告别轮询，SSE 流式传输可太香了！

今天想和大家分享的一个技术是 **SSE 流式传输** 。如标题所言，通过 SSE 流式传输的方式可以让我们不再通过轮询的方式获取服务端返回的结果，进而提升前端页面的性能。

对于需要轮询的业务场景来说，采用 SSE 确实是一个更好的技术方案。

接下来，我将从 SSE 的概念、与 Websocket 对比、SSE 应用场景多个方面介绍 SSE 流式传输，感兴趣的同学一起来了解下吧！

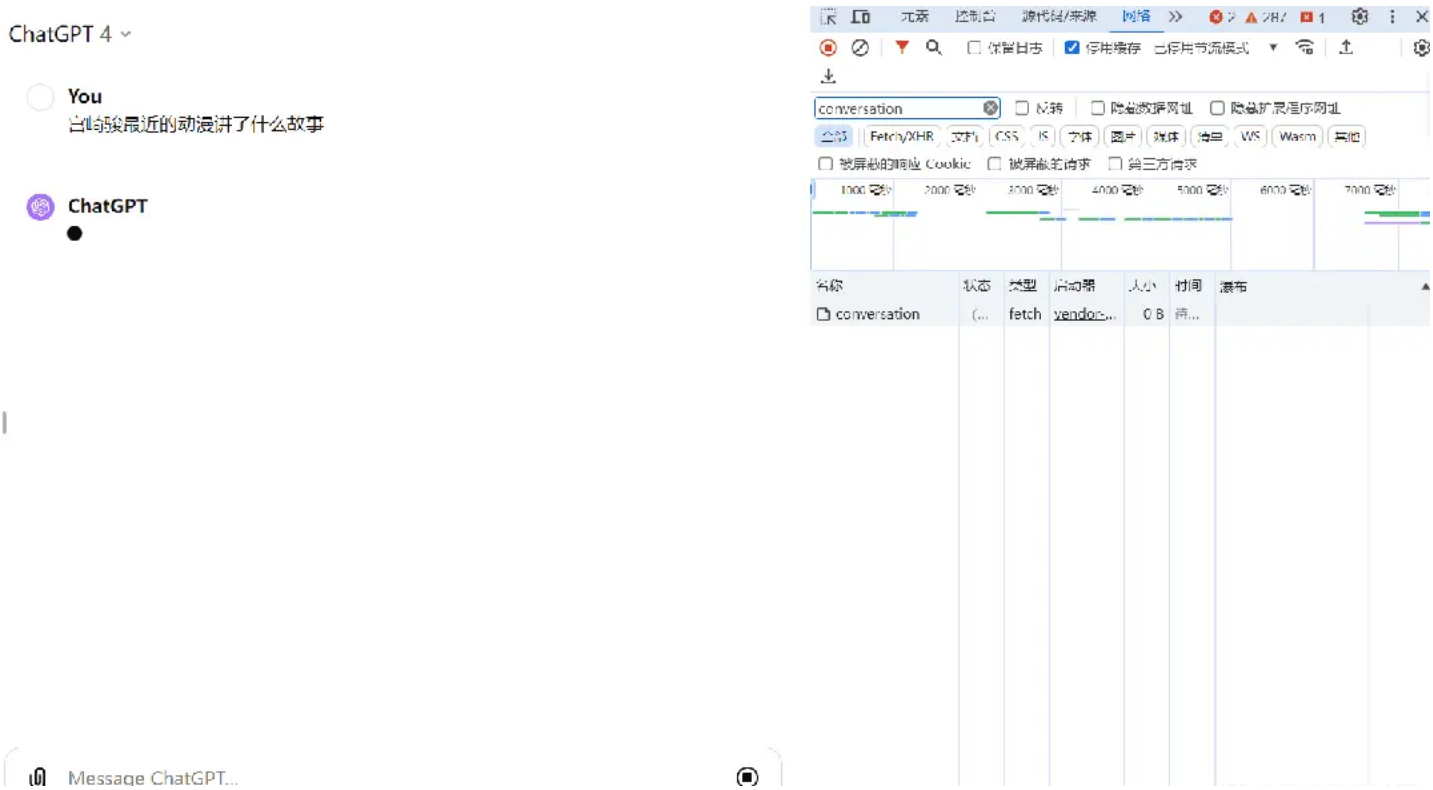
什么是 SSE 流式传输

SSE 全称为 **Server-sent events** , 是一种基于 HTTP 协议的通信技术，允许服务器主动向客户端（通常是Web浏览器）发送更新。

它是 HTML5 标准的一部分，设计初衷是用来建立一个单向的服务器到客户端连接，使得服务器可以实时地向客户端发送数据。

这种服务端实时向客户端发送数据的传输方式，其实就是流式传输。

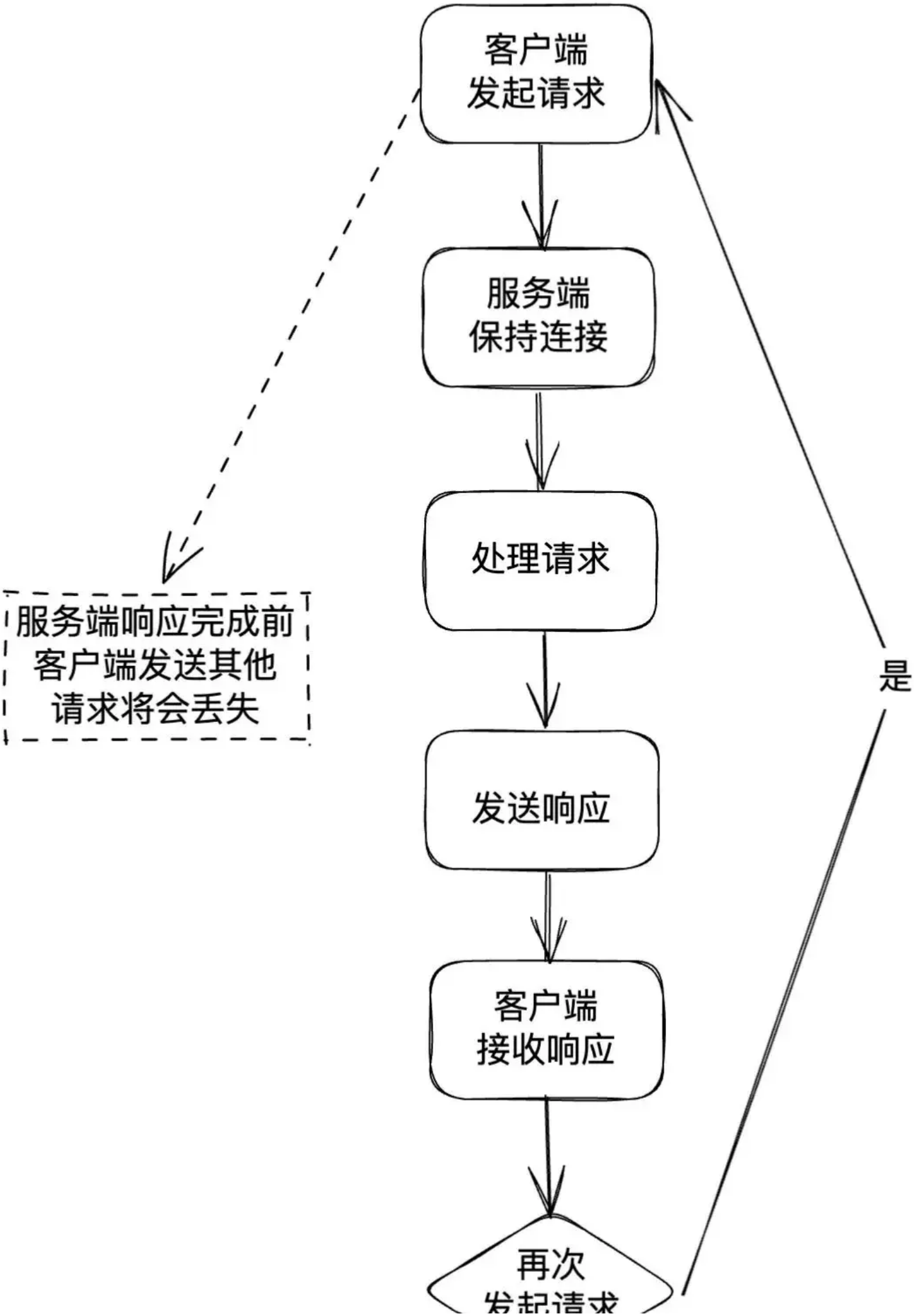
我们在与 ChatGPT 交互时，可以发现 ChatGPT 的响应总是间断完成。细扒 ChatGPT 的网络传输模式，可以发现，用的也是流式传输。



SSE 流式传输的好处

在 **SSE** 技术出现之前，我们习惯把需要等待服务端返回的过程称为长轮询。

长轮询的实现其实也是借助 http 请求来完成，一个完整的长轮询过程如下图所示：


客户端
发起请求
服务端
保持连接
处理请求
发送响应
客户端
接收响应
再次
发起请求
是
服务端响应完成前
客户端发送其他
请求将会丢失

从图中可以发现，**长轮询最大的弊端是当服务端响应请求之前，客户端发送的所有请求都不会被受理。并且服务端发送响应的前提是客户端发起请求。**

前后端通信过程中，我们常采用 ajax 、axios 来异步获取结果，这个过程，其实也是长轮询的过程。

而同为采用 http 协议通信方式的 SSE 流式传输，相比于长轮询模式来说，优势在于可以在不需要客户端介入的情况下，多次向客户端发送响应，直至客户端关闭连接。

这对于需要服务端实时推送内容至客户端的场景可方便太多了！

SSE 技术原理

1. 参数设置

前文说到，SSE 本质是一个基于 http 协议的通信技术。

因此想要使用 **SSE** 技术构建需要服务器实时推送信息到客户端的连接，只需要将传统的 **http 响应头的 contentType 设置为 text/event-stream** 。

并且为了保证客户端展示的是最新数据，需要将 **Cache-Control 设置为 no-cache** 。

在此基础上，**SSE** 本质是一个 TCP 连接，因此为了保证 SSE 的持续开启，需要将 **Connection 设置为 keep-alive** 。

```
Content-Type: text/event-stream
Cache-Control: no-cache
Connection: keep-alive
```

完成了上述响应头的设置后，我们可以编写一个基于 **SSE** 流式传输的简单 Demo 。

2. SSE Demo

服务端代码：

```
const express = require('express');
const app = express();
const PORT = 3000;

app.use(express.static('public'));

app.get('/events', function(req, res) {
    res.setHeader('Content-Type', 'text/event-stream');
    res.setHeader('Cache-Control', 'no-cache');
    res.setHeader('Connection', 'keep-alive');

    let startTime = Date.now();

    const sendEvent = () => {
```

```
        // 检查是否已经发送了10秒
        if (Date.now() - startTime >= 10000) {
            res.write('event: close\ndata: {}\n\n'); // 发送一个特殊事件通知客户端关闭
            res.end(); // 关闭连接
            return;
        }

        const data = { message: 'Hello World', timestamp: new Date() };
        res.write(`data: ${JSON.stringify(data)}\n\n`);

        // 每隔2秒发送一次消息
        setTimeout(sendEvent, 2000);
    };

    sendEvent();
});

app.listen(PORT, () => {
    console.log(`Server running on http://localhost:${PORT}`);
});
```

客户端代码：

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>SSE Example</title>
</head>

<body>
    <h1>Server-Sent Events Example</h1>
    <div id="messages"></div>

    <script>
        const evtSource = new EventSource('/events');
        const messages = document.getElementById('messages');

        evtSource.onmessage = function(event) {
            const newElement = document.createElement("p");
            const eventObject = JSON.parse(event.data);
            newElement.textContent = "Message: " + eventObject.message + " at " + eventObject.timestamp;
```

```
            messages.appendChild(newElement);
        };
    </script>
</body>
</html>
```

当我们在浏览器中访问运行在 localhost: 3000 端口的客户端页面时，页面将会以 **流式模式** 逐步渲染服务端返回的结果：

需要注意的是，为了保证使用 **SSE** 通信协议传输的数据能被客户端正确的接收，服务端和客户端在发送数据和接收数据应该遵循以下规范：

服务端基本响应格式

SSE 响应主要由一系列以两个换行符分隔的事件组成。每个事件可以包含以下字段：

```
data：事件的数据。如果数据跨越多行，每行都应该以data:开始。
id：事件的唯一标识符。客户端可以使用这个ID来恢复事件流。
event：自定义事件类型。客户端可以根据不同的事件类型来执行不同的操作。
retry：建议的重新连接时间（毫秒）。如果连接中断，客户端将等待这段时间后尝试重新连接。
```

字段之间用单个换行符分隔，而事件之间用两个换行符分隔。

客户端处理格式

客户端使用 **EventSource** 接口监听 **SSE** 消息：

```
const evtSource = new EventSource('path/to/sse');
evtSource.onmessage = function(event) {
    console.log(event.data); // 处理收到的数据
};
```

SSE 应用场景

SSE 作为基于 **http** 协议由服务端向客户端单向推送消息的通信技术，对于需要服务端主动推送消息的场景来说，是非常适合的：

SSE 兼容性

除了 IE 和低版本的主流浏览器，目前市面上绝大多数浏览器都支持 **SSE** 通信。

SSE 与 WebSocket 对比

看完 SSE 的使用方式后，细心的同学应该发现了：

SSE 的通信方式和 **WebSocket** 很像啊，而且 WebSocket 还支持双向通信，为什么不直接使用 WebSocket？

下表展示了两者之间的对比：

特性/因素	SSE	WebSockets
协议	基于HTTP，使用标准HTTP连接	单独的协议（ws:// 或 wss://），需要握手升级
通信方式	单向通信（服务器到客户端）	全双工通信
数据格式	文本（UTF-8编码）	文本或二进制
重连机制	浏览器自动重连	需要手动实现重连机制
实时性	高（适合频繁更新的场景）	非常高（适合高度交互的实时应用）
浏览器支持	良好（大多数现代浏览器支持）	非常好（几乎所有现代浏览器支持）
适用场景	实时通知、新闻feed、股票价格等需要从服务器推送到客户端的场景	在线游戏、聊天应用、实时交互应用
复杂性	较低，易于实现和维护	较高，需要处理连接的建立、维护和断开
兼容性和可用性	基于HTTP，更容易通过各种中间件和防火墙	可能需要配置服务器和网络设备以支持WebSocket
服务器负载	适合较低频率的数据更新	适合高频率消息和高度交互的场景

可以发现，**SSE** 与 **WebSocket** 各有优缺点，对于需要客户端与服务端高频交互的场景，WebSocket 确实更适合；但对于只需要服务端单向数据传输的场景，**SSE 确实能耗更低，且不需要客户端感知**。