switch 求成绩等级(练 switch)
+-- isPrime             题3:循环判素数(练 for/while/break)
+-- compareInt          通用:比对期望值与实际值
```

方法清单 (逐个)

方法	签名	职责	关键逻辑
入口	`public static void main(String[] a)`	跑三道题	顺序调 judgeAplusB() 等, 打印分隔
解法 1	`static int solve(int a, int b)`	模拟用户 A+B 解法	`return a + b;`
判题 1	`static void judgeAplusB()`	内置用例-> 逐组判-> 统计	见下方算法
比对	`static boolean compareInt(int expected, int actual)`	判断一组是否通过	`return expected == actual;`
解法 2	`static char grade(int score)`	分数-> 等级 (练 switch)	`switch(score/10){case 10,9->'A'; ...}`
解法 3	`static boolean isPrime(int n)`	判素数 (练循环)	`for(i=2;i*i<=n;i++) if(n%i==0) return false;`

> 题 2/题 3 只需各写一个简版 judgeXxx() 仿照 judgeAplusB, 确认 switch / 循环都用上即可, 不必三题都做满——够覆盖知识点就停。

judgeAplusB 算法 (核心, 讲清数组 + 循环 + 分支)

```
// 用例:每行 {a, b};期望:对应 a+b
int[][] inputs   = { {1,2}, {10,20}, {0,0} };
int[] expected   = {   3,     30,      0  };
```

```
int passed = 0;
for (int i = 0; i < inputs.length; i++) {          // 数组 length + for 遍历
    int actual = solve(inputs[i][0], inputs[i][1]);
    if (compareInt(expected[i], actual)) {          // 分支
        passed++;
    } else {
        System.out.println("case#" + i + " 期望 " + expected[i] + " 实际 " + actual);
        // 可 continue / break,体会两者区别
    }
}
String status = (passed == inputs.length) ? "AC" : "WA";   // 三元 + 关系运算
System.out.println("A+B: " + status + " " + passed + "/" + inputs.length);
```

> 故意制造一组错的期望值, 观察输出从 AC 变 WA 并打出失败行——这就是”判题”的内核。

动手步骤

```
cd ~/mini-oj
nvim Main.java            # 按上面方法清单补全
javac Main.java && java Main
# 期望输出形如:
#   A+B: AC 3/3
#   Grade(85): B
#   isPrime(7): true
```

验收标准

- ☐ 三道题都能跑,A+B 打印 AC 3/3。
- ☐ 把某组 expected 改错 -> 输出变 WA 2/3 且打出失败的 case 行。
- ☐ 代码里确实用到了: 数组遍历、if/三元、switch、for(或 while)。
- ☐ 能说清”为什么现在代码很难扩展到多道题”——为下一章引入”类/对象”埋伏笔。

本章产物 -> 下一章

一个单文件、面向过程的判题器。痛点: 题目、用例、结果全是散落的数组和变量, 加一道题要改一堆地方。第 3 章 (M2) 用类与对象把它们封装成 Problem/TestCase/JudgeResult。

03 ·第 4 章对象建模 (里程碑 M2)

考试相关度 ★★★(编程大题核心) ·真题考点参考: 定义类 (属性/构造/get-set/封装)。[第一步] 考试向 用考试级写法把 OJ 的 TestCase/Problem/JudgeResult 写成最简普通类 (字段 + 构造 +get/set), 能 new 能打印;[第二步] 工程向 收成 ProblemMeta + static 计数 + 包/jar。

目标: 把 M1 散落的数组/变量, 封装成 oj.core 的几个类; 判题逻辑收进一个 SimpleJudge(还没学接口, 先用具体类)。学会包、jar、javadoc。

知识点 (大纲对照)

要点	处理
类/成员变量/方法/构造方法/this	动手 (贯穿)
封装 (private + getter)	动手
实例成员 vs 类成员 (static)	动手 (Submission.counter)
方法重载、可变参数	动手 (构造重载 + TestCase...)
对象数组	动手 (Problem.cases)
包 package / import / 访问权限	动手 (迁进 oj.core)
可运行 jar、javadoc	动手
基本类型的类封装、var 局部变量	了解即可, 可不专门写

要写的类 (本章共 6 个)

oj.core: Status(enum) TestCase Problem JudgeResult Submission
oj.judge: Solution(本章先做成具体类) SimpleJudge

Status(enum) — oj/core/Status.java

```
package oj.core;
public enum Status { AC, WA, TLE, MLE, RE, CE, PE }   // 权威定义见 OO §5;MLE/CE 主要由判
↪ 题机 (第 11 章) 产生,Java 内判题用不到
```

TestCase — oj/core/TestCase.java

```
package oj.core;
public class TestCase {
    private final String input;
    private final String expected;
    public TestCase(String input, String expected) {
```

```
        this.input = input;            // this 区分参数与字段
        this.expected = expected;
    }
    public String getInput()    { return input; }
    public String getExpected() { return expected; }
}
```

Problem — oj/core/Problem.java(对象数组 + 构造重载 + 可变参数)

```
package oj.core;
public class Problem {
    private final int id;
    private final String title;
    private final TestCase[] cases;        // 对象数组
    private final long timeLimitMs;
    // 构造重载①: 默认时限 1000ms; 可变参数 TestCase... 本质就是 TestCase[]
    public Problem(int id, String title, TestCase... cases) {
        this(id, title, 1000, cases);      // this(...) 调另一个构造方法
    }
    // 构造重载②: 显式时限
    public Problem(int id, String title, long timeLimitMs, TestCase... cases) {
        this.id = id; this.title = title;
        this.timeLimitMs = timeLimitMs; this.cases = cases;
    }
    public int getId()           { return id; }
    public String getTitle()     { return title; }
    public TestCase[] getCases() { return cases; }
    public long getTimeLimitMs() { return timeLimitMs; }
}
```

注: 为可变参数好用, 把 timeLimitMs 放在 cases 前; 签名与 OO 契约等价 (TestCase... 即 TestCase[])。

JudgeResult — oj/core/JudgeResult.java

```
package oj.core;
public class JudgeResult {
    private final Status status;
    private final int passed, total;
```

```
    private final String detail;
    private final long elapsedMs;
    public JudgeResult(Status status, int passed, int total, String detail, long
     ↪  elapsedMs) {
        this.status=status; this.passed=passed; this.total=total;
        this.detail=detail; this.elapsedMs=elapsedMs;
    }
    public Status getStatus()  { return status; }
    public int getPassed()     { return passed; }
    public int getTotal()      { return total; }
    public String getDetail()  { return detail; }
    public long getElapsedMs() { return elapsedMs; }
    public boolean isAccepted(){ return status == Status.AC; }
    @Override public String toString() {
        return status + " " + passed + "/" + total + " (" + elapsedMs + "ms)";
    }
}
```

Submission — oj/core/Submission.java(类变量 static)

```
package oj.core;
public class Submission {
    private static int counter = 0;     // 类变量: 所有提交共享
    private final int id;
    private final int problemId;
    private final String userName;
    private JudgeResult result;         // 判完回填
    public Submission(int problemId, String userName) {
        this.id = ++counter;            // 自增得到唯一 id
        this.problemId = problemId;
        this.userName = userName;
    }
    public int getId()            { return id; }
    public int getProblemId()     { return problemId; }
    public String getUserName()   { return userName; }
    public JudgeResult getResult(){ return result; }
    public void setResult(JudgeResult r){ this.result = r; }
    public static int count()     { return counter; }   // 类方法
}
```

> `time` 字段 (LocalDateTime) 第 5 章 (M4) 再加, 避免本章引入日期类。

Solution(本章 = 具体类) — `oj/judge/AplusB.java`

```
package oj.judge;
import java.util.Scanner;
public class AplusB {                    // 第 6 章会抽象成 Solution 接口
    public String solve(String input) {
        Scanner sc = new Scanner(input);
        return String.valueOf(sc.nextInt() + sc.nextInt());
    }
}
```

SimpleJudge — `oj/judge/SimpleJudge.java`(本章核心)

```
package oj.judge;
import oj.core.*;
public class SimpleJudge {
    // 方法重载示例: 可指定/不指定用户名
    public JudgeResult judge(Problem p, AplusB solution) {
        return judge(p, solution, "anonymous");
    }
    public JudgeResult judge(Problem p, AplusB solution, String user) {
        TestCase[] cases = p.getCases();
        int passed = 0;
        String detail = "";
        long start = System.currentTimeMillis();
        for (int i = 0; i < cases.length; i++) {
            String actual   = solution.solve(cases[i].getInput()).trim();
            String expected = cases[i].getExpected().trim();
            if (actual.equals(expected)) {
                passed++;
            } else if (detail.isEmpty()) {
                detail = "case#" + i + " 期望 [" + expected + "] 实际 [" + actual + "]";
            }
        }
        long elapsed = System.currentTimeMillis() - start;
        Status st = (passed == cases.length) ? Status.AC : Status.WA;
        return new JudgeResult(st, passed, cases.length, detail, elapsed);
    }
```

```
}
```

Main(演示) — oj/Main.java

```
package oj;
import oj.core.*;
import oj.judge.*;
public class Main {
    public static void main(String[] args) {
        Problem p = new Problem(1, "A+B",
            new TestCase("1 2", "3"),
            new TestCase("10 20", "30"));
        JudgeResult r = new SimpleJudge().judge(p, new AplusB(), "alice");
        System.out.println(p.getTitle() + " -> " + r);   // 用到 toString
        System.out.println(" 总提交数: " + Submission.count());
    }
}
```

动手步骤 (编译/jar/javadoc)

目录:src/oj/...。写个 Makefile:

```
build:
    javac -d build $(shell find src -name '*.java')
run: build
    java -cp build oj.Main
jar: build
    jar cfe mini-oj.jar oj.Main -C build .
doc:
    javadoc -d doc -sourcepath src -subpackages oj
```

```
make run          # 跑演示
make jar && java -jar mini-oj.jar
make doc          # 生成 API 文档到 doc/
```

动手实现: 两步走 (javac / java 实操)

约定: 源码放 src/oj/..., 编译到 build/, 全程 javac/java, 不用 make。

第一步 考试向 — 把 OJ 的数据写成最简普通类

src/oj/core/TestCase.java

```
package oj.core;
public class TestCase {
    private String input, expected;
    public TestCase(String input, String expected) { this.input = input; this.expected
     ↪  = expected; }
    public String getInput()    { return input; }
    public String getExpected() { return expected; }
}
```

讲解: 一组测试 = 输入 + 期望输出。考试编程题的”普通类”写法: 私有字段 + 构造方法 + getter。

src/oj/core/Problem.java

```
package oj.core;
public class Problem {
    private int id; private String title; private TestCase[] cases;
    public Problem(int id, String title, TestCase[] cases) { this.id = id; this.title
     ↪  = title; this.cases = cases; }
    public int getId()           { return id; }
    public String getTitle()     { return title; }
    public TestCase[] getCases() { return cases; }   // 对象数组
}
```

讲解: 题目 = 编号 + 标题 + 一组测试点 (对象数组)。

src/oj/core/JudgeResult.java

```
package oj.core;
public class JudgeResult {
    private String status; private int passed, total;
    public JudgeResult(String status, int passed, int total) { this.status = status;
     ↪  this.passed = passed; this.total = total; }
    @Override public String toString() { return status + " " + passed + "/" + total; }
}
```

src/oj/Demo.java

```
package oj;
import oj.core.*;
public class Demo {
```

```
    public static void main(String[] args) {
        Problem p = new Problem(1, "A+B", new TestCase[]{ new TestCase("1 2","3"), new
↪   TestCase("10 20","30") });
        System.out.println(p.getTitle() + " 有 " + p.getCases().length + " 组用例");
        System.out.println(new JudgeResult("AC", 2, 2));
    }
}
```

编译运行:

```
javac -d build src/oj/core/TestCase.java src/oj/core/Problem.java
↪   src/oj/core/JudgeResult.java src/oj/Demo.java
java -cp build oj.Demo
# A+B 有 2 组用例
# AC 2/2
```

第二步 工程向 —— 升级为契约版

把 Status 改 enum、JudgeResult 补 detail/elapsedMs、抽出 ProblemMeta(标题/判题类名/时限) 让 Problem = id + ProblemMeta + 测试点、Submission 加 static 计数 (代码见上文【新产物架构】)。整项目一条命令编译:

```
javac -d build $(find src -name '*.java')
java -cp build oj.Main
```

验收标准

- ☐ M1 的判题逻辑现在全部由对象承载,make run 输出 A+B -> AC 2/2 (...)。
- ☐ 用到了:private+getter、构造重载、this、static 计数、对象数组、TestCase... 可变参数、包与 import。
- ☐ java -jar mini-oj.jar 能跑;doc/index.html 能打开。
- ☐ 能说清”加一道新题”现在只需 new Problem(...), 比 M1 改数组干净多了。

本章产物 -> 下一章

SimpleJudge 只会”精确比对”, 且写死在一个类里。第 5-7 章 (M3) 学接口/抽象类/多态后, 把它重构成 Judge 接口 + AbstractJudge + 多种判题器, 并用异常处理用户解法崩溃。

04 ·第 5-7 章判题器: 接口/继承/多态/异常 (里程碑 M3)

考试相关度 ★★★(编程大题核心) ·真题考点参考: 抽象类/继承/重写/多态/异常。[第一步] 考试向 用考试级写法写一个简单判题 (把一题判出 AC/WA)+ try-catch 兜异常;[第二步] 工程向 抽成 Judge 接口 / AbstractJudge 模板 + 多态 + 自定义异常。

本章是面向对象的重头戏。我们把 M2 写死的 SimpleJudge 重构成一套可扩展的判题器: Judge 接口 -> AbstractJudge 抽象类 (模板)-> StandardJudge 子类 (继承 + 重写); 并用自定义异常处理用户解法崩溃。主线一句话: 抽象出”判题器”这个概念, 以后加新判法只加类、不改老代码 (开闭原则)。

知识点 (大纲对照)——本章覆盖最密

主题	要点	处理
接口 (Ch6)	interface/实现/接口回调/函数接口 +Lambda/接口参数/接口多态	动手 (Solution、Judge)
继承 (Ch5)	子类 extends/方法重写 @Override/super/final/上转型/多态	动手 (AbstractJudge-> 子类)
抽象 (Ch5)	abstract 类与方法/面向抽象/开闭原则	动手 (模板方法)
内部类 (Ch7)	内部类/匿名类/Lambda 代替匿名类	动手 (比较器回调)
异常 (Ch7)	try-catch/自定义异常类/断言	动手 (RE 处理 + JudgeException)
成员变量隐藏	(Ch5)	了解即可, 不刻意制造

第一步: 把 Solution 变成接口 (Ch6)

oj/judge/Solution.java

```
package oj.judge;
public interface Solution {            // 函数式接口 (只一个抽象方法)-> 可用 Lambda
    String solve(String input);
}
```

原来的 AplusB 改成实现接口:

```
package oj.judge;
import java.util.Scanner;
public class AplusB implements Solution {
```

```
    @Override public String solve(String input) {
        Scanner sc = new Scanner(input);
        return String.valueOf(sc.nextInt() + sc.nextInt());
    }
}
```

接口 = 函数式接口的好处: 用户解法可以直接用 Lambda 写, 不必建类:

```
Solution s = input -> {                    // Lambda 实现 Solution
    Scanner sc = new Scanner(input);
    return String.valueOf(sc.nextInt() + sc.nextInt());
};
```

第二步:Judge 接口 + AbstractJudge 抽象类 (Ch5+Ch6 合体)

接口 oj/judge/Judge.java

```
package oj.judge;
import oj.core.*;
public interface Judge {
    JudgeResult judge(Problem problem, Solution solution);
}
```

抽象类 (模板方法)oj/judge/AbstractJudge.java——本章核心

```
package oj.judge;
import oj.core.*;
public abstract class AbstractJudge implements Judge {
    // final: 模板流程固定, 子类不许改; 子类只能定制 compare(开闭原则的" 闭")
    @Override
    public final JudgeResult judge(Problem p, Solution s) {
        TestCase[] cases = p.getCases();
        int passed = 0;
        String detail = "";
        long start = System.currentTimeMillis();
        try {
            for (int i = 0; i < cases.length; i++) {
                String actual = s.solve(cases[i].getInput());   // 用户解法, 可能抛异常
                if (compare(cases[i].getExpected(), actual)) {  // 调子类实现 (多态)
                    passed++;
```

```
                } else if (detail.isEmpty()) {
                    detail = "case#" + i + " 期望 [" + cases[i].getExpected().trim()
                           + "] 实际 [" + (actual == null ? "null" : actual.trim()) +
                            ↪  "]";
                }
            }
        } catch (RuntimeException e) {                          // 捕获运行期异常 -> RE
            long t = System.currentTimeMillis() - start;
            return new JudgeResult(Status.RE, passed, cases.length,
                                   " 运行异常: " + e.getMessage(), t);
        }
        long elapsed = System.currentTimeMillis() - start;
        assert passed <= cases.length : "passed 不应超过 total";   // 断言: 内部不变量
        Status st = (passed == cases.length) ? Status.AC : Status.WA;
        return new JudgeResult(st, passed, cases.length, detail, elapsed);
    }

    // 抽象方法:" 怎么比对一组输出" 留给子类 -> 这就是可扩展点
    protected abstract boolean compare(String expected, String actual);
}
```

这一个抽象方法 compare 是整套设计的” 活接头”: 换判法 = 写个新子类重写 compare, 模板流程一行不动。

子类: 精确判题器 oj/judge/StandardJudge.java(继承 + 重写 + super)

```
package oj.judge;
public class StandardJudge extends AbstractJudge {
    @Override
    protected boolean compare(String expected, String actual) {
        if (actual == null) return false;
        return expected.trim().equals(actual.trim());       // 去首尾空白后精确比对
    }
}
```

super 的典型用法 (可选演示): 子类构造里 super(...) 调父类构造; 本章父类无显式构造, 故从略, 留到有字段的子类时再讲。

第三步: 多态 + 开闭原则 (Ch5)

```
Judge judge;
if (useFloat) judge = new SpecialJudge();   // 第 8 章 (M4) 实现的浮点判题器
else          judge = new StandardJudge();  // 上转型: 父类型引用指向子类对象
JudgeResult r = judge.judge(problem, solution);   // 运行时多态: 调到具体子类的 compare
```

开闭原则: 将来要支持”忽略空白”“按行无序”等新判法, 只新增 XxxJudge extends AbstractJudge, Main/调用方代码完全不用改。

第四步: 内部类 / 匿名类 / Lambda(Ch7)

判题里常要一个”输出比较器”回调。三种写法演示演化关系:

```
// 1) 命名内部类
class IgnoreCaseComparator implements OutputComparator {
    public boolean same(String a, String b) { return
     ↪  a.trim().equalsIgnoreCase(b.trim()); }
}
// 2) 匿名类 (就地实现)
OutputComparator c2 = new OutputComparator() {
    public boolean same(String a, String b) { return a.trim().equals(b.trim()); }
};
// 3) Lambda 代替匿名类 (函数式接口才行)
OutputComparator c3 = (a, b) -> a.trim().equals(b.trim());
```

其中:

```
package oj.judge;
public interface OutputComparator {     // 函数式接口
    boolean same(String a, String b);
}
```

体会: 匿名类 -> Lambda 是同一件事的简写。OutputComparator 属”可选增强”, 核心判题用 compare 即可, 不强求全用上。

第五步: 自定义异常 (Ch7)

oj/exception/JudgeException.java

```
package oj.exception;
public class JudgeException extends Exception {     // 受检异常: 判题流程的异常基类
```

```
    public JudgeException(String message) { super(message); }
}
```

oj/exception/TimeLimitException.java

```
package oj.exception;
public class TimeLimitException extends JudgeException {   // 继承自定义异常
    public TimeLimitException(String message) { super(message); }
}
```

本章先定义这两个异常类 (练"怎么写自定义异常、怎么继承异常")。TimeLimitException 的真正触发在第 12 章 (M6a) 用线程做超时控制时; 本章 RE 用捕获 RuntimeException 即可。

第六步: 更新 Main 演示

```
package oj;
import oj.core.*;
import oj.judge.*;
public class Main {
    public static void main(String[] args) {
        Problem p = new Problem(1, "A+B",
            new TestCase("1 2", "3"), new TestCase("10 20", "30"));

        Judge judge = new StandardJudge();          // 多态: 用接口类型接收
        Solution good = new AplusB();
        System.out.println(" 正常: " + judge.judge(p, good));        // AC 2/2

        Solution bad = input -> { throw new RuntimeException("boom"); };  // Lambda 造一
↪   个会崩的解法
        System.out.println(" 崩溃: " + judge.judge(p, bad));         // RE
    }
}
```

跑断言需加 -ea:java -ea -cp build oj.Main。

动手实现: 两步走 (javac / java 实操)

接上一章 M2 的 core 类 (Problem/TestCase/JudgeResult)。

第一步 考试向 —— 一个简单判题器, 判出 AC/WA, 并用 try-catch 兜异常

src/oj/judge/Solution.java(函数式接口, 考点 Ch6)

```
package oj.judge;
public interface Solution { String solve(String input); }   // 输入一组数据 -> 产生输出
```

src/oj/judge/SimpleJudge.java

```
package oj.judge;
import oj.core.*;
public class SimpleJudge {
    public JudgeResult judge(Problem p, Solution s) {
        TestCase[] cs = p.getCases();
        int passed = 0;
        try {
            for (TestCase c : cs)
                if (s.solve(c.getInput()).trim().equals(c.getExpected().trim()))
                 ↪  passed++;
        } catch (RuntimeException e) {
            return new JudgeResult("RE", passed, cs.length);   // 用户解法崩溃 -> RE
        }
        return new JudgeResult(passed == cs.length ? "AC" : "WA", passed, cs.length);
    }
}
```

讲解: 遍历用例、trim().equals() 比对、try-catch 把崩溃兜成 RE——考试读程序/编程题的常见写法。

src/oj/Demo3.java

```
package oj;
import oj.core.*;
import oj.judge.*;
public class Demo3 {
    public static void main(String[] args) {
        Problem p = new Problem(1, "A+B", new TestCase[]{ new TestCase("1 2","3"), new
 ↪  TestCase("10 20","30") });
        Solution good = in -> { String[] t = in.split("\\s+"); return "" +
 ↪  (Integer.parseInt(t[0]) + Integer.parseInt(t[1])); };
        Solution bad  = in -> { throw new RuntimeException("boom"); };   // Lambda 实现
 ↪  接口
        System.out.println(new SimpleJudge().judge(p, good));   // AC 2/2
        System.out.println(new SimpleJudge().judge(p, bad));    // RE 0/2
```

```
    }
}
```

编译运行:

```
javac -d build src/oj/core/*.java src/oj/judge/Solution.java
 ↪  src/oj/judge/SimpleJudge.java src/oj/Demo3.java
java -cp build oj.Demo3
# AC 2/2
# RE 0/2
```

第二步 工程向 — 抽象化: 接口 + 模板 + 多态 + 自定义异常

把 SimpleJudge 重构成 Judge 接口 + AbstractJudge(模板方法,final judge + 抽象 compare)+ StandardJudge/SpecialJudge 子类, 加 oj.exception.JudgeException(代码见上文【新产物架构】)。多态调用 + 开闭原则; 断言要加 -ea:

```
javac -d build $(find src -name '*.java')
java -ea -cp build oj.Main
```

验收标准

- ☐ Judge/Solution 是接口;AbstractJudge 是抽象类且实现了 Judge;StandardJudge 继承 AbstractJudge 并 @Override compare。
- ☐ 用接口类型变量接收子类对象, 跑出多态 (judge.judge(...) 调到子类逻辑)。
- ☐ 会崩溃的解法被判 **RE** 而不是让程序挂掉。
- ☐ 至少演示了" 匿名类 -> Lambda" 两种等价写法之一。
- ☐ 定义了自定义异常 JudgeException/TimeLimitException(extends), 并能说清它们的继承关系。
- ☐ 能讲清开闭原则: 新增一种判法时, 你只新增了类、没动 AbstractJudge 和 Main。

本章产物 -> 下一章

判题器框架成型, 但只能" 精确/浮点" 地比。第 8 章 (M4) 用 **String**/正则/反射实现 SpecialJudge、做格式校验 (PE), 并用反射按类名动态加载判题器, 让判法彻底" 可插拔"。

05 ·M4 第二代判题机 (Ch8 字符串/反射 + Ch10 单文件)

考试相关度 ★★**(Ch8 String)**·反射/单文件属工程·真题考点参考:String ==/equals、StringBuffer、Random/Math。[第一步] 考试向 用 `trim/equals/split+StringBuilder` 做 OJ 的输出比对与判题报告;[第二步] 工程向 `config.txt` + `Class.forName` 反射工厂、单文件读写。

第一代判题机能跑、但题目是硬编码的，每新增一道题都要改 Java 源码重新编译。第二代的目标是：读一份配置文件，自动决定用哪个判题器、跑哪组测试点——“配置驱动，反射解耦”。

【核心痛点】

M1–M3 的代码大致如下：

```
// 硬编码时代 ——全是 magic number 和写死的类名
Problem p = new Problem(1, "A+B", new TestCase[]{
    new TestCase("1 2", "3"),
    new TestCase("10 20", "30")
});
Judge judge = new StandardJudge();   // 换题换判题器？改代码、重新编译
JudgeResult r = judge.judge(p, new MySolution());
```

三个直接缺陷：

1. 题目元数据硬编码：标题、时间限制、判题器类名全散在 `main` 里，加一道题就改一处代码。
2. 判题器写死：`new StandardJudge()` 或 `new SpecialJudge()` 在源码里固定，无法按题目切换。
3. 测试点无处管理：`TestCase` 对象在代码里手打，完全无法复用，也无法离线维护。

M4 的答案：把元数据写进 `config.txt`，把测试点写进 `input.txt/output.txt`，用 `Class.forName` 反射造判题器——代码不动，换题只换文件。

【引入课本知识点】

Ch8 ·字符串处理

`String.split` 切分 key=value

`config.txt` 每行形如 `title=A+B Problem`，读出来之后：

```
String[] parts = line.split("=", 2);   // 限制 2 份，防止 value 里含 "="
String key   = parts[0].trim();
String value = parts[1].trim();
```

`split("=", 2)` 是本节最小可用的正则应用：`=` 是字面量分隔符，`2` 是 limit 参数——只切第一个等号，value 部分原封不动保留。

`Math.abs` 浮点容差（SpecialJudge 核心逻辑）

精确比对对浮点题不公平，标准做法：

```
double expected = Double.parseDouble(exp.trim());
double actual   = Double.parseDouble(out.trim());
boolean ok = Math.abs(expected - actual) < 1e-6;
```

`Math.abs` 返回绝对值，`1e-6` 是科学计数法字面量——两个 Ch8 知识点同时落地。

Ch8 ·反射基础

`Class.forName(String name)` 在运行期按全限定名查找并加载类；`.getDeclaredConstructor().newInstance()` 无参构造出对象——这是 Java 反射的最小实践：

```
Class<?> clazz = Class.forName("oj.judge.StandardJudge");
Judge judge = (Judge) clazz.getDeclaredConstructor().newInstance();
```

不需要提前 `import`，不需要写死类型——判题器类名从配置文件读进来就能用。

Ch10 ·文件读写基础

```
File dir = new File("problems/1/");
File cfg = new File(dir, "config.txt");   // 拼路径，平台无关
BufferedReader br = new BufferedReader(new FileReader(cfg));
String line;
while ((line = br.readLine()) != null) {
    // 逐行处理
}
br.close();
```

`File` 是路径抽象，`BufferedReader` 带缓冲逐行读——Ch10 两个核心 API 的教学入口。`readLine()` 返回 `null` 表示 EOF，这是初学者最常犯错的边界条件，此处正好强调。

【三代演进定位】

代次	阶段	题目来源	判题器决定方式	测试点来源
第一代（M1–M3）	已完成	硬编码 `new Problem(...)`	`new StandardJudge()` 写死	`new TestCase(...)` 手打
第二代（**M4**，本章）	本章	**`config.txt`** 读元数据	**`Class.forName`** 反射造判题器	**`input.txt/output.txt`** 文件读取
第三代（M5a）	下一章	多组 `N.in/N.out` 文件	调外部 C++ 判题机二进制	文件系统扫目录

第二代的核心贡献是“配置驱动”闭环：

```
problems/1/
+-- config.txt        <- title / judgeClass / timeLimitMs
+-- input.txt         <- 单组输入
+-- output.txt        <- 单组期望输出
```

`SingleFileProblemLoader` 读这三个文件，组装出 `Problem` 对象；`JudgeFactory` 用 `config.txt` 里的 `judgeClass` 字段反射造出对应 `Judge`——从此换题不改代码。

> 第三代会把”单文件单组”扩展成”多文件多组”，并把 JVM 内判题替换为外部二进制——但那是 M5a 的事，本章只做第二代的”最小可用闭环”。

【新产物架构】

涉及类/接口总览

类 / 接口	包	职责
`ProblemMeta`	`oj.core`	不可变值对象，持有题目元数据
`ConfigFile`	`oj.io`	静态工具类，解析 `config.txt` -> `ProblemMeta`
`SingleFileProblemLoader`	`oj.io`	静态工具类，聚合三文件 -> `Problem`
`JudgeFactory`	`oj.judge`	静态工厂，`Class.forName` 反射造 `Judge`

类 / 接口	包	职责
Problem（更新）	oj.core	新增 meta 字段，judgeClass 移入 ProblemMeta

以下沿用（不改动）：TestCase、JudgeResult、Status、Judge、AbstractJudge、StandardJudge、SpecialJudge、Solution。

---

oj.core.ProblemMeta

元数据值对象。字段全 final，只有 getter，不可变。

```
package oj.core;

public final class ProblemMeta {
    private final String title;
    private final String judgeClass;   // 判题器全限定名，如 "oj.judge.StandardJudge"
    private final long timeLimitMs;

    public ProblemMeta(String title, String judgeClass, long timeLimitMs) {
        this.title = title;
        this.judgeClass = judgeClass;
        this.timeLimitMs = timeLimitMs;
    }

    public String getTitle() { return title; }
    public String getJudgeClass() { return judgeClass; }
    public long getTimeLimitMs() { return timeLimitMs; }
}
```

---

oj.core.Problem（M4 版本）

相比 M1–M3 版本，新增 meta 字段，getTitle()/getJudgeClass()/getTimeLimitMs() 委托给 meta。测试点仍用 TestCase[]（M5a 才迁移到 List<TestCase>）。

```
package oj.core;

public class Problem {
```

```
    private final int id;
    private final ProblemMeta meta;
    private final TestCase[] cases;

    public Problem(int id, ProblemMeta meta, TestCase[] cases) {
        this.id = id;
        this.meta = meta;
        this.cases = cases;
    }

    public int getId() { return id; }
    public ProblemMeta getMeta() { return meta; }

    // 便捷代理，让调用方无需先拿 meta
    public String getTitle() { return meta.getTitle(); }
    public String getJudgeClass() { return meta.getJudgeClass(); }
    public long getTimeLimitMs() { return meta.getTimeLimitMs(); }

    public TestCase[] getCases() { return cases; }
}
```

---

oj.io.ConfigFile

解析 config.txt，返回 ProblemMeta。

config.txt 格式约定（顺序不限，# 开头为注释行）：

```
title=A+B Problem
judge=oj.judge.StandardJudge
timeLimitMs=1000
```

```
package oj.io;

import oj.core.ProblemMeta;
import java.io.*;

public class ConfigFile {

    /**
     * 解析 dir 下的 config.txt，返回 ProblemMeta。
     * 未识别的 key 直接跳过，不抛异常。
```

```
     */
    public static ProblemMeta read(File dir) throws IOException {
        File cfg = new File(dir, "config.txt");
        String title = "";
        String judgeClass = "oj.judge.StandardJudge";   // 默认值
        long timeLimitMs = 1000L;

        BufferedReader br = new BufferedReader(new FileReader(cfg));
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty() || line.startsWith("#")) {
                continue;   // 跳过空行和注释
            }
            String[] parts = line.split("=", 2);
            if (parts.length < 2) {
                continue;   // 格式不合法，跳过
            }
            String key   = parts[0].trim();
            String value = parts[1].trim();
            switch (key) {
                case "title":
                    title = value;
                    break;
                case "judge":
                    judgeClass = value;
                    break;
                case "timeLimitMs":
                    timeLimitMs = Long.parseLong(value);
                    break;
                default:
                    // 未知 key 忽略，方便未来扩展
                    break;
            }
        }
        br.close();
        return new ProblemMeta(title, judgeClass, timeLimitMs);
    }
}
```

教学要点：`split("=", 2)` 的 limit 参数防止 value 里含 = 被误切；`switch` 上的字符串（Java 7+）比 `if-else if` 链更易扩展。

---

oj.io.SingleFileProblemLoader

聚合三个文件，组装 Problem。单组测试点，TestCase[] 长度始终为 1。

```
package oj.io;

import oj.core.*;
import java.io.*;
import java.nio.file.*;

public class SingleFileProblemLoader {

    /**
     * 读 dir 下的 config.txt / input.txt / output.txt,
     * 返回含单组测试点的 Problem。
     */
    public static Problem load(int id, File dir) throws IOException {
        ProblemMeta meta = ConfigFile.read(dir);

        String input    = readAll(new File(dir, "input.txt"));
        String expected = readAll(new File(dir, "output.txt"));

        TestCase[] cases = { new TestCase(input, expected) };
        return new Problem(id, meta, cases);
    }

    /** 读整个文件为字符串，行尾统一用 \n。 */
    private static String readAll(File f) throws IOException {
        StringBuilder sb = new StringBuilder();
        BufferedReader br = new BufferedReader(new FileReader(f));
        String line;
        while ((line = br.readLine()) != null) {
            sb.append(line).append('\n');
        }
        br.close();
        return sb.toString().trim();   // 去掉末尾多余空行
    }
}
```

教学要点：private static readAll 体现”封装小工具”原则——重复的文件读逻辑只写一次，两处调用（input / output）共享同一实现。trim() 消除末尾换行符，避免比对时因空白差异误判为 WA。

---

oj.judge.JudgeFactory

反射工厂。核心只有一行：Class.forName(className).getDeclaredConstructor().newInstance()。

```
package oj.judge;

public class JudgeFactory {

    /**
     * 按全限定类名反射造 Judge 实例。
     * 要求目标类：① 实现 Judge 接口；② 有无参构造器。
     *
     * @param className 如 "oj.judge.StandardJudge" 或 "oj.judge.SpecialJudge"
     * @return 对应的 Judge 实例
     * @throws ReflectiveOperationException 类不存在、无无参构造、类型不匹配时抛出
     */
    public static Judge create(String className) throws ReflectiveOperationException {
        Class<?> clazz = Class.forName(className);
        return (Judge) clazz.getDeclaredConstructor().newInstance();
    }
}
```

教学要点：- Class<?> 用通配符，因为编译期不知道具体类型。- getDeclaredConstructor() 比 newInstance()(已废弃)更规范。- 强制转型（Judge）若配置文件写错类名会在运行期抛 ClassCastException——这正是向学生演示”运行期类型检查”的绝佳时机。

“文件 -> 反射 -> 判题”闭环示例

以下是 main 里的完整使用示例，串起所有新产物：

```
import oj.core.*;
import oj.io.*;
import oj.judge.*;

public class M4Demo {
    public static void main(String[] args) throws Exception {
        // 1. 从文件系统加载题目（读三个文件）
        File problemDir = new File("problems/1/");
        Problem problem = SingleFileProblemLoader.load(1, problemDir);
```

```
        System.out.println(" 题目: " + problem.getTitle());
        System.out.println(" 判题器: " + problem.getJudgeClass());
        System.out.println(" 时间限制: " + problem.getTimeLimitMs() + "ms");

        // 2. 反射造判题器（判题器类名来自 config.txt，不写死）
        Judge judge = JudgeFactory.create(problem.getJudgeClass());

        // 3. 准备一个 Solution（教学占位，仍在 JVM 内运行）
        Solution solution = input -> {
            String[] nums = input.trim().split("\\s+");
            int a = Integer.parseInt(nums[0]);
            int b = Integer.parseInt(nums[1]);
            return String.valueOf(a + b);
        };

        // 4. 判题，打印结果
        JudgeResult result = judge.judge(problem, solution);
        System.out.println(result);   // 输出如: AC 1/1 (3ms)
    }
}
```

运行时目录结构：

```
problems/
+-- 1/
    +-- config.txt    ->  title=A+B Problem
    |                     judge=oj.judge.StandardJudge
    |                     timeLimitMs=1000
    +-- input.txt     ->  1 2
    +-- output.txt    ->  3
```

将 config.txt 里的 judge 改为 oj.judge.SpecialJudge，无需改代码，重新运行即切换为浮点判题器——这就是”配置驱动”的含义。

动手实现: 两步走 (javac / java 实操)

第一步 考试向 — 用 String 写 OJ 的输出比对 + 报告

src/oj/judge/OutputChecker.java

```java
package oj.judge;
public class OutputChecker {
    /** 逐行去首尾空白、忽略末尾空行后比较（OJ 最常用的"宽松精确"比对） */
    public static boolean same(String expected, String actual) {
        return norm(expected).equals(norm(actual));
    }
    private static String norm(String s) {
        StringBuilder sb = new StringBuilder();
        for (String line : s.split("\n", -1)) sb.append(line.trim()).append('\n');
        return sb.toString().trim();
    }
}
```

讲解:split 切行、trim() 去空白、StringBuilder 拼接、equals 比较——全是 Ch8 考点，用在 OJ 自己的判题上。

src/oj/Demo4.java

```java
package oj;
import oj.judge.OutputChecker;
public class Demo4 {
    public static void main(String[] args) {
        System.out.println(OutputChecker.same("3\n", "  3  "));        // true(空白无关)
        System.out.println(OutputChecker.same("1 2 3", "1 2 4"));      // false
        StringBuilder report = new StringBuilder("判题报告:\n");
        report.append("case#1: ").append(OutputChecker.same("3", "3") ? "AC" :
 ↪  "WA").append('\n');
        System.out.print(report);
    }
}
```

编译运行:

```
javac -d build src/oj/judge/OutputChecker.java src/oj/Demo4.java
java -cp build oj.Demo4
# true
# false
# 判题报告:
# case#1: AC
```

第二步 工程向 — 配置驱动 + 反射工厂

加 ConfigFile.read(dir) 解析 config.txt、JudgeFactory.create(类名) 用 Class.forName 反射造判题器、SingleFileProblemLoader 读单组用例 (代码见上文【新产物架构】)。换判题器只改配置:

```
javac -d build $(find src -name '*.java')
java -cp build oj.Main
```

验收标准

1. **ConfigFile.read**：给定合法 config.txt，能正确解析 title、judge、timeLimitMs 三个字段；遇到空行、# 注释行、未知 key 不抛异常。
2. **SingleFileProblemLoader.load**：读取 problems/1/ 下三个文件后，返回的 Problem 对象 getTitle() / getJudgeClass() / getCases().length == 1 均符合预期。
3. **JudgeFactory.create("oj.judge.StandardJudge")**：返回的对象 instanceof StandardJudge 为 true；传入不存在的类名抛 ClassNotFoundException（不是静默失败）。
4. **JudgeFactory.create("oj.judge.SpecialJudge")**：仅修改 config.txt，不改任何 Java 源码，判题器自动切换。
5. 闭环冒烟测试：M4Demo.main 在 problems/1/ 目录存在时输出 AC 1/1 (Xms)；将 output.txt 改为错误答案后输出 WA 0/1 (Xms)。
6. **Problem.getJudgeClass()**：调用链 problem.getJudgeClass() 实际委托给 problem.getMeta().getJudgeCla 两者返回值相同。

06 ·M5a 第三代判题机

考试相关度 ★★(Ch15 集合)·C++ 判题机属工程·真题考点参考:ArrayList/LinkedList/HashMap/Iterator。[第一步] 考试向 用 ArrayList/HashMap 把 OJ 的多组用例/题目装成内存题库;[第二步] 工程向 ProblemLoader/Repository + ProcessBuilder 调外部 C++ 判题机 + 序列化。

到目前为止，Java 在 JVM 里模拟判题——既无法真正卡住 CPU 时间，也无法强制限制内存。是时候让外部 C++ 沙箱接管真正的资源控制了。

下载与使用 C++ 判题机

第三代判题机是一个独立的 C++ 单文件程序 judge.cpp——与 Java 解耦,Java 只负责”调用 + 读结果”。它支持 cpp/c/python/java, 判 AC/WA/PE/TLE/MLE/RE/CE, 带 setrlimit + 进程组掐断 + 编译超时, 健壮性不影响教学推进。

v 下载 judge.cpp(单文件源码) v 下载完整判题机 (含 Makefile + 示例)

① 放置位置 —— 放到项目根目录下的 judge/:

```
mini-oj/
+-- src/oj/...            # Java 代码(MachineJudge 在 oj.judge)
+-- problems/<id>/        # 题目:config.txt + N.in/N.out
+-- judge/
    +-- judge.cpp
    +-- judge             # ② 编译出来的二进制
```

② 编译 (一次即可, 需要 g++):

```
cd judge && g++ -O2 -std=c++17 -o judge judge.cpp   # 或 make
```

③ 命令行用法:

```
./judge/judge --problem problems/1 --src sub.cpp --lang cpp --time-ms 1000 --mem-mb 256
 ↪  [--special]
# -> {"status":"AC","passed":3,"total":3,"time_ms":5,"mem_kb":1840,"detail":""}
```

④ **Java** 怎么调 —— 就是下面 MachineJudge 干的事 (ProcessBuilder 启动 + 正则解析 JSON):

```
MachineJudge mj = new MachineJudge("judge/judge");
JudgeResult r = mj.judge("problems/1", "sub.cpp", "cpp", 1000, 256);
```

判题机完整说明见仓库 judge/README.md; 它在本仓库已编译验证 (AC/WA/TLE/RE/CE/MLE/PE + python + 浮点 special 全通过)。

【核心痛点】

第一代、第二代判题的天花板

M1–M4 的判题链路始终在 JVM 内部：Solution.solve() 直接返回字符串，StandardJudge 做字符串比对。这种做法在教学早期够用，但有三个根本缺陷：

1. 无法真正限制 **CPU**：Java 没有 setrlimit(RLIMIT_CPU, ...)，Thread.interrupt() 只能请求终止，无法强杀死循环。
2. 无法真正限制内存：JVM 堆本身就是几百 MB 的大容器，-Xmx 限制的是 JVM 堆，不是选手代码的实际内存分配。
3. 无法支持多语言：第二代只能判 Java，而 OJ 需要支持 C++/Python/Java 等语言，每种语言的编译/运行命令完全不同。

破局方案：Java 只做”调度员”，真正的编译、运行、资源限制全部委托给 C++ 编写的外部 judge 二进制。Java 用 ProcessBuilder 启动这个二进制，传入源文件路径和时间/内存限制，读取其标准输出的一行 JSON，正则解析后回填到 JudgeResult。

【引入课本知识点】

Ch15 ·泛型与集合

泛型类与接口声明

泛型让容器在编译期就确定元素类型，消除强制转型：

```
// 编译器保证 list 里只有 TestCase, 取出无需 (TestCase) 强转
List<TestCase> cases = new LinkedList<>();
cases.add(new TestCase("1 2", "3"));
TestCase tc = cases.get(0);  // 类型安全
```

ProblemLoader 用 List<TestCase> 替代 M4 的 TestCase[]，原因有二：- 测试点数量未知，扫目录时逐条 add()，不需要预先 new TestCase[n]。- List 提供迭代器和 Stream，后续统计更方便。

HashMap<Integer, Problem> 题目映射

```
Map<Integer, Problem> cache = new HashMap<>();
cache.put(1001, problem);
Problem p = cache.get(1001);  // O(1) 查找
```

ProblemRepository 用此结构做内存缓存：第一次从磁盘加载，之后命中缓存直接返回。

Stream 流式统计示例

```
// 统计一批提交里各 Status 出现次数
Map<Status, Long> stat = submissions.stream()
    .map(s -> s.getResult().getStatus())
    .collect(Collectors.groupingBy(s -> s, Collectors.counting()));
```

这是 M5a 结尾的统计演示，让学生感受 Stream 的链式风格。

Ch10 ·目录遍历与对象序列化

目录遍历

```
File dir = new File("problems/1001");
File[] files = dir.listFiles((d, name) -> name.endsWith(".in"));
```

ProblemLoader 用此方式扫 problems/<id>/ 下所有 N.in/N.out 文件对。

对象序列化（**Serializable**）

实现 Serializable 的对象可以用 ObjectOutputStream 直接写入文件，用 ObjectInputStream 原样读回，

不需要手写 CSV 格式。SubmissionStore 正是用这种方式把 Submission 对象持久化到 submissions.dat。

【三代演进定位】

代际	判题入口	资源限制	多语言	定位
第一代（M1–M3）	Solution.solve()	无	仅 Java	JVM 内模拟，教学启蒙
第二代（M4）	反射 JudgeFactory.create()	无	仅 Java	反射解耦，配置驱动
第三代（**M5a**）	**MachineJudge + ProcessBuilder**	**setrlimit（C++ 端）**	**C++/Python/Java**	跨界真判题，工业级沙箱

第三代的关键跨越：Java 进程通过 ProcessBuilder 启动一个预编译好的 C++ 二进制（judge），后者接收命令行参数（源文件、语言、时间/内存限制、题目目录），在 Linux 内核层面用 setrlimit 施加资源上限，运行完毕后向标准输出打印一行 JSON：

```
{"status":"AC","passed":3,"total":3,"time_ms":45,"mem_kb":2048,"detail":""}
```

Java 用正则表达式解析这行 JSON，构造 JudgeResult 返回。C++ 侧代码不在本章展开，本章聚焦 **Java** 侧的调用接口与数据流。

【新产物架构】

本节给出 M5a 四个新产物的完整实现。

1. oj.io.ProblemLoader —— 扫目录，返回 List<TestCase>

职责：扫描 problems/<id>/ 目录，按编号顺序读取 1.in/1.out、2.in/2.out……构造 List<TestCase>。

```
package oj.io;

import oj.core.TestCase;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.ArrayList;
import java.util.Arrays;
```

```java
import java.util.List;

public class ProblemLoader {

    private final String problemsRoot;

    public ProblemLoader(String problemsRoot) {
        this.problemsRoot = problemsRoot;
    }

    /**
     * 扫描 problems/<id>/ 目录，返回按序号排列的测试点列表。
     * 每对 N.in / N.out 构成一个 TestCase。
     *
     * @param id 题目 ID
     * @return 有序的 TestCase 列表，若目录不存在则返回空列表
     */
    public List<TestCase> cases(int id) {
        File dir = new File(problemsRoot, String.valueOf(id));
        List<TestCase> result = new ArrayList<>();
        if (!dir.isDirectory()) {
            return result;
        }

        // 收集所有 .in 文件，按文件名数字排序
        File[] inFiles = dir.listFiles((d, name) -> name.endsWith(".in"));
        if (inFiles == null || inFiles.length == 0) {
            return result;
        }
        Arrays.sort(inFiles, (a, b) -> {
            int na = indexOf(a.getName());
            int nb = indexOf(b.getName());
            return Integer.compare(na, nb);
        });

        for (File inFile : inFiles) {
            String baseName = inFile.getName().replace(".in", "");
            File outFile = new File(dir, baseName + ".out");
            if (!outFile.exists()) {
                continue;
            }
```

```
            try {
                String input = new String(Files.readAllBytes(inFile.toPath())).trim();
                String expected = new
                 ↪  String(Files.readAllBytes(outFile.toPath())).trim();
                result.add(new TestCase(input, expected));
            } catch (IOException e) {
                System.err.println("[ProblemLoader] 读取测试点失败: " +
                 ↪  inFile.getName());
            }
        }
        return result;
    }

    /** 从 "3.in" 这样的文件名中提取数字 3 */
    private int indexOf(String filename) {
        try {
            return Integer.parseInt(filename.replaceAll("[^0-9]", ""));
        } catch (NumberFormatException e) {
            return Integer.MAX_VALUE;
        }
    }
}
```

目录结构约定：

```
problems/
  1001/
    config.txt        <- 题目元数据（M4 沿用）
    1.in  1.out
    2.in  2.out
    3.in  3.out
```

---

2. oj.io.ProblemRepository — HashMap 缓存题目

职责：组合 ConfigFile（M4 产物，读 config.txt）与 ProblemLoader，对外提供 get(id) 接口；内部用 HashMap<Integer, Problem> 缓存，同一题目只从磁盘加载一次。

```
package oj.io;

import oj.core.Problem;
```

```
import oj.core.ProblemMeta;
import oj.core.TestCase;

import java.io.File;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class ProblemRepository {

    private final String problemsRoot;
    private final ProblemLoader loader;
    private final Map<Integer, Problem> cache = new HashMap<>();

    public ProblemRepository(String problemsRoot) {
        this.problemsRoot = problemsRoot;
        this.loader = new ProblemLoader(problemsRoot);
    }

    /**
     * 返回指定 ID 的题目，优先命中缓存。
     * 若缓存未命中则从磁盘加载 config.txt + 测试点，存入缓存后返回。
     *
     * @param id 题目 ID
     * @return Problem 对象，若不存在则返回 null
     */
    public Problem get(int id) {
        if (cache.containsKey(id)) {
            return cache.get(id);
        }
        Problem p = load(id);
        if (p != null) {
            cache.put(id, p);
        }
        return p;
    }

    private Problem load(int id) {
        File dir = new File(problemsRoot, String.valueOf(id));
        File configFile = new File(dir, "config.txt");
        if (!configFile.exists()) {
```

```
            return null;
        }
        ProblemMeta meta = ConfigFile.read(dir);  // M4 契约：传目录，内部读 config.txt
        List<TestCase> cases = loader.cases(id);
        return new Problem(id, meta, cases);
        // Problem 构造器在 M5a 重构为接受 List<TestCase>
    }

    /** 清除缓存（测试用途） */
    public void evict(int id) {
        cache.remove(id);
    }

    /** 已缓存的题目数量 */
    public int cacheSize() {
        return cache.size();
    }
}
```

---

3. oj.io.SubmissionStore —— 对象序列化持久化

职责：把 Submission 对象列表追加写入 submissions.dat，支持全量读回。利用 Java 对象序列化（ObjectOutputStream/ObjectInputStream）完成持久化，不需要手写文本格式。

```
package oj.io;

import oj.core.Submission;

import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

public class SubmissionStore {
```

```
    private final File storeFile;

    public SubmissionStore(String path) {
        this.storeFile = new File(path);
    }

    /**
     * 将一条提交追加写入持久化文件。
     * 注意：ObjectOutputStream 写多个对象到同一文件时，
     * 后续追加必须跳过文件头 (magic + version)，用内部类 AppendObjectOutputStream 实现。
     *
     * @param submission 要持久化的提交对象
     */
    public void save(Submission submission) throws IOException {
        if (storeFile.exists() && storeFile.length() > 0) {
            try (AppendObjectOutputStream out =
                     new AppendObjectOutputStream(new FileOutputStream(storeFile,
                      ↪  true))) {
                out.writeObject(submission);
            }
        } else {
            try (ObjectOutputStream out =
                     new ObjectOutputStream(new FileOutputStream(storeFile, true))) {
                out.writeObject(submission);
            }
        }
    }

    /**
     * 读取文件中所有提交记录。
     *
     * @return 提交列表，若文件不存在则返回空列表
     */
    public List<Submission> loadAll() throws IOException, ClassNotFoundException {
        List<Submission> list = new ArrayList<>();
        if (!storeFile.exists() || storeFile.length() == 0) {
            return list;
        }
        try (ObjectInputStream in =
                 new ObjectInputStream(new FileInputStream(storeFile))) {
            while (true) {
```

```
                try {
                    list.add((Submission) in.readObject());
                } catch (EOFException e) {
                    break;  // 正常结束，文件读完
                }
            }
        }
        return list;
    }

    /**
     * 追加写模式的 ObjectOutputStream：覆盖 writeStreamHeader()，
     * 避免在文件中段再次写入 magic bytes，防止读回时报 StreamCorruptedException。
     */
    private static class AppendObjectOutputStream extends ObjectOutputStream {
        AppendObjectOutputStream(FileOutputStream out) throws IOException {
            super(out);
        }

        @Override
        protected void writeStreamHeader() throws IOException {
            reset();  // 只重置状态，不写 magic + version
        }
    }
}
```

教学提示：AppendObjectOutputStream 是一个常见陷阱。直接用 new ObjectOutputStream(new FileOutputStream(file, true)) 追加写，第二条对象写入时会再次写入 4 字节 magic（0xACED）和 2 字节版本（0x0005），导致读回时 StreamCorruptedException。覆盖 writeStreamHeader() 改为 reset() 是标准解法。

---

4. oj.judge.MachineJudge — 第三代判题机核心

职责：通过 ProcessBuilder 启动外部 C++ judge 二进制，传入判题参数，读取其标准输出的 JSON 行，用正则解析后构造 JudgeResult 返回。

```
package oj.judge;

import oj.core.JudgeResult;
```

```java
import oj.core.Status;

import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.concurrent.TimeUnit;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class MachineJudge {

    /**
     * C++ judge 二进制输出的 JSON 正则。
     * 示例: {"status":"AC","passed":3,"total":3,"time_ms":45,"mem_kb":2048,"detail":""}
     */
    private static final Pattern JSON_PATTERN = Pattern.compile(
        "\\{\"status\":\"(\\w+)\"," +
        "\"passed\":(\\d+)," +
        "\"total\":(\\d+)," +
        "\"time_ms\":(\\d+)," +
        "\"mem_kb\":(\\d+)," +
        "\"detail\":\"([^\"]*)\"\\}"
    );

    private final String judgeBinary;  // C++ judge 二进制的绝对路径

    public MachineJudge(String judgeBinary) {
        this.judgeBinary = judgeBinary;
    }

    /**
     * 调用外部 C++ 判题机，返回判题结果。
     *
     * @param problemDir 题目目录（含 N.in/N.out）
     * @param srcFile    选手源文件路径（.cpp / .py / .java）
     * @param lang       语言标识："cpp" / "python" / "java"
     * @param timeMs     时间限制（毫秒）
     * @param memMb      内存限制（MB）
     * @return JudgeResult，若 judge 进程异常则返回 RE
     */
```

```
    public JudgeResult judge(String problemDir, String srcFile,
                             String lang, long timeMs, int memMb) {
        ProcessBuilder pb = new ProcessBuilder(
            judgeBinary,
            "--problem", problemDir,
            "--src",     srcFile,
            "--lang",    lang,
            "--time",    String.valueOf(timeMs),
            "--mem",     String.valueOf(memMb)
        );
        pb.redirectErrorStream(false);  // 不合并 stderr，只读 stdout
        pb.directory(new File("."));

        long wallStart = System.currentTimeMillis();
        Process proc;
        try {
            proc = pb.start();
        } catch (IOException e) {
            return errorResult(" 启动 judge 进程失败: " + e.getMessage());
        }

        // 读取 judge 标准输出（预期一行 JSON）
        String jsonLine = "";
        try (BufferedReader reader =
                 new BufferedReader(new InputStreamReader(proc.getInputStream()))) {
            jsonLine = reader.readLine();
        } catch (IOException e) {
            return errorResult(" 读取 judge 输出失败: " + e.getMessage());
        }

        // 等待进程退出，最多 timeMs + 5000ms 宽限
        boolean finished;
        try {
            finished = proc.waitFor(timeMs + 5000, TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            proc.destroyForcibly();
            return errorResult(" 等待 judge 进程被中断");
        }
        if (!finished) {
            proc.destroyForcibly();
```

```
            return errorResult("judge 进程超时未退出");
        }

        long elapsed = System.currentTimeMillis() - wallStart;

        if (jsonLine == null || jsonLine.isBlank()) {
            return errorResult("judge 无输出");
        }
        return parseJson(jsonLine.trim(), elapsed);
    }

    /**
     * 用正则解析 C++ judge 输出的 JSON 行，构造 JudgeResult。
     * C++ 侧若报 ERR（沙箱内部错误），映射为 RE。
     */
    private JudgeResult parseJson(String json, long elapsed) {
        Matcher m = JSON_PATTERN.matcher(json);
        if (!m.find()) {
            return errorResult("JSON 格式不匹配: " + json);
        }

        String rawStatus = m.group(1);
        int passed  = Integer.parseInt(m.group(2));
        int total   = Integer.parseInt(m.group(3));
        long timeMs = Long.parseLong(m.group(4));
        // mem_kb 目前记录在 detail，供日志使用
        int memKb   = Integer.parseInt(m.group(5));
        String detail = m.group(6);

        Status status = mapStatus(rawStatus);
        String fullDetail = detail.isBlank()
            ? String.format("mem=%dKB", memKb)
            : String.format("%s mem=%dKB", detail, memKb);

        return new JudgeResult(status, passed, total, fullDetail, timeMs);
    }

    /**
     * 将 C++ judge 返回的状态字符串映射到 oj.core.Status。
     * ERR（沙箱内部异常）统一映射为 RE。
     */
```

```
    private Status mapStatus(String raw) {
        switch (raw.toUpperCase()) {
            case "AC":  return Status.AC;
            case "WA":  return Status.WA;
            case "TLE": return Status.TLE;
            case "MLE": return Status.MLE;
            case "RE":  return Status.RE;
            case "CE":  return Status.CE;
            case "PE":  return Status.PE;
            case "ERR": return Status.RE;   // 沙箱内部错误 -> RE
            default:    return Status.RE;
        }
    }

    /** 构造一个表示 Java 侧调用失败 (RE) 的 JudgeResult */
    private JudgeResult errorResult(String detail) {
        return new JudgeResult(Status.RE, 0, 0, detail, 0L);
    }
}
```

为什么换 C++？

能力	Java（第一/二代）	C++ judge（第三代）
CPU 时间限制	不可靠（Thread.interrupt）	`setrlimit(RLIMIT_CPU, t)` 内核强制
内存限制	不可靠（JVM 堆外无法控制）	`setrlimit(RLIMIT_AS, m)` 地址空间硬限
多语言编译	仅 Java	调 g++/python3/javac
隔离性	无	chroot / seccomp（可扩展）
性能	JVM 启动 ~200ms	native 进程 ~5ms

---

5. Stream 统计演示（整合示例）

以下代码展示如何用 Stream 对一批提交做状态统计，通常放在 `Main` 或测试类中：

```
import oj.core.Status;
import oj.core.Submission;
import oj.io.SubmissionStore;
```

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class StatsDemo {
    public static void main(String[] args) throws Exception {
        SubmissionStore store = new SubmissionStore("submissions.dat");
        List<Submission> all = store.loadAll();

        // Stream: 按 Status 分组计数
        Map<Status, Long> stat = all.stream()
            .map(s -> s.getResult().getStatus())
            .collect(Collectors.groupingBy(s -> s, Collectors.counting()));

        stat.forEach((status, count) ->
            System.out.printf("%-4s : %d 次%n", status, count));

        // Stream: 筛选 AC 提交，打印用户名
        System.out.println("--- AC 提交者 ---");
        all.stream()
            .filter(s -> s.getResult().isAccepted())
            .map(Submission::getUserName)
            .distinct()
            .sorted()
            .forEach(System.out::println);
    }
}
```

类关系总览

```
ProblemRepository
  +- uses ConfigFile          (M4 产物，读 config.txt)
  +- uses ProblemLoader
          +- returns List<TestCase>   <- Ch15 泛型集合

MachineJudge
  +- ProcessBuilder -> [C++ judge binary]
  +- Pattern.compile -> JudgeResult
```

```
SubmissionStore
  +- ObjectOutputStream （写）  <- Ch10 序列化
  +- ObjectInputStream  （读）
```

动手实现: 两步走 (javac / java 实操)

第一步 考试向 —— 用集合把 OJ 题库装进内存

src/oj/io/Bank.java

```
package oj.io;
import oj.core.*;
import java.util.*;
public class Bank {
    private Map<Integer, Problem> problems = new HashMap<>();   // 题号 -> 题目
    public void add(Problem p)   { problems.put(p.getId(), p); }
    public Problem get(int id)   { return problems.get(id); }
    public int size()            { return problems.size(); }
    public List<Integer> ids()   { return new ArrayList<>(problems.keySet()); }
}
```

讲解:HashMap<Integer,Problem> 做题号映射、ArrayList 列题号——Ch15 集合考点, 用在 OJ 题库管理上。

src/oj/Demo5a.java

```
package oj;
import oj.core.*;
import oj.io.Bank;
import java.util.*;
public class Demo5a {
    public static void main(String[] args) {
        Bank bank = new Bank();
        bank.add(new Problem(1, "A+B", new TestCase[]{ new TestCase("1 2","3") }));
        bank.add(new Problem(2, " 平方", new TestCase[]{ new TestCase("3","9"), new
 ↪  TestCase("4","16") }));
        for (int id : bank.ids())                                   // 遍历 (类似
         ↪  Iterator)
            System.out.println(id + " => " + bank.get(id).getTitle() + " (" +
             ↪  bank.get(id).getCases().length + " 用例)");
        long total = bank.ids().stream().mapToInt(id ->
         ↪  bank.get(id).getCases().length).sum();  // Stream 统计
        System.out.println(" 总用例数: " + total);
```

```
    }
}
```

编译运行:

```
javac -d build src/oj/core/*.java src/oj/io/Bank.java src/oj/Demo5a.java
java -cp build oj.Demo5a
# 1 => A+B (1 用例)
# 2 => 平方 (2 用例)
# 总用例数: 3
```

第二步 工程向 — 扫目录题库 + 外部 C++ 判题机

`ProblemLoader.cases(id)`(扫 `problems/<id>/N.in/N.out -> List<TestCase>`)、`ProblemRepository`(HashMap 缓存)、`MachineJudge`(ProcessBuilder 调判题机, 代码见上文【新产物架构】)。先编译判题机, 再跑 Java:

```
cd judge && g++ -O2 -std=c++17 -o judge judge.cpp && cd ..
javac -d build $(find src -name '*.java')
java -cp "build:lib/*" oj.Main
```

验收标准

1. **`ProblemLoader.cases(1001)`** 能从 `problems/1001/` 扫到所有 `N.in/N.out` 对，按编号升序返回 `List<TestCase>`，编号不连续（如缺少 `2.in`）时跳过不报错。
2. **`ProblemRepository.get(1001)`** 第一次调用触发磁盘读取，第二次调用命中缓存（`cacheSize()` 不增加），两次返回同一个对象引用。
3. **`SubmissionStore`** 连续 `save()` 三条提交后，`loadAll()` 能完整读回三条，顺序一致，无 `StreamCorruptedExceptio`
4. **`MachineJudge.judge()`** 在 judge 二进制返回合法 JSON 时正确构造 `JudgeResult`；返回 `ERR` 状态时映射为 `RE`；judge 进程超时未退出时强制销毁并返回 `RE`。
5. **Stream** 统计：对含 AC/WA/TLE 混合提交的 `SubmissionStore` 执行 `StatsDemo`，输出各 Status 计数正确，AC 用户名列表去重且排序。
6. 不出现任何 **Socket/ServerSocket**/网络相关代码，不出现第一代 `Solution` 接口的调用。

07 ·M5b 数据库衔接（Ch11 JDBC + MySQL）

考试相关度 ★★ ·真题考点参考:Connection/Statement/ResultSet 查询入 List、更新。[第一步] 考试向 用最简 JDBC(Statement) 把 OJ 的提交/题目存取 MySQL;[第二步] 工程向 ProblemDao/事务/PreparedStatement/ProblemService 的 FS/DB 分工。

文件能存代码、能存测试点，但它不会帮你在一秒内告诉你”过了 AC 的选手有几个”。当题目数量破百、提交记录过万，纯文件方案的代价才真正暴露出来。

【核心痛点】

M5a 的 ProblemRepository 把全部 ProblemMeta（包括判题器类名、时间限制）缓存在内存的 HashMap 里，启动时从文件系统扫 problems/<id>/config.txt 逐一解析。SubmissionStore 则把 Submission 对象序列化进 .ser 文件，一个文件一条提交记录。

这两个方案在教学原型里够用，但三个场景会让它原形毕露：

1. 统计：想查某题通过率、某用户 AC 数，要把所有 .ser 文件全部反序列化再在内存里过滤——O(N) 读 I/O，N 一大就超时。
2. 并发安全：多线程判题队列同时往同一目录写 .ser，没有任何互斥保障，文件名冲突或写到一半被读的情况真实发生。
3. 元数据变更：改一道题的时间限制要手动编辑 config.txt，没有原子性，改到一半系统崩溃就留下半截文件。

本节引入 MySQL + JDBC，让 元数据与历史记录进库、测试点文件留 **FS**，形成 FS/DB 分工格局，同时完整实践 Ch11 的 JDBC 知识点。

【引入课本知识点】

JDBC 四步固定套路

```
DriverManager.getConnection(url, user, pwd)   // 建连
conn.prepareStatement(sql)                    // 预编译
ps.setXxx(index, value)                       // 绑参
ps.executeQuery() / ps.executeUpdate()        // 执行
```

PreparedStatement 的核心价值是防 **SQL** 注入：参数占位符 ? 由驱动负责转义，用户输入里的单引号、注释符永远不会被解释成 SQL 语法。

ResultSet 游标

rs.next() 把游标从”结果集首行之前”推进一行，返回 false 表示遍历完毕。列索引从 1 开始，或用列名 rs.getString("title")。

事务

MySQL 默认 AUTO_COMMIT=true，每条语句独立提交。需要原子操作时：

```
conn.setAutoCommit(false);
// ... 多条 SQL ...
conn.commit();          // 全成功
// catch -> conn.rollback();
```

SubmissionDao.save 在单次提交里写两张表（submissions 插入 + 可选计数更新），必须在事务里完成，否则中途崩溃会留下孤儿记录。

连接复用

每次 getConnection 都要经历 TCP 握手 + 认证，开销约 10–30 ms。教学阶段用单例连接（Db 持有一个静态 Connection），生产级做法是连接池（HikariCP 等），了解即可，本节不实现。

【三代演进定位】

本节 不属于判题机三代的演进节点，而是为第三代 **MachineJudge** 提供配置来源：

数据	来源（M5a 之前）	来源（M5b 之后）
timeLimitMs / memLimitMb	config.txt -> ConfigFile.read	problems 表 -> ProblemDao.meta
判题器类名 judgeClass	config.txt	problems 表
提交历史	.ser 文件 -> SubmissionStore	submissions 表 -> SubmissionDao.save
测试点 .in / .out	文件系统（保留）	文件系统（保留，不变）

MachineJudge 调用外部 C++ 判题机时传入的 timeMs、memMb 由 ProblemService.load 从 DB 取出，路径不变、语义不变，只是数据源从 config.txt 换成了 MySQL。

【新产物架构】

Schema

```
CREATE TABLE IF NOT EXISTS users (
    id       INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(64) NOT NULL UNIQUE,
    pwd_hash VARCHAR(128) NOT NULL
);
```

```
CREATE TABLE IF NOT EXISTS problems (
    id              INT PRIMARY KEY,
    title           VARCHAR(256) NOT NULL,
    judge_class     VARCHAR(256) NOT NULL,
    time_limit_ms   BIGINT NOT NULL DEFAULT 2000,
    mem_limit_mb    INT    NOT NULL DEFAULT 256
);

CREATE TABLE IF NOT EXISTS submissions (
    id          INT PRIMARY KEY AUTO_INCREMENT,
    problem_id  INT         NOT NULL,
    username    VARCHAR(64) NOT NULL,
    lang        VARCHAR(16) NOT NULL,
    status      VARCHAR(8)  NOT NULL,
    passed      INT         NOT NULL DEFAULT 0,
    total       INT         NOT NULL DEFAULT 0,
    time_ms     BIGINT      NOT NULL DEFAULT 0,
    src_path    VARCHAR(512),
    submitted_at DATETIME   NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (problem_id) REFERENCES problems(id)
);
```

src_path 存选手源码在文件系统上的绝对路径，文件本身不入库（大文本存 LONGTEXT 会让 submissions 表膨胀，且无法 diff）。

---

oj.db.Db —— 单例连接管理

```
package oj.db;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Db {
    private static final String URL  =
     ↪  "jdbc:mysql://localhost:3306/minioj?useSSL=false&serverTimezone=UTC";
    private static final String USER = "oj";
    private static final String PASS = "ojpass";
```

```
    private static Connection conn;

    private Db() {}

    public static Connection get() throws SQLException {
        if (conn == null || conn.isClosed()) {
            conn = DriverManager.getConnection(URL, USER, PASS);
        }
        return conn;
    }

    public static void close() {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            } finally {
                conn = null;
            }
        }
    }
}
```

教学要点：isClosed() 保护让单例在连接被数据库服务端主动断开（如 wait_timeout）后能自动重连，避免”幽灵连接”导致的 Communications link failure。

---

oj.db.ProblemDao — 元数据 CRUD

```
package oj.db;

import oj.core.ProblemMeta;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.LinkedHashMap;
```

```
import java.util.Map;

public class ProblemDao {

    /**
     * 按 id 读取题目元数据，不含测试点。
     * 测试点由 ProblemLoader 从 FS 读取，两者在 ProblemService 里合并。
     */
    public ProblemMeta meta(int id) throws SQLException {
        Connection conn = Db.get();
        String sql = "SELECT title, judge_class, time_limit_ms FROM problems WHERE id =
        ↪  ?";
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setInt(1, id);
            try (ResultSet rs = ps.executeQuery()) {
                if (!rs.next()) {
                    throw new SQLException("Problem not found: " + id);
                }
                return new ProblemMeta(
                    rs.getString("title"),
                    rs.getString("judge_class"),
                    rs.getLong("time_limit_ms")
                );
            }
        }
    }

    /**
     * 列出全部题目元数据（题目下拉用），按 id 升序。
     */
    public Map<Integer, ProblemMeta> listMeta() throws SQLException {
        Connection conn = Db.get();
        String sql = "SELECT id, title, judge_class, time_limit_ms FROM problems ORDER
        ↪  BY id";
        Map<Integer, ProblemMeta> all = new LinkedHashMap<>();
        try (PreparedStatement ps = conn.prepareStatement(sql);
             ResultSet rs = ps.executeQuery()) {
            while (rs.next()) {
                all.put(rs.getInt("id"), new ProblemMeta(
                    rs.getString("title"),
                    rs.getString("judge_class"),
```

```
                    rs.getLong("time_limit_ms")));
            }
        }
        return all;
    }

    /**
     * 插入或更新题目（管理端录题时调用）。
     * INSERT ... ON DUPLICATE KEY UPDATE 保持幂等。
     */
    public void upsert(int id, ProblemMeta meta, int memLimitMb) throws SQLException {
        Connection conn = Db.get();
        String sql = "INSERT INTO problems (id, title, judge_class, time_limit_ms,
         ↪  mem_limit_mb) " +
                     "VALUES (?, ?, ?, ?, ?) " +
                     "ON DUPLICATE KEY UPDATE " +
                     "title=VALUES(title), judge_class=VALUES(judge_class), " +
                     "time_limit_ms=VALUES(time_limit_ms),
                      ↪  mem_limit_mb=VALUES(mem_limit_mb)";
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setInt(1, id);
            ps.setString(2, meta.getTitle());
            ps.setString(3, meta.getJudgeClass());
            ps.setLong(4, meta.getTimeLimitMs());
            ps.setInt(5, memLimitMb);
            ps.executeUpdate();
        }
    }
}
```

注意 `meta(int)` 只返回 **ProblemMeta**，不返回 `Problem`，因为 `Problem` 还需要测试点，而测试点来自 FS——这个组合动作交给 `ProblemService` 完成，职责分离。

`oj.db.SubmissionDao` —— 提交历史持久化（带事务）

```
package oj.db;

import oj.core.JudgeResult;
import oj.core.Submission;
```

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

public class SubmissionDao {

    /**
     * 将一条提交结果持久化进 submissions 表。
     * 整个操作在事务内：先 INSERT submissions，若判题已完成同步写 status/passed/total。
     * 任何一步失败整体 rollback，保证不产生孤儿记录。
     */
    public void save(Submission sub) throws SQLException {
        Connection conn = Db.get();
        boolean prev = conn.getAutoCommit();
        conn.setAutoCommit(false);
        try {
            String sql = "INSERT INTO submissions " +
                         "(problem_id, username, lang, status, passed, total, time_ms,
                          ↪  src_path) " +
                         "VALUES (?, ?, ?, ?, ?, ?, ?, ?)";
            try (PreparedStatement ps = conn.prepareStatement(sql)) {
                JudgeResult r = sub.getResult();
                ps.setInt(1, sub.getProblemId());
                ps.setString(2, sub.getUserName());
                ps.setString(3, sub.getLang());
                ps.setString(4, r != null ? r.getStatus().name() : "PENDING");
                ps.setInt(5, r != null ? r.getPassed() : 0);
                ps.setInt(6, r != null ? r.getTotal() : 0);
                ps.setLong(7, r != null ? r.getElapsedMs() : 0L);
                ps.setString(8, sub.getSrcPath());
                ps.executeUpdate();
            }
            conn.commit();
        } catch (SQLException e) {
            conn.rollback();
            throw e;
        } finally {
```

```
            conn.setAutoCommit(prev);
        }
    }

    /**
     * 查询某用户在某题上的所有提交，按时间倒序，最多返回 limit 条。
     */
    public List<Submission> listByUser(String username, int problemId, int limit)
     ↪  throws SQLException {
        Connection conn = Db.get();
        String sql = "SELECT id, problem_id, username, lang, status, passed, total,
         ↪  time_ms, src_path " +
                     "FROM submissions " +
                     "WHERE username = ? AND problem_id = ? " +
                     "ORDER BY submitted_at DESC LIMIT ?";
        List<Submission> result = new ArrayList<>();
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setString(1, username);
            ps.setInt(2, problemId);
            ps.setInt(3, limit);
            try (ResultSet rs = ps.executeQuery()) {
                while (rs.next()) {
                    Submission sub = new Submission(
                        rs.getInt("problem_id"),
                        rs.getString("username"),
                        rs.getString("lang"),
                        rs.getString("src_path")
                    );
                    oj.core.Status st = oj.core.Status.valueOf(rs.getString("status"));
                    JudgeResult jr = new JudgeResult(
                        st,
                        rs.getInt("passed"),
                        rs.getInt("total"),
                        "",
                        rs.getLong("time_ms")
                    );
                    sub.setResult(jr);
                    result.add(sub);
                }
            }
        }
```

```
        return result;
    }
}
```

事务模式说明：save 先保存原始 conn.getAutoCommit() 状态，finally 里恢复，这样单例连接被后续调用复用时不会携带残留的 autoCommit=false 状态。

---

oj.service.ProblemService — FS/DB 合并入口

```
package oj.service;

import oj.core.Problem;
import oj.core.ProblemMeta;
import oj.core.TestCase;
import oj.db.ProblemDao;
import oj.io.ProblemLoader;

import java.sql.SQLException;
import java.util.List;
import java.util.Map;

/**
 * 统一加载入口：DB 提供元数据，FS 提供测试点，两者在此合并为完整 Problem。
 * 调用方（OjFrame、JudgeWorker）只依赖本类，不直接接触 ProblemDao 或 ProblemLoader。
 */
public class ProblemService {

    private final ProblemDao dao;
    private final ProblemLoader loader;

    public ProblemService(ProblemDao dao, ProblemLoader loader) {
        this.dao = dao;
        this.loader = loader;
    }

    /**
     * 按题目 id 返回含完整测试点的 Problem。
     *
     * @param id 题目编号
```

```
     * @return 合并了 DB 元数据与 FS 测试点的 Problem 对象
     * @throws SQLException      数据库访问异常
     * @throws RuntimeException 题目目录不存在或测试点格式错误
     */
    public Problem load(int id) throws SQLException {
        ProblemMeta meta = dao.meta(id);              // DB: 元数据
        List<TestCase> cases = loader.cases(id);      // FS: 测试点
        return new Problem(id, meta, cases);
    }

    /** 只取元数据（GUI 取时限/标题用，不读测试点）。 */
    public ProblemMeta meta(int id) throws SQLException {
        return dao.meta(id);
    }

    /** 列出全部题目元数据（题目下拉用）。 */
    public Map<Integer, ProblemMeta> listMeta() throws SQLException {
        return dao.listMeta();
    }
}
```

ProblemService 是本节最关键的组合点：它让上层（JudgeWorker、OjFrame）只看到一个 load(int) 接口，完全不感知底层是 DB 还是文件。将来换成远程存储只改 ProblemService 内部，调用方零修改。

模块间数据流（M5b 全景）

```
OjFrame / JudgeWorker
        |
        v
 ProblemService.load(id)
   +- ProblemDao.meta(id)   -->  MySQL · problems 表
   +- ProblemLoader.cases(id) -> FS · problems/<id>/N.in,N.out
        |
        v
   Problem (含 ProblemMeta + List<TestCase>)
        |
        v
 MachineJudge.judge(...)  -->  C++ 外部判题机
        |
```

```
        v
SubmissionDao.save(sub)  -->  MySQL · submissions 表（事务）
```

测试点文件（.in / .out）与选手源码（.java / .cpp）始终留在文件系统，MySQL 只存可查询的结构化字段。

> 自 M5b 起，提交历史改由 SubmissionDao 入库；M5a 的 SubmissionStore（.ser 序列化）停用，不再两套并存。

动手实现: 两步走 (javac / java 实操)

> 需先装好 MySQL、建库 mini_oj, 并把驱动 mysql-connector-j-*.jar 放进 lib/。

第一步 考试向 — 最简 JDBC: 建表/插入/查询 (Statement)

src/oj/db/SimpleDb.java

```java
package oj.db;
import java.sql.*;
import java.util.*;
public class SimpleDb {
    static final String URL = "jdbc:mysql://localhost:3306/mini_oj?serverTimezone=UTC";
    public static void main(String[] args) throws Exception {
        try (Connection c = DriverManager.getConnection(URL, "root", "root");
             Statement st = c.createStatement()) {
            st.executeUpdate("CREATE TABLE IF NOT EXISTS sub(id INT, name VARCHAR(32),
 ↪  status VARCHAR(8))");
            st.executeUpdate("INSERT INTO sub VALUES(1,'alice','AC')");      // 增
            ResultSet rs = st.executeQuery("SELECT id,name,status FROM sub");// 查
            List<String> rows = new ArrayList<>();
            while (rs.next())                                                // 游标遍历
                rows.add(rs.getInt("id") + " " + rs.getString("name") + " " +
 ↪  rs.getString("status"));
            rows.forEach(System.out::println);
        }
    }
}
```

讲解:DriverManager.getConnection -> Statement -> executeUpdate/executeQuery -> ResultSet.next() 游标——JDBC 四步固定套路, 用在 OJ 自己的提交表上。

编译运行 (驱动挂 classpath):

```
javac -cp "lib/*" -d build src/oj/db/SimpleDb.java
java -cp "build:lib/*" oj.db.SimpleDb
# 1 alice AC
```

第二步 工程向 — DAO + 事务 + 预处理 + FS/DB 分工

升级为 Db.get() 单例、ProblemDao.meta/listMeta、SubmissionDao.save(PreparedStatement 防注入 + setAutoCommit(false)/commit/rollback 事务)、ProblemService 合并 DB 元数据 + FS 测试点(代码见上文【新产物架构】):

```
javac -cp "lib/*" -d build $(find src -name '*.java')
java -cp "build:lib/*" oj.Main
```

验收标准

1. 执行 schema.sql 建库建表后，向 problems 插入一条记录，运行 ProblemDao.meta(id) 能正确返回 ProblemMeta，字段值与数据库一致。
2. ProblemService.load(id) 能同时取到 DB 元数据（getTitle()、getTimeLimitMs()）和 FS 测试点（getCases().size() > 0），两者均正确。
3. 构造一个带 JudgeResult（status=AC, passed=3, total=3, elapsedMs=42）的 Submission，调用 SubmissionDao.save，用 SELECT * FROM submissions 验证数据库中出现对应行，status 字段值为 "AC"。
4. 模拟 save 执行到一半时抛出异常（可在第二条 SQL 前手动 throw new SQLException("test")），验证 submissions 表中没有半截记录（事务 rollback 生效）。
5. listByUser 按 submitted_at DESC 返回，最新的提交排在 get(0)，结果条数不超过传入的 limit。
6. 在 Db.get() 关闭后再次调用，能自动重建连接，不抛 "Connection is closed" 异常。
7. ProblemDao.meta 接受含单引号的 title（如 "It's a Trap!"），查询结果正确返回，不抛 SQL 语法异常（验证 PreparedStatement 防注入有效）。

08 ·M5c Swing 桌面客户端（Ch9，大前端）

考试相关度 ★★★(编程 + 填空双大题) · 真题考点参考:JFrame/布局/JButton/JTextField/JLabel/ActionListener。[第一步] 考试向 用考试级 Swing 写 OJ 提交窗口最简版 (选题 + 输入 + 提交按钮 + 标签显示判题结果);[第二步] 工程向 OjFrame/OjController MVC 解耦 + SwingWorker。

> 判题逻辑已经具备：数据库存题、C++ 判题机真正跑代码。现在缺少的是一扇窗户——用户还在用命令行粘贴代码、手动查库。本章用 Java Swing 为 Mini-OJ 装上可视化界面，把前三代的成果串联成一个完整的桌面应用。

【核心痛点】

M5b 之后，系统后端已经相当完善：ProblemService 从数据库读取题目元数据，ProblemLoader 加载测试点，MachineJudge 调用外部 C++ 判题机真实编译运行。但所有交互都停留在命令行或单元测试层面。

具体痛点如下：

1. 无题目浏览：用户不知道有哪些题，必须手动查数据库或翻代码。
2. 无语言选择：提交时语言类型硬编码在测试脚本里。
3. 无源码编辑区：用户只能把源文件路径写死，无法在界面里直接粘贴或编写代码。
4. 无结果展示：判题完成后结果只打印在控制台，没有弹窗或表格呈现给用户。

本章目标：用 Swing 构建一个 MVC 结构的桌面客户端，让用户在图形界面里选题、选语言、贴代码、提交，然后看到 AC/WA/TLE 等结果。

【引入课本知识点】

本章对应 Ch9——Java 图形用户界面（Swing）。

JFrame 与 BorderLayout

JFrame 是 Swing 的顶层窗口容器。BorderLayout 是其默认布局管理器，将窗口分为 NORTH/SOUTH/CENTER/EAST/W 五个区域，适合”工具栏在上、内容在中、操作按钮在下”的经典布局。

```
JFrame frame = new JFrame("Mini-OJ");
frame.setLayout(new BorderLayout());
frame.add(toolbar, BorderLayout.NORTH);
frame.add(scrollPane, BorderLayout.CENTER);
frame.add(buttonPanel, BorderLayout.SOUTH);
```

JComboBox — 下拉选择框

用于语言选择(cpp / python)和题目选择。泛型版本 JComboBox<String> 类型安全，通过 getSelectedItem() 获取当前选中值。题目下拉框统一用 JComboBox<String> 显示 “id 标题”，不要把它当成 JComboBox<Integer>，取题号要靠 OjFrame 内部维护的 id 列表换算。

```
JComboBox<String> langBox = new JComboBox<>(new String[]{"cpp", "python"});
String lang = (String) langBox.getSelectedItem();
```

JTextArea — 多行文本编辑区

源码编辑区的核心组件。需要用 JScrollPane 包裹以支持滚动，设置等宽字体提升代码可读性。

```
JTextArea codeArea = new JTextArea(20, 60);
codeArea.setFont(new Font("Monospaced", Font.PLAIN, 13));
JScrollPane scroll = new JScrollPane(codeArea);
```

JButton + Lambda 事件监听

Ch9 的重点之一：用函数式接口 ActionListener（单方法接口）配合 Lambda 表达式绑定按钮事件，替代匿名内部类写法，代码更简洁。

```
JButton submitBtn = new JButton(" 提交");
submitBtn.addActionListener(e -> controller.onSubmit());
```

ActionListener 是函数式接口（@FunctionalInterface），actionPerformed(ActionEvent e) 是其唯一抽象方法，Lambda e -> ... 正是该方法的实现体。

JLabel — 状态标签

工具栏里放一个 JLabel，用来实时显示当前状态（“就绪”、判题结果文本等），比每次都弹窗更轻量。本章 OjFrame 会真实持有这个状态标签字段并对外暴露 getResultLabel()，下一章（M6a）会直接复用它做” 评测中…“提示。

JOptionPane — 状态弹窗

用于在判题完成后弹出结果对话框，无需手动创建 JDialog。

```
JOptionPane.showMessageDialog(frame, result.toString(), " 判题结果",
    JOptionPane.INFORMATION_MESSAGE);
```

JTable — 提交历史表格

通过 DefaultTableModel 动态添加行，展示历史提交的题号、语言、状态、耗时。

EDT（Event Dispatch Thread）规则

Swing 不是线程安全的，所有 UI 操作必须在 EDT 上执行。长时间操作（如调用 MachineJudge）必须在后台线程（SwingWorker 或普通线程）里执行，完成后用 SwingUtilities.invokeLater 回到 EDT 更新界面。

【三代演进定位】

本章不产生新判题机代。Swing 客户端是” 大前端”，其定位是：

```
用户 -> OjFrame (View) -> OjController (Controller)
          |                        |
    ProblemService (Model)    MachineJudge (第三代)
          |                        |
      数据库 (M5b)              C++ 判题机进程
```

```
        |                          |
SubmissionDao.save()      JudgeResult 返回
```

- 第一代（M1–M3，已完成）：Solution.solve 在 JVM 内执行，题目硬编码，无 UI。
- 第二代（M4）：反射工厂 + 单文件配置，解耦题目元数据，仍无 UI。
- 第三代（M5a）：ProcessBuilder 调用外部 C++ 判题机，真实编译运行，仍无 UI。
- **M5c**（本章）：Swing 客户端直接调用第三代 MachineJudge 和 M5b 的 DAO，不经过网络（不使用 **Socket/ServerSocket**），所有调用均为本地方法调用。

删除说明：Ch13 网络内容已从课程中移除，因此客户端与判题机之间不存在任何 **Socket** 通信。OjController 直接持有 MachineJudge 和 SubmissionDao 的引用，在同一 JVM 进程内完成全部工作。

【新产物架构】

涉及的类与接口

类/接口	包	职责
`OjFrame`	`oj.gui`	View 层；持有全部 Swing 组件；不含业务逻辑
`OjController`	`oj.gui`	Controller 层；处理提交事件；协调 Model
`ProblemService`	`oj.service`	Model 层；从 DB + FS 加载题目（M5b 已实现）
`MachineJudge`	`oj.judge`	第三代判题机；ProcessBuilder 调用外部二进制
`SubmissionDao`	`oj.db`	持久化提交记录到数据库（M5b 已实现）
`Submission`	`oj.core`	提交数据模型（M5a 已实现）
`JudgeResult`	`oj.core`	判题结果数据模型

OjFrame（View 层）

OjFrame 负责所有 Swing 组件的创建和布局，对外暴露 getter 让 Controller 读取输入、写入结果，不直接调用任何业务逻辑。题目下拉用 `JComboBox<String>`，内部用一个 `problemIds` 列表把下标映射回真正的题号，`getSelectedProblemId()` 在没有选中时返回 `-1`。工具栏里有一个真实的状态标签 `resultLabel`，并通过 `getResultLabel()` 暴露给 Controller（下一章 M6a 会复用它）。

```
package oj.gui;

import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.*;
import java.util.List;

public class OjFrame extends JFrame {

    private final JComboBox<String> problemBox;
    private final JComboBox<String> langBox;
    private final JTextArea codeArea;
    private final JButton submitBtn;
    private final JLabel resultLabel;
    private final DefaultTableModel historyModel;
    private final JTable historyTable;

    private List<Integer> problemIds;   // 与 problemBox 下标对应

    public OjFrame() {
        super("Mini-OJ 桌面客户端");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(900, 650);
        setLayout(new BorderLayout(6, 6));

        // -- 北部工具栏 ----------------------------------------
        JPanel toolbar = new JPanel(new FlowLayout(FlowLayout.LEFT, 8, 4));

        toolbar.add(new JLabel(" 题目："));
        problemBox = new JComboBox<>();
        problemBox.setPreferredSize(new Dimension(260, 26));
        toolbar.add(problemBox);

        toolbar.add(new JLabel(" 语言："));
        langBox = new JComboBox<>(new String[]{"cpp", "python"});
        toolbar.add(langBox);

        submitBtn = new JButton(" 提交");
        toolbar.add(submitBtn);

        // 状态标签：实时显示判题状态，供 Controller 写入
```

```
        resultLabel = new JLabel(" 就绪");
        resultLabel.setForeground(new Color(0x33, 0x66, 0x99));
        toolbar.add(resultLabel);

        add(toolbar, BorderLayout.NORTH);

        // -- 中部源码编辑区 ------------------------------------
        codeArea = new JTextArea();
        codeArea.setFont(new Font("Monospaced", Font.PLAIN, 13));
        codeArea.setTabSize(4);
        JScrollPane codeScroll = new JScrollPane(codeArea);
        codeScroll.setBorder(BorderFactory.createTitledBorder(" 源码"));
        add(codeScroll, BorderLayout.CENTER);

        // -- 南部提交历史表格 ----------------------------------
        String[] cols = {" 提交 ID", " 题目 ID", " 语言", " 状态", " 通过/总计", " 耗时
        ↪  (ms)"};
        historyModel = new DefaultTableModel(cols, 0) {
            @Override
            public boolean isCellEditable(int row, int col) {
                return false;
            }
        };
        historyTable = new JTable(historyModel);
        historyTable.setFillsViewportHeight(true);
        JScrollPane tableScroll = new JScrollPane(historyTable);
        tableScroll.setPreferredSize(new Dimension(0, 180));
        tableScroll.setBorder(BorderFactory.createTitledBorder(" 提交历史"));
        add(tableScroll, BorderLayout.SOUTH);

        setLocationRelativeTo(null);
    }

    // -- 初始化题目下拉列表（由 Controller 在加载完数据后调用）--
    public void loadProblems(java.util.List<Integer> ids, java.util.List<String>
    ↪  titles) {
        this.problemIds = ids;
        problemBox.removeAllItems();
        for (int i = 0; i < ids.size(); i++) {
            problemBox.addItem(ids.get(i) + " " + titles.get(i));
        }
```

```
    }

    // -- 向历史表格追加一行 ----------------------------------
    public void appendHistory(int subId, int probId, String lang,
                              String status, String passTotal, long ms) {
        historyModel.addRow(new Object[]{subId, probId, lang, status, passTotal, ms});
        int last = historyModel.getRowCount() - 1;
        historyTable.scrollRectToVisible(historyTable.getCellRect(last, 0, true));
    }

    // -- Getters 供 Controller 读取 --------------------------
    public int getSelectedProblemId() {
        int idx = problemBox.getSelectedIndex();
        if (idx < 0 || problemIds == null) return -1;
        return problemIds.get(idx);
    }

    public String getSelectedLang() {
        return (String) langBox.getSelectedItem();
    }

    public String getSourceCode() {
        return codeArea.getText();
    }

    public JButton getSubmitBtn() {
        return submitBtn;
    }

    public JLabel getResultLabel() {
        return resultLabel;
    }
}
```

OjController（Controller 层）

OjController 持有 Model 的全部引用，构造签名为 `(OjFrame view, ProblemService svc, MachineJudge judge, SubmissionDao dao)`。构造时用 `svc.listMeta()` 把题目填进下拉框，并绑定提交按钮的 Lambda 事件监听器。提交操作在后台线程执行，避免阻塞 EDT。

注意几个与全局契约对齐的关键点：

- ProblemService 用 `listMeta()` / `meta(int)`，不是 `listAll()`。
- `SubmissionDao.save(Submission)` 返回 `void`，所以拿提交 ID 要先 `dao.save(sub)` 再 `sub.getId()`（id 由 Submission 静态计数自增）。
- 临时源文件写好后不要立即删除——它的路径要存进 Submission 的 `srcPath`，留作记录。

```
package oj.gui;

import oj.core.JudgeResult;
import oj.core.ProblemMeta;
import oj.core.Submission;
import oj.db.SubmissionDao;
import oj.judge.MachineJudge;
import oj.service.ProblemService;

import javax.swing.*;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class OjController {

    private final OjFrame view;
    private final ProblemService svc;
    private final MachineJudge judge;
    private final SubmissionDao dao;

    private static final String PROBLEMS_DIR = "problems";
    private static final String USER_NAME    = "student";   // 演示用固定用户

    public OjController(OjFrame view,
                        ProblemService svc,
                        MachineJudge judge,
                        SubmissionDao dao) {
        this.view  = view;
        this.svc   = svc;
        this.judge = judge;
        this.dao   = dao;
```

```
        initProblems();
        bindEvents();
    }

    // 从数据库加载题目列表，填充下拉框
    private void initProblems() {
        try {
            Map<Integer, ProblemMeta> all = svc.listMeta();
            List<Integer> ids    = new ArrayList<>(all.keySet());
            List<String>  titles = new ArrayList<>();
            ids.forEach(id -> titles.add(all.get(id).getTitle()));
            view.loadProblems(ids, titles);
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(view,
                "加载题目失败：" + ex.getMessage(), "错误",
                JOptionPane.ERROR_MESSAGE);
        }
    }

    // 绑定提交按钮：Lambda -> 后台线程执行判题，SwingUtilities 回 EDT 更新 UI
    private void bindEvents() {
        view.getSubmitBtn().addActionListener(e -> onSubmit());
    }

    void onSubmit() {
        int problemId = view.getSelectedProblemId();
        if (problemId < 0) {
            JOptionPane.showMessageDialog(view, "请先选择一道题目。",
                "提示", JOptionPane.WARNING_MESSAGE);
            return;
        }

        String src = view.getSourceCode().trim();
        if (src.isEmpty()) {
            JOptionPane.showMessageDialog(view, "源码不能为空。",
                "提示", JOptionPane.WARNING_MESSAGE);
            return;
        }

        String lang = view.getSelectedLang();
        view.getSubmitBtn().setEnabled(false);
```

```
        view.getResultLabel().setText(" 评测中…");

        // 后台线程：写源文件 -> 调 MachineJudge -> 入库 -> 回 EDT 更新 UI
        new Thread(() -> {
            JudgeResult result = null;
            int subId = -1;
            try {
                // 1. 落源文件（保留路径，存进 Submission.srcPath，不立即删除）
                String ext = lang.equals("cpp") ? ".cpp" : ".py";
                Path   tmp = Files.createTempFile("oj_submit_", ext);
                Files.write(tmp, src.getBytes(StandardCharsets.UTF_8));

                // 2. 获取题目元数据（时间限制）
                ProblemMeta meta = svc.meta(problemId);
                long timeMs = meta.getTimeLimitMs();
                int  memMb  = 256;   // 默认内存限制

                // 3. 调用第三代 MachineJudge（本地进程，无网络）
                String problemDir = PROBLEMS_DIR + "/" + problemId;
                result = judge.judge(problemDir, tmp.toString(),
                    lang, timeMs, memMb);

                // 4. 构造提交记录并持久化到数据库
                //    save() 返回 void，提交 ID 由 Submission 自增，存库后用 getId() 取回
                Submission sub = new Submission(
                    problemId, USER_NAME, lang, tmp.toString());
                sub.setResult(result);
                dao.save(sub);
                subId = sub.getId();

            } catch (IOException | InterruptedException ex) {
                Thread.currentThread().interrupt();
                final String msg = ex.getMessage();
                SwingUtilities.invokeLater(() -> {
                    view.getResultLabel().setText(" 提交出错");
                    view.getSubmitBtn().setEnabled(true);
                    JOptionPane.showMessageDialog(view,
                        " 提交出错：" + msg, " 错误",
                        JOptionPane.ERROR_MESSAGE);
                });
                return;
```

```
            }

            // 回到 EDT 更新界面
            final JudgeResult finalResult = result;
            final int         finalSubId  = subId;
            SwingUtilities.invokeLater(() -> {
                view.getSubmitBtn().setEnabled(true);
                view.getResultLabel().setText(finalResult.toString());

                // 追加历史表格一行
                view.appendHistory(
                    finalSubId, problemId, lang,
                    finalResult.getStatus().name(),
                    finalResult.getPassed() + "/" + finalResult.getTotal(),
                    finalResult.getElapsedMs());

                // 弹窗显示判题结果
                int msgType = finalResult.isAccepted()
                    ? JOptionPane.INFORMATION_MESSAGE
                    : JOptionPane.WARNING_MESSAGE;
                JOptionPane.showMessageDialog(view,
                    finalResult.toString() + "\n" + finalResult.getDetail(),
                    " 判题结果", msgType);
            });
        }).start();
    }
}
```

程序入口（Main 类）

Main 按全局契约组装 Model 层：ProblemDao / SubmissionDao 都是无参构造（内部用 Db.get() 取单例连接，没有 Db.connect()），ProblemService(ProblemDao, ProblemLoader)、MachineJudge(String judgeBinary) 都按契约构造。所有 Swing 操作放进 SwingUtilities.invokeLater。

```
package oj.gui;

import oj.db.ProblemDao;
import oj.db.SubmissionDao;
import oj.io.ProblemLoader;
import oj.judge.MachineJudge;
```

```java
import oj.service.ProblemService;

import javax.swing.*;

public class Main {

    public static void main(String[] args) {
        // 初始化 Model 层（DAO 无参构造，内部用 Db.get() 单例连接）
        ProblemService svc   = new ProblemService(new ProblemDao(),
                                                  new ProblemLoader("problems"));
        MachineJudge   judge = new MachineJudge("judge/judge");
        SubmissionDao  dao   = new SubmissionDao();

        // 所有 Swing 操作必须在 EDT 上执行
        SwingUtilities.invokeLater(() -> {
            OjFrame f = new OjFrame();
            new OjController(f, svc, judge, dao);
            f.setVisible(true);
        });
    }
}
```

关键交互时序（文字版）

```
[EDT] 用户点击"提交"
  -> OjController.onSubmit()
      -> 校验输入（EDT）
      -> 禁用按钮 + 状态标签显示"评测中…"（EDT）
      -> 启动新线程（非 EDT）
          -> Files.createTempFile / Files.write  <- 写源文件（路径留作 srcPath）
          -> svc.meta(problemId)                 <- 取题目时限
          -> judge.judge(...)                    <- ProcessBuilder，第三代
          -> new Submission(...) + setResult     <- 构造提交记录
          -> dao.save(sub); subId = sub.getId()  <- JDBC 持久化（save 返回 void）
          -> SwingUtilities.invokeLater(...)     <- 回 EDT
              -> 状态标签写入结果文本
              -> view.appendHistory(...)         <- 更新表格
              -> JOptionPane.showMessageDialog   <- 弹窗
```

-> 恢复按钮

没有任何 **Socket/ServerSocket**/网络调用。MachineJudge 通过 `ProcessBuilder` 启动本地可执行文件（系统进程），SubmissionDao 通过 JDBC 访问本地数据库，两者均为进程内的本地调用。

动手实现: 两步走 (javac / java 实操)

第一步 考试向 — 最简 OJ 提交窗口 (单文件)

src/oj/gui/SimpleOj.java

```
package oj.gui;
import javax.swing.*;
import java.awt.*;
public class SimpleOj {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {                // 在 EDT 上建界面
            JFrame f = new JFrame("Mini-OJ 提交");
            f.setLayout(new FlowLayout());
            f.setSize(380, 150);
            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            JComboBox<String> prob = new JComboBox<>(new String[]{ "1 A+B", "2 平方" });
            JTextField input = new JTextField(12);
            JButton submit = new JButton(" 提交");
            JLabel result = new JLabel(" 等待提交");
            submit.addActionListener(e ->                 // Lambda 监听 ActionEvent
                result.setText(" 已提交 [" + prob.getSelectedItem() + "] 输出 =" +
 ↪  input.getText()));
            f.add(new JLabel(" 题目")); f.add(prob);
            f.add(new JLabel(" 输出")); f.add(input);
            f.add(submit); f.add(result);
            f.setVisible(true);
        });
    }
}
```

讲解:JFrame+FlowLayout+JComboBox(选题)+JTextField(输入)+JButton+JLabel+ActionListener Lambda——Ch9 全部考点, 组成 OJ 的提交窗口。

编译运行 (需图形界面环境):

```
javac -d build src/oj/gui/SimpleOj.java
java -cp build oj.gui.SimpleOj
```

第二步 工程向 — MVC 解耦 + 后台判题

拆成 OjFrame(View)/OjController(Controller)/ProblemService+MachineJudge(Model)，点提交后用 SwingWorker 在后台调判题机、回 EDT 刷新，界面不卡（代码见上文【新产物架构】）：

```
javac -cp "lib/*" -d build $(find src -name '*.java')
java -cp "build:lib/*" oj.gui.Main
```

验收标准

1. 题目下拉：启动后 JComboBox<String> 自动填充数据库中全部题目（“id 标题”），按 id 升序，无需手动输入题号；取题号一律走 getSelectedProblemId()，未选中返回 -1。
2. 语言选择：JComboBox 包含 cpp 和 python 两项，选中值传递给 MachineJudge。
3. 源码编辑：JTextArea 使用等宽字体，支持 Tab 缩进，可粘贴任意长度源码。
4. 提交不冻结 **UI**：点击提交后按钮立即变灰、状态标签显示”评测中…“，判题期间界面保持可响应；判题完成后按钮恢复。
5. 状态标签：OjFrame 工具栏里有一个真实的 JLabel（getResultLabel() 暴露），提交过程中显示”评测中…“，完成后显示 JudgeResult.toString()（如 AC 3/3 (47ms)）。
6. 结果弹窗：判题完成后弹出 JOptionPane，AC 时显示 INFORMATION_MESSAGE，非 AC 时显示 WARNING_MESSAGE，内容为 JudgeResult.toString() 加 detail 字段。
7. 历史表格：每次提交后向 JTable 追加一行，包含提交 ID、题目 ID、语言、状态、通过/总计、耗时，表格自动滚动到最新行。
8. 持久化：提交记录通过 dao.save(sub)（返回 void）写入数据库，提交 ID 用 sub.getId() 取回；源文件路径作为 srcPath 一并入库（写文件后不立即删除）。
9. 构造契约：DAO 无参构造（内部 Db.get()），ProblemService(ProblemDao, ProblemLoader)、MachineJudge(String) 按契约组装；OjController 构造签名为 (OjFrame, ProblemService, MachineJudge, SubmissionDao)。
10. 无网络依赖：全程无 Socket、ServerSocket、URL、HttpURLConnection 等任何网络 API；MachineJudge 和 SubmissionDao 均为本地调用。
11. **MVC** 分离：OjFrame 不包含任何业务逻辑，OjController 不直接操作 Swing 组件样式，ProblemService/Machine
不感知 UI 层。
12. **EDT** 合规：所有 view.* 调用均在 EDT 上执行；MachineJudge 调用在独立线程；通过 SwingUtilities.invokeLat
回到 EDT，无线程安全问题。

09 ·M6a 多线程并发判题（Ch12，收官总决赛）

考试相关度 ★★ ·真题考点参考:`extends Thread`/run/start、join、synchronized。[第一步] 考试向 给 OJ 判题加最简并发（`Thread/Runnable` + `synchronized/join`）;[第二步] 工程向 泛型阻塞队列 + 工作线程池 + 调 C++ 判题机。

至此，Mini-OJ 终于能同时接待多名同学提交代码，而不必让后来者苦苦等待——这一章是整个项目的收官之战。

【核心痛点】

在 M5 完成之后，系统已经能调用真实的 C++ 判题机对外部源码进行编译、运行和评测。然而有两个致命问题依然悬而未决：

问题一：串行阻塞。如果两位同学同时点击”提交”，第二位提交会被完全阻塞，直到第一份代码评测完毕才能开始。课堂演示时只要稍微多几个人同时操作，界面就会卡住无响应。

问题二：死循环挂死系统。提交一份 `while(true){}` 的 Java 代码，在 M3 的纯 JVM 方案里会把整个 JVM 进程卡死。在 M5 虽然调用了外部判题机，但如果 Java 主线程同步等待 `Process.waitFor()`，同样会造成 GUI 无响应。

解决方案的核心是两步：第一步，把判题任务放进一个阻塞队列，让提交操作立即返回；第二步，用工作线程异步消费队列，每个工作线程调起 C++ 判题机——TLE 由 C++ 端的墙钟看门狗加 `setrlimit` 掐断死循环并返回 TLE 状态，Java 这侧不再需要外套超时计时器。

【引入课本知识点】

Ch12：Thread 与 Runnable

Java 并发的两种启动方式：

```
// 方式一：继承 Thread
class MyThread extends Thread {
    public void run() { /* ... */ }
}

// 方式二：实现 Runnable（推荐，保留继承槽位）
Thread t = new Thread(new MyRunnable());
t.setDaemon(true); // 守护线程：主程序退出时不阻止 JVM 关闭
t.start();
```

Ch12：synchronized + wait/notify

`synchronized` 锁住对象监视器，保证同一时刻只有一个线程进入临界区。`wait()` 让当前线程释放锁并挂起，`notify()` 唤醒一个等待中的线程（`notifyAll()` 唤醒全部）。

```
synchronized (lock) {
    while (条件不满足) {
        lock.wait();   // 释放锁，挂起
    }
    // 操作共享数据
    lock.notifyAll(); // 唤醒等待方
}
```

`wait()` 必须在 `while` 循环里检查条件，而不是 `if`——这是防范虚假唤醒（**spurious wakeup**）的标准写法。

Ch15：泛型阻塞队列

教材 Ch15 讲泛型容器。我们用泛型让 `JudgeQueue<T>` 既能装 `JudgeTask`，将来也能换其他任务类型，而不需要修改队列本身——这正是开闭原则的实践：对扩展开放，对修改关闭。

SwingWorker：EDT 与后台线程

Swing 的所有界面操作必须在事件派发线程（**EDT**）上执行；而判题调用（ProcessBuilder、文件 IO）必须在后台线程上执行，否则界面冻结。`SwingWorker<T, V>` 提供了标准的两阶段模型：

- `doInBackground()`：后台线程执行耗时操作，返回结果。
- `done()`：EDT 线程取回结果并刷新界面。

【三代演进定位】

代次	所在里程碑	判题执行方式	并发能力	超时控制
第一代	M1–M3	Java 对象 `Solution.solve()` 在 JVM 内直接调用	无，全程串行	无法控制死循环
第二代	M4	反射工厂 `JudgeFactory` 动态加载 `Judge` 实现	无，全程串行	无法控制死循环

代次	所在里程碑	判题执行方式	并发能力	超时控制
第三代（串行）	M5a	MachineJudge 通过 ProcessBuilder 调外部 C++ 判题机	无，waitFor() 同步阻塞	C++ 侧 setrlimit 控制
第三代（并发化）	**M6a**（本章）	同 **M5a**，但由工作线程异步消费 **JudgeQueue**	**N** 个守护工作线程并行	**C++** 墙钟看门狗 + **setrlimit**，**Java** 不外套超时

本章是第三代的并发化收官：判题执行逻辑不变，变的是调度层——从单线程同步调用，升级为生产者-消费者队列驱动的多线程并发调用。

【新产物架构】

本章新增或改造以下类，均位于全局契约规定的包路径下。

类关系总览

```
OjController（生产者）
    | onSubmit()
    |  +- SwingWorker.doInBackground()
    |       +- JudgeQueue.put(task)      <--- 返回立即，不阻塞 EDT
    |
JudgeQueue<JudgeTask>  （阻塞队列，synchronized + wait/notify）
    |
JudgeWorker（消费者，Runnable，守护线程）
    |  run() 死循环
    |   +- JudgeQueue.take()             <--- 空时挂起
    |   +- 把源码字节写到持久文件
    |   +- MachineJudge.judge(...)       <--- 调 C++ 判题机
    |   +- new Submission(...) + save    <--- 入库
    |   +- JudgeTask.complete(result)    <--- 回填，唤醒等待方
    |
JudgeTask（工作项，携带源码 + synchronized 的 complete/await）
    +- SwingWorker.done()  <--- EDT 刷新界面
```

---

oj.core.JudgeTask

携带一次提交的全部信息，同时充当 **Future** 的简化版：`await()` 挂起调用者直到评测完成，`complete()` 回填结果并唤醒。

```
package oj.core;

import java.io.Serializable;

public class JudgeTask implements Serializable {
    private static final long serialVersionUID = 1L;

    private final int submissionId;
    private final int problemId;
    private final String userName;
    private final String lang;
    private final byte[] source;      // 源码字节，避免依赖临时文件路径
    private JudgeResult result;
    private boolean done;

    public JudgeTask(int submissionId, int problemId,
                     String userName, String lang, byte[] source) {
        this.submissionId = submissionId;
        this.problemId    = problemId;
        this.userName     = userName;
        this.lang         = lang;
        this.source       = source;
        this.done         = false;
    }

    /** 工作线程调用：回填结果，唤醒所有 await() 调用者 */
    public synchronized void complete(JudgeResult r) {
        this.result = r;
        this.done   = true;
        notifyAll();
    }

    /** 提交方调用：阻塞到评测完成后返回结果 */
    public synchronized JudgeResult await() throws InterruptedException {
        while (!done) {
            wait();
        }
```

```
        return result;
    }

    public int getSubmissionId() { return submissionId; }
    public int getProblemId()    { return problemId; }
    public String getUserName()  { return userName; }
    public String getLang()      { return lang; }
    public byte[] getSource()    { return source; }
    public JudgeResult getResult() { return result; }
}
```

---

oj.judge.queue.JudgeQueue

泛型阻塞队列，用 LinkedList 作底层存储，通过 synchronized + wait/notify 实现生产者-消费者协调。

```
package oj.judge.queue;

import java.util.LinkedList;
import java.util.Queue;

public class JudgeQueue<T> {
    private final Queue<T> queue = new LinkedList<>();
    private final int capacity;

    public JudgeQueue(int capacity) {
        this.capacity = capacity;
    }

    /** 生产者：队列满时挂起，有空位后入队并唤醒消费者 */
    public synchronized void put(T item) throws InterruptedException {
        while (queue.size() >= capacity) {
            wait();
        }
        queue.offer(item);
        notifyAll();
    }

    /** 消费者：队列空时挂起，有元素后出队并唤醒生产者 */
    public synchronized T take() throws InterruptedException {
        while (queue.isEmpty()) {
```

```
            wait();
        }
        T item = queue.poll();
        notifyAll();
        return item;
    }

    public synchronized int size() {
        return queue.size();
    }
}
```

设计说明： capacity 参数让队列在请求洪峰时有背压（back-pressure）能力，防止内存无限增长。课堂场景设 64 即可。

---

oj.judge.queue.JudgeWorker

消费者，实现 Runnable，以守护线程方式运行。核心流程：从队列取任务 -> 把源码字节写到持久文件（如 submissions/sub<id>.<ext>，不要写完立刻删，因为这个路径要存进 Submission 的 srcPath）-> 调 MachineJudge -> new Submission(...) + setResult + dao.save(sub) -> 回填 JudgeTask。

关键契约点：

- 不要 new MachineJudge() / new SubmissionDao()——它们从构造器传入。
- SubmissionDao.save(Submission) 返回 void，没有 save(int, JudgeResult) 这种重载；要先构造 Submission 再 save。
- 任何异常都要 catch 住，绝不让工作线程崩溃退出；异常时给 task.complete() 一个 RE 结果。

```
package oj.judge.queue;

import oj.core.JudgeResult;
import oj.core.JudgeTask;
import oj.core.Status;
import oj.core.Submission;
import oj.db.SubmissionDao;
import oj.judge.MachineJudge;

import java.io.IOException;
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Path;
```

```
import java.nio.file.Paths;

public class JudgeWorker implements Runnable {
    private static final String PROBLEMS_DIR    = "problems";
    private static final String SUBMISSIONS_DIR = "submissions";

    private final JudgeQueue<JudgeTask> queue;
    private final SubmissionDao dao;
    private final MachineJudge machine;

    public JudgeWorker(JudgeQueue<JudgeTask> queue,
                       SubmissionDao dao,
                       MachineJudge machine) {
        this.queue   = queue;
        this.dao     = dao;
        this.machine = machine;
    }

    @Override
    public void run() {
        while (true) {                          // 守护线程，随 JVM 退出
            JudgeTask task = null;
            try {
                task = queue.take();            // 队列空时挂起，不自旋
                JudgeResult result = evaluate(task);

                // 构造提交记录并持久化（save 返回 void，没有 save(id,result) 重载）
                Submission sub = new Submission(
                    task.getProblemId(), task.getUserName(),
                    task.getLang(), srcPathFor(task));
                sub.setResult(result);
                dao.save(sub);

                task.complete(result);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            } catch (Exception e) {
                // 任何异常都不能让工作线程崩溃退出；回填一个 RE
                JudgeResult err = new JudgeResult(
                    Status.RE, 0, 0, "Worker error: " + e.getMessage(), 0L);
```

```
                if (task != null) {
                    task.complete(err);
                }
            }
        }
    }

    private JudgeResult evaluate(JudgeTask task) throws IOException {
        // 1. 把源码字节写到持久文件（保留，不删除；路径要存进库）
        Path dir = Paths.get(SUBMISSIONS_DIR);
        Files.createDirectories(dir);
        Path src = Paths.get(srcPathFor(task));
        try (OutputStream out = Files.newOutputStream(src)) {
            out.write(task.getSource());
        }

        // 2. 题目测试点目录
        String problemDir = PROBLEMS_DIR + "/" + task.getProblemId();

        // 3. 调外部 C++ 判题机（TLE / MLE 由 C++ 侧 setrlimit 控制）
        return machine.judge(
            problemDir,
            src.toString(),
            task.getLang(),
            2000L,   // 最大墙钟 ms，C++ 判题机内部会按题目配置截断
            256      // 最大内存 MB
        );
    }

    // 持久化源文件路径：submissions/sub<id>.<ext>
    private String srcPathFor(JudgeTask task) {
        return SUBMISSIONS_DIR + "/sub" + task.getSubmissionId()
             + langToExt(task.getLang());
    }

    private String langToExt(String lang) {
        switch (lang.toLowerCase()) {
            case "java":   return ".java";
            case "python": return ".py";
            case "c":      return ".c";
            default:       return ".cpp";
```

```
        }
    }
}
```

---

oj.gui.OjController — SwingWorker 提交

OjController 是本章改造最重的类。构造签名仍按全局契约：(OjFrame view, ProblemService svc, MachineJudge judge, SubmissionDao dao)。构造里做三件事：建 JudgeQueue<JudgeTask>、启动 N 个守护 JudgeWorker（复用传入的 **dao**、**judge**，不要 **new MachineJudge() / new SubmissionDao()**）、用 svc.listMeta() 初始化题目下拉并绑定提交。

提交按钮的响应逻辑从直接调用 MachineJudge，改为：在 SwingWorker.doInBackground() 里把任务放进队列、调 await() 等结果，在 done() 里回到 EDT 刷新 getResultLabel()、追加历史、非 AC 弹窗。题号一律走 view.getSelectedProblemId()，不要把下拉框当 JComboBox<Integer> 强转。

```
package oj.gui;

import oj.core.JudgeResult;
import oj.core.JudgeTask;
import oj.core.ProblemMeta;
import oj.db.SubmissionDao;
import oj.judge.MachineJudge;
import oj.judge.queue.JudgeQueue;
import oj.judge.queue.JudgeWorker;
import oj.service.ProblemService;

import javax.swing.*;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.atomic.AtomicInteger;

public class OjController {
    private static final int WORKER_COUNT   = 4;
    private static final int QUEUE_CAPACITY = 64;
    private static final String USER_NAME   = "student";

    private final OjFrame view;
    private final ProblemService svc;
```

```
    private final MachineJudge judge;
    private final SubmissionDao dao;

    private final JudgeQueue<JudgeTask> queue;
    private final AtomicInteger submissionIdGen = new AtomicInteger(1);

    public OjController(OjFrame view,
                        ProblemService svc,
                        MachineJudge judge,
                        SubmissionDao dao) {
        this.view  = view;
        this.svc   = svc;
        this.judge = judge;
        this.dao   = dao;
        this.queue = new JudgeQueue<>(QUEUE_CAPACITY);

        startWorkers();
        initProblems();
        bindEvents();
    }

    // 启动 N 个守护工作线程，复用传入的 dao / judge（不要 new）
    private void startWorkers() {
        for (int i = 0; i < WORKER_COUNT; i++) {
            Thread t = new Thread(
                new JudgeWorker(queue, dao, judge),
                "judge-worker-" + i);
            t.setDaemon(true);   // JVM 退出时不等待工作线程
            t.start();
        }
    }

    // 用 svc.listMeta() 填充题目下拉框
    private void initProblems() {
        try {
            Map<Integer, ProblemMeta> all = svc.listMeta();
            List<Integer> ids    = new ArrayList<>(all.keySet());
            List<String>  titles = new ArrayList<>();
            ids.forEach(id -> titles.add(all.get(id).getTitle()));
            view.loadProblems(ids, titles);
        } catch (Exception ex) {
```

```
            JOptionPane.showMessageDialog(view,
                " 加载题目失败: " + ex.getMessage(), " 错误",
                JOptionPane.ERROR_MESSAGE);
        }
    }

    private void bindEvents() {
        view.getSubmitBtn().addActionListener(e -> onSubmit());
    }

    private void onSubmit() {
        // 1. 从界面读取输入（在 EDT 上执行，安全）
        int problemId = view.getSelectedProblemId();
        if (problemId < 0) {
            JOptionPane.showMessageDialog(view, " 请先选择一道题目。",
                " 提示", JOptionPane.WARNING_MESSAGE);
            return;
        }

        String lang = view.getSelectedLang();
        String src  = view.getSourceCode();
        if (src.isBlank()) {
            JOptionPane.showMessageDialog(view, " 源码不能为空。",
                " 提示", JOptionPane.WARNING_MESSAGE);
            return;
        }

        int subId = submissionIdGen.getAndIncrement();
        byte[] srcBytes = src.getBytes(StandardCharsets.UTF_8);
        final JudgeTask task = new JudgeTask(
            subId, problemId, USER_NAME, lang, srcBytes);

        // 2. 禁用提交按钮，显示等待状态
        view.getSubmitBtn().setEnabled(false);
        view.getResultLabel().setText(" 评测中…");

        // 3. 后台线程: 入队 + 等待结果
        SwingWorker<JudgeResult, Void> worker = new SwingWorker<>() {
            @Override
            protected JudgeResult doInBackground() throws Exception {
                queue.put(task);      // 若队列满则挂起（背压）
```

```
                return task.await(); // 等待工作线程回填结果
            }

            @Override
            protected void done() {  // 回到 EDT
                view.getSubmitBtn().setEnabled(true);
                try {
                    JudgeResult r = get();
                    view.getResultLabel().setText(r.toString());
                    view.appendHistory(
                        task.getSubmissionId(), task.getProblemId(),
                        task.getLang(), r.getStatus().name(),
                        r.getPassed() + "/" + r.getTotal(),
                        r.getElapsedMs());
                    if (!r.isAccepted()) {
                        JOptionPane.showMessageDialog(
                            view,
                            r.getDetail(),
                            r.getStatus().name(),
                            JOptionPane.WARNING_MESSAGE);
                    }
                } catch (Exception ex) {
                    view.getResultLabel().setText(" 提交失败: " + ex.getMessage());
                }
            }
        };
        worker.execute();
    }
}
```

---

oj.gui.OjFrame — 界面骨架

OjFrame 沿用 08 的 API（已含 `getResultLabel()`），本章不新增字段、不重命名 getter。控制器需要的方法就是 08 那一套：`getSubmitBtn()`、`getSelectedProblemId()`、`getSelectedLang()`、`getSourceCode()`、`getResultLabel()`、`loadProblems(...)`、`appendHistory(...)`。

动手实现: 两步走 (javac / java 实操)

第一步 考试向 — 最简并发判题:Thread + synchronized + join

src/oj/Demo6a.java

```
package oj;
public class Demo6a {
    static int done = 0;
    static synchronized void finish(String who) {     // 同步：保护共享计数
        done++;
        System.out.println(who + " 判完，累计 " + done);
    }
    public static void main(String[] args) throws InterruptedException {
        Runnable job = () -> {
            try { Thread.sleep(50); } catch (InterruptedException e) { }
            finish(Thread.currentThread().getName());
        };
        Thread t1 = new Thread(job, "worker-1");
        Thread t2 = new Thread(job, "worker-2");
        t1.start(); t2.start();                          // 两个判题线程并发
        t1.join();  t2.join();                           // 主线程等它们判完
        System.out.println(" 全部完成：" + done);
    }
}
```

讲解:Runnable + Thread.start() 并发、synchronized 保护共享变量、join() 让主线程等待——Ch12 考点，模拟 OJ 多份提交并发判题。

编译运行:

```
javac -d build src/oj/Demo6a.java
java -cp build oj.Demo6a
# worker-1 判完，累计 1
# worker-2 判完，累计 2   (顺序可能不同)
# 全部完成：2
```

第二步 工程向 — 泛型阻塞队列 + 工作线程池 + 调 C++ 判题机

JudgeTask(wait/notify 回填结果)、JudgeQueue<T>(泛型阻塞队列)、JudgeWorker(N 个守护线程消费队列、落临时文件、调 MachineJudge)、GUI 用 SwingWorker(代码见上文【新产物架构】):

```
cd judge && g++ -O2 -std=c++17 -o judge judge.cpp && cd ..
javac -cp "lib/*" -d build $(find src -name '*.java')
java -cp "build:lib/*" oj.gui.Main
```

验收标准

1. 并发提交不阻塞 **EDT**：同时点击提交按钮两次，两份代码均能在队列中排队，状态标签”评测中…“正常显示，界面不卡顿。
2. 死循环源码返回 **TLE**：提交 `while(true){}` 的 Java 或 C++ 代码，C++ 判题机在配置的时间限制后返回 `TLE`，Java 侧正常展示结果，不挂死。
3. 工作线程数量正确：用调试输出或线程 dump 确认恰好有 `WORKER_COUNT`(默认 4)个名为 `judge-worker-N` 的守护线程在运行，且它们复用构造器传入的同一个 `dao` 与 `judge`，没有任何 `new MachineJudge()` / `new SubmissionDao()`。
4. 结果最终一致：任意一次提交的 `JudgeResult`，在 `OjFrame` 展示的内容与 `SubmissionDao` 持久化到数据库的内容完全一致；源文件落在 `submissions/sub<id>.<ext>` 且未被删除，路径存进 Submission 的 `srcPath`。
5. 异常不崩溃工作线程：向工作线程注入一个不存在的题目 ID，验证捕获 `Exception` 后工作线程仍然存活并继续处理后续任务，对应任务返回 `RE` 状态。
6. 背压生效：将 `QUEUE_CAPACITY` 设为 2，同时发起 5 次提交，第 3–5 次的 `SwingWorker.doInBackground()` 应阻塞在 `queue.put()`，不丢任务也不抛异常。
7. **getter 与 08 一致**：OjController 全程使用 08 定义的 OjFrame API（`getSubmitBtn` / `getSelectedProblemId` / `getSelectedLang` / `getSourceCode` / `getResultLabel`），题号走 `getSelectedProblemId()`，不把下拉当 `JComboBox<Integer>` 强转；OjController 构造签名为 `(OjFrame, ProblemService, MachineJudge, SubmissionDao)`。
8. 项目整体可运行：从 M1 到 M6a 的全部测试用例（含 M4 单文件配置、M5a 外部判题机、M5b 数据库持久化、M5c Swing 界面）在同一个项目内编译通过并正确运行，不存在孤立分支或未接入的模块。

> 项目收官说明 M6a 是 Mini-OJ 的最后一个里程碑。至此，系统完整地串联了面向对象设计（M1–M3）、反射与配置（M4）、外部进程与文件 IO（M5a）、数据库持久化（M5b）、Swing GUI（M5c），以及本章的多线程并发（M6a）。Ch13 网络编程不纳入本项目，`JudgeServer`/`JudgeClient`/`Socket`/`ServerSocket` 均不出现在任何章节中。同学们掌握了这条主线，便已综合运用了一学期 Java 课程的核心知识。

Mini-OJ 第三代判题机 (judge)

单文件、编译型的判题机: 真编译、真运行、真限资源, 跑全部测试点, 比对输出, 输出一行 **JSON** 判定。是 Mini-OJ「判题机三代演进」里的第三代;Java OJ(`MachineJudge`) 只负责 `ProcessBuilder` 调用它 + 正则解析结果。

能力一览

- 多语言:`cpp` / `c` / `python` / `java`(自动编译 / 语法检查 / 构造运行命令)。
- 完整判定:`AC` / `WA` / `PE` / `TLE` / `MLE` / `RE` / `CE` / `ERR`。
- 资源限制:`setrlimit`(CPU/AS/输出/栈)+ 父进程墙钟看门狗 + 进程组整组 **SIGKILL**(连同子孙进程一

起掐); 死循环必出 TLE、爆内存必出 MLE。

- 编译保护: 编译也有墙钟上限 (g++ 10s / javac 15s), 防模板炸弹挂死。
- 比对: 逐行去尾空白 + 去末尾空行;--special 开浮点容差 (1e-6, 逐 token); 仅空白排布不同判 PE。
- 健壮: 缺文件 / 无用例 / 编译器缺失 / 异常输出都有明确状态, 绝不崩。

编译

```
make            # 等价于 g++ -O2 -std=c++17 -o judge judge.cpp
```

用法

```
./judge --problem <题目目录> --src <提交源码> --lang <cpp|c|python|java> \
        [--time-ms 1000] [--mem-mb 256] [--special]
# --time / --mem 是 --time-ms / --mem-mb 的别名
```

题目目录约定 (成对、按数字排序; 非 .in/.out 文件忽略):

```
problems/<id>/
+-- 1.in   1.out
+-- 2.in   2.out
+-- 3.in   3.out
```

输出 (stdout 一行 JSON)

```
{"status":"AC","passed":3,"total":3,"time_ms":3,"mem_kb":1840,"detail":""}
```

字段	含义
status	AC/WA/PE/TLE/MLE/RE/CE/ERR
passed / total	通过测试点数 / 总数
time_ms	最大测试点 CPU 耗时
mem_kb	峰值内存 (ru_maxrss)
detail	失败说明 (WA/PE/RE 的 case 号、CE 的编译错误等)

状态:AC 通过·WA 答案错·**PE** 格式错 (仅空白差异) ·TLE 超时·MLE 超内存·RE 运行错 (崩溃/非零退出) ·CE 编译/语法错·ERR 判题机内部错。

例子

```
make test       # 跑全部示例:AC/WA/TLE/RE/CE/MLE/PE + python + 浮点 special
./judge --problem examples/problems/aplusb --src examples/solutions/ac.cpp  --lang cpp
./judge --problem examples/problems/avg    --src examples/solutions/avg.cpp --lang cpp
 ↪  --special
```

放置位置 (集成进 Mini-OJ 项目)

```
mini-oj/
+-- src/oj/...            # Java 代码(MachineJudge 在 oj.judge)
+-- problems/<id>/        # 题目:config.txt + N.in/N.out
+-- judge/
    +-- judge.cpp
    +-- judge             # make 出来的二进制(.gitignore)
```

Java OJ 如何调用 (第三代核心:ProcessBuilder + 正则解析)

```
// oj.judge.MachineJudge
Process p = new ProcessBuilder(
    "judge/judge", "--problem", "problems/1", "--src", "sub.cpp", "--lang", "cpp",
    "--time-ms", "1000", "--mem-mb", "256"
).start();
String json = new String(p.getInputStream().readAllBytes(), StandardCharsets.UTF_8);
p.waitFor();
// 用正则从 json 抽 status/passed/total/time_ms, 构造 oj.core.JudgeResult
```

完整封装见教程 **M5a** ·第三代判题机一章的 MachineJudge。

限制与边界 (教学/单机版)

- 用 setrlimit + 进程组隔离, 但未做完整沙箱 (无 seccomp/namespace/chroot、不禁系统调用、不隔离文件系统)。只在你信任的本地/实验室环境运行提交。
- 内存判定以 ru_maxrss 为准;RLIMIT_AS 只对 c/cpp 设硬上限 (java/python 解释器预留巨量虚拟内存, 设 AS 会误杀)。
- CPU 时限到点可能由内核以 SIGXCPU 或硬限 SIGKILL 触发, 判题机据"CPU 已超限" 统一归 TLE。
- java 提交的 public 类名须为 Main(判题机会把源码复制成 Main.java 再 javac)。