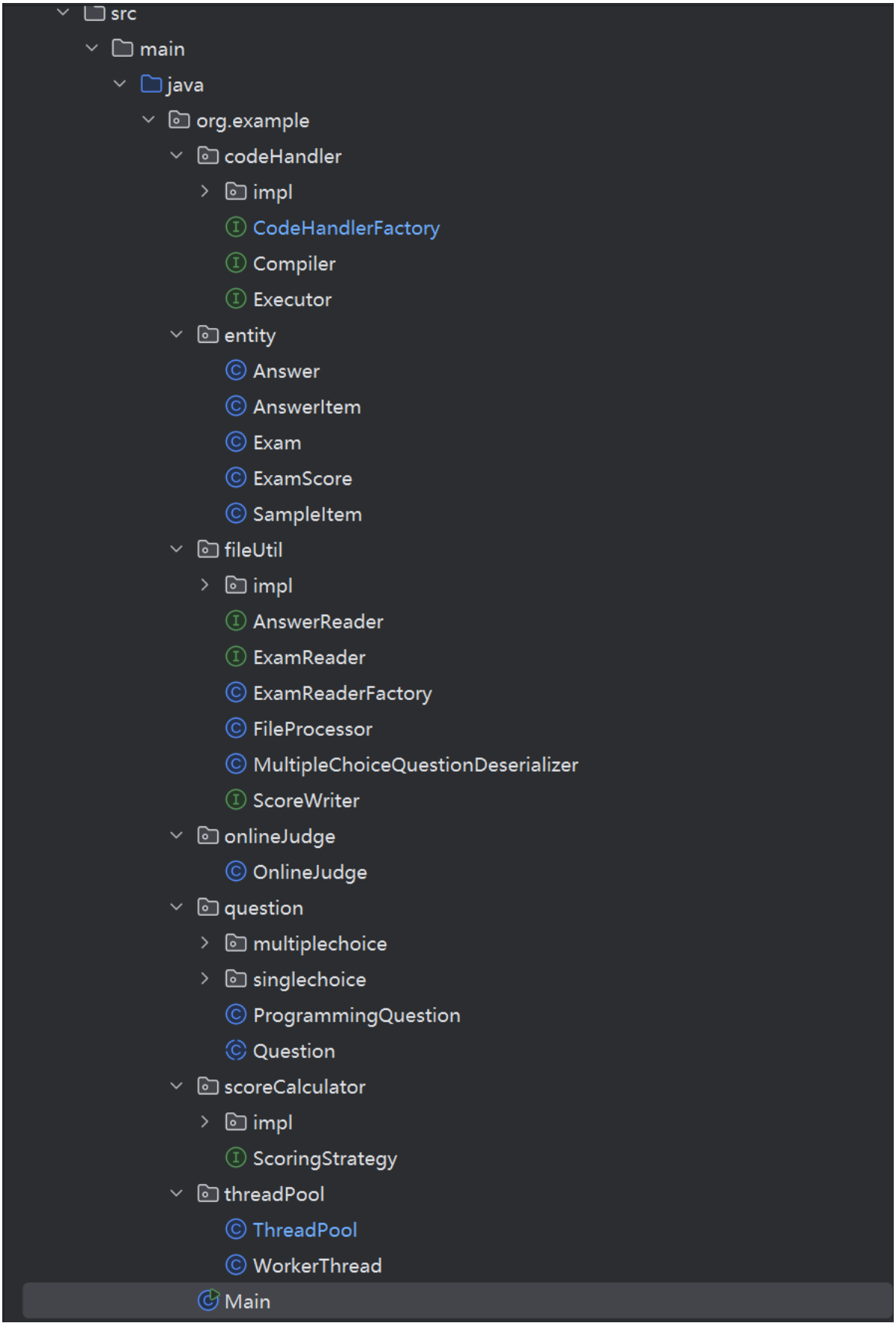
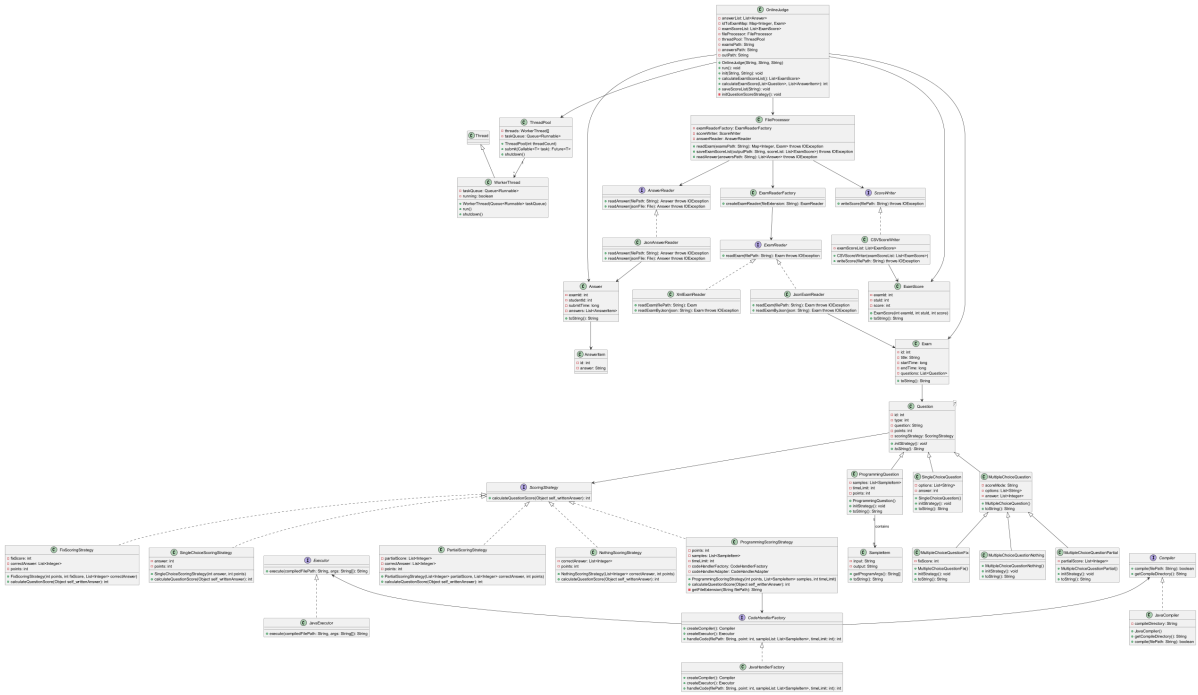
设计文档

211870287 丁旭

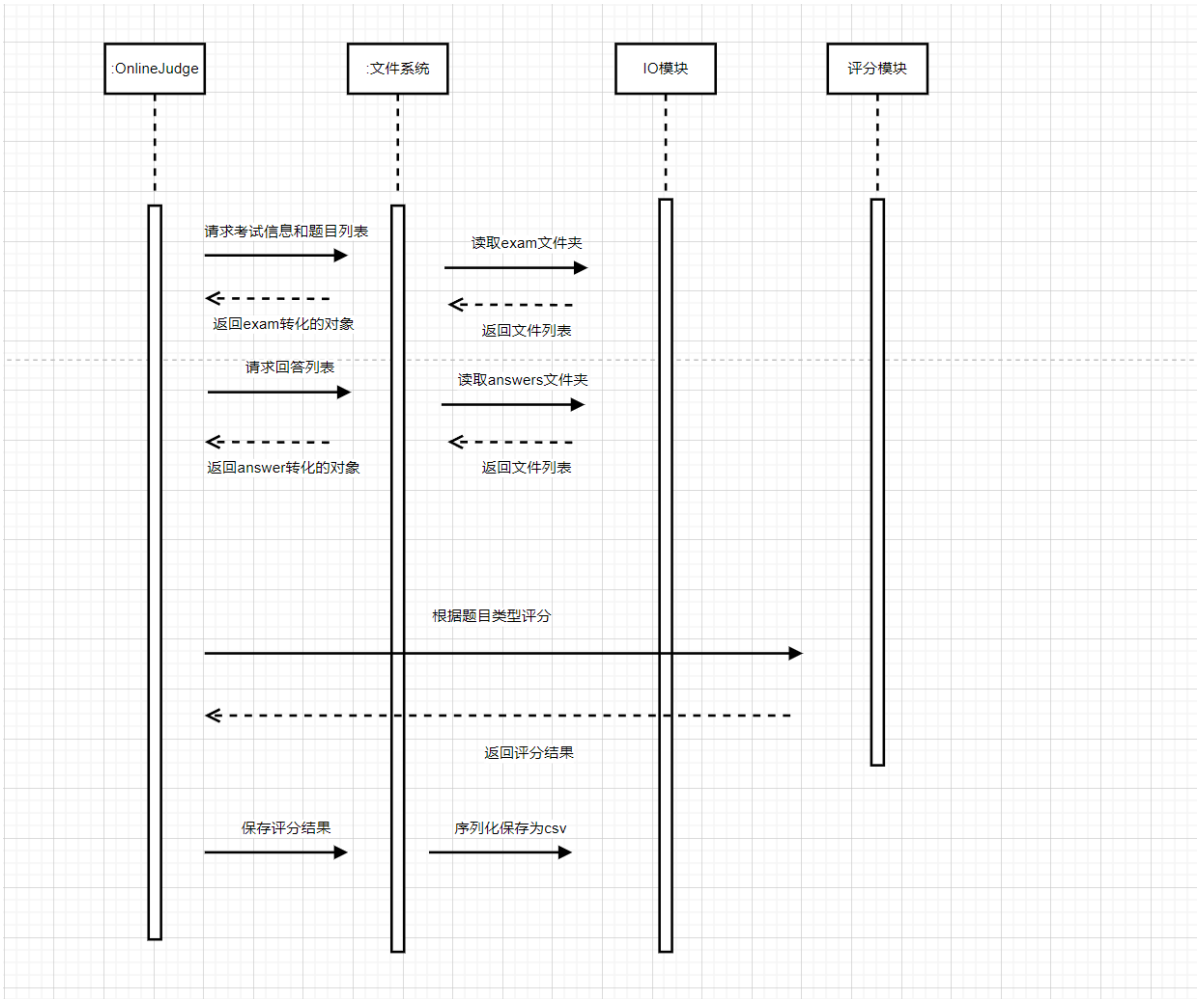
代码目录

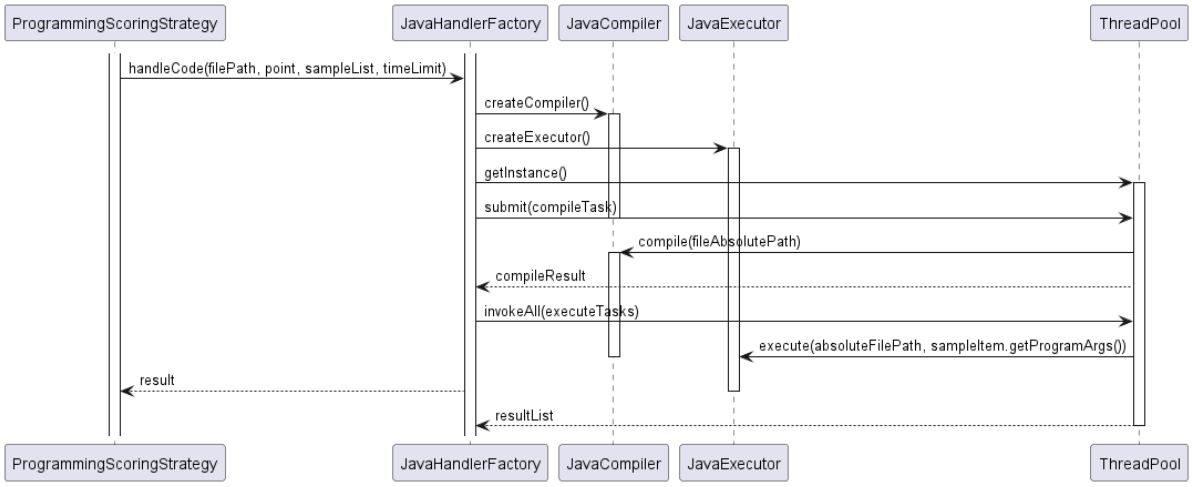


类图



流程图


:OnlineJudge
:文件系统
IO模块
评分模块
请求考试信息和题目列表
读取exam文件夹
返回exam转化的对象
返回文件列表
请求回答列表
读取answers文件夹
返回answer转化的对象
返回文件列表
根据题目类型评分
返回评分结果
保存评分结果
序列化保存为csv



设计原则

- [x] 单一职责

  如ExamScore类，仅作为对csv文件的映射，将每一个学生的成绩保存在表中，作为一个实体类即成绩单

```java
@Data
public class ExamScore {
    2 usages
    @JsonProperty(index = 0)
    private int examId;
    2 usages
    @JsonProperty(index = 1)
    private int stuId;
    2 usages
    @JsonProperty(index = 2)
    private int score;

    // 构造函数
    2 usages  211870287
    public ExamScore(int examId, int stuId, int score) {
        this.examId = examId;
        this.stuId = stuId;
        this.score = score;
    }

    211870287
    @Override
    public String toString() {
        return "ExamScore{" +
                "examId='" + examId + '\'' +
                ", stuId='" + stuId + '\'' +
                ", score=" + score +
                '}';
    }
}
```

- [x] 开闭原则

使用抽象类question作为所有子类的父类，包含最基本的属性id，type，question，points，scoringStrategy（对问题的计算策略，这里的scoringStrategy是一个接口，根据不同question的计分策略按需实现）

```
@Data
public abstract class Question<T> {

    1 usage
    private int id;
    1 usage
    private int type;
    1 usage
    private String question;
    1 usage
    private int points;

    private ScoringStrategy scoringStrategy;

    1 usage   5 overrides   211870287
    public void initStrategy(){}

    6 overrides   211870287
    @Override
    public String toString() {...}
}
```

- [x] 里氏代换原则

这里使用到Question父类进行计分策略的初始化，可以用其任意子类来进行各自的计分策略的初始化。

```
1 usage   211870287
private  void initQuestionScoreStrategy(){
    for (Exam exam:idToExamMap.values()){
        exam.getQuestions().forEach(Question::initStrategy);
    }
}
```

- [x] 依赖倒转原则

Exam依赖于抽象类的列表List,而非具体的 多选题、 单选题、编程题

```java
@Data
public class Exam {
    1 usage
    private int id;
    1 usage
    private String title;
    1 usage
    private long startTime;
    1 usage
    private long endTime;

    1 usage
    private List<Question> questions;

    211870287
    @Override
    public String toString() {...}
}
```

- [x] 接口隔离原则

将io处理拆分成三个细分的专业接口

1. 满足**单一职责原则**，将一组相关的操作定义在一个接口中，且在满足高内聚的前提下，接口中的方法越少越好。
2. 可以在进行系统设计时采用**定制服务**的方式，即**为不同的客户端提供宽窄不同的接口**

```java
2 usages  1 implementation  211870287
public interface AnswerReader {
    1 implementation  211870287
    Answer readAnswer(String filePath) throws IOException;
}
```

```java
8 usages  2 implementations  211870287
public interface ExamReader {
    3 usages  2 implementations  211870287
    Exam readExam(String filePath) throws IOException;
}
```

```java
8 usages  2 implementations  211870287
public interface ExamReader {
    3 usages  2 implementations  211870287
    Exam readExam(String filePath) throws IOException;
}
```

- [x] 合成复用原则

Question类组合ScoringStrategy，组合复用以使其可以黑箱复用

```java
@Data
public abstract class Question<T> {

    1 usage
    private int id;
    1 usage
    private int type;
    1 usage
    private String question;
    1 usage
    private int points;

    private ScoringStrategy scoringStrategy;

    1 usage  5 overrides  211870287
    public void initStrategy(){}

    6 overrides  211870287
    @Override
    public String toString() {...}
}
```

- [x] 迪米特法则

在设计的过程中，尽可能的减少了类之间的不必要的通信，使用抽象工厂和接口来减少不同类之间的通信

```java
public class FileProcessor {
    2 usages
    private ExamReaderFactory examReaderFactory = new ExamReaderFactory();
    2 usages
    private ScoreWriter scoreWriter;
    2 usages
    private AnswerReader answerReader;

    1 usage  211870287
    public Map<Integer, Exam> readExam(String examsPath) throws IOException {...}

    1 usage  211870287
    public void saveExamScoreList(String outputPath, List<ExamScore> scoreList) throws IOException {...}

    1 usage  211870287
    public List<Answer> readAnswer(String answersPath) throws IOException {...}
}
```

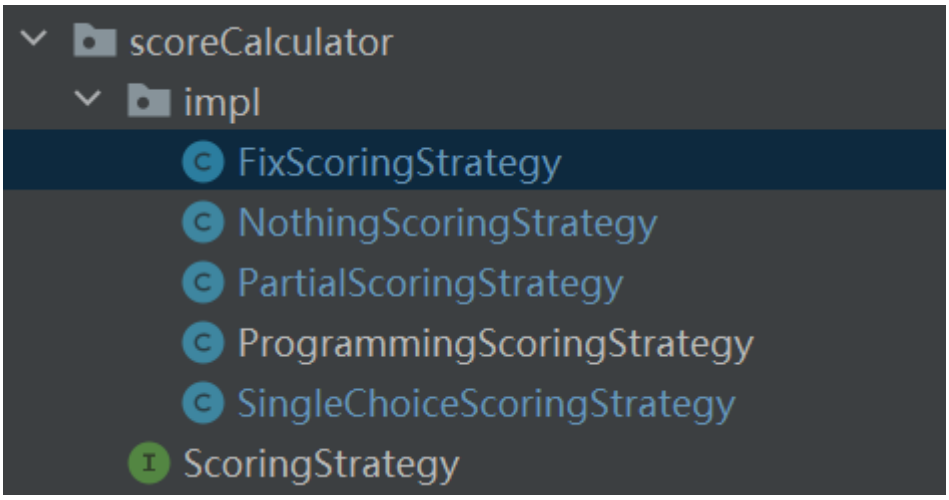
设计模式：

策略模式

定义接口

```
public interface ScoringStrategy {
    1 usage   5 implementations   211870287
    public int calculateQuestionScore(Object self_writtenAnswer);
}
```

实现不同的**计分策略**



再由对应的Question**组合复用**

```java
public abstract class Question<T> {

    1 usage
    private int id;
    1 usage
    private int type;
    1 usage
    private String question;
    1 usage
    private int points;

    private ScoringStrategy scoringStrategy;

    1 usage   5 overrides   211870287
    public void initStrategy(){}

    6 overrides   211870287
    @Override
    public String toString() {...}
}
```

简单工厂模式

设计读取exam的简单工厂

```java
public class ExamReaderFactory {
    2 usages   211870287
    public static ExamReader createExamReader(String fileExtension) {
        switch (fileExtension) {
            case "json":
                return new JsonExamReader();
            case "xml":
                return new XmlExamReader();
            default:
                throw new IllegalArgumentException("Unsupported file type: " + fileExtension);
        }
    }
}
```

根据不同的参数来获取不同的“产品”——对应的reader类

单例模式

```
public class ThreadPool {  7 usages  211870287 *
    private static final int threadNum = 5;  1 usage

    private static final ThreadPool instance = new ThreadPool(threadNum);  1 usage
    private final WorkerThread[] threads;  4 usages
    private final Queue<Runnable> taskQueue;  5 usages

    // 对外提供静态方法获取该对象
    public static ThreadPool getInstance() { return instance; }

    private ThreadPool(int threadCount) {...}

    public <T> Future<T> submit(Callable<T> task) {...}

    public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)  1 usage  211870287
            throws InterruptedException {...}

    public void shutdown() {...}
```

1. **ThreadPool 类**：
   - ThreadPool 类是一个单例类，使用了单例模式确保系统中只有一个线程池实例。
   - 它通过私有构造函数和静态方法 `getInstance()` 提供了对唯一实例的访问。
2. **线程池管理**：
   - 线程池中维护了一个固定数量的工作线程（`WorkerThread`），默认为 5 个。
   - 在构造函数中创建了这些工作线程，并启动它们。
3. **任务提交**：
   - 提供了 `submit(Callable<T> task)` 方法，用于提交一个可执行的任务（Callable）到线程池中执行。
   - 任务被添加到任务队列（`taskQueue`）中，然后被工作线程取出执行。
4. **任务执行**：
   - `WorkerThread` 类表示线程池中的工作线程，它从任务队列中获取任务并执行。
   - 当有任务提交时，工作线程会从任务队列中取出任务并执行。
5. **任务执行状态管理**：
   - 提供了 `invokeAll(Collection<? extends Callable<T>> tasks)` 方法，用于提交一批任务，并等待所有任务执行完成。
   - 方法会将所有任务提交到线程池执行，并返回一个包含任务执行结果的列表。
6. **线程池关闭**：
   - 提供了 `shutdown()` 方法，用于关闭线程池。
   - 调用 `shutdown()` 方法后，线程池会停止接受新任务，并等待当前执行的任务执行完成。
7. **线程安全**：
   - 线程池的方法使用了同步机制确保线程安全。
   - 通过对任务队列进行同步，线程池能够确保多个线程安全地提交和执行任务。

抽象工厂模式

```
public interface CodeHandlerFactory {  4 usages  1 implementation  211870287 *
    Compiler createCompiler();  1 usage  1 implementation  211870287
    Executor createExecutor();  1 usage  1 implementation  211870287
    int handleCode(String filePath, int point, List<SampleItem> sampleList, int timeLimit);
}
```

1. **接口定义**：
   - `CodeHandlerFactory` 是一个接口，它定义了一组用于创建编译器（Compiler）和执行器（Executor）对象的方法，以及处理代码的方法。
2. **创建方法**：
   - `createCompiler()` 方法用于创建编译器对象。
   - `createExecutor()` 方法用于创建执行器对象。
   - 这两个方法提供了对编译器和执行器的抽象，使得具体实现可以根据需要创建不同类型的编译器和执行器。
3. **处理代码方法**：
   - `handleCode(String filePath, int point, List<SampleItem> sampleList, int timeLimit)` 方法用于处理代码。
   - 它接受代码文件路径、分数、样本列表和时间限制等参数，并返回处理结果。
   - 这个方法是抽象工厂的核心方法，它将创建的编译器和执行器对象用于处理代码，然后返回处理结果。
4. **抽象性和灵活性**：
   - `CodeHandlerFactory` 接口提供了对编译器和执行器的抽象，使得客户端代码可以针对接口编程，而不是具体的实现类。
   - 这种抽象性和灵活性使得系统更易于扩展和维护，可以轻松地替换或添加新的编译器和执行器实现，而不需要修改客户端代码。

接口设计

```
public interface AnswerReader {
    public Answer readAnswer(String filePath) throws IOException;
    public Answer readAnswer(File jsonFile) throws IOException;
}
```

```
public interface ExamReader {
    Exam readExam(String filePath) throws IOException;
}
```

```
public interface ScoreWriter {
    void writeScore(String filePath) throws IOException;
}
```

```
public interface ScoringStrategy {
    public int calculateQuestionScore(Object self_writtenAnswer);
}
```

```
public interface Executor {
    String execute(String compiledFilePath, String[] args);
}
```

```
public interface Compiler {
    boolean compile(String filePath);
    String getCompileDirectory();
}
```

```
public interface CodeHandlerFactory {
    Compiler createCompiler();
    Executor createExecutor();
    int handleCode(String filePath, int point, List<SampleItem> sampleList, int
timeLimit);
}
```

抽象类设计

```
public abstract class Question<T> {

    private int id;
    private int type;
    private String question;
    private int points;

    private ScoringStrategy scoringStrategy;

    public void initStrategy(){}

    @Override
    public String toString() {
        return "Question{" +
                "id=" + id +
                ", type=" + type +
                ", question='" + question + '\'' +
                ", points=" + points +
                '}';
    }
}
```

实体设计

```
public class Answer {
    @JsonProperty("examId")
    private int examId;

    @JsonProperty("stuId")
```

```
    private int studentId;

    @JsonProperty("submitTime")
    private long submitTime;

    @JsonProperty("answers")
    private List<AnswerItem> answers;

    @Override
    public String toString() {
        return "Answer{" +
                "\nexamId=" + examId +
                "\nstuId=" + studentId +
                "\nsubmitTime=" + submitTime +
                "\nanswers=" + answers +
                "\n}";
    }

}
```

```
public class AnswerItem {
    private int id;
    private String answer;
}
```

```
public class Exam {
    private int id;
    private String title;
    private long startTime;
    private long endTime;

    private List<Question> questions;

    @Override
    public String toString() {
        return "Exam{\n" +
                "id=" + id +
                "\ntitle='" + title + '\'' +
                "\nstartTime=" + startTime +
                "\nendTime=" + endTime +
                "\nquestions=\n" + questions +
                '}';
    }
}
```

```
public class ExamScore {
    @JsonProperty(index = 0)
    private int examId;
```

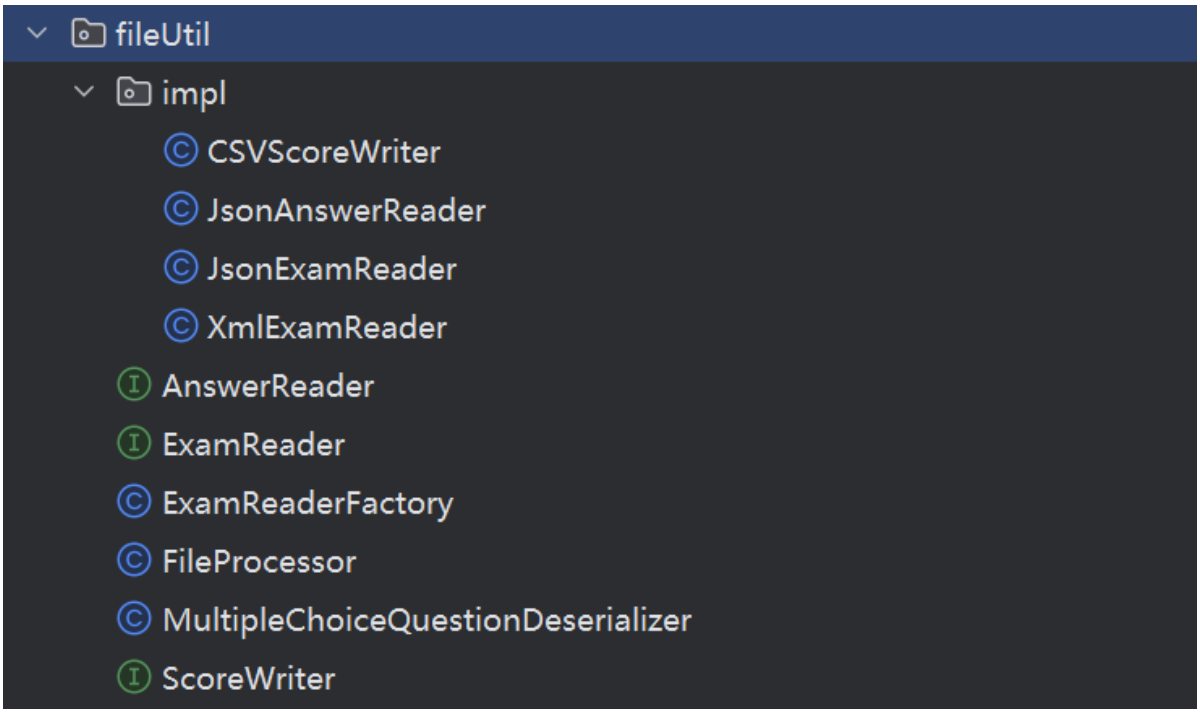
```
    @JsonProperty(index = 1)
    private int stuId;
    @JsonProperty(index = 2)
    private int score;

    // 构造函数
    public ExamScore(int examId, int stuId, int score) {
        this.examId = examId;
        this.stuId = stuId;
        this.score = score;
    }

    @Override
    public String toString() {
        return "ExamScore{" +
                "examId='" + examId + '\'' +
                ", stuId='" + stuId + '\'' +
                ", score=" + score +
                '}';
    }
}
```

职责划分

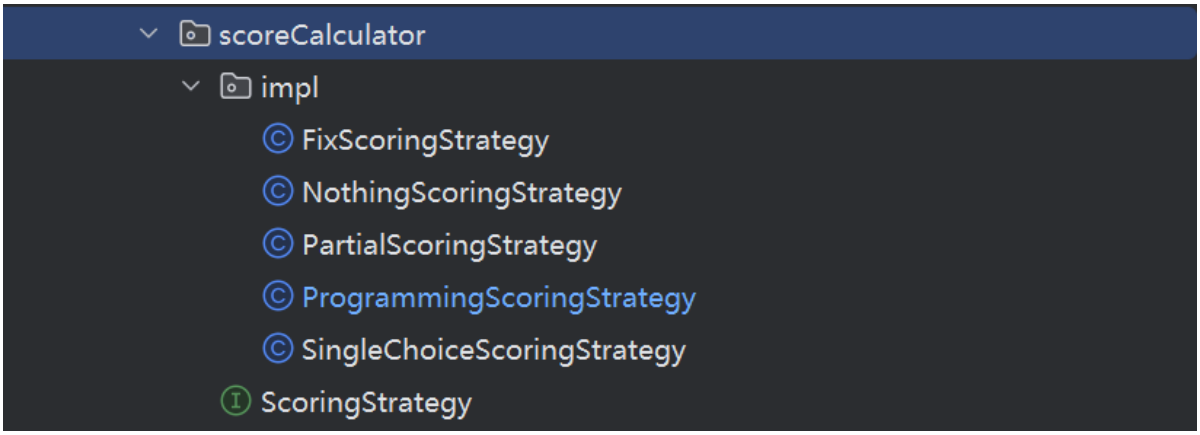
codeHandler 包用于处理代码，包括对代码预处理（编译）、执行

entity 包用于存储实体的信息，如exam、Answer。保存了多张表。

fileUtil 包用于对文件的IO处理，如读取xml、json文件，并实例化到entity对应的实体中。



scoreCalculator 包是用于对于不同策略的给分模式



threadPool 包用于多线程的处理

功能演示

2.1. 整体流程

- 创建编译任务：使用给定的编译器对代码进行编译。
- 创建执行任务列表：为每个样本创建执行任务，其中包括编译后的文件的绝对路径和样本程序参数。
- 等待编译任务完成：等待编译任务在线程池中执行完成。
- 执行任务并获取结果：执行任务列表中的每个任务，并获取执行结果。
- 比较执行结果：将执行结果与样本输出进行比较，如果与样本输出不一致，则返回得分为 0。

- 返回得分：如果所有任务执行成功且输出与样本一致，则返回给定的分数。

```
try {
    // 创建编译任务
    Callable<Boolean> compileTask = () -> compiler.compile(fileAbsolutePath);
    Future<Boolean> compileFuture = threadPool.submit(compileTask);

    // 创建执行任务列表
    List<Callable<String>> executeTasks = new ArrayList<>();
    for (SampleItem sampleItem : sampleList) {
        // 创建编译后的文件的绝对路径
        String absoluteFilePath = compileDirectory + fileName;
        // 创建执行任务
        Callable<String> executeTask = () -> executor.execute(absoluteFilePath, sampleItem.getProgramArgs());
        executeTasks.add(executeTask);
    }

    // 等待编译任务完成
    if (compileFuture.get()) {
        // 执行任务并获取结果
        List<Future<String>> futures = threadPool.invokeAll(executeTasks);

        int len = futures.size();
        for (int i = 0; i < len; i++) {
            if (! futures.get(i).get().equals(sampleList.get(i).getOutput())){
                return 0;
            }
        }

        return point;
    }
    return 0;
```

2.2. 代码（编程题作答结果）的预处理

编译过程：

1. **构建编译命令**：创建一个 ProcessBuilder 实例，并传入 javac 命令及相关参数，如编译目标目录和源文件路径。并保存于filePath路径
2. **启动进程**：通过 ProcessBuilder 的 start() 方法启动外部进程，并执行编译命令。
3. **等待编译完成**：调用进程的 waitFor() 方法等待编译过程完成，并获取退出代码（exit code）。
4. **处理编译结果**：根据退出代码判断编译是否成功，如果退出代码为 0 表示编译成功，否则表示编译失败。

异常处理：

- 如果在编译过程中出现 IOException 或 InterruptedException 异常，则会捕获异常并处理。
- 在异常处理过程中，会打印异常信息，并返回 false 表示编译失败。

```java
public boolean compile(String filePath) {
      System.out.println("正在编译.."  + filePath);
    try {
        // 构建编译命令
        ProcessBuilder processBuilder = new ProcessBuilder( ...command: "javac", "-d", compileDirectory, filePath);
          processBuilder.directory(new File(filePath).getParentFile()); // 设置工作目录为 Java 文件所在目录

        // 启动进程并等待其完成
        Process process = processBuilder.start();

        int exitCode = process.waitFor();
        if ( exitCode == 0){
              System.out.println("编译成功");
            return true;
        }
          System.out.println("编译结果exitCode: " + exitCode);
        // 返回编译结果
        return false;
    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
        return false; // 返回 -1 表示编译过程中出现异常
    }
}
```

2.3. 代码（编程题作答结果）的执行

- 接收编译后的 Java 类文件路径和运行参数。
- 构建执行命令，并通过外部进程执行 Java 类文件。
- 读取进程的标准输出流，并将输出结果返回。

执行过程：

1. **构建执行命令**：根据给定的类文件路径和参数，构建执行命令列表。
2. **构建进程**：使用 `ProcessBuilder` 类构建外部进程，并指定命令列表。
3. **启动进程**：启动外部进程，并将执行结果输出到标准输出流。
4. **等待执行完成**：等待外部进程执行完成，并获取退出代码。
5. **处理执行结果**：如果执行成功（退出代码为 0），则返回执行结果；否则，抛出异常。

异常处理：

- 如果在执行过程中出现 IOException 或 InterruptedException 异常，则捕获异常并抛出运行时异常。
- 如果外部进程的退出代码不为 0，则抛出运行时异常。

```
    public String execute(String compiledFilePath, String[] args) {
//        System.out.println("执行中");
        try {
            // 获取最后一个斜杠的索引位置
            int lastIndex = compiledFilePath.lastIndexOf( str: "/");

            // 获取父目录路径（src/test/resources/cases/answers/bin）
            String directoryPath = compiledFilePath.substring(0, lastIndex);
//            System.out.println("directoryPath: "+directoryPath);

            String className = compiledFilePath.substring(lastIndex + 1, compiledFilePath.lastIndexOf( str: "."));
//            System.out.println("className: "+className);
            // 构建执行命令
            List<String> command = new ArrayList<>();
            command.add("java");
            command.add(className);
            command.addAll(Arrays.asList(args));

            // 构建进程
            ProcessBuilder processBuilder = new ProcessBuilder(command);
            processBuilder.directory(new java.io.File(directoryPath));

            // 启动进程并等待其完成
            Process process = processBuilder.start();

            // 读取命令输出
            BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));
            StringBuilder re = new StringBuilder();
            String line;
            while ((line = reader.readLine()) != null) {
                re.append(line);
//                System.out.println(line);
            }
```

2.4. 并发需求

1. 固定的、有限个的线程创建，为了统一，我们指定线程数量为5。在线程池创建之初就直接创建好固

定的线程并放在线程池之中。

```
public class ThreadPool {
    private static final int threadNum = 5;

    private static final ThreadPool instance = new ThreadPool(threadNum);
    private final WorkerThread[] threads;
    private final Queue<Runnable> taskQueue;

    // 对外提供静态方法获取该对象
    public static ThreadPool getInstance() {
        return instance;
    }

    private ThreadPool(int threadCount) {
        threads = new WorkerThread[threadCount];
        taskQueue = new LinkedList<>();

        for (int i = 0; i < threadCount; i++) {
            threads[i] = new WorkerThread(taskQueue);
            threads[i].start();
        }
    }

    public <T> Future<T> submit(Callable<T> task) {
```

```
        FutureTask<T> futureTask = new FutureTask<>(task);
        synchronized (taskQueue) {
            taskQueue.add(futureTask);
            taskQueue.notify();
        }
        return futureTask;
    }

    public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
            throws InterruptedException {
        if (tasks == null)
            throw new NullPointerException();
        ArrayList<Future<T>> futures = new ArrayList<>(tasks.size());
        boolean done = false;
        try {
            for (Callable<T> t : tasks) {
                Future<T> f = submit(t);
                futures.add(f);
            }
            for (int i = 0, size = futures.size(); i < size; i++) {
                Future<T> f = futures.get(i);
                if (!f.isDone()) {
                    try {
                        f.get();
                    } catch (CancellationException | ExecutionException ignore) {
                    }
                }
            }
            done = true;
            return futures;
        } finally {
            if (!done)
                for (int i = 0, size = futures.size(); i < size; i++)
                    futures.get(i).cancel(true);
        }
    }

    public void shutdown() {
        for (WorkerThread thread : threads) {
            thread.shutdown();
        }
    }
}
```

2. 自动的任务调度。FIFO调度，采用任务队列，先进先出

```
private final Queue<Runnable> taskQueue;
```

```
public void run() {
    while (running) {
        Runnable task;
        synchronized (taskQueue) {
            while (taskQueue.isEmpty()) {
                try {
```

```
                    taskQueue.wait(); // Wait for task
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    return;
                }
            }
            task = taskQueue.poll();//从队列中取出并移除队首的元素
        }
        try {
            task.run(); // Execute task
        } catch (RuntimeException e) {
            System.out.println("Thread pool is interrupted due to an issue: " +
e.getMessage());
        }
    }
}
```

3. 线程池和任务类型需要是解耦的，也就是说，一个线程池类可以执行任意种任务，在本次迭代中，需要同时支持代码的预处理和执行。

```
public <T> Future<T> submit(Callable<T> task) {  2 usages   211870287
    FutureTask<T> futureTask = new FutureTask<>(task);
    synchronized (taskQueue) {
        taskQueue.add(futureTask);
        taskQueue.notify();
    }
    return futureTask;
}
```

2.5 评分

编程题需要测试用例全部通过，任一一个测试用例不过均为0分。

```
// 等待编译任务完成
if (compileFuture.get()) {
    // 执行任务并获取结果
    List<Future<String>> futures = threadPool.invokeAll(executeTasks);

    int len = futures.size();
    for (int i = 0; i < len; i++) {
        if (! futures.get(i).get().equals(sampleList.get(i).getOutput())){
            return 0;
        }
    }

    return point;
}
```