Com.plexpt.chatgpt package 中的ChatGPT.java类的单元测试分析

以成员函数为划分，DD路径和数据流分析如下

DD路径测试采用 分支覆盖(c1);

数据流测试采用全定义准则

ChatGPT{

Private:

String apiKey

List<String> apiKeyList

String apiHost

Api apiClient

OkHttpClient okHttpClient

long timeout

Proxy proxy = Proxy.NO\_PROXY

Public函数(测试目标):

**ChatGPT init()**

**该函数比较特殊，相当于在构造的时候写入了两个lamda函数，所以需要依次测试其主逻辑和lamda函数逻辑**

DD路径:

决策点主要有

93 if (apiKeyList != null && !apiKeyList.isEmpty())

106 if (!response.isSuccessful())

111 if (Objects.*nonNull*(baseResponse.getError()))

123 if (Objects.*nonNull*(proxy))

DD路径如下

主逻辑

Start -> 89 -> [90-122] ->123{------->}126-136->end

{->124->}

Lamda 函数1

Start -> 80-92 ->93{------> }97-102

{-> 94 ->}

Lamda 函数2

Start ->103-105->106{->117->end

{->107-110->111{->115->end

{->112-113->end

数据流:

|  |  |  |  |
| --- | --- | --- | --- |
| 变量 | 定义节点 | 使用节点 | 全定义测试的DC路径 |
| client | 89,90-118,120,121,122 | 126 | <122,126> |
| original(1) | 91 | 97,100 | <91,100> |
| key | 92,94 | 98 | <92,98><94,98> |
| request | 97-101 | 102 | <97,102> |
| original(2) | 104 | 105 | <104,105> |
| response | 105 | 106,107,117 | <105,107><105,117> |
| errorMsg | 107 | 109,110 | <107,110> |
| baseResponse | 110 | 111,112,113 | <110,113> |
| --以下为函外定义 |  |  |  |
| apiKey | 构造函数 | 92 | <89,92> |
| apiKeyList | 构造函数 | 93,94 | <89,94> |
| timeout | 构造函数 | 120,121,122 | <89,122> |
| proxy | 构造函数 | 123,124 | <89,124> |
| okHttpClient | （构造函数），126 | 130 | <126,130> |
| apiClient | (构造函数),128 | \ | \ |
| apiHost | 构造函数 | 129 | <89,129> |

根据DD路径分析和数据流分析，设计测试用例如下:

(由于除了key定义外每个分支里都有对于数据有所使用，同时我选择的**DC路径均尽可能达到了最后一次可能的使用**，同时key定义分支互不相关，所以DD路径和数据流测试用例在我的设计和分析下是一致的)

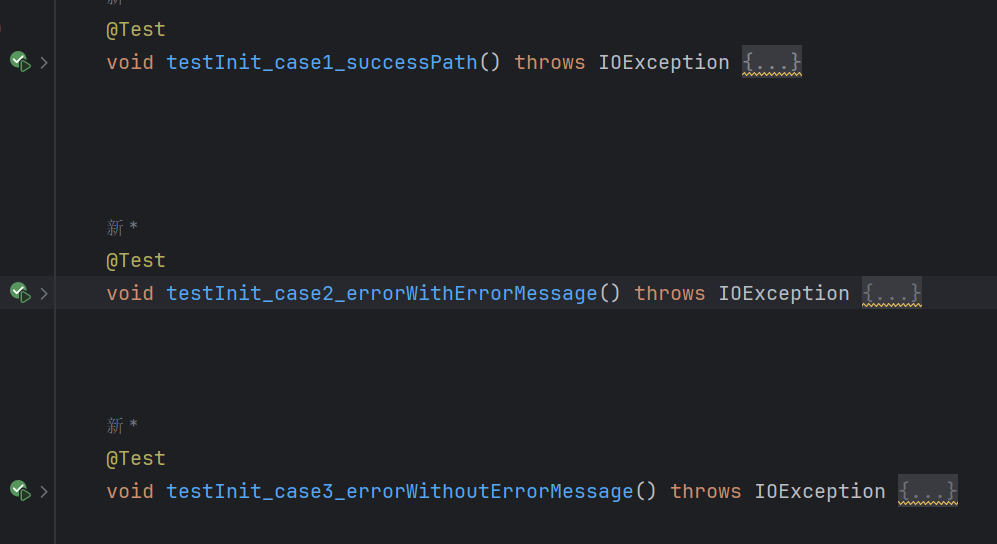
（假如baseResponse DC设置为<110,111>，则会损失一条分支的测试）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试用例 | proxy | apiKeyList | Response | baseResponse |
| 1 | Not null | Not null  Not Empty | Successful | - |
| 2 | Null | Null | !Successful | baseResponse.getError())  ! = null |
| 3 | Null | Null | !Successful | baseResponse.getError())  == null |

使用 **JUnit 5** 和 **Mockito** 实现的测试函数

由于代码中使用了lamda函数来对client进行addInterceptor的操作，所以我们不仅要测试初始化过程，还需要发起伪请求,通过调用interceptors = client.interceptors();提取出构造的两个拦截器，然后分别用两个拦截器传入mock好的chain，对代码逻辑进行测试.

总计需要对两个拦截器发送五次请求。

三个测试函数:

**其他函数大多为线性的简短的函数，在此统一进行简要分析**

1. chatCompletion(ChatCompletion chatCompletion)

**DD路径**：chatCompletion 是入参；传入后用于调用 apiClient.chatCompletion()，返回结果直接 blockingGet()。

**数据流分析**：定义后立即使用，无分支。

**测试点**：模拟 apiClient 返回特定响应，验证 blockingGet 正确解析。

2. chatCompletion(List<Message> messages)

**DD路径**：messages → chatCompletion → 复用前一个重载。

**数据流分析**：messages 构造后立即调用另一方法，无额外路径。

**测试点**：传入 message 列表，验证传递链正确、最终调用的响应结果正确。

…

chat(String message)

createEmbeddings(EmbeddingRequest request)

createEmbeddings(String input, String user)

imageGeneration(Generations generations)

imageEdit(File image, File mask, Edits edits)

imageVariation(File image, Variations variations)

audioTranscription(File audio, Transcriptions transcriptions)

audioSpeech(SpeechRequest speechRequest)

audioTranslation(File audio, Transcriptions transcriptions)

balance(String key)

creditGrants()

listFiles()

uploadFile(String purpose, MultipartBody.Part file)

deleteFile(String fileId)

retrieveFile(String fileId)

retrieveFileContent(String fileId)

这些都是简短的单一无分支逻辑的函数，没有必要对其进行分析和复杂测试，简单mock之后测试即可

Deprecated public BigDecimal balance() 稍长，但仍没有逻辑分支

**数据流分析**：

subscriptionData 派生 total

useageResponse 派生 used

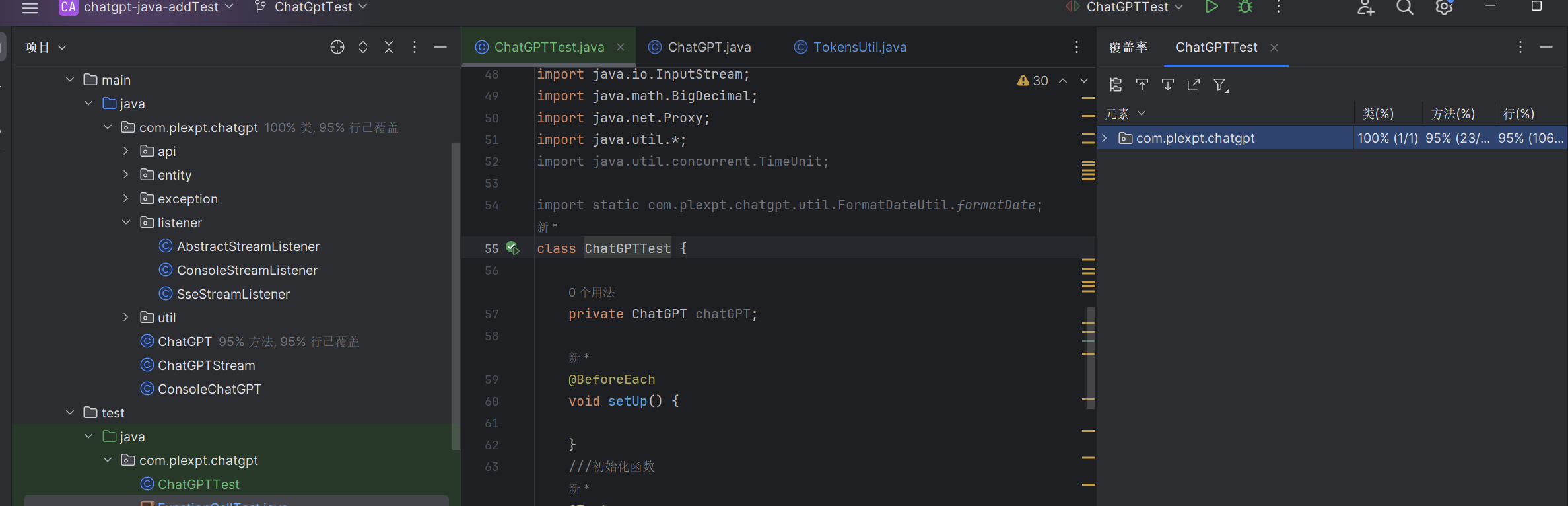
两者组合用于最终结果

使用若干个个大的测试函数对这些功能进行测试:



由于

balance(String key)已经被标记为弃用并且难以测试，于是便没有对其进行测试

}

测试结果

覆盖率95%