

23 | 多轮对话：时间线与记忆

月影 · 跟月影学前端智能体开发



你好，我是月影。

接下来，我们就要正式进入复杂多轮对话的实现环节，这是 AI 应用里很复杂、很具有挑战的一部分。不过不要担心，跟住我的节奏，你就能掌握其中精髓。

在具体实现之前，我们要先定义一些概念。

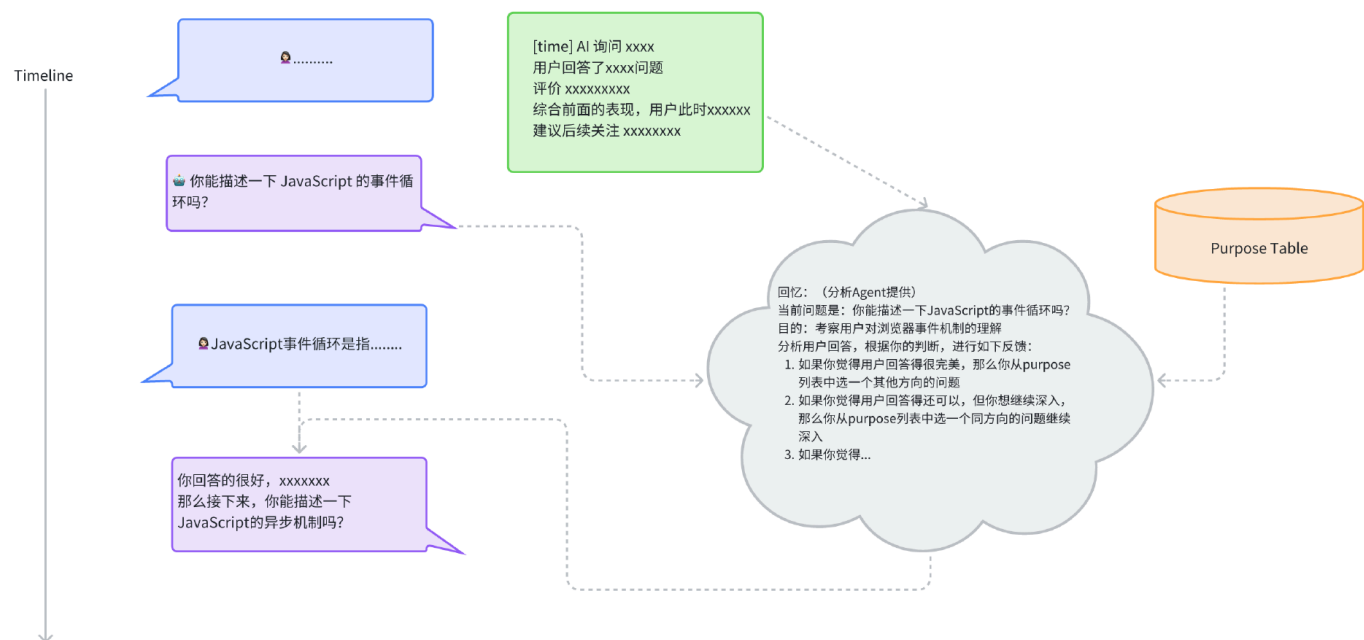
会话 (session)：用来标记一轮面试，我们用同一个会话表示一个面试，相同会话的 Agent（智能体）之间才能共享和交换数据。

记忆 (memory)：用来记录之前的内容，根据前一节设计的 MoE 模型，我们会用一个 Agent 来专门负责记忆。在真正的应用中，记忆是需要存储的，通常采用 redis、数据库等方式进行存储。但为了同学们实操方便，在这里我们将它简化为存储在内存的对象中，不影响对基本原理的理解。


时间线 (timeline): 这是一份配置，用来控制面试流程，它的每个节点包含起、止时间，和在该区间内的整体目标 (objective)。在实际代码里，我们还会有一个控制器来操纵时间线，让 AI 有一定的自主权。你可能会想 AI 为什么要控制时间线，这个问题我先抛出来供你思考，我们后面深入讲解的时候再解释。

目的表 (purpose table): 这也是一份配置表，用于思考的 Agent 根据时间线和记忆，参考目的表来筛选出当前轮次的目的，根据目的来生成一组具体的行动方针 (actions)，对话 Agent 会根据候选人的回答和行动方针，参考目的表来进行下一轮的提问。

下面是一个典型的对话、分析和思考配合过程的示意图。



根据上面的示意图，当 AI 询问“你能描述一下 JavaScript 的事件循环吗？”之后，等待候选人回答的期间，AI 将制定如下**行动方针**：

 复制代码


- 1 - 如果你觉得用户回答得很完美，那么你就从purpose列表中选一个其他方向的问题
- 2 - 如果你觉得用户回答得还可以，但你想继续深入，那么你就从purpose列表中选一个同方向的问题继续深入
- 3 - 如果你觉得.....

这里需要强调的是，就是整体 Agent 协同的过程是一个异步过程。仔细看上面的图，当 AI 询问一个问题时，候选人开始回答，**同时** AI 开始准备下一步的策略，生成行动方针。注意行动方针并不考虑候选人实际是如何回答的，而是**将候选人任何一种回答的可能性都考虑进去，并针对该可能性制定计划（即行动方针）**。正是这样的诀窍，大大提升了面试流程的思考和对话并行能力，极大提高了面试流程的实时性。

时间线（Timeline）

前面已经提到过，面试整体要遵循一定的节奏，所以我们可以通过配置时间线，来让 AI 针对不同的时间点思考不同的面试对策。这一点和真正的面试官是一样的，我们在真实的面试中，面试官也会考虑如何安排整体时间，从而把握整个面试节奏。

在 Server 端，时间线是一份配置文件，我们创建 `timeline.config.ts`，内容如下：

 复制代码

```
1  /*
2   定义面试过程的时间线
3   0~3 分钟 自我介绍
4   3~10 分钟 项目讨论
5   10~17 分钟 技术讨论
6   17~25 分钟 代码和算法讨论
7   25~30 分钟 非技术问题讨论
8   30~32 分钟 反问
9   32 分钟以上 结束
10 */
11
12 interface TimelineStep {
13   startTime: number;
14   endTime: number;
15   focus: string; // 聚焦的问题，例如项目、技术、代码、算法、非技术问题
16   prompt: string; // 提示语
17 }
18
19 interface TimelineConfig {
20   steps: TimelineStep[];
21 }
22
23 export const timelineConfig: TimelineConfig = {
24   steps: [
25     {
26       startTime: 0,
27       endTime: 3,
```

```
28     focus: '自我介绍',
29     prompt: `当前在自我介绍阶段，你要求候选人提供简历或者自我介绍，简历和介绍内容包括：
30 1. 个人信息
31 2. 教育背景
32 3. 工作经历
33 4. 项目经历
34 5. 擅长的技术
35 6. 自我评价
36
37 你首先根据<memory>的信息，判断候选人是否完成了介绍，如果没有完成，你询问并收集候选人缺失的信息，
38     `，
39     }, {
40         startTime: 3,
41         endTime: 10,
42         focus: '项目讨论',
43         prompt: `当前在项目讨论阶段，你要求候选人讨论他/她在项目中的表现。
44
45 你首先根据<memory>的信息，回顾 interviewSummary，针对你感兴趣的项目经历或技术细节，深入询问他
46     `，
47     }, {
48         startTime: 10,
49         endTime: 17,
50         focus: '技术讨论',
51         prompt: `当前在技术讨论阶段，你要求候选人讨论他/她在技术方面的表现。
52
53 如果你之前在和候选人讨论TA过往项目，你可以收尾，然后开始讨论前端工程师的技术栈和基础知识。
54
55 首先根据候选人应聘的岗位描述，判断对前端技术的总体要求，并结合<memory>的信息，针对你感兴趣的技术
56
57 注意整体节奏遵循先JS（包括Node.js如果岗位要求或候选人简历提到的话）、再CSS/HTML、然后是框架和
58
59 你通过<memory>中的askedQuestions回溯之前的问题，以把控整体节奏。
60     `，
61     }, {
62         startTime: 17,
63         endTime: 25,
64         focus: '代码和算法讨论',
65         prompt: `当前在代码和算法讨论阶段，你要求候选人讨论他/她在代码和算法方面的表现。
66
67 如果你之前在和候选人讨论前端技术栈和基础知识，你可以收尾，然后开始讨论代码问题和算法题。
68
69 首先根据候选人应聘的岗位描述，判断对代码和算法的总体要求，并结合<memory>的信息，针对你感兴趣的代
70
71 整体节奏上你先出一道简单的代码题，让候选人说出代码运行结果，然后你再出一道算法题，让候选人说出算
72
73 你通过<memory>中的askedQuestions回溯之前的问题，以把控整体节奏。
74     `，
75     }, {
76         startTime: 25,
```

```

77     endTime: 30,
78     focus: '非技术问题讨论',
79     prompt: `当前在非技术问题讨论阶段，你要求候选人讨论他/她在软素质方面的表现。
80
81 如果你之前在和候选人讨论代码和算法，你可以收尾，然后开始讨论非技术问题。
82
83 首先根据候选人应聘的岗位描述，判断对软素质的总体要求，并结合<memory>的信息，针对你感兴趣的软素质
84
85 整体节奏上你按照价值观、团队沟通协作、项目管理、时间管理、学习能力、沟通表达能力、解决问题能力、自
86
87 你通过<memory>中的askedQuestions回溯之前的问题，以把控整体节奏。
88 、
89     }, {
90         startTime: 30,
91         endTime: 32,
92         focus: '反问',
93         prompt: `当前在反问阶段，你要求候选人反问面试官，补充判断候选人对岗位是否有兴趣。
94
95 候选人提问后，你进行回答，如果候选人说没有问题了，你可以结束面试。
96 、
97     }, {
98         startTime: 32,
99         endTime: Infinity,
100        focus: '结束',
101        prompt: `当前在结束阶段，你礼貌地与候选人沟通，结束面试。`
102    }
103 ]
104 };

```

在这里，我们适当缩减了面试整体时间，将面试节奏把控在大约 35 分钟之内，这是为了体验和测试方便。在真实产品中，我们可以根据需要进行调整时间节奏，延长面试的总体时间。

我们先看一下上面的代码，TimelineConfig 是一份配置数据，它的 steps 是一个 TimelineStep 数组，数组中每一个元素表示当前时间下的面试阶段配置，它由以下属性构成：

startTime 当前阶段开始时间，以分钟为单位

endTime 当前阶段结束时间，以分钟为单位

focus 当前面试阶段聚焦的问题


prompt 当前面试阶段的主要策略提示词

在实际使用的时候，我们将根据前端传入的时间参数来控制当前具体的阶段。当然，这是一种简易的办法，根据产品特点的不同，还有其他决定时间的方式。

因为我们的产品主要卖点是模拟面试，所以可以采用简单的方式，不用担心用户自己更改了传入的参数，而如果产品最终的用途是替代真正的面试官完成真实面试，那么考虑到安全性，可能我们就要用服务端的时间来计算具体的 Timeline 阶段了。

配套地，在前端我们通过实现一个 UI 组件来展示 Timeline。

我们封装一个 Vue 组件。首先创建 `src/components/Timeline.vue`，内容如下：

 复制代码

```
1 <script setup lang="ts">
2 import { ref, defineProps, defineEmits, watch, onBeforeUnmount } from 'vue';
3
4 const props = defineProps<{
5   currentTime: number;
6   totalDuration: number;
7   started: boolean;
8 }>();
9
10 const emit = defineEmits(['updateTime']);
11
12 const timePoints = ref<number[]>([]);
13 const startTime = ref<number>(0);
14 const intervalId = ref<number | null>(null);
15
16 // 生成时间点数组，从0到总时长，每5分钟一个刻度
17 for (let i = 0; i <= props.totalDuration; i += 5) {
18   timePoints.value.push(i);
19 }
20
21 // 监听started状态变化
22 watch(() => props.started, (newValue) => {
23   if (newValue && !intervalId.value) {
24     // 记录开始时间
25     startTime.value = Date.now();
26     // 启动计时器，每秒更新一次时间
27     intervalId.value = window.setInterval(() => {
28       const elapsedMinutes = (Date.now() - startTime.value) / 60000;
29       // 确保不超过总时长
30       const newTime = Math.min(elapsedMinutes, props.totalDuration);
31       if (newTime !== props.currentTime) {
```


```
32     emit('updateTime', newTime);
33   }
34   }, 1000);
35 }
36 });
37
38 // 组件卸载时清除计时器
39 onBeforeUnmount(() => {
40   if (intervalId.value) {
41     clearInterval(intervalId.value);
42   }
43 });
44 </script>
45
46 <template>
47   <div class="timeline-container">
48     <div class="timeline">
49       <div
50         v-for="time in timePoints"
51         :key="time"
52         class="time-point"
53         :class="{ active: time === currentTime }"
54       >
55         <div class="time-marker"></div>
56         <div class="time-label">{{ time }}分钟</div>
57       </div>
58       <div
59         class="current-time-indicator"
60         :class="{ active: intervalId !== null }"
61         :style="{ top: `${(currentTime / totalDuration) * 100}%` }"
62       ></div>
63     </div>
64   </div>
65 </template>
66
67 <style scoped>
68 .timeline-container {
69   width: 160px;
70   height: 100%;
71   padding: 20px 0;
72   display: flex;
73   justify-content: center;
74   box-sizing: border-box;
75 }
76
77 .timeline {
78   position: relative;
79   height: 100%;
80   width: 2px;
```

```
81     background-color: #ddd;
82     display: flex;
83     flex-direction: column;
84     justify-content: space-between;
85 }
86
87 .time-point {
88     position: relative;
89     cursor: pointer;
90     display: flex;
91     align-items: center;
92 }
93
94 .time-marker {
95     width: 10px;
96     height: 2px;
97     background-color: #ddd;
98     position: absolute;
99     left: 0;
100 }
101
102 .time-label {
103     position: absolute;
104     left: 15px;
105     font-size: 12px;
106     color: #666;
107     white-space: nowrap;
108 }
109
110 .time-point.active .time-marker {
111     background-color: #646cff;
112 }
113
114 .time-point.active .time-label {
115     color: #646cff;
116     font-weight: bold;
117 }
118
119 .current-time-indicator {
120     position: absolute;
121     width: 12px;
122     height: 12px;
123     border-radius: 50%;
124     background-color: brown;
125     left: -5px;
126     transform: translateY(-50%);
127     box-shadow: 0 0 0 3px rgba(100, 108, 255, 0.2);
128 }
129
```



```
130 .current-time-indicator.active {
131   background-color: #646cff;
132 }
133 </style>
```

在上面的代码中，我们创建了一个 Timeline 的 UI，它通过开关控制是否开启计时，具体实现上是用 watch 方法监听 props.started 状态。

 复制代码

```
1 // 监听started状态变化
2 watch(() => props.started, (newValue) => {
3   if (newValue && !intervalId.value) {
4     // 记录开始时间
5     startTime.value = Date.now();
6     // 启动计时器，每秒更新一次时间
7     intervalId.value = window.setInterval(() => {
8       const elapsedMinutes = (Date.now() - startTime.value) / 60000;
9       // 确保不超过总时长
10      const newTime = Math.min(elapsedMinutes, props.totalDuration);
11      if (newTime !== props.currentTime) {
12        emit('updateTime', newTime);
13      }
14    }, 1000);
15  }
16 });
```

当计时开启后，Timeline 会启动计时器，即通过 window.setInterval 定时器动态更新时间，并且通过 updateTime 方法将更新的时间发给父级组件，这样父级组件调用 AI 对话的时候，就可以将具体的时间参数传给 Server 端。

在 UI 方面，通过更新元素的样式来控制小圆点在时间轴中的位置：

复制代码

```
1 <div
2   class="current-time-indicator"
3   :class="{ active: intervalId !== null }"
4   :style="{ top: `${(currentTime / totalDuration) * 100}%` }"
5 ></div>
```

最终的 UI 效果如下图所示：



图里面的动态效果展示得不太明显，但是你可以看到随着候选人发送消息后，面试正式开始，左侧时间轴的蓝色小圆点就开始缓慢向下移动了。

这个页面的其他部分功能还没有实现，所以我发送消息后，AI 并没有回复，我们先不用管它，在后续章节里，我们再逐步完善，到时候也会详细讲解右侧这一部分的前端逻辑和实现。

记忆 (memory)

现在我们先回过头来看一下记忆（memory）模块。

在 AI 智能体和应用的设计中，记忆（memory）是一个非常重要的特性，因为我们很多时候必须让 AI 能够记住之前发生的事情，从而进行下一步的动作。就拿面试官应用来说，如果 AI 没有记忆，那它就很可能问重复的问题，这就和真实的面试过程不符了。

一般来说，记忆又分为长期记忆和短期记忆。通常情况下，短期记忆是通过聊天上下文，由大模型自己管理的；而长期记忆，则是通过记录和整理文本（知识库）、向量数据库等方式实现，对于不同的应用，有不同的要求。


对我们的产品而言，如果不考虑多轮面试，那么最重要的还是短期记忆。不过我们不能依靠默认的聊天上下文，这是因为，面试过程比较长，对话上下文比较复杂，而且面试有比较严格的流程，只依赖单纯的聊天上下文，是达不到效果的。

所以我们用另外一种短期记忆的方式，即**结构化短期记忆**。

其实结构化短期记忆的实现非常简单，也就是我们不是简单记录聊天上下文，在下一次对话时提交之前的聊天记录，而是用一个负责记忆的智能体，来整理记录每轮聊天之后的内容，将其记录为 JSON 数据，这对应了我们前面说的 **MoE** 结构中信息分析记录的部分。

在这一节里，我们先不说它的具体逻辑实现，只定义它的数据结构。

我们将 Memory 定义如下，首先创建 `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 export function createInterviewMemory(sessionId: string): InterviewMemory {
33     return {
34         sessionId,
35         conversationIndex: 0,
36         lastConversation: '',
37         candidateIntroduction: '',
38         askedQuestions: [],
39         candidateEvaluation: {
40             technicalSkills: '',
41             problemSolving: '',
42             communication: '',
43             codingStyle: '',
44             overallImpression: '',
45         },
46         interviewSummary: '',
47         additionalNotes: [],
48         lastUpdateTime: Date.now(),
49     }
50 }

```

根据上面的代码, 我们记忆的数据格式如下。

sessionId: 会话 ID, 在一场面试中唯一, 同样的会话 ID 将同一场面试中的对话、记忆和思考关联起来。

conversationIndex：对话轮次，多轮对话中我们会异步更新记忆，所以当前记忆已经更新到第几轮对话了，需要记录下来，这样我们才能匹配记忆的“版本”，具体内容在后续章节中会详细说明。

lastConversation：上一轮对话内容，我们记录下较少的历史聊天上下文，因为有了memory 不需要记录太多轮。

candidateInstruction：候选人介绍，在自我介绍和项目讨论阶段主要更新这部分记忆。

askedQuestions：已经问过的问题，这个列表很重要，AI 要记住问过的问题，以免重复询问。

candidateEvaluation：候选人评价，通过几个维度对候选人做出评价，在每一轮面试中分析智能体会根据评价制定后续提问策略，面试结束后，负责给出评价的智能体也需要根据这部分内容给出最终的评价。

interviewSummary：面试摘要，每轮面试对话后都会更新摘要，记录面试过程里比较重要的信息。

additionalNotes：特别备注，面试过程中需要额外留意的信息，面试官的自我提醒。

lastUpdateTime：记忆最后更新时间。

以上就是整体的记忆表，在每一轮对话后，负责记录的智能体就会将需要更新的信息更新到该表中，智能体提问前，也会通过将记忆信息放入系统提示词中，控制下一轮的询问。

具体详细的流程我们后续的章节中会一一展开细说。

要点总结

这一节里，我们具体了解了时间线（Timeline）和记忆（Memory）的概念定义和数据结构，它们是后续实现核心工作流的重要对象。

另外，我们还实现了时间线的前端 UI 组件。下一节课，我们还会继续探索，看看如何用时间线搭配记忆，来实现节奏合理的高质量面试对话。

课后练习

这节课虽然没有涉及 UI 核心功能，但对于前端工程师来说，交互体验也特别重要。

时间线 UI 组件还有些细节值得优化，你可以考虑在它之上加入一些小功能，比如鼠标移动到时间线小圆点上时，具体显示对应的时间，以及对时间线上某个时间点设置一个倒计时，这样就可以在面试中候选人需要较长时间的思考时提醒对方（比如写代码环节里，设置一个 10 分钟的提醒）。

你可以动手试试看，将你的改进分享到评论区。

AI智能总结

1. 实现复杂多轮对话是 AI 应用中的挑战之一，需要掌握会话、记忆、时间线和目的表等概念。
2. 记忆在实际应用中需要存储，通常采用 redis、数据库等方式进行存储，但在实操中可以简化为存储在内存的对象中。
3. 时间线是一份配置，用来控制面试流程，包含起、止时间和整体目标，AI 会有一个控制器来操纵时间线，让 AI 有一定的自主权。
4. 目的表是一份配置表，用于思考的 Agent 根据时间线和记忆，参考目的表来筛选出当前轮次的目的，根据目的来生成一组具体的行动方针。
5. 对话 Agent 会根据候选人的回答和行动方针，参考目的表来进行下一轮的提问。
6. 整体 Agent 协同的过程是一个异步过程，AI 在候选人回答的期间开始准备下一步的策略，生成行动方针，大大提升了面试流程的思考和对话并行能力，极大提高了面试流程的实时性。

© 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。

精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。