# LLM

"text/event-stream" 和 WebSocket 是用于实现实时通信的两种不同协议，它们在某些方面有相似的功能，但也存在一些区别。

主要区别如下：

协议类型： "text/event-stream" 使用 HTTP 协议，而 WebSocket 则是一种独立的协议。

连接性质： "text/event-stream" 基于持久连接（long-polling），即客户端与服务器之间的连接保持打开状态，服务器可以随时向客户端发送数据。WebSocket 采用全双工通信方式，允许客户端和服务器之间进行双向实时通信。

数据格式： "text/event-stream" 使用纯文本格式，通常是纯文本或 JSON，每个事件以换行符分隔。WebSocket 则支持任意的二进制或文本数据传输。

浏览器支持：当涉及到浏览器中的实时通信时，WebSocket 得到了更广泛的支持。大多数现代浏览器都原生支持 WebSocket，而 "text/event-stream" 的支持相对较少。

适用场景：

"text/event-stream" 适用于需要从服务器获取实时数据更新，但不需要频繁的双向通信的场景。它适合用于服务器向客户端推送事件和数据的应用程序，例如股票市场报价、新闻推送等。

WebSocket 适用于需要实时双向通信的场景，例如即时聊天应用、实时游戏、协作编辑工具等。WebSocket 提供了更高效、低延迟和更灵活的双向通信能力，允许服务器主动向客户端发送消息，并且客户端也可以在任何时候向服务器发送消息。

总结来说，如果你只需要从服务器获取实时数据更新，而无需频繁的双向通信，则可以选择使用 "text/event-stream"。如果你需要实现实时双向通信，允许服务器主动推送消息并允许客户端主动发送消息，则应选择 WebSocket。

Event-stream server

func main() {  
 http.HandleFunc("/", eventStreamHandler)  
 log.Fatal(http.ListenAndServe(":8080", nil))  
}  
  
func eventStreamHandler(w http.ResponseWriter, r \*http.Request) {  
 *// 设置响应头，指定内容类型为 text/event-stream* w.Header().Set("Content-Type", "text/event-stream")  
 *// 设置响应头，禁用缓存* w.Header().Set("Cache-Control", "no-cache")  
 *// 设置响应头，允许跨域访问* w.Header().Set("Access-Control-Allow-Origin", "\*")  
 *// 设置响应头，保持长连接* w.Header().Set("Connection", "keep-alive")  
  
 *// 创建一个新的事件通道* eventChannel := make(chan string, 100)  
  
 *// 创建一个关闭通道，用于通知协程关闭连接* closeChannel := make(chan struct{})  
  
 *// 获取 CloseNotifier 接口  
 //closeNotifier, ok := w.(http.CloseNotifier)  
 //if !ok {  
 // http.Error(w, "Streaming unsupported!", http.StatusInternalServerError)  
 // return  
 //}  
  
 // 启动一个协程，用于发送事件到客户端* go func() {  
 *// 计数器，记录发送的事件次数* count := 0  
  
 for {  
 *// 模拟生成事件数据* eventData := time.Now().Format("2006-01-02 15:04:05")  
  
 select {  
 case eventChannel <- eventData:  
 *// 发送事件数据到通道* count++  
  
 if count == 10 {  
 *// 达到指定次数后发送关闭信号* close(closeChannel)  
 close(eventChannel)  
 return  
 }  
 case <-closeChannel:  
 *// 收到关闭信号，退出协程* close(eventChannel)  
 close(eventChannel)  
 return  
 }  
  
 *// 延迟一秒钟* time.Sleep(time.Second)  
 }  
 }()  
  
 *// 遍历事件通道，将事件数据写入响应流中* for eventData := range eventChannel {  
 fmt.Fprintf(w, "data: %s\n", eventData)  
 flusher, ok := w.(http.Flusher)  
 if !ok {  
 http.Error(w, "Streaming unsupported!", http.StatusInternalServerError)  
 return  
 }  
 flusher.Flush()  
  
 *// 检查连接是否已关闭  
 //if closeNotifier.CloseNotify() != nil {  
 // // 收到关闭通知，发送关闭信号到协程并退出循环  
 // close(closeChannel)  
 // break  
 //}* }  
  
 fmt.Println("0000000000000000")  
}

Event stream client

func main() {  
 client := http.Client{}  
  
 req, err := http.NewRequest("GET", "http://localhost:8080/events", nil)  
 if err != nil {  
 fmt.Println("Failed to create request:", err)  
 return  
 }  
  
 resp, err := client.Do(req)  
 if err != nil {  
 fmt.Println("Request failed:", err)  
 return  
 }  
 defer resp.Body.Close()  
  
 *// 检查响应头部是否为 text/event-stream 类型* contentType := resp.Header.Get("Content-Type")  
 if contentType != "text/event-stream" {  
 fmt.Println("Invalid Content-Type. Expected text/event-stream, got", contentType)  
 return  
 }  
  
 *// 创建一个无限循环来读取和处理服务器发送的事件和数据* for {  
 reader := bufio.NewReader(resp.Body) *// 替换为你要读取的 io.Reader 对象* line, err := reader.ReadString('\n')  
 if err != nil {  
 fmt.Println("Failed to read line:", err)  
 break  
 }  
 line = strings.ReplaceAll(line, "\n", "")  
  
 fmt.Println("11111 " + line)  
 }  
}