

迭代二设计文档

首先明晰一下迭代二的功能需求：编程题作答结果的预处理、执行和并发需求，我就依据流程来进行设计了。

整体设计

设计类图：

Score接口设计如下:

```

public interface Score {
    Integer compileCode(String outputPath, String sourcePath);
    Integer executeCode(String classFilePath, ProgrammingQuestion question, String outputFilePa
    Integer score(ProgrammingQuestion programmingQuestion, String outputFilePa
}

```

三个模块分别负责预处理、执行和编程题评分。采取单一职责的设计思想，将功能解耦。

由于多处需要使用文件路径，通过参数传递的方式方法与方法之间耦合度极高，故抽离出一个类存放常量：

```

public class Constant {
    private static String examsPath;
    private static String answersPath;
    private static String outputPath;

    public static void setExamsPath(String exam) {
        examsPath = exam;
    }

    public static void setAnswerPath(String answer) {
        answersPath = answer;
    }

    public static void setOutputPath(String output) {
        outputPath = output;
    }

    public static String getExamsPath() {
        return examsPath;
    }

    public static String getAnswerPath() {
        return answersPath;
    }

    public static String getOutputPath() {
        return outputPath;
    }
}

```

代码的预处理

还是使用一个简单工厂来生产具体的实现类：

```
CodeFactory codeFactory = new CodeFactory("java");  
return codeFactory.getScore((ProgrammingQuestion) question, answer);
```

Java预处理相关代码：

```
public Integer compileCode(String outputPath, String sourcePath) {  
    Future<Integer> future = threadPool.submit(new CompileTaskImpl(outputPath, sourcePath));  
    try {  
        return future.get();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    return 0;  
}
```

通过将下面的任务放入线程池进行具体的任务执行。将具体任务的实现类与实体类抽离开来，易于进行不同任务的替换装配。

```

public class CompileTaskImpl implements Callable<Integer> {

    private final String outputPath;
    private final String sourcePath;

    public CompileTaskImpl(String outputPath, String sourcePath) {
        this.outputPath = outputPath;
        this.sourcePath = sourcePath;
    }

    @Override
    public Integer call() throws Exception {
        try {
            // 调用命令行命令编译Java代码
            Process process = Runtime.getRuntime().exec("javac -d " + outputPath + " " + sourcePath);
            return process.waitFor();
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
            return -1;
        }
    }
}

```

由于后面需要支持任意中任务的线程池，故将其设计继承自 `Callable<Integer>`

代码的执行

```

List<Future<Integer>> futures = new ArrayList<>();

for (Sample sample : samples) {
    Future<Integer> future = threadPool.submit(new ExecuteTaskImpl(classFilePath, sample));
    futures.add(future);
}

```

具体实际线程实现：

```

public Integer call() {
    try {
        // 执行程序并将输出重定向到文件
        String[] inputArgs = sample.getInput().split(" ");
        List<String> commandList = new ArrayList<>();
        commandList.add("java");
        commandList.add("-cp");
        commandList.add(Constant.getAnswerPath() + System.getProperty("file.separator") + "c
        commandList.add(classFilePath);
        commandList.addAll(Arrays.asList(inputArgs));

        ProcessBuilder processBuilder = new ProcessBuilder(commandList);
        processBuilder.redirectOutput((ProcessBuilder.Redirect.appendTo(new File(outputFile)
        Process process = processBuilder.start();
        int exitCode = process.waitFor();
        if (exitCode != 0) {
            // 运行出错, 返回0分
            // 获取错误输出流
            InputStream errorStream = process.getErrorStream();
            BufferedReader errorReader = new BufferedReader(new InputStreamReader(errorStre
            String errorLine;
            StringBuilder errorOutput = new StringBuilder();
            // 读取错误输出
            while ((errorLine = errorReader.readLine()) != null) {
                errorOutput.append(errorLine).append("\n");
            }
            // 输出错误信息
            System.out.println("命令行运行报错信息: " + errorOutput);
            return 0;
        }
    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    }
    return 1;
}

```

将结果重定向到对应文件。

并发需求

简单线程池的实现, 为了支持任务拥有返回值, 使用Callable, 一个简单线程池如下:

```

public class ThreadPool {

    private static final int poolSize = 5;
    private final Worker[] workers;
    private final BlockingQueue<FutureTask<Integer>> taskQueue;

    public ThreadPool() {
        this.taskQueue = new LinkedBlockingQueue<>();
        this.workers = new Worker[poolSize];

        for (int i = 0; i < poolSize; i++) {
            workers[i] = new Worker();
            workers[i].start();
        }
    }

    public Future<Integer> submit(Callable<Integer> task) {
        FutureTask<Integer> futureTask = new FutureTask<>(task);
        try {
            taskQueue.put(futureTask);
        } catch (InterruptedException e) {
            // 处理任务添加异常
            e.printStackTrace();
        }
        return futureTask;
    }

    private class Worker extends Thread {
        public void run() {
            while (true) {
                try {
                    FutureTask<Integer> futureTask = taskQueue.take();
                    futureTask.run();
                } catch (InterruptedException e) {
                    // 处理任务执行异常
                    e.printStackTrace();
                }
            }
        }
    }
}

```

由于编程题只有一道，故而我本人虽然觉得预处理编译时多线程意义不大，但执行代码时，采用多线程同时评测多个测试用例确实效率得到提升，整体评测时间由10s左右减少到6s左右。