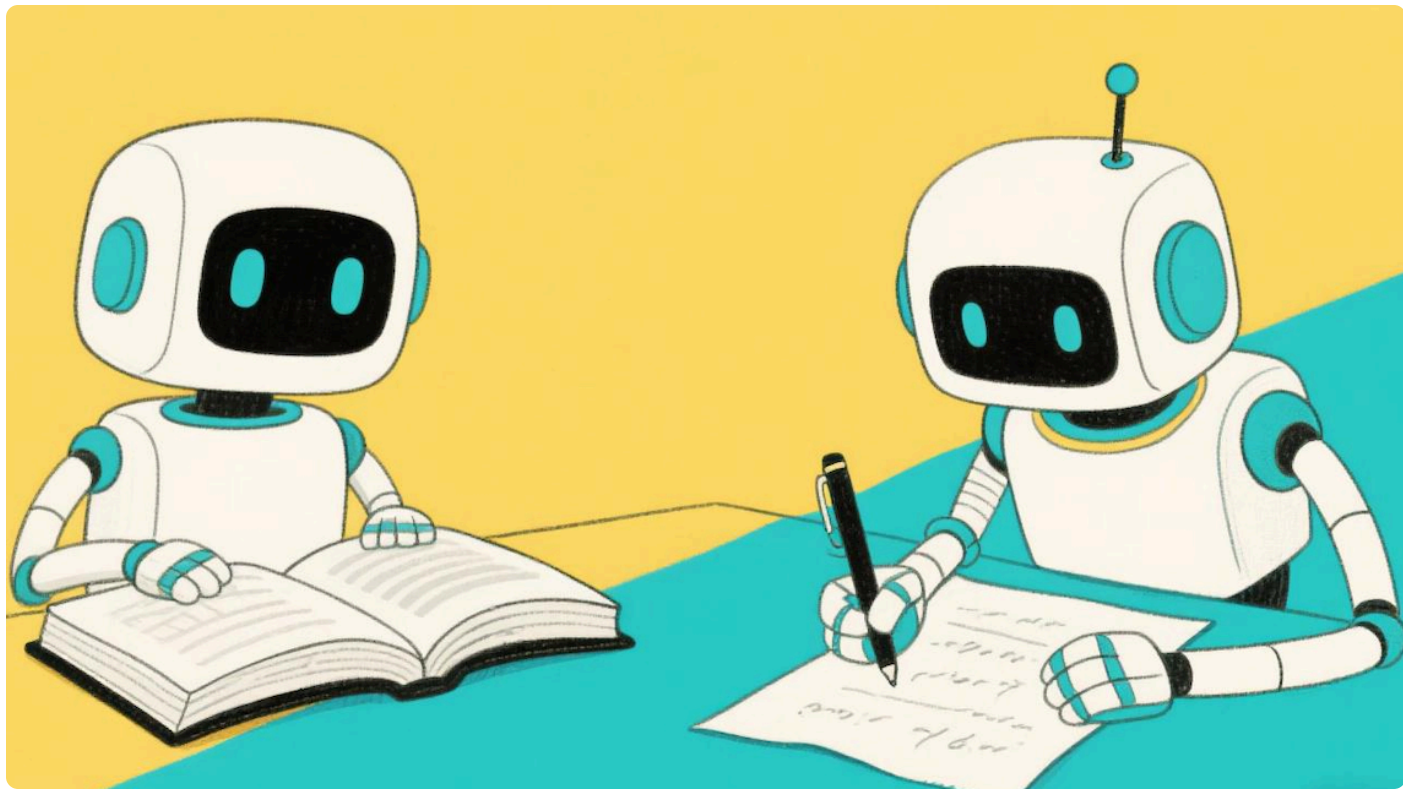


24 | 多轮对话：API 读写分离设计

月影 · 跟月影学前端智能体开发



你好，我是月影。

接下来我们继续实现复杂的多轮对话。在这一讲里，我们在深入业务前，需要解决一个技术难题。

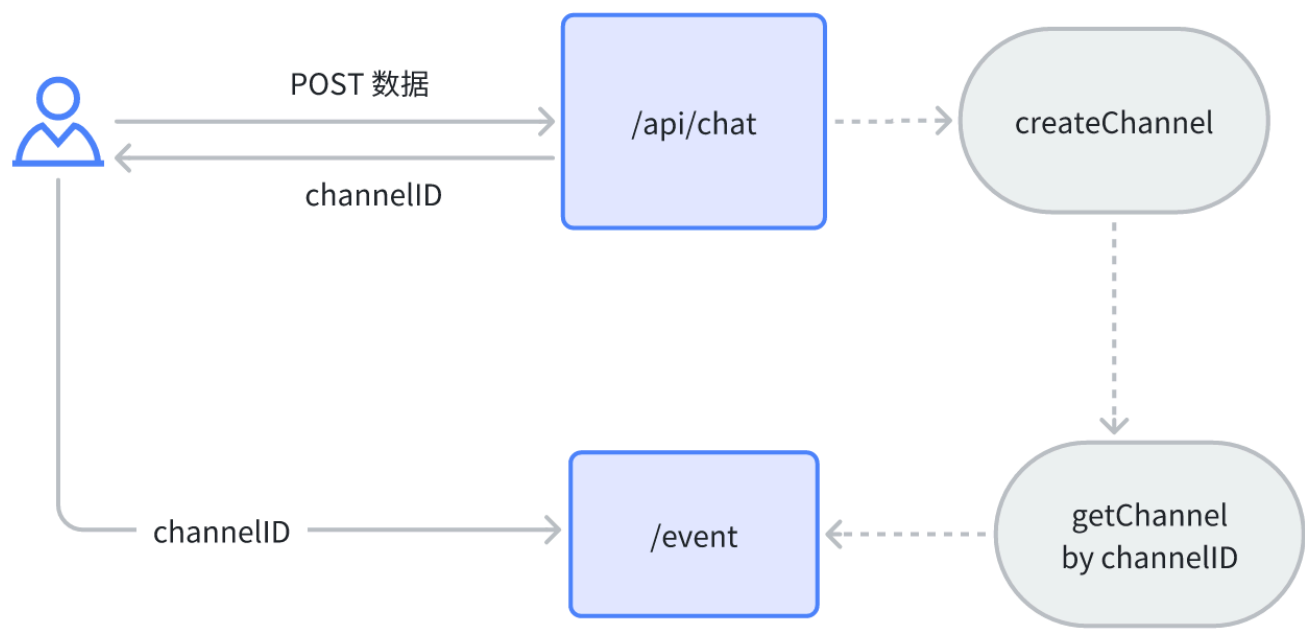
还记得我们在使用 Ling 的时候，推荐大家用 Server-Sent Events 模式，因为 Server-Sent Event 是现代浏览器支持的协议，前端处理比较简单。

但是，Server-Sent Events 有个弊端，那就是根据规范，它只支持 GET 请求。这是因为，SSE 的设计初衷就是客户端建立一个**单向连接**从服务器接收事件，而这个连接只能通过 GET 请求发起，因为它用来“获取资源”的，服务器通过流式传输的方式持续返回数据。

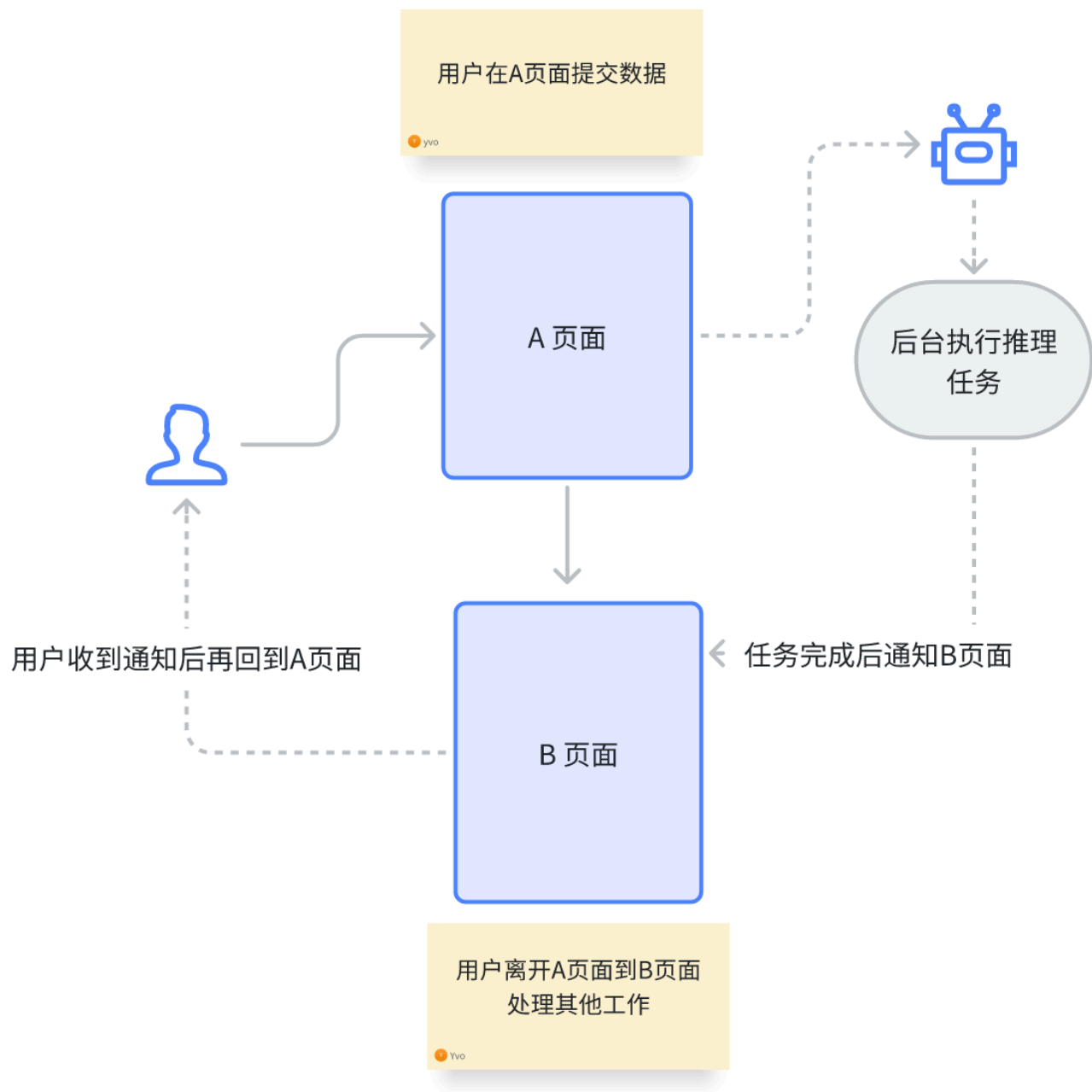
对于前面的波波熊学伴产品而言，由于孩子问的问题内容通常比较简单，所以用 GET 请求的问题不大。因为简单的参数我们完全可以通过浏览器的 URL 查询传递参数。

然而对于面试官这种需求，因为我们提交回答的内容可能会很长，甚至未来有可能提交图片之类非文本的内容，所以通过 GET 的方式就不再合适，我们需要采用 POST 方法提交数据。

那么我们这个时候就要做一个**读写分离**的处理，通过 POST 请求提交数据，建立 EventSource 通道，然后再通过通道来读取流式数据。



在 AI 应用的技术架构中，这实际上是非常常用的一种接口设计范式。因为通常 GET 请求处理 AI 的流式推理的时间，远比 POST 提交用户请求和数据的时间要长。这么设计之后，我们可以让读取数据不会阻塞整个业务流程，也就意味着用户可以提交数据后离开页面，等待 server 的更新，server 更新完成后，再通过轮询或者浏览器自身的消息机制告诉用户。



实现读写分离设计

了解了读写分离的好处，那我们具体来看看，要怎么设计读写分离的结构。

首先，我们打开 Trea IDE，在 Get Offer AI 的项目下，创建 `/lib/service/eventChannel.ts` 模块。

在这里，我们创建一个类 EventChannel：

```
1 class EventChannel {
2   private frequency: number = 100; // 100 ms 异步刷新
3   private eventBuffer: string[] = [];
4   public id: string;
5
6   init(ling: Ling): { message: string; code: number; channel: string;}
7   {
8     ...
9   }
10
11  getStream(
12    lastEventId?: string,
13  ): { stream: ReadableStream; controller: ReadableStreamDefaultController | null
14    ...
15  }
16 }
```

这个类主要有两个方法，其中一个 `init` 方法，它接受 `Ling` 对象实例作为参数，返回一个 JSON 对象，其中 `channel` 参数就是我们后续要使用的通道 ID。

另一个是 `getStream` 方法，因为 Server-Sent Events 自带中断恢复机制，所以它可以接受一个额外的 `lastEventId`，让它从指定的 event 之后的数据再继续发送。

我们看一下具体实现，就可以理解它们的用途。

首先是 `init` 方法：

```
1   init(ling: Ling): { message: string; code: number; channel: string;} {
2     this.id = Math.random().toString(36).substring(2, 10);
3     ling.on('finished', () => {
4       try {
5         // @ts-ignore
6         ling.tube?.controller?.close();
7       } catch (ex) {}
8     });
9
10    const reader = ling.stream.getReader();
11    const events = this.eventBuffer;
```

```

12     reader.read().then(async function processText({ done: _done, value }): Promis
13         if (_done) {
14             return;
15         }
16         events.push(value);
17         return reader.read().then(processText);
18     }).catch((ex) => {
19         console.error(ex);
20     });
21
22     return {
23         message: 'success',
24         code: 201,
25         channel: this.id,
26     }
27 }


```

在 `init` 方法中，我们通过 `ling.stream.getReader()` 得到 `Reader` 对象，然后通过 `reader.read()` 异步读取 `Ling` 发出的流数据，将它们先暂时 `push` 到 `eventBuffer` 中，也就是直接放到了内存中。

注意，在实际的生产环境中，我们通常不会在内存中管理这些数据，而是将它们放到 `Redis` 之类的持久化存储对象中，但是在本课程中，为了让你聚焦主线，这些持久化存储方案就不做展开了，有兴趣的同学，在课后练习中可以自行研究。

这个过程是个异步的过程，但我们不必等待过程处理完，因此我们这里并没有用 `await`，而是直接将随机生成的 `channel` 返回。

接着我们看 `getStream` 方法，它的实现稍微复杂一些，代码如下：

 复制代码

```

1  getStream(
2      lastEventId?: string,
3  ): { stream: ReadableStream; controller: ReadableStreamDefaultController | null
4      let controller: ReadableStreamDefaultController | null = null;
5      const stream = new ReadableStream({
6          start(_controller: ReadableStreamDefaultController) {
7              controller = _controller;
8          },
9      });

```

```
10     const events = this.eventBuffer;
11
12
13     let i = 0;
14     let isFinished = false;
15
16     if (lastEventId) {
17         for( let j = 0; j < events.length; j++) {
18             const event = events[i];
19             if (event.startsWith('data: {"event":"finished"}')) {
20                 isFinished = true;
21                 break;
22             }
23             if (event.includes(`id: ${lastEventId}`)) {
24                 i = j + 1;
25                 break;
26             }
27         }
28     }
29
30     for(; i < events.length; i++) {
31         const event = events[i];
32         if (event.startsWith('data: {"event":"finished"}')) {
33             isFinished = true;
34             break;
35         }
36         if (controller) {
37             (controller as ReadableStreamDefaultController).enqueue(event);
38         }
39     }
40
41
42     (async () => {
43         // 等待更新
44         while (true) {
45             if (isFinished) {
46                 break;
47             }
48             await sleep(this.frequency);
49             for (; i < events.length; i++) {
50                 const event = events[i];
51                 if (event.startsWith('data: {"event":"finished"}')) {
52                     isFinished = true;
53                 }
54                 if (controller) {
55                     (controller as ReadableStreamDefaultController).enqueue(event);
56                 }
57             }
58         }
```


```

59     // console.log(events);
60     this.eventBuffer = [];
61     if (controller) {
62         (controller as ReadableStreamDefaultController).close();
63     }
64     delete EventChannelMap[this.id];
65 }());
66
67 return { stream, controller }
68 }

```

首先我们创建个 ReadableStream 对象，同时获取它内部的 Controller 对象。接着我们对缓存的 eventBuffer 数组进行逐条处理。

如果有 lastEventId，那么我们要先略过之前的数据。这个情况通常发生在前端读取数据过程中网络临时中断。

 复制代码

```

1  if (lastEventId) {
2      for( let j = 0; j < events.length; j++) {
3          const event = events[i];
4          if (event.startsWith('data: {"event":"finished"}')) {
5              isFinished = true;
6              break;
7          }
8          if (event.includes(`id: ${lastEventId}`)) {
9              i = j + 1;
10             break;
11         }
12     }
13 }

```

注意，我们判断消息是否传输结束的条件是看有没有 `data: {"event":"finished"}` 的数据。如果你还记得第二单元讲过的 Ling 框架的内容，那么应该知道这是 Ling 框架内部的约定。后面的处理也是一样，通过这个数据来判断消息是否发送完。

接着我们看一下存量的数据，因为 init 之后，Ling 对象会不断将数据发送到 eventBuffer 中，所以在我们 getStream 函数调用的时候，eventBuffer 里面很可能已经有一部分数据。


```
1  for(; i < events.length; i++) {
2      const event = events[i];
3      if (event.startsWith('data: {"event":"finished"}')) {
4          isFinished = true;
5          break;
6      }
7      if (controller) {
8          (controller as ReadableStreamDefaultController).enqueue(event);
9      }
10 }
```

然后我们等待增量数据到来。这里我们采用一个异步轮询，每 100 毫秒轮询一次，这是一种常用的方式，它可能不是最好的，但是最简单，在大部分场景下就够用了。

```
1  (async () => {
2      // 等待更新
3      while (true) {
4          if (isFinished) {
5              break;
6          }
7          await sleep(this.frequency);
8          for (; i < events.length; i++) {
9              const event = events[i];
10             if (event.startsWith('data: {"event":"finished"}')) {
11                 isFinished = true;
12             }
13             if (controller) {
14                 (controller as ReadableStreamDefaultController).enqueue(event);
15             }
16         }
17     }
18     // console.log(events);
19     this.eventBuffer = [];
20     if (controller) {
21         (controller as ReadableStreamDefaultController).close();
22     }
23     delete EventChannelMap[this.id];
24 })();
25
```


注意我们这里的一个用法细节，我们用一个 `(async () => {})()` 的异步匿名函数将这部分代码套起来，实现异步过程。这是因为 `getStream` 的过程也是异步处理的，我们不需要在这里 `await` 中间的处理过程，所以我们并没有将 `getStream` 声明成一个 `async` 的函数。

好了，这样我们就完成了 `EventChannel` 对象的设计，接着我们只要暴露两个模块方法：

 复制代码

```
1 const EventChannelMap: Record<string, EventChannel> = {};  
2  
3 export function getEventChannel(id: string) {  
4   return EventChannelMap[id];  
5 }  
6 export function createEventChannel(ling: Ling) {  
7   const channel = new EventChannel();  
8   const ret = channel.init(ling);  
9   EventChannelMap[channel.id] = channel;  
10  return ret;  
11 }
```

我们用 `EventChannelMap` 来存储所有的 `EventChannel` 对象到内存中，随机生成的 ID 作为 `channel` 的唯一标识。`getEventChannel` 根据 `channel` 的 `id` 获取指定 `EventChannel` 对象，`createEventChannel` 则创建新的 `EventChannel` 对象。

这样我们就可以在 `server` 中使用 `EventChannel` 了。

我们修改 `server.ts`，增加两个方法：

 复制代码

```
1 import { createEventChannel, getEventChannel } from './lib/service/eventChannel';  
2 import { pipeline } from 'node:stream/promises';  
3  
4 ...  
5  
6 app.post('/chat', async (req, res) => {  
7   const { message, sessionId, timeline } = req.body;  
8   const config = {  
9     model_name: modelName,  
10    api_key: apiKey,
```

```

11     endpoint: endpoint,
12     sse: true,
13 };
14 // ----- The work flow start -----
15 const ling = new Ling(config);
16 const bot = ling.createBot();
17 bot.chat(message);
18
19 ling.close();
20
21 res.send(createEventChannel(ling));
22 });
23
24 app.get('/event', (req, res) => {
25     const lastEventId = req.headers['last-event-id'] as string | undefined;
26     const eventChannel = getEventChannel(req.query.channel as string);
27
28     res.setHeader('Content-Type', 'text/event-stream');
29     res.setHeader('Cache-Control', 'no-cache');
30     res.setHeader('Connection', 'keep-alive');
31     res.setHeader('Access-Control-Allow-Origin', '*');
32     res.flushHeaders();
33
34     const { stream, controller } = eventChannel.getStream(lastEventId);
35     try {
36         pipeline(stream as any, res);
37     } catch (ex) {
38         console.log(ex);
39         controller?.close();
40     }
41 });

```

首先是 /chat 方法，它是一个 POST 方法，接受 message、sessionId 和 timeline 三个参数。这里我们目前先只用到了 message，没关系，后面课程我们会继续使用 sessionId 和 timeline。


我们先试一下 EventChannel 的效果，所以我们创建一个 Ling 对象，创建一个默认的 Bot 对象，将 message 发送给它。然后我们通过 createEventChannel(ling) 创建一个通道，并将结果通过 res.send 返回给前端。

这样，我们在前端能拿到 channel ID，就可以调用 /event 方法，它是流式 GET 方法。首先我们通过 req.headers['last-event-id'] 判断一下是否存在 lastEventId，这是浏览器默认的

SSE 续传机制。接着我们通过传递来的 channel 参数获取对应的 eventChannel 对象。然后我们通过 getStream 方法获取 stream 和 controller 对象，再通过 pipeline 将内容发送给前端。

这样我们就实现了服务端的逻辑。

接着我们处理前端部分。我们这样修改 App.vue。

 复制代码

```
1  const sessionId = Math.random().toString(36).slice(2, 9);
2
3  // 发送新消息
4  const sendMessage = async (content: string) => {
5    const newMessage: Message = {
6      id: messages.value.length + 1,
7      sender: 'user',
8      content,
9      timestamp: new Date()
10   };
11
12   messages.value.push(newMessage);
13
14   const res = await fetch('/api/chat', {
15     method: 'POST',
16     headers: {
17       'Content-Type': 'application/json'
18     },
19     body: JSON.stringify({
20       message: newMessage.content,
21       sessionId,
22       timeline: currentTime.value
23     })
24   });
25
26   const { channel } = await res.json();
27
28   const eventSource = new EventSource(`/api/event?channel=${channel}`);
29   let aiResponse: Message = {
30     id: messages.value.length + 1,
31     sender: 'ai',
32     content: '',
33     timestamp: new Date()
34   };
35   eventSource.addEventListener("message", function (e: any) {
```

```


36     let { uri, delta } = JSON.parse(e.data);
37     if (aiResponse.content === '') {
38         aiResponse.content = delta;
39         messages.value.push(aiResponse);
40     } else {
41         aiResponse.content += delta;
42         messages.value = [...messages.value]; // 强制更新
43     }
44     console.log(uri, delta);
45 });
46 eventSource.addEventListener('finished', () => {
47     console.log('传输完成');
48     eventSource.close();
49 });
50 };
51

```

当用户在面试输入框中输入文字，点击发送后，我们首先通过 `/api/chat` 拿到 channel，然后再用 `/api/event?channel=${channel}` 去请求真正的数据，最终处理数据，通过 `mesages.value.push(aiResponse)` 更新数据。

这里我们还设计了一个 ChatDisplay 的 Vue 组件来显示聊天记录，它的具体代码如下：

ChatDisplay.vue

 复制代码

```

1  <script setup lang="ts">
2  import { ref, watch } from 'vue';
3
4  interface Message {
5      id: number;
6      sender: 'ai' | 'user';
7      content: string;
8      timestamp: Date;
9  }
10
11  const props = defineProps<{
12      messages: Message[];
13  }>();
14
15  // 创建聊天容器的引用
16  const chatDisplayRef = ref<HTMLElement | null>(null);

```

```

17 const emit = defineEmits(['content-change']);
18
19 // 记录是否已经触发过内容变化事件
20 const hasEmittedContentChange = ref(false);
21
22 // 监听消息变化，自动滚动到底部并在首次变化时触发事件
23 watch(() => props.messages, (newMessages, oldMessages) => {
24   scrollToBottom();
25
26   // 如果消息数量变化且之前没有触发过内容变化事件
27   if (newMessages.length > 1 && !hasEmittedContentChange.value) {
28     emit('content-change');
29     hasEmittedContentChange.value = true;
30   }
31 }, { deep: true });
32
33 // 滚动到底部的方法
34 const scrollToBottom = () => {
35   setTimeout(() => {
36     if (chatDisplayRef.value) {
37       chatDisplayRef.value.scrollTop = chatDisplayRef.value.scrollHeight;
38     }
39   }, 0);
40 };
41 </script>
42
43 <template>
44   <div class="chat-display" ref="chatDisplayRef">
45     <div v-if="messages.length === 0" class="empty-state">
46       <p>面试即将开始，请准备好...</p>
47     </div>
48     <div v-else class="message-list">
49       <div
50         v-for="message in messages"
51         :key="message.id"
52         class="message"
53         :class="message.sender"
54       >
55         <div class="message-header">
56           <span class="sender">{{ message.sender === 'ai' ? 'AI面试官' : '候选人' }}
57           <span class="timestamp">{{ new Date(message.timestamp).toLocaleTimeStri
58         </div>
59         <div class="message-content">
60           {{ message.content }}
61         </div>
62       </div>
63     </div>
64   </div>
65

```


```
66 </template>
67
68 <style scoped>
69 .chat-display {
70   height: 100%;
71   overflow-y: auto;
72   padding: 16px;
73   background-color: #f9f9f9;
74   border-radius: 8px;
75   box-shadow: inset 0 0 5px rgba(0, 0, 0, 0.1);
76   text-align: left;
77 }
78
79 .empty-state {
80   height: 100%;
81   display: flex;
82   justify-content: center;
83   align-items: center;
84   color: #888;
85   font-style: italic;
86 }
87
88 .message-list {
89   display: flex;
90   flex-direction: column;
91   gap: 16px;
92   margin-bottom: 40px;
93 }
94
95 .message {
96   max-width: 80%;
97   padding: 12px;
98   border-radius: 8px;
99   box-shadow: 0 1px 2px rgba(0, 0, 0, 0.1);
100 }
101
102 .message.ai {
103   align-self: flex-start;
104   background-color: #e1f5fe;
105   border-bottom-left-radius: 0;
106 }
107
108 .message.user {
109   align-self: flex-end;
110   background-color: #e8f5e9;
111   border-bottom-right-radius: 0;
112 }
113
114 .message-header {
```

```

115   display: flex;
116   justify-content: space-between;
117   margin-bottom: 6px;
118   font-size: 12px;
119   color: #666;
120 }
121
122 .sender {
123   font-weight: bold;
124 }
125
126 .message-content {
127   white-space: pre-wrap;
128   word-break: break-word;
129 }
130 /<+>

```

再加上右下方的输入发送消息的组件 MessageInput.vue:

 复制代码

```

1  <script setup lang="ts">
2  import { ref } from 'vue';
3
4  const emit = defineEmits(['sendMessage']);
5
6  const message = ref('');
7
8  const handleSendMessage = () => {
9    if (message.value.trim()) {
10      emit('sendMessage', message.value);
11      message.value = '';
12    }
13  };
14
15  const handleKeydown = (event: KeyboardEvent) => {
16    // Ctrl+Enter 发送消息
17    if (event.ctrlKey && event.key === 'Enter') {
18      handleSendMessage();
19    }
20  };
21 </script>
22
23 <template>
24   <div class="message-input-container">
25     <textarea
26       v-model="message"

```

```
27     class="message-input"
28     placeholder="请输入您的回答..."
29     @keydown="handleKeydown"
30   ></textarea>
31   <button class="send-button" @click="handleSendMessage">发送</button>
32 </div>
33 </template>
34
35 <style scoped>
36 .message-input-container {
37   display: flex;
38   flex-direction: column;
39   gap: 10px;
40   padding: 16px;
41   background-color: #fff;
42   border-top: 1px solid #eee;
43   border-radius: 0 0 8px 8px;
44 }
45
46 .message-input {
47   min-height: 80px;
48   max-height: 120px;
49   padding: 12px;
50   border: 1px solid #ddd;
51   border-radius: 8px;
52   resize: vertical;
53   font-family: inherit;
54   font-size: 14px;
55   line-height: 1.5;
56   outline: none;
57   transition: border-color 0.2s;
58 }
59
60 .message-input:focus {
61   border-color: #646cff;
62   box-shadow: 0 0 0 2px rgba(100, 108, 255, 0.2);
63 }
64
65 .send-button {
66   align-self: flex-end;
67   padding: 8px 16px;
68   background-color: #646cff;
69   color: white;
70   border: none;
71   border-radius: 4px;
72   cursor: pointer;
73   font-weight: 500;
74   transition: background-color 0.2s;
75 }
```



```
76
77 .send-button:hover {
78     background-color: #535bf2;
79 }
80 </style>
```

这两个纯 UI 组件功能比较简单，你可以结合代码看一下，有兴趣的同学也可以自行研究一下它们的实现细节。

这样，我们实际上已经实现了完整的基础对话功能，它的运行效果如下图所示：



要点总结

在这一节课里，我们讲了一个重要的 API 设计范式**读写分离**设计。因为在 AI 应用里，往往 AI 大模型执行推理，发送推理结果给前端的时间远大于前端提交数据或发送指令请求给服务端的时间，所以通常会采用读写分离的设计范式。

要实现读写分离设计，其要点是在处理 POST 请求时，将接收到的流式数据异步写入缓存对象中，该缓存对象可以是内存对象（不推荐），也可以是 Redis 等持久化缓存。在课程里，为了简单起见，我们使用了内存对象。

异步写入数据时，我们将 channel ID 作为唯一标识返回给前端，前端可以通过

```
/api/event?channle=${channel}
```

 这样的方式来异步读取流式数据。

这样既能解决 SSE 限制为 GET 请求的参数传递问题，又可以实现生成内容时不阻塞用户操作，让用户可以继续浏览应用的其他页面，等待内容处理完成后再通知用户，提升用户的体验。

课后练习

前面我们说了，对象缓存在生产环境中一般不会直接用内存，而是用 Redis 等持久化存储对象。你可以试一试，将 EventChannel 修改为使用 Redis 持久化缓存的版本，这个版本对你自己的实际业务也许会有帮助，可以将你的实现版本分享到评论区。

读写分离是一种通用的范式，它也可以用来优化前面我们波波熊学伴的产品，用了读写分离后，我们就能做到用户在生成文章时不必留在页面中等待，而是可以做其他事情，等到生成结束后再收到通知。要实现这个效果，我们应该怎么做，你可以想一想，动手用你的方案来改进波波熊学伴，然后将你的改进结果分享到评论区。

AI智能总结

1. API 读写分离设计解决多轮对话中的技术难题，处理 GET 和 POST 请求的数据提交和读取。
2. 采用读写分离的设计范式在AI应用的技术架构中是常用的，可以提高用户体验，避免阻塞整个业务流程。
3. 通过读写分离设计，用户可以提交数据后离开页面，等待服务器的更新，提高了系统的效率和用户体验。
4. EventChannel 类的设计包括 init 方法和 getStream 方法，用于处理异步读取和存储流数据。
5. 服务端通过 /chat 方法接收消息并创建 EventChannel，通过 /event 方法实现流式 GET 方法，将内容发送给前端。
6. 读写分离设计在多轮对话中的应用，通过 EventChannel 实现异步读取和存储流数据，提高系统效率和用户体验。
7. ChatDisplay 组件用于显示聊天记录，监听消息变化，自动滚动到底部并在首次变化时触发内容变化事件。
8. MessageInput 组件实现了发送消息的功能，提供了简洁的用户界面和交互体验。

全部留言 (1)

最新 精选



机智帅气的小雨

2025-06-15 来自浙江

```
if(lastEventId) {
    for(let j = 0; j < events.length; j++) {
        const event = events[j];
        if(event.startsWith('data {"event":"finished'}')) {
            isFinished = true;
            break;
        }

        if(event.startsWith(`id: ${lastEventId}`)) {
            i = j + 1;
            break;
        }
    }
}
```

这里略过之前的数据，event 应该用到的是 `event = events[j]`，不然结果应该会有异常；源码上同理

