

## 《OpenAI 代码自动评审组件》- 第7节-学习记录

来自：码农会锁



爱海贼的无处不在

2024年08月17日 12:54

学习系列文章目录：

- 1、《OpenAI 代码自动评审组件》- 第1节-学习记录: [https://articles.zsxq.com/id\\_bpapuj05ur03.html](https://articles.zsxq.com/id_bpapuj05ur03.html)
- 2、《OpenAI 代码自动评审组件》- 第2节-学习记录: [https://articles.zsxq.com/id\\_uokpg7mvt1uc.html](https://articles.zsxq.com/id_uokpg7mvt1uc.html)
- 3、《OpenAI 代码自动评审组件》- 第3节-学习记录: [https://articles.zsxq.com/id\\_bi51vtdh36.html](https://articles.zsxq.com/id_bi51vtdh36.html)
- 4、《OpenAI 代码自动评审组件》- 第4节-学习记录: [https://articles.zsxq.com/id\\_mw3srnic84z7.html](https://articles.zsxq.com/id_mw3srnic84z7.html)
- 5、《OpenAI 代码自动评审组件》- 第5节-学习记录: [https://articles.zsxq.com/id\\_s55ppt4hbsmr.html](https://articles.zsxq.com/id_s55ppt4hbsmr.html)
- 6、《OpenAI 代码自动评审组件》- 第6节-学习记录: [https://articles.zsxq.com/id\\_pfn3s3r86fa0.html](https://articles.zsxq.com/id_pfn3s3r86fa0.html)

### 一、概述

本节作者主要讲解AI代码评审后，开始对现有逻辑的重构处理，包括了文件职责的拆分，代码获取的拆分设计、发送评审和网络请求的拆分，结果解析与处理拆分，最终基于模板方法模式实现了基于DDD的代码评审组件的结构设计优化。

### 二、正文

1、首先第一个改变就是Github Action工作流配置文件的改变，可以看到博主增加了新的节点内容：

```
- name: Copy openai-code-review-sdk jar
  run: mvn dependency:copy -Dartifact=cn.aijavapro:openai-code-review-sdk:1.0 -DoutputDirectory=./libs

- name: Get repository name
  id: repo-name
  run: echo "REPO_NAME=${GITHUB_REPOSITORY##*/}" >> $GITHUB_ENV

- name: Get branch name
  id: branch-name
  run: echo "BRANCH_NAME=${GITHUB_REF#refs/heads/}" >> $GITHUB_ENV

- name: Get commit author
  id: commit-author
  run: echo "COMMIT_AUTHOR=$(git log -1 --pretty=format:'%an <%ae>')" >> $GITHUB_ENV

- name: Get commit message
  id: commit-message
  run: echo "COMMIT_MESSAGE=$(git log -1 --pretty=format:'%s')" >> $GITHUB_ENV
- name: Print repository, branch name, commit author, and commit message
  run: |
    echo "Repository name is ${env.REPO_NAME}"
    echo "Branch name is ${env.BRANCH_NAME}"
    echo "Commit author is ${env.COMMIT_AUTHOR}"
    echo "Commit message is ${env.COMMIT_MESSAGE}"

- name: Run code review
  run: java -jar ./libs/openai-code-review-sdk-1.0.jar
  env:
    GITHUB_TOKEN: ${secrets.GITHUB_TOKEN}
    CHATGPT_KEY: ${secrets.CHATGPT_KEY}
    GIT_USER_TOKEN: ${secrets.GIT_USER_TOKEN}
    GITHUB_REVIEW_LOG_URI: ${secrets.CODE_REVIEW_LOG_URI}
    COMMIT_PROJECT: ${env.REPO_NAME}
    COMMIT_BRANCH: ${env.BRANCH_NAME}
    COMMIT_AUTHOR: ${env.COMMIT_AUTHOR}
    COMMIT_MESSAGE: ${env.COMMIT_MESSAGE}
    # 微信配置 「https://mp.weixin.qq.com/debug/cgi-bin/sandboxinfo?action=showinfo&t=sandbox/index」
    WEIXIN_APPID: ${secrets.WEIXIN_APPID}
    WEIXIN_SECRET: ${secrets.WEIXIN_SECRET}
    WEIXIN_TOUSER: ${secrets.WEIXIN_TOUSER}
    WEIXIN_TEMPLATE_ID: ${secrets.WEIXIN_TEMPLATE_ID}
    # OpenAI - ChatGLM 配置 「https://open.bigmodel.cn/api/paas/v4/chat/completions」、 「https://open.bigmodel.cn/usercenter/apikeys」
    CHATGLM_APIHOST: ${secrets.CHATGLM_APIHOST}
    CHATGLM_APIKEYSECRET: ${secrets.CHATGLM_APIKEYSECRET}
```

这段配置的含义如下：

#### 1. Checkout repository

- 使用actions/checkout@v2动作来检出代码库。这个动作负责从GitHub仓库中检出代码，使得后续的步骤可以在本地执行。
- with:部分设置了额外的参数：
  - fetch-depth: 2: 这个参数用于控制Git检出时保留的提交历史记录深度。这里设置为2，意味着只会下载最近的两个提交，这对于快速检出代码分支非常有用。

#### 2. Set up JDK 11

- 使用actions/setup-java@v2动作来设置Java开发环境。
- with:部分设置了额外的参数：
  - distribution: 'adopt': 指定使用AdoptOpenJDK作为Java的发行版。
  - java-version: '11': 指定使用Java 11版本。

#### 3. Build with Maven



- 运行mvn clean install命令来使用Maven构建项目。
- clean是Maven的生命周期阶段，用于清理之前构建的文件。
- install是Maven的生命周期阶段，用于将项目安装到本地Maven仓库。

4. Copy openai-code-review-sdk jar

- 使用Maven命令mvn dependency:copy来复制特定的jar包到libs目录。
- -Dartifact=cn.aijavapro:openai-code-review-sdk:1.0指定了要复制的jar包的坐标。
- -DoutputDirectory=.libs指定了复制后的jar包应该放置的目录。

5. Get repository name

- 使用shell命令将仓库名称添加到环境变量中，以便后续步骤可以使用。
- GITHUB\_REPOSITORY是GitHub提供的环境变量，包含了仓库名称。
- echo "REPO\_NAME=\${GITHUB\_REPOSITORY##\*/}" >> \$GITHUB\_ENV将仓库名称提取出来，并将其存储在GITHUB\_ENV环境变量中。

6. Get branch name

- 使用shell命令将分支名称添加到环境变量中。
- GITHUB\_REF是GitHub提供的环境变量，包含了分支的完整引用。
- echo "BRANCH\_NAME=\${GITHUB\_REF#refs/heads/}" >> \$GITHUB\_ENV从GITHUB\_REF中提取分支名称，并将其存储在BRANCH\_NAME环境变量中。

7. Get commit author

- 使用shell命令获取最新的提交作者信息，并将其添加到环境变量中。
- git log -1 --pretty=format:'%an <%ae>'命令用于获取最后一次提交的作者姓名和电子邮件地址。
- echo "COMMIT\_AUTHOR=\$(git log -1 --pretty=format:'%an <%ae>')" >> \$GITHUB\_ENV将作者信息存储在COMMIT\_AUTHOR环境变量中。

8. Get commit message

- 使用shell命令获取最新的提交信息，并将其添加到环境变量中。
- git log -1 --pretty=format:'%s'命令用于获取最后一次提交的消息。
- echo "COMMIT\_MESSAGE=\$(git log -1 --pretty=format:'%s')" >> \$GITHUB\_ENV将提交信息存储在COMMIT\_MESSAGE环境变量中。

9. Print repository, branch name, commit author, and commit message

- 使用shell命令打印上面获取到的环境变量信息，以便于查看。


























10. Run code review

- 运行一个Java命令来执行代码审查。
- 使用java -jar .libs/openai-code-review-sdk-1.0.jar命令执行一个名为openai-code-review-sdk的jar文件。
- env:部分设置了多个环境变量，这些变量包含了执行代码审查所需的各种凭证和配置信息，如GitHub令牌、微信配置、OpenAI ChatGLM配置等。这些变量通过GitHub的secrets机制进行安全存储。

2、既然这里有了这些变量，那我们就需要Github中添加上这些内容

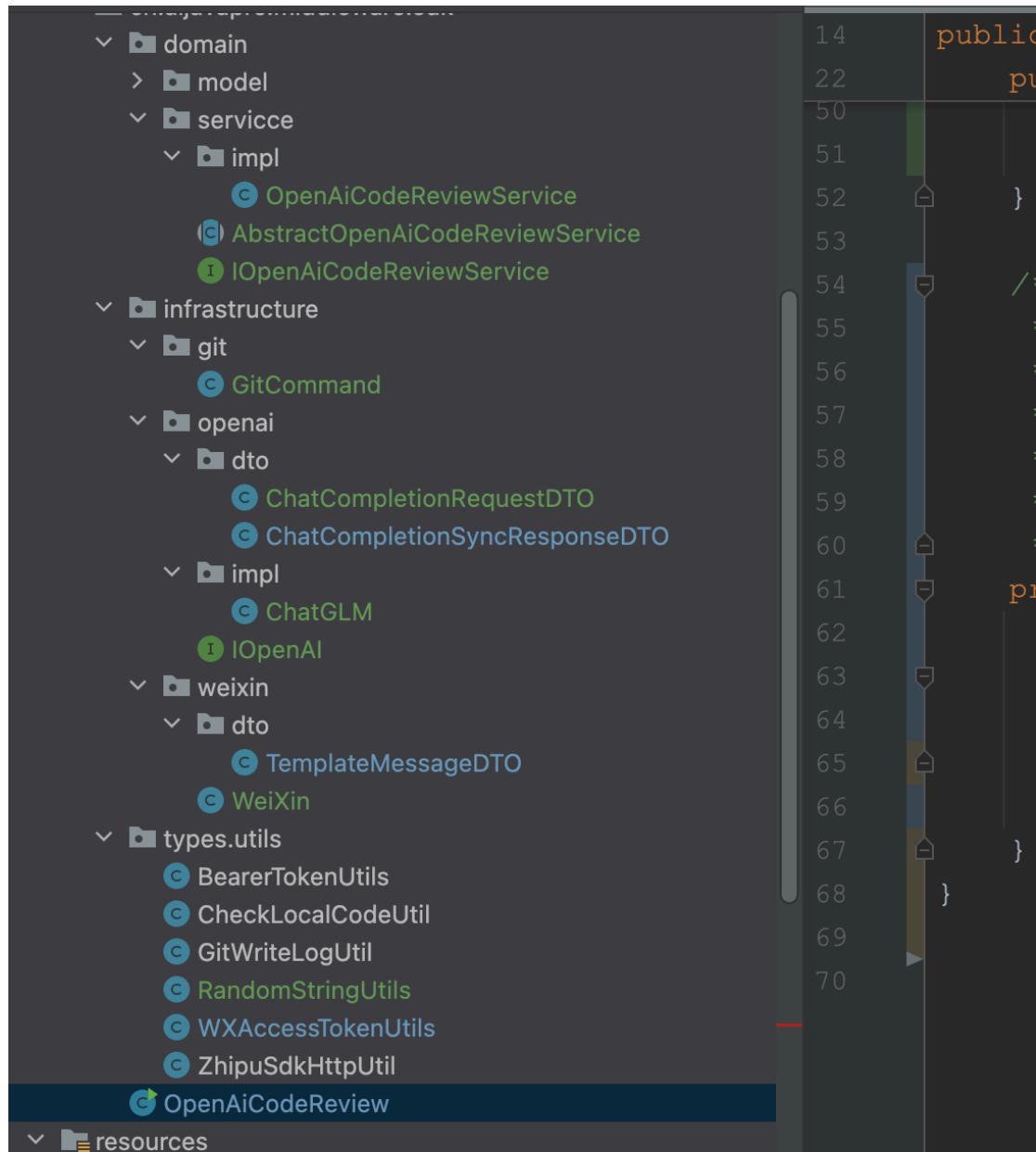
Repository secrets

New repository secret

| Name                  | Last updated  |   |
|--|---------------|---|
|  CHATGLM_APIHOST      | 3 minutes ago |   |
|  CHATGLM_APIKEYSECRET | 4 minutes ago |   |
|  GIT_USER_TOKEN       | 2 weeks ago   |   |
|  REVIEW_LOG_URI       | now           |   |
|  WEIXIN_APPID         | 2 minutes ago |   |
|  WEIXIN_SECRET        | 2 minutes ago |   |
|  WEIXIN_TEMPLATE_ID   | 1 minute ago  |   |
|  WEIXIN_TOUSER        | 1 minute ago  |   |

3、接下来在看看作者的工程架构最新的样子，针对当前组件中的基础设施层，增加了GIT、OPENAI、WEIXIN这三个基础设施模块，分别维护了GIT的操作、AI的操作、微信的操作，结构清晰，同时在一定程度上做到了第三方与项目依

赖的解耦。



首先来看看GitCommand这个类，这个核心的Diff方法的处理逻辑是值得学习的，对于我们是非常有帮助的，也帮助我们了解下Java操作系统进程的API，和Git Diff命令：

```
/**
 * 获取两个提交之间的差异。
 * 此方法通过调用Git命令行工具，获取当前分支上最新提交（HEAD）与它之前的一次提交之间的差异。
 * 差异结果将以文本形式返回。
 *
 * <p>处理流程如下：</p>
 * <ol>
 *   <li>创建一个ProcessBuilder实例来执行"git log"命令，获取最新提交的哈希值。</li>
 *   <li>启动进程，并读取输出流以获取最新提交的哈希值。</li>
 *   <li>创建另一个ProcessBuilder实例来执行"git diff"命令，比较最新提交与它之前的一次提交。</li>
 *   <li>启动进程，并读取输出流以获取差异内容。</li>
 *   <li>如果"git diff"命令执行失败（即进程退出码非零），则抛出运行时异常。</li>
 *   <li>返回获取的差异内容字符串。</li>
 * </ol>
 *
 * @return 包含当前分支最新提交与之前一次提交之间差异的文本内容。
 * @throws IOException 如果在执行Git命令或读取输出时发生I/O错误。
 * @throws InterruptedException 如果当前线程在执行Git命令时被中断。
 */
public String diff() throws IOException, InterruptedException {
    // 创建ProcessBuilder实例以获取最新提交的哈希值
    ProcessBuilder logProcessBuilder = new ProcessBuilder("git", "log", "-1", "--pretty=format:%H");
    logProcessBuilder.directory(new File("."));
    Process logProcess = logProcessBuilder.start();
    // 读取最新提交的哈希值
    BufferedReader logReader = new BufferedReader(new InputStreamReader(logProcess.getInputStream()));
    String latestCommitHash = logReader.readLine();
    logReader.close();
    logProcess.waitFor(); // 等待进程结束
    // 创建ProcessBuilder实例以获取两个提交之间的差异
    ProcessBuilder diffProcessBuilder = new ProcessBuilder("git", "diff", latestCommitHash + "^", latestCommitHash);
    diffProcessBuilder.directory(new File("."));
    Process diffProcess = diffProcessBuilder.start();
    // 读取差异内容
    StringBuilder diffCode = new StringBuilder();
    BufferedReader diffReader = new BufferedReader(new InputStreamReader(diffProcess.getInputStream()));
    String line;
    while ((line = diffReader.readLine()) != null) {
        diffCode.append(line).append("\n");
    }
    diffReader.close();
    diffProcess.waitFor(); // 等待进程结束
    // 检查是否有错误发生
    int exitCode = diffProcess.waitFor();
    if (exitCode != 0) {
```

这里面我们可以学到一个命令如下：

```
git log -1 --pretty=format:%H
```

这个命令的含义如下：

命令 `git log -1 --pretty=format:%H` 是Git版本控制系统中用来获取最近一次提交的哈希值的命令。下面是各个部分的含义：

- `git log`: 这是Git命令行工具中的一个基本命令，用于显示提交历史记录。
- `-1`: 这是一个参数，用于指定要显示的提交历史记录的条目数量。在这里，`-1` 意味着只显示最近的一次提交。
- `--pretty=format:`: 这是一个选项，用于控制提交信息的格式。`format:` 后面跟着的是一个格式化字符串。
- `%H`: 这是一个格式化占位符，表示提交的哈希值。

所以，这个命令的整体作用是：

- 使用 `git log` 获取提交历史。
- `-1` 限制输出只显示一条记录，即最新的一次提交。
- `--pretty=format:%H` 指示Git使用 `%H` 占位符来格式化输出，只输出每个提交的哈希值。

在执行这个命令后，你将得到一个唯一的字符串，这个字符串代表Git仓库中最近一次提交的标识。哈希值通常是一长串字母和数字的组合，例如 `e4759c3dfb2f3fd299d9b31a9e0a605f0b1b6d95`。这个哈希值在Git中用于唯一标识每个提交。

这里面我们可以学到第二个命令如下：

```
git diff e4759c3dfb2f3fd299d9b31a9e0a605f0b1b6d95^ e4759c3dfb2f3fd299d9b31a9e0a605f0b1b6d95
```

这个含义如下：

命令 `git diff e4759c3dfb2f3fd299d9b31a9e0a605f0b1b6d95^ e4759c3dfb2f3fd299d9b31a9e0a605f0b1b6d95` 是用来比较两个提交之间的差异的Git命令。以下是命令中各个部分的含义：

- `git diff`: 这是Git的一个命令，用于显示两个或多个提交、分支或文件之间的差异。
- `e4759c3dfb2f3fd299d9b31a9e0a605f0b1b6d95`: 这是第一个提交的哈希值，表示你想要查看其差异的提交。
- `^`: 在Git中，`^` 符号用于引用当前分支的最近一次提交。如果前面有数字，它会引用该分支的第N个最近提交。在这个命令中，`^` 指的是最新提交。
- `e4759c3dfb2f3fd299d9b31a9e0a605f0b1b6d95`: 这是第二个提交的哈希值，也是你想要查看其差异的提交。

结合以上信息，这个命令的作用如下：

- 首先比较第一个提交 `e4759c3dfb2f3fd299d9b31a9e0a605f0b1b6d95` 和它的上一个提交（由 `^` 表示）之间的差异。
- 因为 `e4759c3dfb2f3fd299d9b31a9e0a605f0b1b6d95` 本身就是最新的提交，所以这个命令实际上是在比较最新的提交和它的前一个提交之间的差异。

这个命令通常用于查看最近一次提交引入了哪些更改，因为它会显示从上一个提交到当前提交的所有变更。如果命令行工具中存在名为 e4759c3dfb2f3fd299d9b31a9e0a605f0b1b6d95 的提交，那么命令会显示从该提交的前一个提交到该提交的所有差异。如果该提交是当前HEAD（即最新提交），则显示从它之前的一次提交到当前提交的差异。

然后针对领域层的IOpenAiCodeReviewService类，作者通过抽象类和模板方法模式完成了代码设计，这个为未来自己扩展和开发提供了扩展口，抽象类的设计在工作中非常重要，很多常年写CRUD的小伙子，几乎没有用过抽象类，一旦你开始用了抽象类，那么你的设计能力就要开始提升了。

4、我们再来仔细看下这个抽象类的核心主流程，这是一个非常清晰的执行步骤：

```
*/
public abstract class AbstractOpenAiCodeReviewService implements IOpenAiCodeReviewService {

    // 日志记录器，用于记录操作日志
    private final Logger logger = LoggerFactory.getLogger(AbstractOpenAiCodeReviewService.class);

    // GitCommand实例，用于执行Git命令以获取代码差异
    protected final GitCommand gitCommand;
    // IOpenAI接口的实现，用于与OpenAI API交互
    protected final IOpenAI openAI;
    // WeiXin实例，用于发送微信通知
    protected final WeiXin weiXin;

    /**
     * 构造函数，用于创建AbstractOpenAiCodeReviewService实例。
     */
    /**
     * @param gitCommand GitCommand实例
     * @param openAI IOpenAI接口的实现
     * @param weiXin WeiXin实例
     */
    public AbstractOpenAiCodeReviewService(GitCommand gitCommand, IOpenAI openAI, WeiXin weiXin) {
        this.gitCommand = gitCommand;
        this.openAI = openAI;
        this.weiXin = weiXin;
    }

    /**
     * 执行代码评审的抽象方法。
     * 子类需要实现这个方法来具体执行代码评审流程。
     */
    @Override
    public void exec() {
        try {
            // 1. 获取提交代码的差异
            String diffCode = getDiffCode();
            // 2. 对获取的差异代码进行评审
            String recommend = codeReview(diffCode);
            // 3. 记录评审结果，并返回日志地址
            String logUrl = recordCodeReview(recommend);
            // 4. 通过微信发送消息通知，包含日志地址和通知内容
            pushMessage(logUrl);
        } catch (Exception e) {
            // 如果在执行过程中发生异常，记录错误日志
            logger.error("openai-code-review error", e);
        }
    }
}
```

先执行获取代码的差异，然后进行代码评审，然后返回结果，然后发送消息通知。这里面其实还隐藏这其他方面的设计思想，即：

#### Pipeline流水线设计思想/Pipeline设计模式

Pipeline翻译过来就是水管的意思，Pipeline设计模式其实很简单，就像是我们常用的CI/CD的Pipeline一样，一个环节做一件事情，最终串联成一个完整的Pipeline，Pipeline的优势在于通过配置化来灵活地实现不同的业务走不同的流程。实现统一化和差异化的完美结合。而当前这个组件的场景就是一个CI/CD的流水线，就是一个Pipeline模式应用场景。

Pipeline的本质是「数据结构和设计模式的灵活应用」，来应对流程复杂多变的业务场景。它并不是一个新的东西，也不是一个固定的设计模式，而是一种灵活的设计思想。感兴趣的小伙伴未来可以尝试使用这个模式来重构组件。当前这个组件也可以集成上责任链模式、策略模式（可能会需要根据不同的业务线，有不同的逻辑）、模板方法模式（目前作者已经提供了初版设计）、工厂方法模式（生成一条Pipeline的时候，用工厂模式来实现比较好），未来可以通过调用一个Pipeline Factory来实现。

```
Pipeline pipeline = PipelineFactory.create(pipelineConfig);
pipeline.start();
```

在本个类中，作者针对核心扩展口提供了抽象方法，未来可以自己进行实现：

```

/**
 * 获取代码差异的抽象方法。
 * 子类需要实现这个方法来获取特定的代码差异。
 *
 * @return 代码差异的字符串表示
 * @throws Exception 如果获取代码差异时发生异常
 */
protected abstract String getDiffCode() throws Exception;

/**
 * 对代码进行评审的抽象方法。
 * 子类需要实现这个方法来进行代码评审，并返回评审结果。
 *
 * @param diffCode 代码差异的字符串表示
 * @return 评审建议的字符串表示
 * @throws Exception 如果评审过程中发生异常
 */
protected abstract String codeReview(String diffCode) throws Exception;

/**
 * 记录代码评审结果的抽象方法。
 * 子类需要实现这个方法记录评审结果，并返回日志地址。
 *
 * @param recommend 评审建议的字符串表示
 * @return 评审日志的URL
 * @throws Exception 如果记录过程中发生异常
 */
protected abstract String recordCodeReview(String recommend) throws Exception;

/**
 * 推送消息的抽象方法。
 * 子类需要实现这个方法发送通知消息。
 *
 * @param logUrl 评审日志的URL
 * @throws Exception 如果发送消息时发生异常
 */
protected abstract void pushMessage(String logUrl) throws Exception;

```

5、然后再来看下触发调用的方法代码，可以看到这里面定义了基本的参数的获取，然后调用领域层进行业务的执行，从一定程度上来说，当前类也承担了一点应用层/业务编排层的职责：

```

/**
 * 程序的入口点。
 *
 * @param args 启动参数，目前未使用。
 * @throws Exception 如果发生任何异常。
 */
public static void main(String[] args) throws Exception {
    // 打印启动信息
    System.out.println("openai 代码评审，测试执行");

    // 创建GitCommand实例，需要从环境变量中获取配置信息
    GitCommand gitCommand = new GitCommand(
        getEnv("REVIEW_LOG_URI"),
        getEnv("GIT_USER_TOKEN"),
        getEnv("COMMIT_PROJECT"),
        getEnv("COMMIT_BRANCH"),
        getEnv("COMMIT_AUTHOR"),
        getEnv("COMMIT_MESSAGE")
    );

    // 创建WeiXin实例，用于发送微信消息，同样需要配置信息
    WeiXin weiXin = new WeiXin(
        getEnv("WEIXIN_APPID"),
        getEnv("WEIXIN_SECRET"),
        getEnv("WEIXIN_TOUSER"),
        getEnv("WEIXIN_TEMPLATE_ID")
    );

    // 创建OpenAI实例，用于与ChatGLM API交互
    IOpenAI openAI = new ChatGLM(getEnv("CHATGLM_APIHOST"), getEnv("CHATGLM_APIKEYSECRET"));

    // 创建OpenAiCodeReviewService实例，将所有组件组合在一起
    OpenAiCodeReviewService openAiCodeReviewService = new OpenAiCodeReviewService(gitCommand, openAI, weiXin);

    // 执行代码评审流程
    openAiCodeReviewService.exec();
}

```

在这个类中，也隐藏着一种设计思想，即入**生命周期模式 (Lifecycle Pattern) 思想**，通过生命周期模式，未来可以使其更加灵活和可扩展。生命周期模式通常用于描述一个对象从创建到销毁的整个过程，它通过定义一系列事件或状态转换来管理对象的生命周期。这样就可以实现在评审前、评审后、评审报错时的一系列扩展设计，例如：

1. **定义生命周期接口**：创建一个接口来定义生命周期事件的回调方法，比如 LifecycleListener。

```

public interface LifecycleListener {
    void onBeforeExec();
    void onAfterExec();
    void onError(Exception e);
}

```



```
// 可以根据需要添加更多生命周期事件  
}
```

**2. 维护生命周期监听器列表：**在 `AbstractOpenAiCodeReviewService` 类中维护一个监听器列表。

```
private List<LifecycleListener> lifecycleListeners = new ArrayList<>();
```

**3. 注册和注销监听器：**提供方法来注册和注销监听器。

```
public void addLifecycleListener(LifecycleListener listener) {  
    lifecycleListeners.add(listener);  
}  
  
public void removeLifecycleListener(LifecycleListener listener) {  
    lifecycleListeners.remove(listener);  
}
```

**4. 触发生命周期事件：**在生命周期事件发生时（比如执行开始、执行完成、发生错误等），通知所有注册的监听器。

```
private void notifyBeforeExec() {  
    for (LifecycleListener listener : lifecycleListeners) {  
        listener.onBeforeExec();  
    }  
}  
  
private void notifyAfterExec() {  
    for (LifecycleListener listener : lifecycleListeners) {  
        listener.onAfterExec();  
    }  
}  
  
private void notifyOnError(Exception e) {  
    for (LifecycleListener listener : lifecycleListeners) {  
        listener.onError(e);  
    }  
}
```

**5. 修改 exec 方法：**在 `exec` 方法中，适当的位置调用生命周期事件通知方法。

```
@Override  
public void exec() {  
    try {  
        notifyBeforeExec();  
        // ... 执行流程 ...  
        notifyAfterExec();  
    } catch (Exception e) {  
        notifyOnError(e);  
        logger.error("openai-code-review error", e);  
    }  
}
```

**6. 在子类中实现生命周期逻辑：**子类可以覆盖这些方法来添加特定的生命周期逻辑。

```
public class ConcreteOpenAiCodeReviewService extends AbstractOpenAiCodeReviewService {  
    // ... 其他代码 ...  
  
    @Override  
    protected void onBeforeExec() {  
        // 实现具体的生命周期逻辑  
    }  
}
```

```
@Override
protected void onAfterExec() {
    // 实现具体的生命周期逻辑
}

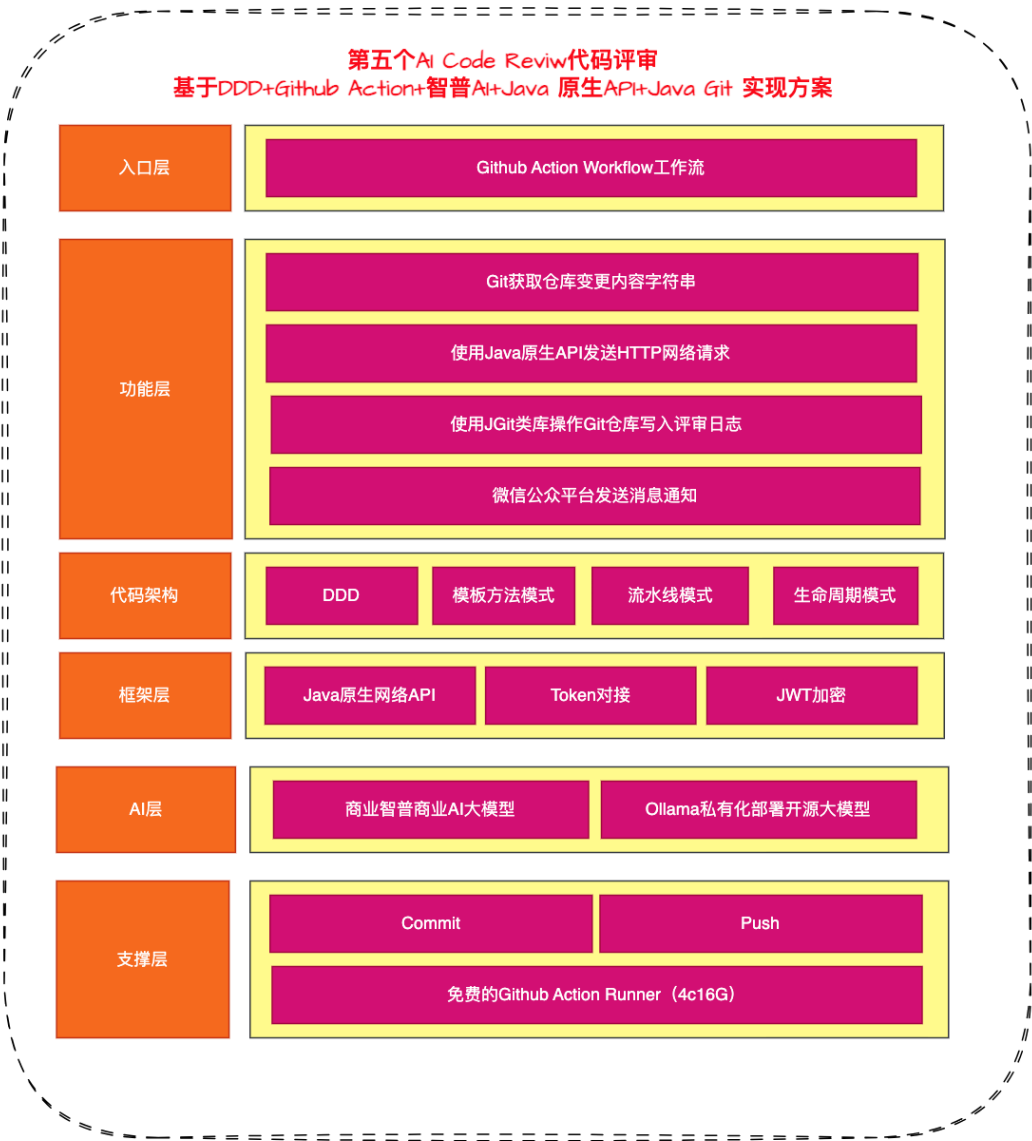
@Override
protected void onError(Exception e) {
    // 实现具体的生命周期逻辑
}
}
```

过这种方式，OpenAiCodeReview 类不仅能够更好地管理其内部流程，还能够通过监听器机制来扩展新的生命周期事件处理逻辑，而不需要修改核心代码，这符合开闭原则（对扩展开放，对修改封闭）。

个人觉得本文核心的就是这几个内容，作者将组件的业务逻辑流程进行了抽取和重构，充分考虑到了未来的扩展和实现，给我们带来的思考也是要提高抽象能力、重构能力。

这里不得不推荐那本马丁·福勒经典的书籍：重构：改善既有代码的设计，程序员这辈子都要读一读的图书。

因此，到这里，我们新的一个代码评审架构图出现了如下所示：



本节作者重构了50分钟，给大家分享了代码如何重构与基本的封装处理，思想和设计理念也是大家可以借鉴和学习的。

今天的记录到这里就结束了，OpenAI 代码自动评审组件项目一期即将结束，距离OpenAI 代码自动评审组件项目的二期建设与记录也即将开始分享、我的企业级的AI代码评审的分享也要开始分享了，两个分享分别是：

当前组件的二期建设内容

以及我的AI代码评审建设分享



希望继续帮助大家和我自己。

