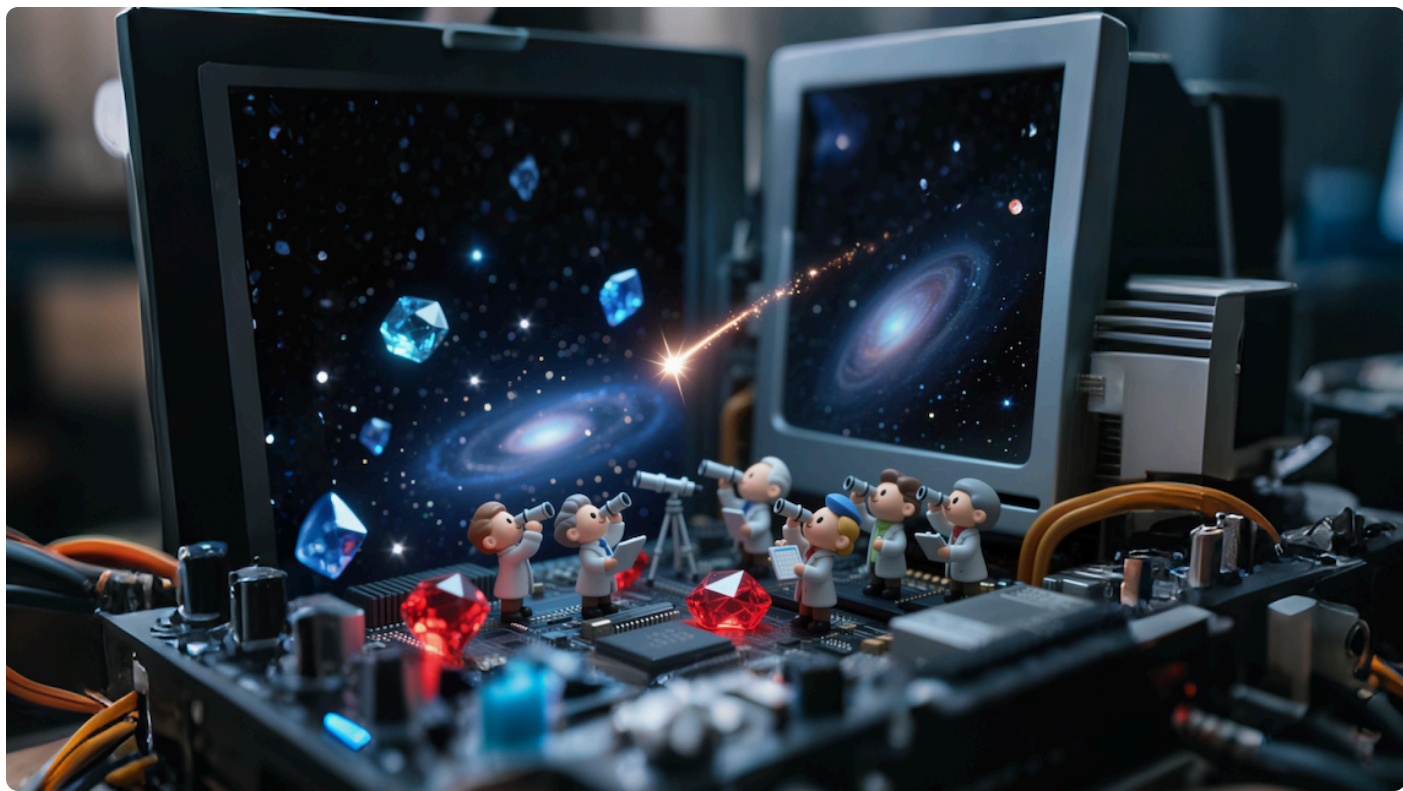


26 | 交互细节功能的丰富和功能优化

月影 · 跟月影学前端智能体开发




你好，我是月影。

上一节课，我们几乎已经实现了 AI 面试官项目完整的流程，给面试官智能体赋予了人设、上下文和记忆。

但是，在这里，我们还需要进行一些细节优化，以保证面试效果和面试体验。

异步更新记忆及前端优化

我们上一节课有一个细节，那就是我们在工作流中对 memory 的更新采用了 quiet 的方式，在 Ling 框架的介绍中我提到过，当我们给 Bot 设置 quiet 参数后，它在执行推理的过程里，并不会将任何内容发送给前端。

 复制代码

```
1 // 对话同时更新记忆
2 const memoryBot = ling.createBot('memory', {}, {
```

```
3         quiet: true,  
4     }  
    }
```


所以，我们无需等待这部分内容更新结束，就可以继续前端的对话。这样就不会因为记忆的更新需要的时间过长，而导致用户等待很久，让面试沟通变得不够流畅。

不过这带来另一个问题，那就是我们持续对话的时候，有可能记忆还未更新，这会导致 AI 记住的内容和当前对话不一定同步。例如面试对话已经进行到了第 5 轮，而记忆可能才更新到第 3 轮。所以我们还是需要传递给 AI 少量的上下文，以防止这种情况下 AI 对话丢失了最近的上下文信息，产生不恰当的提问或者重复对话等问题。

所以，这时候，我们还是需要把最近的一部分历史对话传给大模型。上一节课，我们通过 historyMap 记录了所有的历史对话，因此我们可以取最近两轮历史对话，也就是 histories 的最后 3 条记录传给当前对话。

这部分历史对话内容虽然不是很多，但还是要消耗部分 token，因此我们需要适当增加 max_tokens 配置，这样才能够确保对话时 token 数量够用。

我们修改一下 server.ts：

 复制代码


```
1 app.post('/chat', async (req, res) => {  
2     const { message, sessionId, timeline } = req.body;  
3  
4     const config = {  
5         model_name: modelName,  
6         api_key: apiKey,  
7         endpoint: endpoint,  
8         sse: true,  
9     };  
10  
11     historyMap[sessionId] = historyMap[sessionId] || [];  
12     const histories = historyMap[sessionId];  
13     histories.push({role: 'user', content: message});  
14  
15     const context = getContext(timeline);  
16     const memory = getInterviewMemory(sessionId);  
17 }
```

```
18     const memoryStr = `# memory
19
20     ${JSON.stringify(memory)}
21     `;
22
23     // ----- The work flow start -----
24     const ling = new Ling(config, {
25         max_tokens: 8192,
26     });
27     const bot = ling.createBot('reply', {}, {
28         response_format: { type: "text" },
29     });
30
31     bot.addPrompt(context);
32     bot.addPrompt(memoryStr);
33
34     bot.addHistory(histories.slice(-3));
35
36     // 对话同时更新记忆
37     const memoryBot = ling.createBot('memory', {}, {
38         quiet: true,
39     });
40
41     memoryBot.addListener('inference-done', (content) => {
42         const memory = JSON.parse(content);
43         updateInterviewMemory(sessionId, memory);
44     });
45
46     memoryBot.addPrompt(memoryPrompt, { memory: memoryStr });
47
48     memoryBot.chat(`# 历史对话内容
49
50     ## 提问
51     ${histories[histories.length - 2]?.content || ''}
52
53     ## 回答
54     ${histories[histories.length - 1]?.content || ''}
55
56     请更新记忆`);
57
58     bot.chat(message);
59
60     bot.addListener('inference-done', (content) => {
61         histories.push({role: 'assistant', content});
62     });
63
64     bot.addListener('response', () => {
65         ling.sendEvent({event: 'response-finished'});
66     });
```

```
67     });  
68  
69     ling.close();  
70  
71     res.send(createEventChannel(ling));  
72 }):
```


上面的代码里，我们主要修改了三个地方。

首先是我们给 Ling 对象增加了一个 max_tokens 参数，设置为 8192。如果是 moonshot-8k，这个是最大值，如果是上下文更大的模型，可以设置为更大值，比如 16k 或者 32k。

 复制代码

```
1     const ling = new Ling(config, {  
2         max_tokens: 8192,  
3     });
```

其次，我们给面试对话的 bot 增加了一组最近的聊天记录，上面说过我们取最后三段历史聊天结果：

 复制代码

```
1 bot.addHistory(histories.slice(-3));
```

最后，我们在 bot 输出结束时，给前端发送一个 `response-finished` 事件，这个事件可以用来控制前端 UI 展示，因为此时 memoryBot 还在处理，所以 Ling workflow 整体还未结束，前端还收不到 Ling 本身的 finished 事件。

这样，我们在前端可以进行 UI 处理，为了防止用户重复提交，我们在用户提交后，将提交按钮状态置为 disabled，等收到 `response-finished` 事件后，再恢复按钮可用。

我们修改 src/components/MessageInput 组件。

```
1 <script setup lang="ts">
2 ...
3
4 const buttonDisabled = ref(false);
5 ...
6
7 const handleSendMessage = () => {
8   if (message.value.trim()) {
9     buttonDisabled.value = true;
10    emit('sendMessage', message.value, buttonDisabled);
11    message.value = '';
12  }
13 };
14
15 <template>
16 ...
17   <button class="send-button" @click="handleSendMessage" :disabled="buttonDisab
18 ...
19 </template>
20
21 <style scoped>
22 ...
23 .send-button:disabled {
24   background-color: #ccc;
25   cursor: not-allowed;
26 }
27 </style>
```

上面的代码里，我们在发送消息给 AI 时，将按钮状态通过 buttonDisabled 置为不可用，并将该变量传递给父级组件。因此我们在父级组件，即 App.vue 中也要修改代码。

```
1 <script setup lang="ts">
2 ...
3
4 // 发送新消息
5 const sendMessage = async (content: string, buttonDisabled: any) => {
6   ...
7   eventSource.addEventListener('response-finished', () => {
8     buttonDisabled.value = false;
9   });
10   ...
11 }
```

```
12 </script>
13 ...
```


这样当 AI 回复完成时，我们就恢复提交按钮状态，用户就可以继续进行对话了。

语音输入

接下来，我们要实现语音输入功能，因为面试对话时候选人回答通常比较多，语音输入能极大降低交流成本提升体验。


在第 21 节课程中，我们已经讲解过语音输入的解决方案了，在这里我们可以直接照搬过来。

首先修改 `.env.local` 增加配置。

 复制代码

```
1 # Azure STT
2 VITE_AZURE_SPEECH_KEY=2JdU*****niFW
3 VITE_AZURE_SPEECH_REGION=southeastasia
```

接着编辑 `server.ts` 添加鉴权接口。


 复制代码

```
1 app.get('/voice-token', async (req, res) => {
2   const region = process.env.VITE_AZURE_SPEECH_REGION;
3   const key = process.env.VITE_AZURE_SPEECH_KEY;
4
5   const headers: any = {
6     'Ocp-Apim-Subscription-Key': key,
7     'Content-Type': 'application/x-www-form-urlencoded',
8   };
9
10  const token = await (
11    await fetch(`https://${region}.api.cognitive.microsoft.com/sts/v1.0/issue
12      method: 'POST',
13      headers,
14    })
15  ).text();
```

```
16     res.send({
17       data: {
18         token: token,
19         region: region,
20       }
21     });
22   });
23 }
```


然后实现前端功能。首先安装必要的依赖：

```
1 pnpm i universal-cookie microsoft-cognitiveservices-speech-sdk
```

 复制代码

接着在项目下创建 `src/lib/voice/helper.ts` 文件：

```
1 import Cookie from 'universal-cookie';
2 import * as speechsdk from 'microsoft-cognitiveservices-speech-sdk';
3
4 async function getTokenOrRefresh() {
5   const cookie = new Cookie();
6   const speechToken = cookie.get('speech-token');
7
8   console.log(speechToken, 'speechToken');
9
10  if (speechToken === undefined || speechToken === 'undefined:undefined') {
11    try {
12      const res = await fetch('/api/voice-token', {
13        method: 'GET',
14        headers: {
15          'Content-Type': 'application/json',
16        },
17      });
18      const { data } = await res.json();
19
20      // console.log(data);
21
22      const token = data.token;
23      const region = data.region;
24      cookie.set('speech-token', `${region}:${token}`, {
25        maxAge: 540,
```

 复制代码

```
26     path: '/',
27   });
28
29   console.log('Token fetched from back-end: ' + token);
30   return { authToken: token, region: region };
31 } catch (err: any) {
32   console.log(err);
33   return { authToken: null, error: err.response.data };
34 }
35 } else {
36   // console.log('Token fetched from cookie: ' + speechToken);
37   const idx = speechToken.indexOf(':');
38   return {
39     authToken: speechToken.slice(idx + 1),
40     region: speechToken.slice(0, idx),
41   };
42 }
43 }
44
45 export async function sttFromMic(onMessage: (text: string, delta: string, isClose
46   const tokenObj = await getTokenOrRefresh();
47   const speechConfig = speechsdk.SpeechConfig.fromAuthorizationToken(tokenObj.aut
48   speechConfig.speechRecognitionLanguage = 'zh-CN';
49
50   const audioConfig = speechsdk.AudioConfig.fromDefaultMicrophoneInput();
51   const recognizer = new speechsdk.SpeechRecognizer(speechConfig, audioConfig);
52
53   console.log('speak into your microphone...');
54
55   let text = '';
56
57   recognizer.recognizing = (s, e) => {
58     // console.log(`RECOGNIZING: Text=${e.result.text}`);
59     (recognizer as any).startRecord = true;
60   };
61
62   recognizer.recognized = (s, e) => {
63     if (e.result.reason == speechsdk.ResultReason.RecognizedSpeech) {
64       // console.log(`RECOGNIZED: Text=${e.result.text}`);
65       text += e.result.text;
66       onMessage(text, e.result.text, false);
67     } else if (e.result.reason == speechsdk.ResultReason.NoMatch) {
68       // console.log('NOMATCH: Speech could not be recognized.');
```




```

75     if (e.reason == speechsdk.CancellationReason.Error) {
76         console.log(`CANCELED: ErrorCode=${e.errorCode}`);
77         console.log(`CANCELED: ErrorDetails=${e.errorDetails}`);
78         console.log('CANCELED: Did you set the speech resource key and region value
79     }
80
81     recognizer.stopContinuousRecognitionAsync();
82 };
83
84 recognizer.sessionStopped = (s, e) => {
85     // console.log('\n    Session stopped event. ');
86     recognizer.stopContinuousRecognitionAsync();
87     onMessage(text, '', true);
88 };
89 recognizer.startContinuousRecognitionAsync();
90
91 return recognizer;
92 }

```

然后创建 `src/lib/voice/index.ts`。

 复制代码

```

1  import { sttFromMic } from './helper';
2
3  export class Recognizer {
4      private recorder: MediaRecorder | null = null;
5      private initialized: boolean = false;
6      private isRecording: boolean = false;
7      private azureSTT: any = null;
8
9      onAudioRecording?: (event: MessageEvent<any>) => void;
10     onAudioTranscription?: ((transcription: { text: string; delta: string; done: bo
11
12     async init() {
13         const stream: MediaStream = await navigator.mediaDevices.getUserMedia({
14             audio: true,
15         });
16
17         const audioContext = new AudioContext();
18         const mediaStreamSource = audioContext.createMediaStreamSource(stream);
19         await audioContext.audioWorklet.addModule('/worklet/vumeter.js');
20         const node = new AudioWorkletNode(audioContext, 'vumeter');
21         node.port.onmessage = event => {
22             if (event.data.volume) {
23                 if (this.isRecording) {

```

```


24         this.onAudioRecording && this.onAudioRecording(event);
25     }
26 }
27 };
28 mediaStreamSource.connect(node).connect(audioContext.destination);
29
30 this.initialized = true;
31 }
32
33 async cancel() {
34     await this.stop();
35 }
36
37 get state() {
38     return this.recorder?.state;
39 }
40
41 async start() {
42     if (this.isRecording) return;
43     this.isRecording = true;
44     this.azureSTT = await sttFromMic((text, delta, done) => {
45         if (this.onAudioTranscription) {
46             this.onAudioTranscription({
47                 text,
48                 delta,
49                 done,
50             });
51         }
52     });
53     if (!this.initialized) await this.init();
54 }
55
56 async stop() {
57     if (!this.isRecording) return;
58     this.isRecording = false;
59     return new Promise(resolve => {
60         setTimeout(() => {
61             this.azureSTT.close();
62             if (!this.azureSTT.startRecord) {
63                 this.onAudioTranscription &&
64                 this.onAudioTranscription({
65                     text: '',
66                     delta: '',
67                     done: true,
68                 });
69             }
70             resolve(null);
71         }, 500);
72     });

```

```
73   }
74
75   async destroy() {
76     this.onAudioTranscription = null;
77     await this.stop();
78   }
79 }
```

接着我们在 MessageInput 组件中引入语音输入功能。

我们修改 MessageInput.vue，添加后面的内容，后面的代码你对照注释也很容易理解。

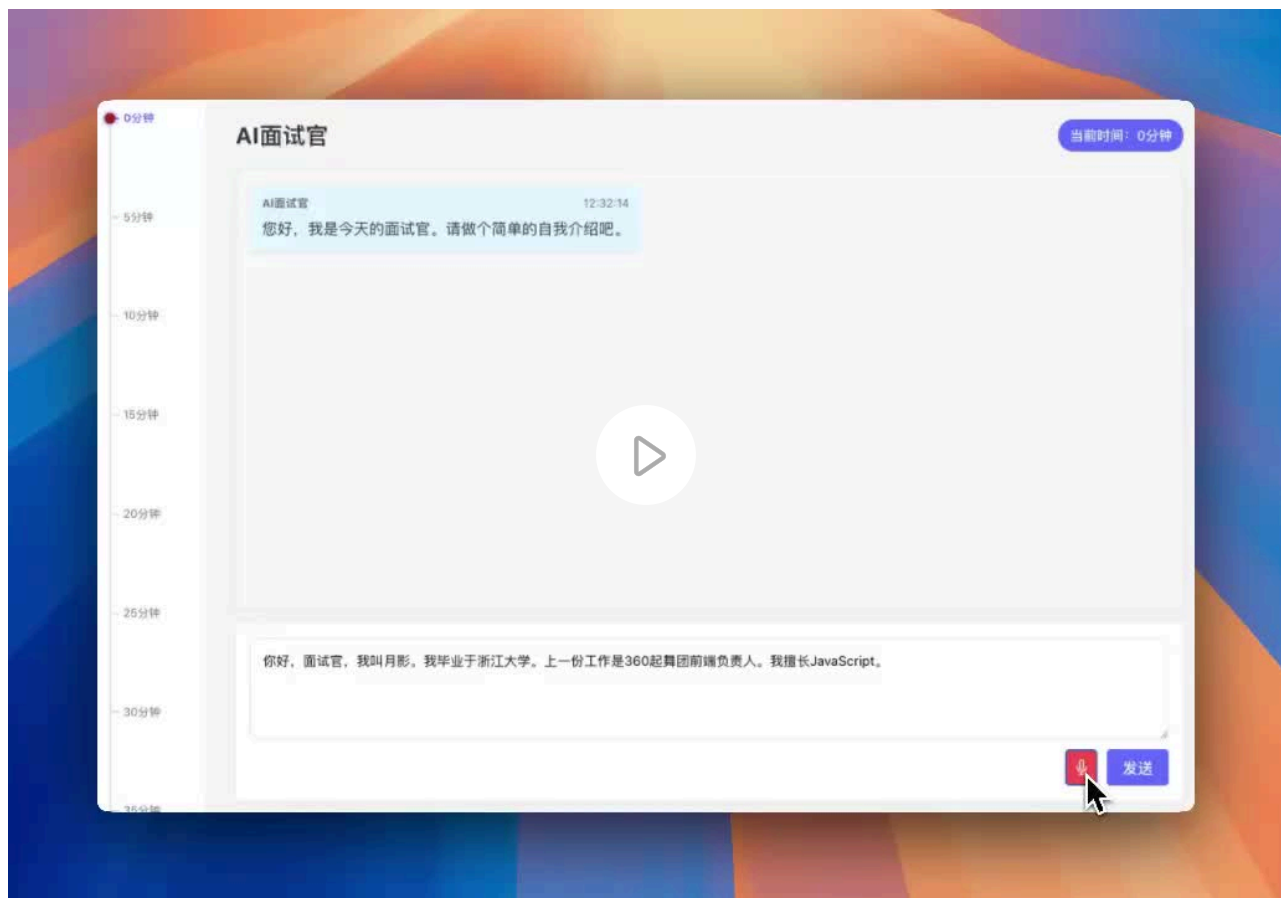
 复制代码

```
1  <script setup lang="ts">
2  import { ref, onUnmounted } from 'vue';
3  import { Recognizer } from '../lib/voice';
4  ...
5
6  const isRecording = ref(false);
7  const recognizer = ref<Recognizer | null>(null);
8
9  // 初始化语音识别器
10 const initRecognizer = () => {
11   if (!recognizer.value) {
12     recognizer.value = new Recognizer();
13     recognizer.value.onAudioTranscription = (transcription) => {
14       if (transcription.text) {
15         message.value = transcription.text;
16       }
17     };
18   }
19 };
20
21 // 开始录音
22 const startRecording = async () => {
23   initRecognizer();
24   await recognizer.value?.start();
25   isRecording.value = true;
26 };
27
28 // 停止录音
29 const stopRecording = async () => {
30   if (!isRecording.value) {
31     return;
```

```
32   }
33   isRecording.value = false;
34   setTimeout(() => {
35     recognizer.value?.stop();
36   }, 1000);
37 };
38
39 // 处理语音按钮按下事件
40 const handleVoiceButtonDown = () => {
41   startRecording();
42 };
43
44 // 处理语音按钮释放事件
45 const handleVoiceButtonUp = () => {
46   stopRecording();
47 };
48
49 ...
50
51 // 组件卸载时销毁语音识别器
52 onUnmounted(() => {
53   recognizer.value?.destroy();
54 });
55 </script>
56
57 <template>
58 ...
59   <div class="button-container">
60     <button
61       class="voice-button"
62       @mousedown="handleVoiceButtonDown"
63       @mouseup="handleVoiceButtonUp"
64       @mouseleave="handleVoiceButtonUp"
65       :class="{ 'recording': isRecording }"
66       title="长按进行语音输入"
67     >
68       <svg xmlns="http://www.w3.org/2000/svg" width="16" height="16" viewBox="0
69         <path d="M12 1a3 3 0 0 0-3 3v8a3 3 0 0 0 6 0V4a3 3 0 0 0-3-3z"></path>
70         <path d="M19 10v2a7 7 0 0 1-14 0v-2"></path>
71         <line x1="12" y1="19" x2="12" y2="23"></line>
72         <line x1="8" y1="23" x2="16" y2="23"></line>
73       </svg>
74     </button>
75     <button class="send-button" @click="handleSendMessage" :disabled="buttonDis
76   </div>
77 ...
78 </template>
79
80 <style scoped>
```

```
81 .button-container {
82   display: flex;
83   justify-content: flex-end;
84   gap: 10px;
85 }
86
87 .voice-button {
88   padding: 8px;
89   background-color: #f5f5f5;
90   color: #646cff;
91   border: 1px solid #ddd;
92   border-radius: 4px;
93   cursor: pointer;
94   transition: all 0.2s;
95   display: flex;
96   align-items: center;
97   justify-content: center;
98 }
99
100 .voice-button:hover {
101   background-color: #eaeaea;
102 }
103
104 .voice-button.recording {
105   background-color: #ff4d4f;
106   color: white;
107   border-color: #ff4d4f;
108   animation: pulse 1.5s infinite;
109 }
110
111 @keyframes pulse {
112   0% {
113     box-shadow: 0 0 0 0 rgba(255, 77, 79, 0.4);
114   }
115   70% {
116     box-shadow: 0 0 0 10px rgba(255, 77, 79, 0);
117   }
118   100% {
119     box-shadow: 0 0 0 0 rgba(255, 77, 79, 0);
120   }
121 }
122 </style>
```

这样我们就在界面上添加了语音输入的 UI 效果，实际效果你可以对照后面的演示视频看一下。



提高容错性

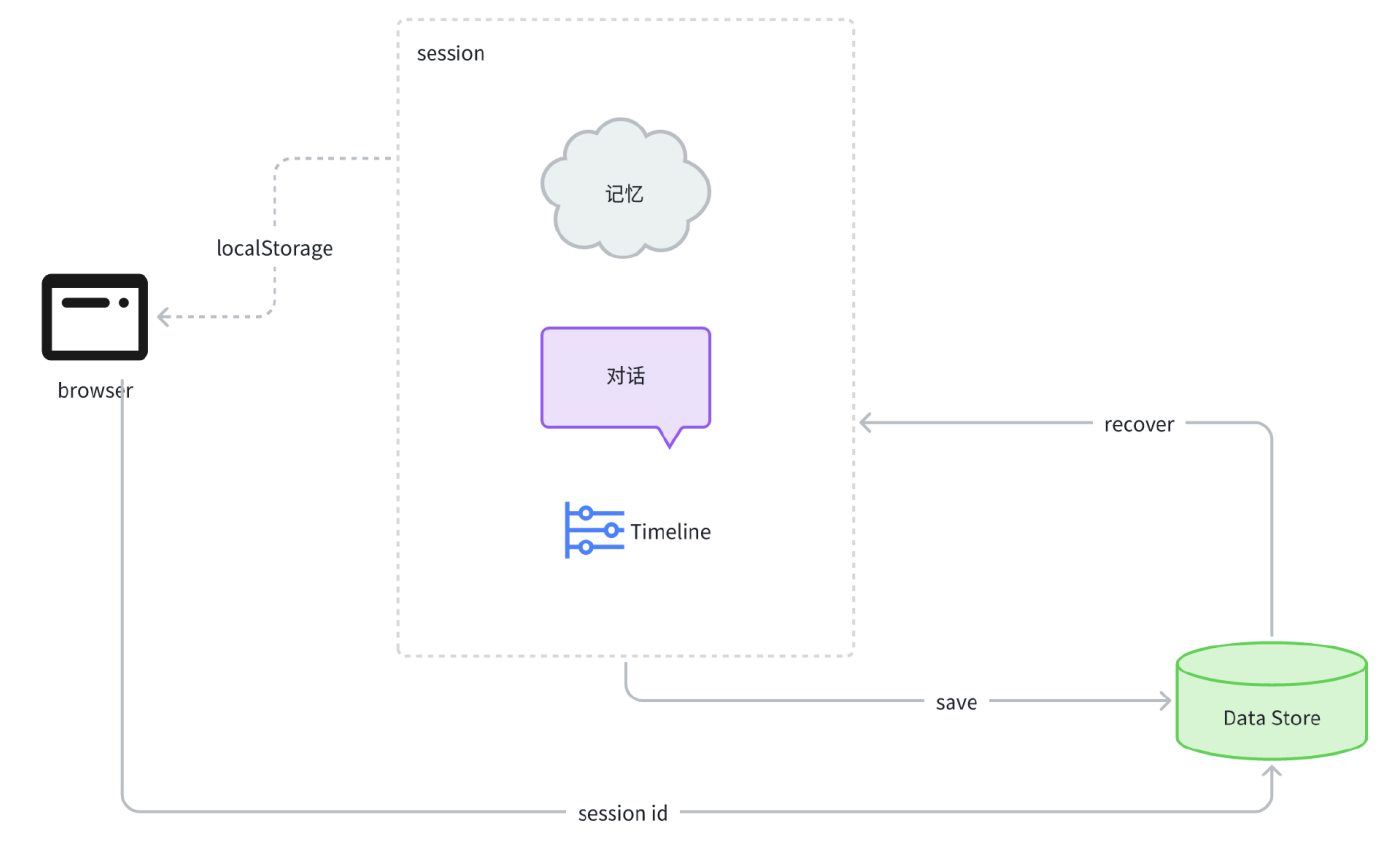
到此为止，正常的前端面试功能我们已经完全实现了，大家可以体验看看，体会一下这个面试官的水平到底如何。

这个面试官智能体，作为一个 Demo 演示，它的功能是比较完整的，但是作为一款用于模拟面试的成熟产品，肯定还有很多需要优化的地方。其中最大的问题是容错性，我们这就来看看有什么方式做改善。

面试记录的存储及恢复

因为面试过程通常时间较长，所以用户在这个过程中可能会中断面试，此时我们应该将当前 Timeline、Memory、以及 History 记录到持久化的存储对象中，可以是数据库或者 Redis，以便于用户在适当的时候恢复面试。当用户处理完手头事情，回到产品中，可以通过前端持久化的 session id 来读取服务器持久化存储的内容，然后继续面试。

前端持久化，可以用 Vue 的 store 持久化机制来实现，比如 [pinia](#) 就是不错的选择。鉴于课程内容主要还是聚焦在 AI 方向，因此与 AI 无关的细节就不再展开细讲了，我将这套机制画成下面的流程图，就留给有兴趣的同学们自行研究和完善吧。



控制时间线，跳过当前面试阶段

在实际面试中，如果候选人表现正常，那么面试通常是按照 Timeline 的节奏来进行的，但如果候选人表现在正常范围之外，比如某个阶段表现特别差，或者特别完美，那么真实面试中，面试官是有可能跳过这一面试阶段的，那么如何跳过面试阶段呢？实际上有很多办法，这里我提供一个思路，有兴趣的同学可以具体研究。

首先我们可以通过 Ling 的 `sendEvent` 能力，通知前端更新 Timeline，具体方法是，我们发送一个 `update-timeline` 事件给前端，参数传递 `timeline` 的值：

```
server.ts
```


```

1  ...
2
3  import { skipToNextSection } from './lib/config/timeline.config';
4
5  ...
6  const bot = ling.createBot('reply', {}, {
7    response_format: { type: "json_object" },
8  });
9  bot.addFilter("updateTimeline"); // 对前端屏蔽这个字段
10 ...
11 bot.addListener('response', (content) => {
12 ...
13     const message = JSON.parse(content);
14     if(message.updateTimeline) {
15         const newTimeline = skipToNextSection(timeline);
16         ling.sendEvent({event: 'update-timeline', delta: newTimeline});
17     }
18 });
19 ...

```

在前面的代码里，我们根据返回的内容中是否要求更新 timeline 来判断是否跳过面试阶段，如果跳过，则调用 skipToNextSection 获得新的 timeline 值，然后通过 `update-timeline` 事件发送给前端，前端获得 timeline 数据后更新该数值即可。

注意如果这么修改，那么 bot 返回格式就**不能是 text 而应该是 json_object** 了，输出内容也应该是修改成后面这样的 JSON 结构。


 复制代码

```

1  {
2    "reply": "",
3    "updateTimeline": false
4  }

```

在系统提示词中，可以插入如下规则。

 复制代码

```

1  根据 memory 和当前用户回答进行判断，如果该阶段用户回答得非常好，或者回答得非常差，那么输出 `upc

```


这样我们就可以进行控制了。具体的细节有兴趣的同学可以自行实现。

产品化及其他细节优化

为了产品化，我们还要做一些事情，比如将 Job Description 配置项抽取出来，以便于针对不同岗位，让 AI 有侧重点地去问问题。另外还可以将 AI 的人设也抽取出来，赋予面试官不同的性格偏好。还有我们面试结束时，也需要通过 Memory 中的内容，进行整体的面试回顾和评估，让 AI 给出面试结论。

此外，在前端体验上，我们面试过程中有写代码环节，在前面初始化项目的时候，我们也添加了 vue-codemirror。我们在前端可以添加输入切换，让输入过程可以切换输入框为多行文本或者 CodeMirror 编辑器，以便于获得更好的体验，这些产品细节都是需要花时间去慢慢打磨的。

这些内容不在我们课程核心内容的范畴内，因此我将这些任务留给大家，希望大家多去思考，多去练习。因为前端技能只有在不断实战练习中，才能得到巩固和提升。

要点总结

这一节课，我们讲了异步更新记忆的前端优化，以及添加语音输入功能，这些改进能够进一步优化面试体验。

我们还了解了进一步优化产品的方向，例如通过支持面试记录的持久化存储来支持面试过程的中断与恢复，以及通过控制时间线允许跳过当前阶段来更好地掌控面试流程，这些改进提高了产品容错性，在自身服务异常或者用户胡乱回答的时候，不至于出现大的问题。

最后，我们探讨了一些可能的产品化和其他细节优化方向，至此我们这个单元就基本上结束了。希望大家通过这个实战例子理解并掌握了复杂对话流程的智能体实现的思路。

从下一节课开始，我们将进入一个新的单元，讨论 AI 如何助力个人成长，这是一个全新的，但也是非常关键的话题，我相信这个话题对你的职业成长将会非常有帮助。

课后练习

可能有细心的同学会发现，我们在 22 节课系统设计里，提出的架构，除了对话流、记录与分析，还有一个慢思考的过程，但是我们在后续的章节里并没有实现这个慢思考的过程。

实际上我们并不是没有实现，而是通过 Memory 对象结构，用语义化的 JSON 字段，引导了 AI 在记录 Memory 时的思考，相当于将慢思考和记录分析二合一了。而这么做，对于我们 AI 面试官产品在实际运行测试时得到的结果来看，是没什么问题的。当然，如果再进一步增加深度思考，有可能进一步提升 AI 面试官问问题的能力，让它问问题变得更加犀利，面试效果也会更好。

在这里，我就留下这个作业，有兴趣的同学给我们的面试过程 workflows 再添加一个慢思考的大模型节点，结合 Memory 和当前讨论进行思考和分析，从而给对话模型一些建议，让它能够问出更有针对性的问题。

这个任务并不简单，甚至有一点挑战性，你可以花时间尝试一下，有结果可以分享到评论区。后续有机会的话，我可能会通过加餐或者其他方式，带着大家进一步深入探讨。

AI智能总结

1. 实现语音输入功能，以提升面试对话的体验。
2. 对代码进行了修改，包括给Ling对象增加了max_tokens参数，设置为8192，给面试对话的bot增加了一组最近的聊天记录，以及在bot输出结束时，给前端发送一个response-finished事件。
3. 异步更新记忆及前端优化，包括对话同时更新记忆、传递最近的一部分历史对话给大模型以防止AI对话丢失最近的上下文信息，以及在前端进行UI处理以防止用户重复提交。
4. 控制时间线，跳过当前面试阶段，通过Ling的sendEvent能力，通知前端更新Timeline，具体方法是发送一个`update-timeline`事件给前端，参数传递timeline的值。
5. 产品化及其他细节优化，包括将Job Description配置项抽取出来，将AI的人设抽取出来，以及通过Memory中的内容进行整体的面试回顾和评估，让AI给出面试结论。
6. 在前端体验上，添加输入切换，让输入过程可以切换输入框为多行文本或者CodeMirror编辑器，以便于获得更好的体验。
7. 优化产品的方向，支持面试记录的持久化存储来支持面试过程的中断与恢复，以及通过控制时间线允许跳过当前阶段来更好地掌控面试流程。
8. 课后练习，通过Memory对象结构，用语义化的JSON字段，引导AI在记录Memory时的思考，结合Memory和当前讨论进行思考和分析，从而给对话模型一些建议，让它能够问出更有针对性的问题。

© 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。

