

25 | 多轮对话：人设、上下文和记忆

月影 · 跟月影学前端智能体开发



你好，我是月影。

在上一节课里，我们经过读写分离的设计，已经将对话的基本功能实现了。这节课呢，我们需要进一步实现面试官的人设（Role）和上下文（Context）。这是非常重要的一个环节，因为只有 AI 理解了人设和上下文信息，它才能按照我们的期望扮演好指定角色，完成工作。

接下来，我们就具体看一下这块是怎么设计的。


设计人设与任务提示词

首先，我们创建两个提示词文件，分别是 `lib/prompt/roleAndTask.prompt.ts` 和 `lib/prompt/jd.prompt.ts`，前者表示当前角色和任务，后者表示岗位描述。

在真实项目中，这两个提示词会被实现为业务后台的配置项，但是我们课程里，为了方便我稍微简化一下，直接用两个配置文件来替代。

它们的内容分别如下：

lib/prompt/roleAndTask.prompt.ts

 复制代码

```
1 export default `## 角色
2 你是月影，字节跳动的前端面试官
3
4 ## 任务
5 根据职位需求和候选人背景，提出有针对性的前端技术面试问题，并评估其专业能力、工程实践经验及团队协作能力
6
7 ## 技能
8 - HTML、CSS、JavaScript
9 - React/Vue 等主流框架
10 - 浏览器原理与性能优化
11 - 前端工程化 (Webpack/Vite)
12 - 跨端开发 (如小程序/Flutter)
13 - 计算机基础 (数据结构与算法)
14 - 系统设计与架构能力
15 - 代码审阅与技术沟通能力
16 `;
```


lib/prompt/jd.prompt.ts

 复制代码

```
1 export default `## 🧑‍💻 前端工程师（3-5年经验）岗位 JD
2
3 **岗位名称**：前端工程师（中级）
4 **工作地点**：北京 / 上海 / 深圳（可远程）
5
6 ---
7
8 ### 🛠️ 岗位职责：
9
10 1. 负责核心产品的前端架构设计与功能开发，保障系统的性能、稳定性与可维护性。
11 2. 深度参与产品需求讨论，推动前端方案落地，提升用户体验。
12 3. 持续优化前端工程化流程，包括构建性能、自动化测试、CI/CD 等。
13 4. 与后端、产品、设计等团队高效协作，推进项目高质量交付。
14 5. 关注技术趋势，参与团队技术分享与沉淀。
15
16 ---
17`;
```

```
18  ### 🎯 岗位要求：
19
20  1. 本科及以上学历，计算机或相关专业，3~5年前端开发经验。
21  2. 精通 HTML5、CSS3、JavaScript，熟悉模块化开发、ES6+ 语法。
22  3. 至少熟练掌握一种主流前端框架（如 React、Vue），具备组件化开发经验。
23  4. 理解前端性能优化原理，具备独立分析和优化线上问题的能力。
24  5. 熟悉前端构建工具（如 Webpack、Vite）和常见调试工具。
25  6. 具备良好的编码习惯、文档能力与团队合作精神。
26  7. 有大型项目开发、跨端开发或微前端经验者优先。
27
28  ---
29
30  ### ✨ 加分项：
31
32  - 有开源项目经验或技术博客。
33  - 熟悉 Node.js、服务端渲染（SSR）、图表库或动画库。
34  - 有 Web 安全、可访问性（a11y）、国际化（i18n）等实践经验。
35
36  ---
37
38  ### 💖 我们能提供：
39
40  - 有挑战的项目场景、充足的成长空间。
41  - 专业的技术团队和工程文化。
42  - 有竞争力的薪资、期权和弹性办公环境。
43  `
```

接着，我们还需要一个通用规则的提示词把人设、任务和 JD 串起来。我们创建文件 `lib/prompt/basicRule.prompt.ts`，内容如下：


 复制代码

```
1  export default `你扮演角色和任务中定义的面试官角色，根据岗位描述中定义的职位需求和候选人简历，
2
3  你需要根据候选人简历、上一轮回答和**当前面试阶段信息**，确定下一步的面试动作，包括：
4  - 总结候选人回答
5  - 引导候选人思考
6  - 提出新的问题或结束面试
7
8  ## 注意
9  你应当表现得像是真正的面试官而不是AI，用口语化的沟通风格来引导候选人回答问题，不要复述候选人回答。
10  你正在和候选人面对面沟通，除了对话内容外，你**不要**输出其他任何书面内容，不要输出你心中的未说出
11  `
```

这样我们就创建了基础提示词。

设计和实现上下文配置

接着我们要设计 ContextConfig 的结构。首先创建 `lib/config/timeline.config.ts`，内容如下：

 复制代码

```
1 import { type TimelineStep, timelineConfig } from "../timeline.config";
2 import jd from "../prompt/jd.prompt";
3 import roleAndTask from "../prompt/roleAndTask.prompt";
4 import basicRule from "../prompt/basicRule.prompt";
5
6 type TimelineContext = TimelineStep & {
7   currentTimeline: string;
8   nextTimelineAction: string;
9 }
10
11 interface ContextConfig {
12   basicRule: string;
13   jobDescription: string;
14   roleAndTask: string;
15   currentTimelineContext: TimelineContext;
16 }
17
18 export function getContext(timeline: number): ContextConfig {
19   for(let i = 0; i < timelineConfig.steps.length; i++) {
20     const step = timelineConfig.steps[i];
21     const nextTimelineAction = i === timelineConfig.steps.length - 1 ? '结束面试'
22     if(timeline >= step.startTime && timeline <= step.endTime) {
23       return {
24         basicRule,
25         jobDescription: jd,
26         roleAndTask: roleAndTask,
27         currentTimelineContext: {
28           ...step,
29           currentTimeline: timeline.toFixed(2),
30           nextTimelineAction,
31         }
32       };
33     };
34   }
35   return {
36     basicRule,
37     jobDescription: jd,
```

```


38     roleAndTask: roleAndTask,
39     currentTimelineContext: {
40         ...timelineConfig.steps[timelineConfig.steps.length - 1],
41         currentTimeline: timeline.toFixed(2),
42         nextTimelineAction: '结束面试',
43     }
44 };
45 }
46

```

这个模块并不复杂，我们把 basicRule、jd、roleAndTask 以及 currentTimelineContext 共同组成一个完整的上下文对象，这个对象将在每一轮对话时，作为 AI 系统提示词使用。值得注意的是，它是一个在面试过程中动态变化的对象，通过 getContext 方法，根据 timeline 获得对应的最新的 currentTimelineContext，用来把控整体的面试节奏。

由于这个 JSON 对象比较复杂，为了方便后续大模型的理解以及人工的调试，我们可以添加一个格式化方法，将它格式化为更加易于阅读的形式。我们修改一下

timeline.config.ts，修改后的内容如下：

 复制代码

```

1  import { type TimelineStep, timelineConfig } from "../timeline.config";
2  import jd from "../prompt/jd.prompt";
3  import roleAndTask from "../prompt/roleAndTask.prompt";
4  import basicRule from "../prompt/basicRule.prompt";
5
6  type TimelineContext = TimelineStep & {
7      currentTimeline: string;
8      nextTimelineAction: string;
9  }
10
11  interface ContextConfig {
12      basicRule: string;
13      jobDescription: string;
14      roleAndTask: string;
15      currentTimelineContext: TimelineContext;
16  }
17
18  function formatContextConfig(contextConfig: ContextConfig): string {
19      return `# 基本原则
20  ${contextConfig.basicRule}
21
22  # 你是谁

```

```
23  ${contextConfig.roleAndTask}
24
25  # 你招聘的角色
26  ${contextConfig.jobDescription}
27
28  # 当前面试阶段信息
29
30  ## 当前时间 (minutes)
31  ${contextConfig.currentTimelineContext.currentTimeline}
32
33  ## 开始时间 (minutes)
34  ${contextConfig.currentTimelineContext.startTime}
35
36  ## 结束时间 (minutes)
37  ${contextConfig.currentTimelineContext.endTime}
38
39  ## 当前应聚焦问题
40  ${contextConfig.currentTimelineContext.focus}
41
42  ## 当前阶段任务
43  ${contextConfig.currentTimelineContext.prompt}
44
45  ## 下一阶段
46  ${contextConfig.currentTimelineContext.nextTimelineAction}
47  `;
48  }
49
50  export function getContext(timeline: number): string {
51    for(let i = 0; i < timelineConfig.steps.length; i++) {
52      const step = timelineConfig.steps[i];
53      const nextTimelineAction = i === timelineConfig.steps.length - 1 ? '结束面试'
54      if(timeline >= step.startTime && timeline <= step.endTime) {
55        return formatContextConfig({
56          basicRule,
57          jobDescription: jd,
58          roleAndTask: roleAndTask,
59          currentTimelineContext: {
60            ...step,
61            currentTimeline: timeline.toFixed(2),
62            nextTimelineAction,
63          }
64        });
65      };
66    }
67    return formatContextConfig({
68      basicRule,
69      jobDescription: jd,
70      roleAndTask: roleAndTask,
71      currentTimelineContext: {
```

```
72     ...timelineConfig.steps[timelineConfig.steps.length - 1],
73     currentTimeline: timeline.toFixed(2),
74     nextTimelineAction: '结束面试',
75   }
76 });
77 }
```

在这里，我们添加了一个新的 `formatContextConfig` 方法，它可将 `contextConfig` 对象格式化为 Markdown 文本内容，这样阅读起来更方便。对应地，我们修改 `getContext` 方法返回的内容为格式化后的字符串。

这样我们就实现了对话上下文信息的基本内容。

接着我们可以应用它，我们修改 `server.ts`：

```
1 import { getContext } from './lib/config/context.config';
2 ...
3
4 app.post('/chat', async (req, res) => {
5     const { message, sessionId, timeline } = req.body;
6     const config = {
7         model_name: modelName,
8         api_key: apiKey,
9         endpoint: endpoint,
10        sse: true,
11    };
12    const context = getContext(timeline);
13
14    // ----- The work flow start -----
15    const ling = new Ling(config);
16    const bot = ling.createBot('reply', {}, {
17        response_format: { type: "text" },
18    });
19    bot.addPrompt(context);
20
21    bot.chat(message);
22
23    ling.close();
24
25    res.send(createEventChannel(ling));
26 });
```

这样我们启动服务，进入面试聊天页面，输入“你好”，得到如下的 AI 回复：

0分钟

5分钟

10分钟

15分钟

20分钟

25分钟

AI面试官

当前时间：0分钟

AI面试官

11:24:38

您好，我是今天的面试官。请做个简单的自我介绍吧。

候选人11:25:04

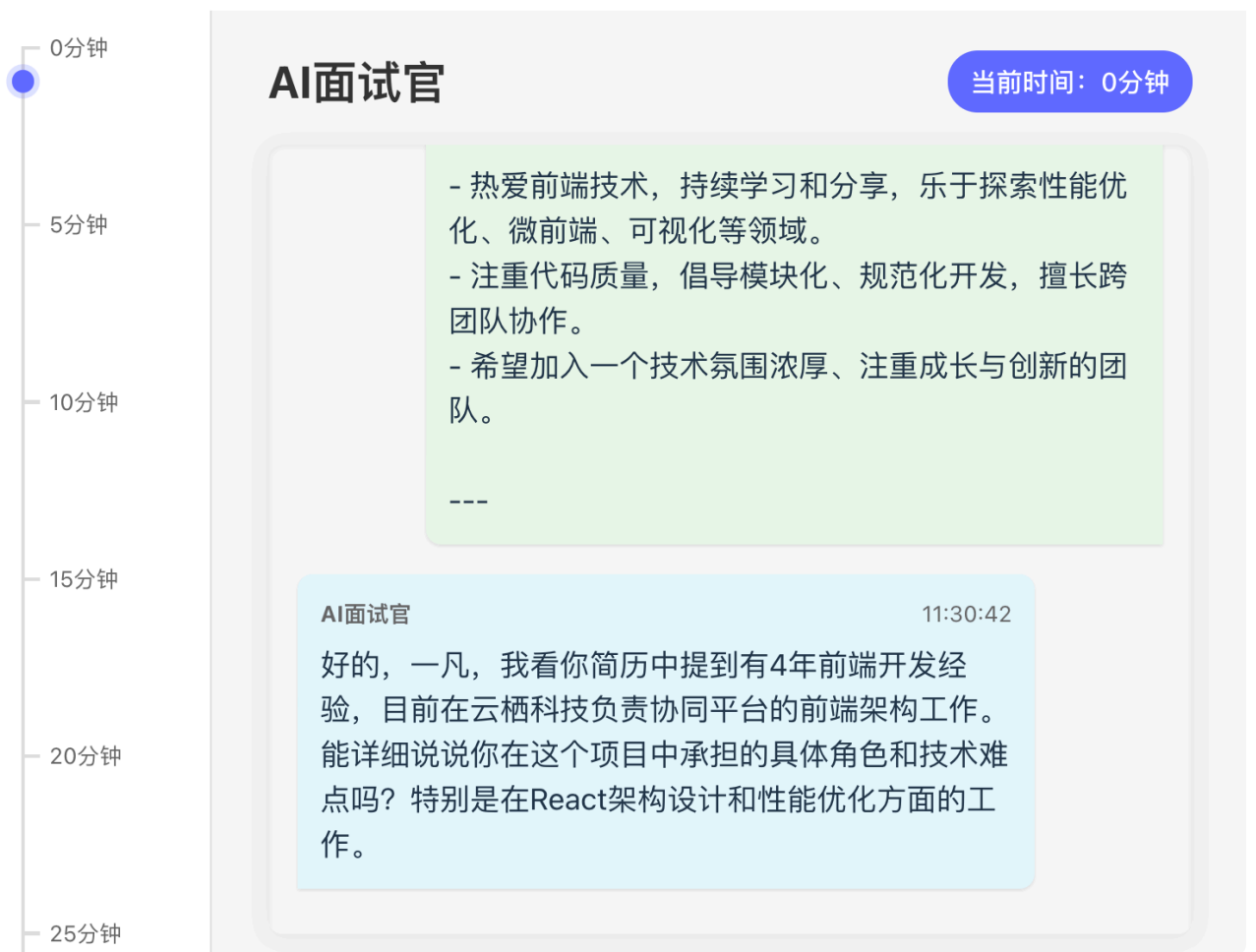
你好

AI面试官

11:25:04

你好，我是月影，今天的面试官。可以请你先做个自我介绍吗？包括你的工作经历、技术栈，以及最近参与的项目情况。

当我们发送简历之后，面试官会推进面试。



但是当我们进一步回答的时候，有时候我们会发现 AI 面试官可能不再继续推进面试，而是重复问之前的问题。

这不是 bug，而因为我们现在还没启用记忆（memory）。而没有记忆的话，对于 AI 来说，现在每一轮对话都是一个全新的内容，所以继续聊下去，AI 就会丢失前面的内容。

要使用记忆，有两种办法。一种方法是，我们每次对话都将前面聊天记录的上下文代入进去。具体做法是，我们可以将每次对话的历史记录完全记录在一个 history 数组里，然后将它通过 `bot.addHistory` 方法传给 bot。


这样做虽然简单，但是前面我们也已经说过，它有一些弊端。首先是这样做很费 token，当聊天进行的轮次过多的时候，是很容易超过大模型处理 token 的上限的。其次，当历史聊天的上下文太长的时候，也会影响 AI 的注意力，从而导致一些问题，比如奇怪的、乱套的面试

节奏，之前已经问过了某一类问题，又倒回去再问一遍，诸如这些问题将会严重影响面试体验。

因此我们会采用另一种办法，即之前的课里提到过的记忆（memory）来解决这个问题。

记录记忆

首先，我们添加一个记忆提示词文件 `lib/prompt/memory.prompt.ts`，内容如下。

 复制代码

```
1 export default `你是一个“面试记录员智能体”，你的任务是：**根据面试过程中的对话内容，提取候选
2
3 请严格遵循以下要求：
4
5 1. 只记录候选人明确表达的信息（包括自我介绍、项目经验、技术栈、职责、难点、解决方案等）。
6 2. 不推测、不补全、不润色，只提取他实际说过的内容。
7 3. 遇到不完整或模糊表达，也不要擅自解释，只保留原始语义。
8 4. 根据 memory 原有内容，结合当前对话进行更新，使用 memory 格式进行记录，严格遵循 memory 的/
9
10 ## 以下是 memory 结构的部分说明
11
12 - askedQuestions 严格记录已经问过的具体问题
13 - candidateEvaluation 目前对候选人的评价，请根据这一轮的问答，在原有评价的基础上，追加对候选
14 - lastUpdateTime 最后更新时间，一个时间戳，每次记录完，直接更新这一字段
15
16 ---
17
18 {{memory}}
19 `
```

接着，我们修改 `lib/service/memory.ts` 内容如下：

 复制代码

```
1 export interface InterviewMemory {
2   sessionId: string;
3
4   conversationIndex: number, // 当前记忆的对话轮次，因为是异步更新的，需要用这个来匹配对话
5
6   lastConversation: string, // 上一轮对话内容
7 }
```

```
8 // 候选人的基本信息和自我介绍
9 candidateIntroduction: string;
10
11 // 面试官已经问过的问题（按顺序记录）
12 askedQuestions: string[];
13
14 // 对候选人各项能力的评价
15 candidateEvaluation: {
16     technicalSkills: string; // 技术能力评价
17     problemSolving: string; // 问题解决能力
18     communication: string; // 沟通表达能力
19     codingStyle?: string; // 可选：编码风格或代码质量
20     overallImpression: string; // 总体印象
21 };
22
23 // 面试摘要：总结整个过程，比如面试重点、表现亮点或不足
24 interviewSummary: string;
25
26 // 特别备注：如迟到、网络问题、态度问题、需进一步确认的信息等
27 additionalNotes: string[];
28
29 lastUpdateTime: number;
30 }
31
32 const interviewMemoryMap = new Map<string, InterviewMemory>();
33 export function getInterviewMemory(sessionId: string): InterviewMemory {
34     let memory = interviewMemoryMap.get(sessionId);
35     if (!memory) {
36         memory = createInterviewMemory(sessionId);
37         interviewMemoryMap.set(sessionId, memory);
38     }
39     return memory;
40 }
41
42 export function updateInterviewMemory(sessionId: string, memory: InterviewMemory) {
43     interviewMemoryMap.set(sessionId, memory);
44 }
45
46 export function clearInterviewMemory(sessionId: string) {
47     interviewMemoryMap.delete(sessionId);
48 }
49
50 function createInterviewMemory(sessionId: string): InterviewMemory {
51     return {
52         sessionId,
53         conversationIndex: 0,
54         lastConversation: '',
55         candidateIntroduction: '',
56         askedQuestions: [],
```

```


57     candidateEvaluation: {
58         technicalSkills: '',
59         problemSolving: '',
60         communication: '',
61         codingStyle: '',
62         overallImpression: '',
63     },
64     interviewSummary: '',
65     additionalNotes: [],
66     lastUpdateTime: Date.now(),
67 }
68 `

```

在上面的代码里，我们主要增加了 interviewMemoryMap，用来将 memory 对象保存在内存中，实际生产环境中我们应当考虑将它缓存在 redis 等分布式持久化缓存中。

然后我们增加了 getInterviewMemory、updateInterviewMemory、clearInterviewMemory 等几个 API，用来管理每一次面试对话的记忆。

最后，我们修改 `server.ts`：

 复制代码

```

1  ...
2  import { getInterviewMemory, updateInterviewMemory } from './lib/service/memory';
3  import memoryPrompt from './lib/prompt/memory.prompt';
4  ...
5
6  const historyMap: Record<string, Array<{role: string, content: string}>> = {};
7
8  ...
9
10 app.post('/chat', async (req, res) => {
11     const { message, sessionId, timeline } = req.body;
12
13     const config = {
14         model_name: modelName,
15         api_key: apiKey,
16         endpoint: endpoint,
17         sse: true,
18     };
19
20     historyMap[sessionId] = historyMap[sessionId] || [];
21     const histories = historyMap[sessionId];

```

```
22     histories.push({role: 'user', content: message});
23
24     const context = getContext(timeline);
25     const memory = getInterviewMemory(sessionId);
26
27
28     const memoryStr = `# memory
29
30 ${JSON.stringify(memory)}
31 `;
32
33     // ----- The work flow start -----
34     const ling = new Ling(config);
35     const bot = ling.createBot('reply', {}, {
36         response_format: { type: "text" },
37     });
38
39
40     bot.addPrompt(context);
41     bot.addPrompt(memoryStr);
42
43     // 对话同时更新记忆
44     const memoryBot = ling.createBot('memory', {}, {
45         quiet: true,
46     });
47
48     memoryBot.addListener('inference-done', (content) => {
49         const memory = JSON.parse(content);
50         console.log('memory', memory);
51         updateInterviewMemory(sessionId, memory);
52     });
53
54     memoryBot.addPrompt(memoryPrompt, { memory: memoryStr });
55
56     memoryBot.chat(`# 历史对话内容
57
58 ## 提问
59 ${histories[histories.length - 2]?.content || ''}
60
61 ## 回答
62 ${histories[histories.length - 1]?.content || ''}
63 请更新记忆`);
64
65
66     bot.chat(message);
67
68     bot.addListener('inference-done', (content) => {
69         histories.push({role: 'assistant', content});
70     });
```

```
71  
72     ling.close();  
73  
74     res.send(createEventChannel(ling));  
75 });
```

我们首先将历史对话记录在 historyMap 中，然后创建一个 memoryBot 对象，将最后一轮历史对话和当前 memory 信息传给这个对象，用这个对象来更新 memory 信息，并最终通过 updateInterviewMemory 更新到 memoryMap 中，这样下一次对话就可以拿到更新后的信息了。

这样我们就实现了一个基本的，带有记忆的多轮面试对话流程。

不过呢，这里仍然有很多细节问题。首先 memoryBot 这个处理过程是比较慢的，通常比面试官回复的 bot 速度慢很多，那这时候我们就需要在流式输出上做一些处理，不能让用户等待面试官更新 memory 结束再进行下一步动作，因为那样的话停顿时间就会比较长，体验就不好。

好在面试过程是一个面试者需要较长时间来回答交流的过程，所以我们可以面试官等待面试者回答的时候，异步更新 memory 内容，这个很像真实面试中，面试官一边在听面试者回答，一边在头脑里思考和用笔记录信息的情况。

这部分具体的优化过程，我们留在后续课程中详细讲解，先来对今天的课程做一个小结。

要点总结

要实现多轮对话面试流程，我们首先要设计面试官人设以及任务提示词，然后再设计上下文配置对象，将上下文配置动态生成，在实际对话的时候作为系统提示词传给 bot 使用。另外，我们用 memory 对象的方式保持记忆，这样 AI 面试官就可以在长上下文的多轮对话中不至于“遗忘”之前的内容了。

由于保持记忆的方法需要运行时间，我们后续要做一些流程和交互上的优化，具体的内容，留待下一节课继续讲解。

课后练习

我们没有对 memory 做内容结构化和进一步说明，这一定程度上可能会影响最终记忆结果的生成，所以这部分是有优化空间的。在真实的 AI 面试官项目里，我们也是根据实际运行情况进行持续优化的，有兴趣的同学可以自己多试试，给 server.ts 加一些 log，看一下 memory 更新的结果有没有优化的余地，如果你有什么优化方面的灵感，欢迎分享到评论区。

AI智能总结

1. 人设和上下文信息对于多轮对话系统的设计至关重要，能够帮助AI系统理解角色和任务，以及根据上下文信息扮演指定角色完成工作。
2. 创建提示词文件，包括角色和任务描述以及岗位描述，这些提示词将在每一轮对话时作为AI系统的提示词使用。
3. 设计和实现上下文配置，通过ContextConfig的结构，动态生成当前面试阶段的上下文信息，用来控制整体的面试节奏。
4. 上下文配置对象是一个在面试过程中动态变化的对象，通过getContext方法，根据时间轴获得对应的最新的上下文信息。
5. 添加格式化方法，将上下文配置对象格式化为易于阅读的形式，方便后续大模型的理解以及人工的调试。
6. 上下文配置对象包括基本原则、角色和任务描述、招聘的角色描述以及当前面试阶段的具体信息，这些信息将在多轮对话中被AI系统使用。
7. 上下文配置对象的动态变化和格式化方法的添加为多轮对话系统的设计提供了更好的可读性和可维护性。
8. 上下文配置对象的设计和实现为AI系统提供了更加智能和灵活的面试角色扮演能力，使得系统能够更好地理解和应对不同的面试情境。
9. 上下文配置对象的设计和实现为多轮对话系统的功能完善和性能优化提供了重要的支持和保障。
10. 上下文配置对象的动态生成和格式化方法的添加为多轮对话系统的可扩展性和可定制性提供了更多的可能性和机会。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (1)

最新 精选



机智帅气的小雨

2025-06-16 来自浙江

使用 deepseek 生成 json 结构的 memory 的时候，用原来的 prompt，即便追加了 response_format 为 json_object，依然大部分时候返回的是 string 类型，memory 包含在 string 里的 ``json`` md 格式中，这一块要如何处理啊？在 bobixiong 的时候，由于也用 ds 替换过 viki，也有类似的问题



