

02 | 如何使用流式传输减少等待时间

月影 · 跟月影学前端智能体开发



你好，我是月影。

前一节课我们使用最简单的 HTTP 协议来处理请求响应，作为前端工程师的你，应该对这块内容并不陌生吧。

接下来呢，我们来了解一下对于初级前端工程师来说稍微复杂一点的内容，那就是通过流式（streaming）的传输方式来使用大模型 API。

为什么要使用流式传输

在具体实践之前，先来说说为什么要使用流式传输。

由于大模型通常是需要实时推理的，Web 应用调用大模型时，它的标准模式是浏览器提交数据，服务端完成推理，然后将结果以 JSON 数据格式通过标准的 HTTP 协议返回给前端，这个我们在上一小节里已经通过例子体会过。

但是这么做有一个问题，**主要是推理所花费的时间和问题复杂度、以及生成的 token 数量有关**。比如像第一节里那样，只是简单问候一句，可能 Deepseek 推理所花费的时间很少，但是如果我们提出稍微复杂一点的要求，比如编写一本小说的章节目录，或者撰写一篇千字的作文，那么 AI 推理的时间会大大增加，这在具体应用中就带来一个显而易见的问题，那就是用户等待的时间。

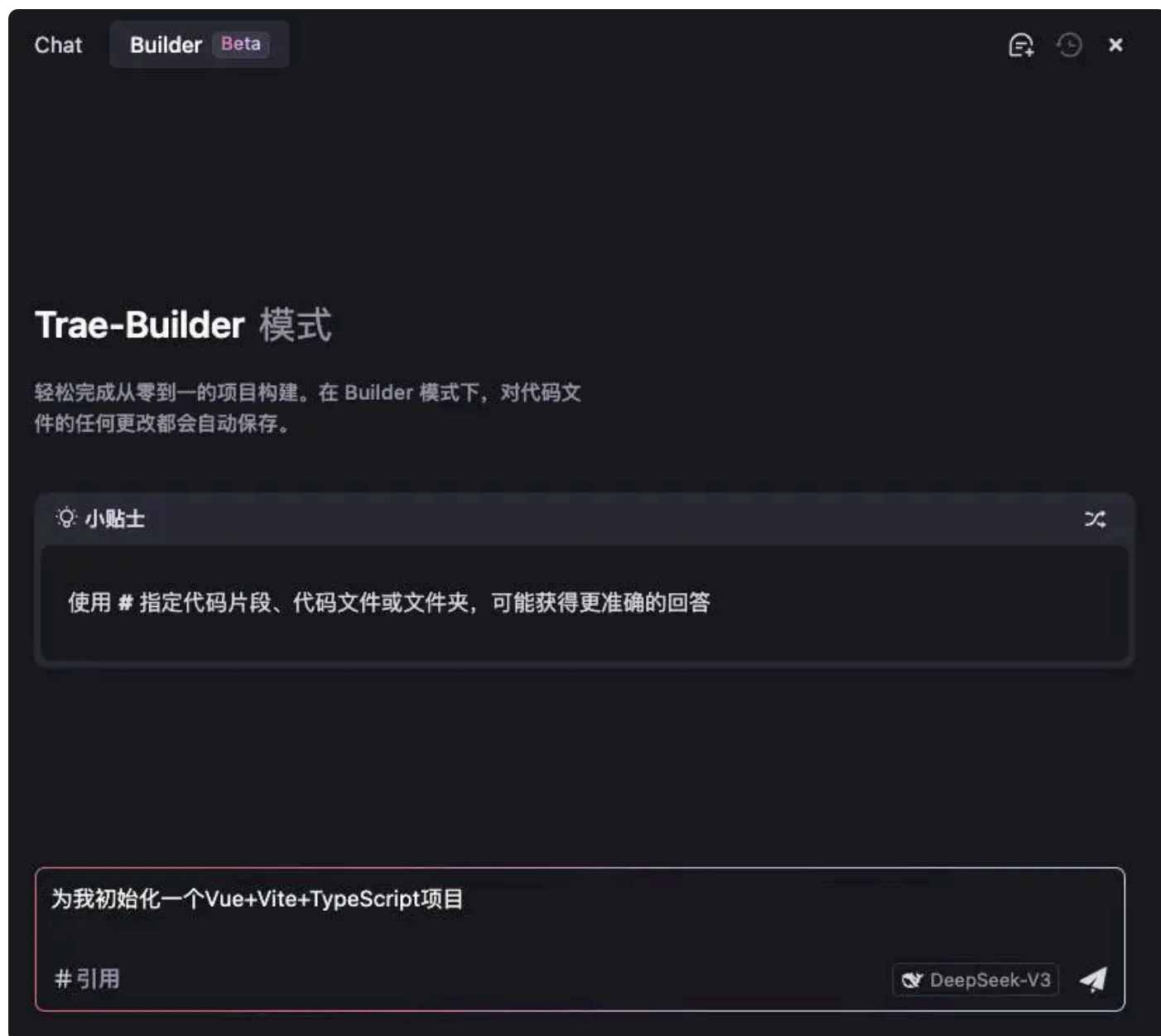
而你应该已经发现，我们在使用线上大模型服务时，不管是哪一家大模型，通常前端的响应速度并没有太慢，这正是因为它们默认采用了流式（streaming）传输，不必等到整个推理完成再将内容返回，而是可以将逐个 token 实时返回给前端，这样就大大减少了响应时间。

如果你是熟悉比较传统的 Web 业务的前端工程师，可能会比较疑惑这种模式具体怎么实现，不要着急，我们接下来通过一个稍微复杂一点例子，来学习和体会这项技术。

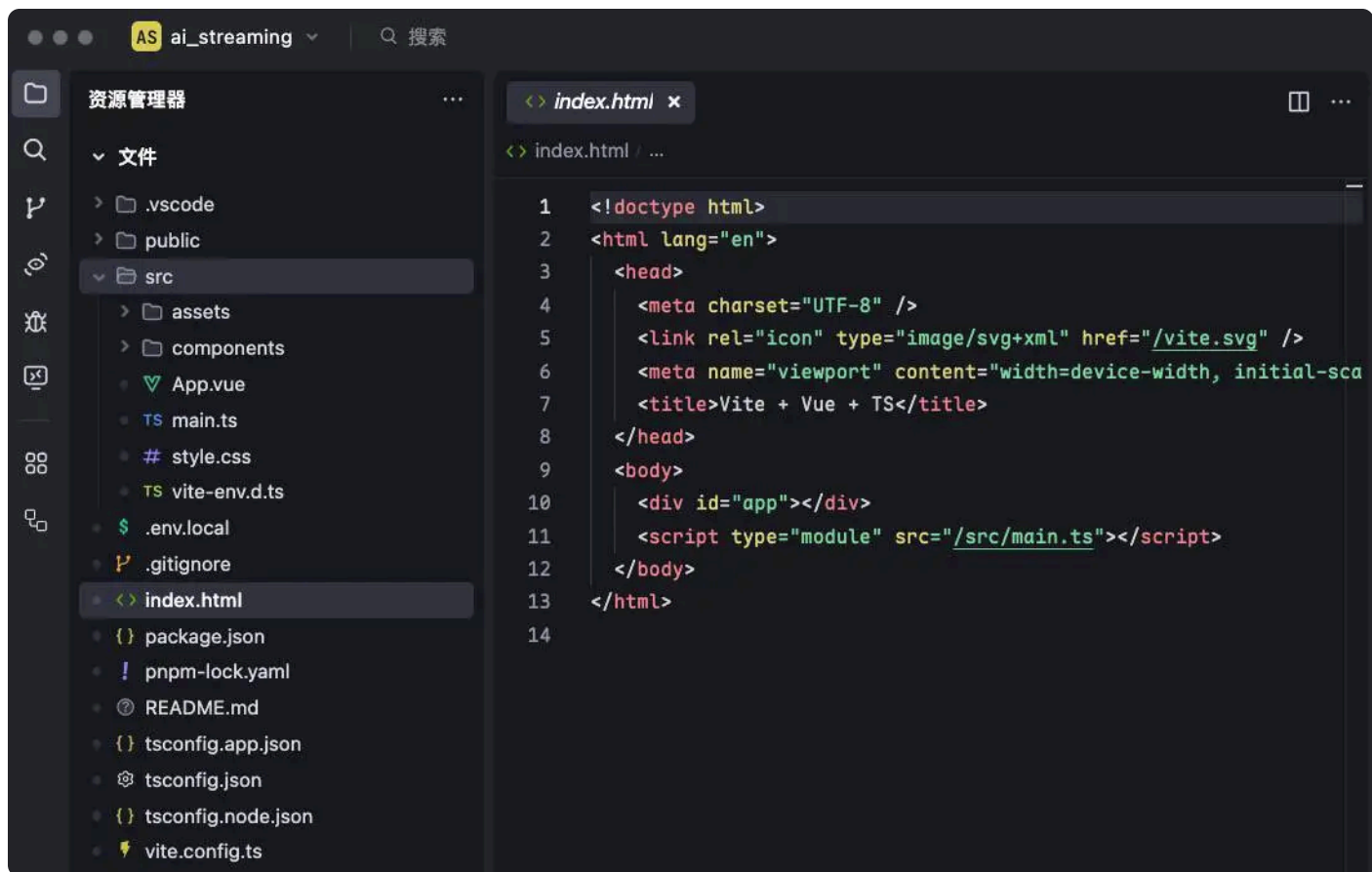
使用流式（streaming）传输减少等待时间

大多数文本模型，都支持使用流式传输来返回内容。在流式传输下，在模型推理过程中，生成的 token 会及时返回，而不用等待推理过程完全结束。在这一小节，我们先看一下 Deepseek Platform 下如何使用流式传输。

首先我们从 Trea 创建一个新项目，这次我们选择创建 Vue+Vite+TypeScript 项目，在后续的课程中，我们基本上以 Vue+Vite+TypeScript 为标配。



创建的项目目录结构如下：



别忘了配置我们的.env.local 文件：

复制代码

```
1 VITE_DEEPSEEK_API_KEY=sk-xxxxxxxxx
```

接着我们修改一下 `App.vue`。

复制代码

```
1 <script setup lang="ts">
2 import { ref } from 'vue';
3
4 const question = ref('讲一个关于中国龙的故事');
5 const content = ref('');
6 const stream = ref(true);
7
8 const update = async () => {
9   if(!question) return;
10   content.value = "思考中...";
11 }
```

```
12 const endpoint = 'https://api.deepseek.com/chat/completions';
13 const headers = {
14     'Content-Type': 'application/json',
15     Authorization: `Bearer ${import.meta.env.VITE_DEEPSEEK_API_KEY}`
16 };
17
18 const response = await fetch(endpoint, {
19     method: 'POST',
20     headers: headers,
21     body: JSON.stringify({
22         model: 'deepseek-chat',
23         messages: [{ role: 'user', content: question.value }],
24         stream: stream.value,
25     })
26 });
27
28 if(stream.value) {
29     content.value = '';
30
31     const reader = response.body?.getReader();
32     const decoder = new TextDecoder();
33     let done = false;
34     let buffer = '';
35
36     while (!done) {
37         const { value, done: doneReading } = await (reader?.read() as Promise<{ val
38         done = doneReading;
39         const chunkValue = buffer + decoder.decode(value);
40         buffer = '';
41
42         const lines = chunkValue.split('\n').filter((line) => line.startsWith('data
43
44         for (const line of lines) {
45             const incoming = line.slice(6);
46             if(incoming === '[DONE]') {
47                 done = true;
48                 break;
49             }
50             try {
51                 const data = JSON.parse(incoming);
52                 const delta = data.choices[0].delta.content;
53                 if(delta) content.value += delta;
54             } catch(ex) {
55                 buffer += incoming;
56             }
57         }
58     }
59 } else {
60     const data = await response.json();
```

```

61     content.value = data.choices[0].message.content;
62   }
63 }
64 </script>
65
66 <template>
67   <div class="container">
68     <div>
69       <label>输入: </label><input class="input" v-model="question" />
70       <button @click="update">提交</button>
71     </div>
72     <div class="output">
73       <div><label>Streaming</label><input type="checkbox" v-model="stream"/></div>
74       <div>{{ content }}</div>
75     </div>
76   </div>
77 </template>
78
79 <style scoped>
80 .container {
81   display: flex;
82   flex-direction: column;
83   align-items: start;
84   justify-content: start;
85   height: 100vh;
86   font-size: .85rem;
87 }
88 .input {
89   width: 200px;
90 }
91 .output {
92   margin-top: 10px;
93   min-height: 300px;
94   width: 100%;
95   text-align: left;
96 }
97 button {
98   padding: 0 10px;
99   margin-left: 6px;
100 }
101


```

运行项目，点击提交按钮，你会看到 AI 正以流式传输的方式输出内容，这样就能减少用户的等待时间。



好，我们来一起看一下代码的关键部分。


首先，流式输出的 API 调用机制，和普通的 HTTPS 输出没有什么区别，都是通过 POST 请求，只不过提交的数据中，将 stream 参数设置为 true。

 复制代码

```
1  const endpoint = 'https://api.deepseek.com/chat/completions';
2  const headers = {
3    'Content-Type': 'application/json',
4    Authorization: `Bearer ${import.meta.env.VITE_DEEPSEEK_API_KEY}`
5  };
6
7  const response = await fetch(endpoint, {
8    method: 'POST',
9    headers: headers,
10   body: JSON.stringify({
11     model: 'deepseek-chat',
12     messages: [{ role: 'user', content: question.value }],
```

```
13     stream: stream.value, // 这里 stream.value 值如果是 true, 采用流式传输
14   })
15   }):
```

在浏览器处理请求的时候, 会通过 HTML5 标准的 [Streams API](#) 来处理数据, 具体处理逻辑如下:

 复制代码


```
1  const reader = response.body?.getReader();
2  const decoder = new TextDecoder();
3  let done = false;
4  let buffer = '';
5
6  while (!done) {
7    const { value, done: doneReading } = await (reader?.read() as Promise<{ value:
8    done = doneReading;
9    const chunkValue = buffer + decoder.decode(value);
10   buffer = '';
11
12   const lines = chunkValue.split('\n').filter((line) => line.startsWith('data: '))
13
14   for (const line of lines) {
15     const incoming = line.slice(6);
16     if(incoming === '[DONE]') {
17       done = true;
18       break;
19     }
20     try {
21       const data = JSON.parse(incoming);
22       const delta = data.choices[0].delta.content;
23       if(delta) content.value += delta;
24     } catch(ex) {
25       buffer += incoming + '\n';
26     }
27   }
28 }
```

首先, 我们利用 ReadableStream API 通过 `getReader()` 获取一个读取器, 并创建 `TextDecoder` 准备对二进制数据进行解码。

然后我们设置控制流标志 `done`，以及一个 `buffer` 变量来缓存数据，因为某些情况下，Stream 数据返回给前端时，不一定传输完整。

接着我们开始循环读取数据，通过 `TextDecoder` 解析数据，将数据转换成文本并按行拆分。

因为 API 返回流式数据的协议是每一条数据以 “`data:`” 开头，后续是一个有效的 JSON 或者 `[DONE]` 表示传输结束，所以我们要对每一行以 “`data:`” 开头的数据进行处理。

 复制代码

```
1 for (const line of lines) {
2   const incoming = line.slice(6);
3   if(incoming === '[DONE]') {
4     done = true;
5     break;
6   }
7   try {
8     const data = JSON.parse(incoming);
9     const delta = data.choices[0].delta.content;
10    if(delta) content.value += delta;
11  } catch(ex) {
12    buffer += incoming + '\n';
13  }
14 }
```

如果数据传输完整，且不是 `[DONE]`，那么它就是合法 JSON，我们从中读取 `data.choices[0].delta.content`，就是需要增量更新的内容，否则说明数据不完整，将它存入缓存，以便后续继续处理。

这样我们就实现了数据的流式传输和浏览器的动态接收。

使用 Server-Sent Events

刚才的做法虽然可以直接使用流式数据，但是处理起来还是略为繁琐。

实际上 Deepseek API 和其他大部分兼容 OpenAI 的平台，AI 返回的流式输出数据都是符合标准的 [Server-Sent Events](#) (SSE) 规范的，现代浏览器几乎都支持更简单的 SSE API，只

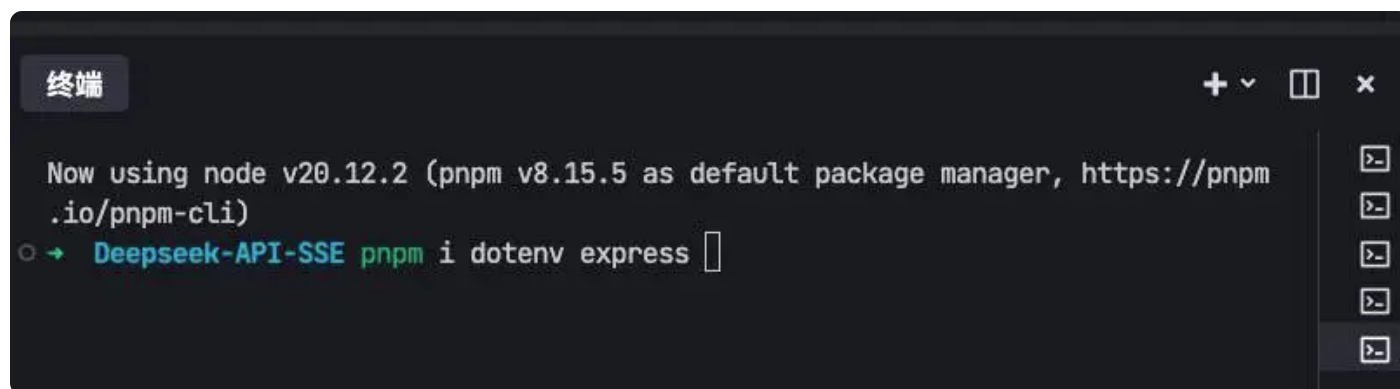
不过我们目前暂时无法在前端直接使用它。

主要原因是，根据标准，SSE 的底层只支持 HTTP GET，并且不能发送自定义的 Header，而我们的授权却需要将 API Key 通过 Authorization Header 发送，而且必须使用 POST 请求。

尽管如此，并不意味着我们前端就不能使用 SSE 来处理流式输出，而是我们需要创建一个 BFF 层，通过 Node Server 来做中转。

首先我们还是用 Trae 创建一个新的 Vue 项目 Deepseek API SSE。

接着在 IDE 终端安装依赖包 dotenv 和 express。



```
终端
Now using node v20.12.2 (pnpm v8.15.5 as default package manager, https://pnpm.io/pnpm-cli)
○ → Deepseek-API-SSE pnpm i dotenv express
```

然后在项目根目录下添加如下 server.js 文件：

复制代码

```
1 import * as dotenv from 'dotenv'
2 import express from 'express';
3
4 dotenv.config({
5   path: ['.env.local', '.env']
6 })
7
8 const openaiApiKey = process.env.VITE_DEEPSEEK_API_KEY;
9 const app = express();
10 const port = 3000;
11 const endpoint = 'https://api.deepseek.com/v1/chat/completions';
12
13 // SSE 端点
14 app.get('/stream', async (req, res) => {
15   // 设置响应头部
```


```
16 res.setHeader('Content-Type', 'text/event-stream');
17 res.setHeader('Cache-Control', 'no-cache');
18 res.setHeader('Connection', 'keep-alive');
19 res.flushHeaders(); // 发送初始响应头
20
21 try {
22     // 发送 OpenAI 请求
23     const response = await fetch(
24         endpoint,
25         {
26             method: 'POST',
27             headers: {
28                 'Authorization': `Bearer ${openaiApiKey}`,
29             },
30             body: JSON.stringify({
31                 model: 'deepseek-chat', // 选择你使用的模型
32                 messages: [{ role: 'user', content: req.query.question }],
33                 stream: true, // 开启流式响应
34             })
35         }
36     );
37
38     if (!response.ok) {
39         throw new Error('Failed to fetch from OpenAI');
40     }
41
42     const reader = response.body.getReader();
43     const decoder = new TextDecoder();
44     let done = false;
45     let buffer = '';
46
47     // 读取流数据并转发到客户端
48     while (!done) {
49         const { value, done: doneReading } = await reader.read();
50         done = doneReading;
51         const chunkValue = buffer + decoder.decode(value, { stream: true });
52         buffer = '';
53
54         // 按行分割数据, 每行以 "data: " 开头, 并传递给客户端
55         const lines = chunkValue.split('\n').filter(line => line.trim() && line.s
56         for (const line of lines) {
57             const incoming = line.slice(6);
58             if(incoming === '[DONE]') {
59                 done = true;
60                 break;
61             }
62             try {
63                 const data = JSON.parse(incoming);
64                 const delta = data.choices[0].delta.content;
```

```

65         if(delta) res.write(`data: ${delta}\n\n`); // 发送数据到客户端
66     } catch(ex) {
67         buffer += incoming;
68     }
69 }
70 }
71
72 res.write('event: end\n'); // 发送结束事件
73 res.write('data: [DONE]\n\n'); // 通知客户端数据流结束
74 res.end(); // 关闭连接
75
76 } catch (error) {
77     console.error('Error fetching from OpenAI:', error);
78     res.write('data: Error fetching from OpenAI\n\n');
79     res.end();
80 }
81 });
82
83 // 启动服务器
84 app.listen(port, () => {
85     console.log(`Server running on http://localhost:${port}`);
86 });

```

完成后，我们在终端启动服务：


 复制代码

```
1 node server.js
```

这个 server.js 的主要作用是在 server 端处理大模型 API 的流式响应，并将数据仍以兼容 SSE（以"data: "开头）的形式逐步发送给浏览器端。

现在我们在 IDE 中可以访问 `http://localhost:3000/stream?question=hello` 进行测试。为了在前端页面上访问，我们可以通过配置 vite 的 server 来进行请求转发。

此时需要修改项目中的 vite.config.js 文件：

 复制代码

```
1 import { defineConfig } from 'vite';
```


```

2 import vue from '@vitejs/plugin-vue';
3 import vueDevTools from 'vite-plugin-vue-devtools';
4
5 // https://vitejs.dev/config/
6 export default defineConfig({
7   server: {
8     allowedHosts: true,
9     port: 4399,
10    proxy: {
11      '/api': {
12        target: 'http://localhost:3000',
13        secure: false,
14        rewrite: path => path.replace(/^\/api/, ''),
15      },
16    },
17  },
18  plugins: [
19    vue(),
20    vueDevTools(),
21  ],
22  })

```

这样 server 请求就被转发到了 `/api/stream`。

最后我们这样实现 App.vue:

 复制代码

```


1 <script setup lang="ts">
2 import { ref } from 'vue';
3
4 const question = ref('讲一个关于中国龙的故事');
5 const content = ref('');
6 const stream = ref(true);
7
8 const update = async () => {
9   if(!question) return;
10  content.value = "思考中...";
11
12  const endpoint = '/api/stream';
13  const headers = {
14    'Content-Type': 'application/json',
15    Authorization: `Bearer ${import.meta.env.VITE_MOONSHOT_API_KEY}`
16  };
17
18  if(stream.value) {

```

```
19     content.value = '';
20     const eventSource = new EventSource(`${endpoint}?question=${question.value}`)
21     eventSource.addEventListener("message", function(e: any) {
22         content.value += e.data;
23     });
24 } else {
25     const response = await fetch(endpoint, {
26         method: 'POST',
27         headers: headers,
28         body: JSON.stringify({
29             model: 'moonshot-v1-8k',
30             messages: [{ role: 'user', content: question.value }],
31             stream: stream.value,
32         })
33     });
34     const data = await response.json();
35     content.value = data.choices[0].message.content;
36 }
37 }
38 </script>
39
40 <template>
41     <div class="container">
42         <div>
43             <label>输入: </label><input class="input" v-model="question" />
44             <button @click="update">提交</button>
45         </div>
46         <div class="output">
47             <div><label>Streaming</label><input type="checkbox" v-model="stream"/></div>
48             <div>{{ content }}</div>
49         </div>
50     </div>
51 </template>
52
53 <style scoped>
54 .container {
55     display: flex;
56     flex-direction: column;
57     align-items: start;
58     justify-content: start;
59     height: 100vh;
60     font-size: .85rem;
61 }
62 .input {
63     width: 200px;
64 }
65 .output {
66     margin-top: 10px;
67     min-height: 300px;
```

```
68   width: 100%;
69   text-align: left;
70 }
71 button {
72   padding: 0 10px;
73   margin-left: 6px;
74 }
75 </style>
```

注意和前面直接通过 Streams API 处理数据相比，有了 server 端处理转发后，浏览器只需使用 SSE，代码如下：

 复制代码

```
1 const eventSource = new EventSource(`${endpoint}?question=${question.value}`);
2 eventSource.addEventListener("message", function(e: any) {
3   content.value += e.data;
4 });
5 eventSource.addEventListener('end', () => {
6   eventSource.close();
7 });
```

这不仅仅让前端代码实现变得简洁很多，而且 SSE 在浏览器内置了自动重连机制。这意味着当网络、服务器或者客户端连接出现问题，恢复后将自动完成重新连接，不需要用户主动刷新页面，这让 SSE 特别适合**长时间保持连接的应用场景**。此外，SSE 还支持通过 lastEventId 来支持数据的续传，这样在错误恢复时，能大大节省数据传输的带宽和接收数据的响应时间。

关于 SSE 的问题，在后续课程中，我们还会有机会继续深入探讨。

要点总结

在大模型的 API 调用方式上，除了传统的 HTTP 调用方式外，还支持流式传输，由于这么做不用等待推理完成就可以实时响应内容，因此能够大大减少用户等待时间，是非常有意义的。

这节课，我们以 Deepseek Platform 为例，探讨了文本大模型使用 Streams API 的流式传输和 Server-Sent Events 的方法。这两种方式，提高了响应实效性，从而能够大大减少用户的

等待时间，带来较好的用户体验。这也是我在实际的 AI 应用产品中推崇并最常使用的两种调用方式。我也希望同为前端的你，能够掌握这些调用方式并将它们运用到实际产品项目中去，从而改进用户的体验。

课后练习

Server-Sent Events 是一种允许服务器主动推送实时更新给浏览器的技术，属于 Web 标准，它除了实时推送数据外，还可以支持自定义事件 (Custom Events) 和内容的续传。你可以通过 [MDN 文档](#) 和询问 AI 进一步学习这部分内容，尝试给我们的例子增加事件通知和连接断开恢复后的数据续传能力，这些能力在实际 AI 应用产品的业务中都是可以用到的。

AI智能总结

1. 流式传输的优势是能够实时返回推理过程中生成的token，减少用户等待时间，提高响应速度。
2. 实现流式传输的关键步骤包括设置stream参数为false，使用POST请求调用API，并在浏览器端利用HTML5标准的ReadableStream API来处理数据，实现动态接收和处理流式数据。
3. 流式传输的处理逻辑涉及在浏览器端利用TextDecoder对二进制数据进行解码，循环读取数据并按行拆分，处理每一条以"data:"开头的的数据，从中提取需要增量更新的内容，实现数据的流式传输和动态接收。
4. 通过示例学习流式传输，可以演示如何在Deepseek Platform下使用流式传输来减少用户等待时间，以及关键的代码实现部分。
5. 流式传输的实际应用范围适用于大多数文本模型，能够在模型推理过程中及时返回生成的token，提高前端工程师处理大模型API的效率。
6. 通过流式传输减少用户等待时间的方法是通过设置stream参数为true，实现AI以流式传输的方式输出内容，从而减少用户的等待时间。
7. 流式传输的实践意义在于处理复杂请求时能够显著减少用户等待时间，提升用户体验，适用于需要实时推理的大模型API。
8. 流式传输的实现机制通过对API调用机制和浏览器处理请求的逻辑进行详细解释，展示了流式传输的实现机制和原理。
9. 流式传输的应用范围适用于处理复杂请求、实时推理的场景，能够提高前端工程师处理大模型API的效率，改善用户体验。
10. Server-Sent Events (SSE) 是一种允许服务器主动推送实时更新给浏览器的技术，除了实时推送数据外，还可以支持自定义事件和内容的续传，适合长时间保持连接的应用场景。

全部留言 (4)

最新 精选



Geek_22eecd

2025-04-09 来自上海

首先，流式输出的 API 调用机制，和普通的 HTTPS 输出没有什么区别，都是通过 POST 请求，只不过提交的数据中，将 stream 参数设置为 false。

这里应该是true, (还有下面的代码部分也是true)

作者回复: 是的，我今天更正一下，感谢反馈

共 2 条评论 >



3



Geek_16ca32

2025-04-11 来自北京

server.js 发送请求时，headers是不是也需要 'Content-Type': 'application/json'

作者回复: 流式传输响应content-type不是json



Geek_16ca32

2025-04-10 来自北京

现在我们在 IDE 中可以访问 <http://localhost:3000/steam?question=hello> 进行测试。steam 更改为 stream

编辑回复: 感谢反馈，我们已经修正了，刷新可见～



立

2025-04-09 来自广东

第一个案例中，开启流式响应，stream 参数设置为 false，是错了吧，应该为true才对，下面的转发案例也是传了true

共 1 条评论 >

 2