

13-服务器推送技术

1.主流服务器推送技术说明

一、需求与背景

二、服务端推送常用技术

全双工通信:WebSocket

服务端主动推送:SSE (Server Send Event)

三、websocket与SSE比较

2.服务端推送事件SSE

一、模拟网络支付场景

二、SSE 起步练习

3.双向实时通信websocket

一、整合websocket

二、兼容HTTPS协议

三、WebSocket编程基础

3.1.连接的建立

3.2.全双工数据交互

3.3.数据发送

浏览器与服务器交换数据

一个用户向其他用户群发

四、websocket实现简单聊天

1.主流服务器推送技术说明

一、需求与背景

若干年前，所有的请求都是由浏览器端发起，浏览器本身并没有接受请求的能力。所以一些特殊需求都是用ajax轮询的方式来实现的。比如：

- 股价展示页面实时的获取股价更新
- 赛事的文字直播，实时更新赛况

- 通过页面启动一个任务，前端想知道任务后台的实时运行状态

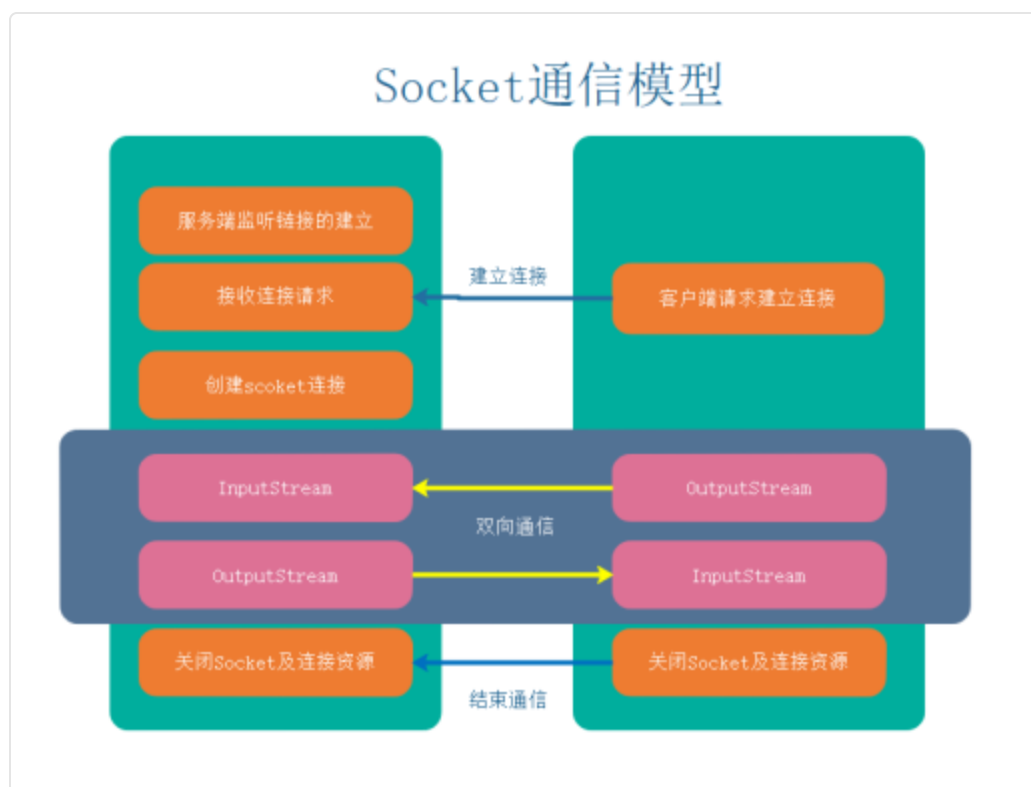
通常的做法就是需要以较小的间隔，频繁的向服务器建立http连接询问任务状态的更新，然后刷新页面显示状态。但这样做的后果就是浪费大量流量，对服务端造成了非常大的压力。

二、服务端推送常用技术

在html5被广泛推广之后，我们可以使用**服务端主动推送数据**，**浏览器接收数据**的方式来解决上面提到的问题。下面我们就为大家介绍两种服务端数据推送技术。

全双工通信:WebSocket

全双工就是双向通信。如果说http协议是“对讲机”之间的通话（你一句我一句，有来有回），那websocket就是移动电话(可以随时发送信息与接收信息，就是全双工)。



本质上是一个额外的tcp连接，建立和关闭时握手使用http协议，其他数据传输不使用http协议，更加复杂一些，比较适用于需要进行复杂双向实时数据通讯的场景。在web网页上面的客服、聊天室一般都是使用WebSocket 协议来开发的。

服务端主动推送:SSE (Server Send Event)

html5新标准，用来从服务端实时推送数据到浏览器端， 直接建立在当前http连接上，本质上是保持一个http长连接，轻量协议 。客户端发送一个请求到服务端，服务端保持这个请求连接直到一个新的消息准备好，将消息返回至客户端。除非主动关闭，这个连接会一直保持。

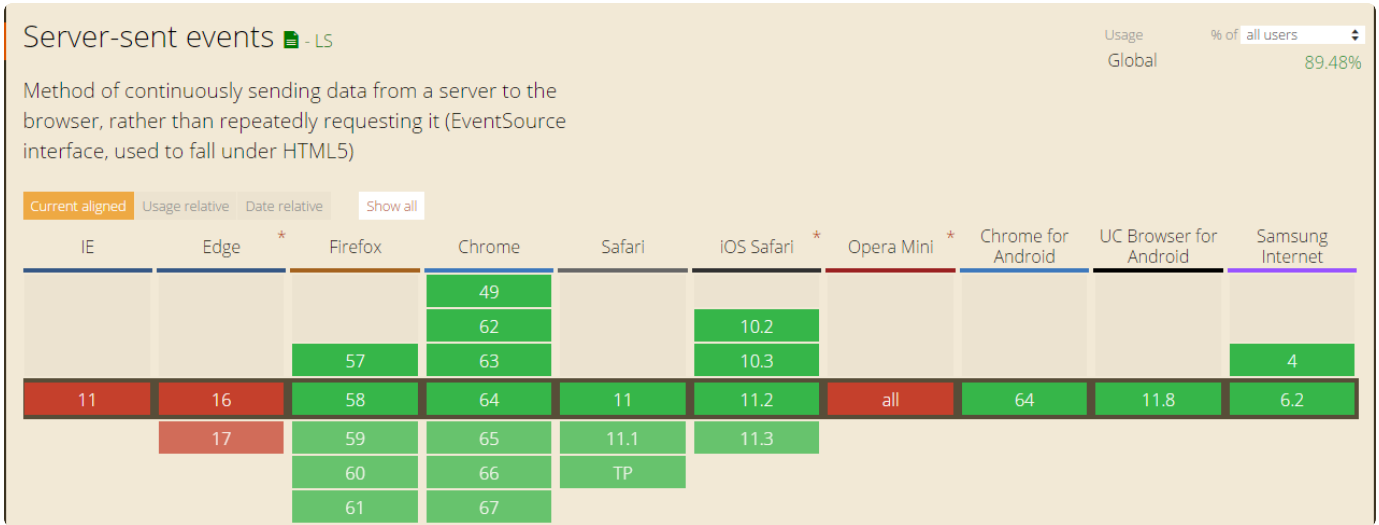
- 建立连接
- 服务端 -> 浏览器(连接保持)
- 关闭连接

SSE的一大特色就是重复利用1个连接来接收服务端发送的消息（又称event），从而避免不断轮询请求建立连接，造成服务资源紧张。

三、websocket与SSE比较

	是否基于新协议	是否双向通信	是否支持跨域
SSE	否（Http）	否（单向）	否（Firefox 支持跨域）
WebSocket	是（ws）	是	是

但是IE和Edge浏览器不支持SSE，所以SSE目前的应用场景比较少。虽然websocket在很多比较旧的版本浏览器上面也不兼容，但是总体上比SSE要好不少。另外还有一些开源的JS前端产品，如 [SockJS](#)，[Socket.IO](#)，在浏览器端提供了更好的websocket前端js编程体验，与浏览器有更好的兼容性。



2.服务端推送事件SSE

一、模拟网络支付场景

大家应该都用过支付系统，比如淘宝买一个产品之后进行扫码支付。如果结合SSE，该如何实现这个过程？

1. 用户扫码向支付系统（支付宝）进行支付
2. 支付完成之后，告知商户系统（淘宝卖家系统）我已经发起支付了（**建立SSE连接**）
3. 支付系统(支付宝)告诉商户系统(淘宝卖家系统)，这个用户确实支付成功了
4. 商户系统(淘宝卖家系统)向用户发送消息：你已经支付成功，跳转到支付成功页面。（**通过SSE连接，由服务器端告知用户客户端浏览器**）

注意：在返回最终支付结果的操作，实现了服务端向客户端的事件推送，可以使用SSE来实现

二、SSE 起步练习

```
1 package com.mqxu.boot.websocket.controller;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.springframework.http.MediaType;
5 import org.springframework.stereotype.Controller;
6 import org.springframework.web.bind.annotation.RequestMapping;
7 import org.springframework.web.bind.annotation.RequestMethod;
8 import
    org.springframework.web.servlet.mvc.method.annotation.ResponseBodyEmitter
    ;
9 import org.springframework.web.servlet.mvc.method.annotation.SseEmitter;
10
11 import java.io.IOException;
12 import java.text.DecimalFormat;
13 import java.util.ArrayList;
14 import java.util.List;
15
16 /**
17  * @description: SSE练习
18  * @author: mxqu
19  * @date: 2022-04-18
20  */
21 @Slf4j
22 @Controller
23 public class SseController {
24
25     @RequestMapping(value = "/server/info", method = {RequestMethod.GET},
26 produces = "text/event-stream;charset=UTF-8")
27     public ResponseBodyEmitter pushMsg() {
28         final SseEmitter emitter = new SseEmitter(0L);
29         log.info("emitter push message .....");
30         List<String> list = new ArrayList<>();
31         list.add("aaa");
32         list.add("bbb");
33         list.add("ccc");
34         try {
35             emitter.send(list.toString(), MediaType.TEXT_EVENT_STREAM);
36         } catch (IOException e) {
37             e.printStackTrace();
38         }
39         return emitter;
40     }
41
42     @RequestMapping(value = "/server/data", method = {RequestMethod.GET},
43 produces = "text/event-stream;charset=UTF-8")
```

```

42  public ResponseBodyEmitter push() {
43      final SseEmitter emitter = new SseEmitter(0L);
44      try {
45          Thread.sleep(1000);
46      } catch (Exception e) {
47          e.printStackTrace();
48      }
49      double money = Math.random() * 10;
50      DecimalFormat df = new DecimalFormat(".00");
51      String param = df.format(money);
52      try {
53          emitter.send("白菜价格行情:" + param + "元" + "\n\n",
54              MediaType.TEXT_EVENT_STREAM);
55      } catch (IOException e) {
56          e.printStackTrace();
57      }
58      return emitter;
59  }
60  }

```

对于服务器端向浏览器发送的数据，浏览器端需要在 JavaScript 中使用 EventSource 对象来进行处理。EventSource 使用的是标准的事件监听器方式，只需要在对象上添加相应的事件处理方法即可。EventSource 提供了三个标准事件

事件名称	事件触发说明	事件处理方法
open	当服务器向浏览器第一次发送数据时产生	onopen
message	当收到服务器发送的消息时产生	onmessage
error	当出现异常时产生	onerror

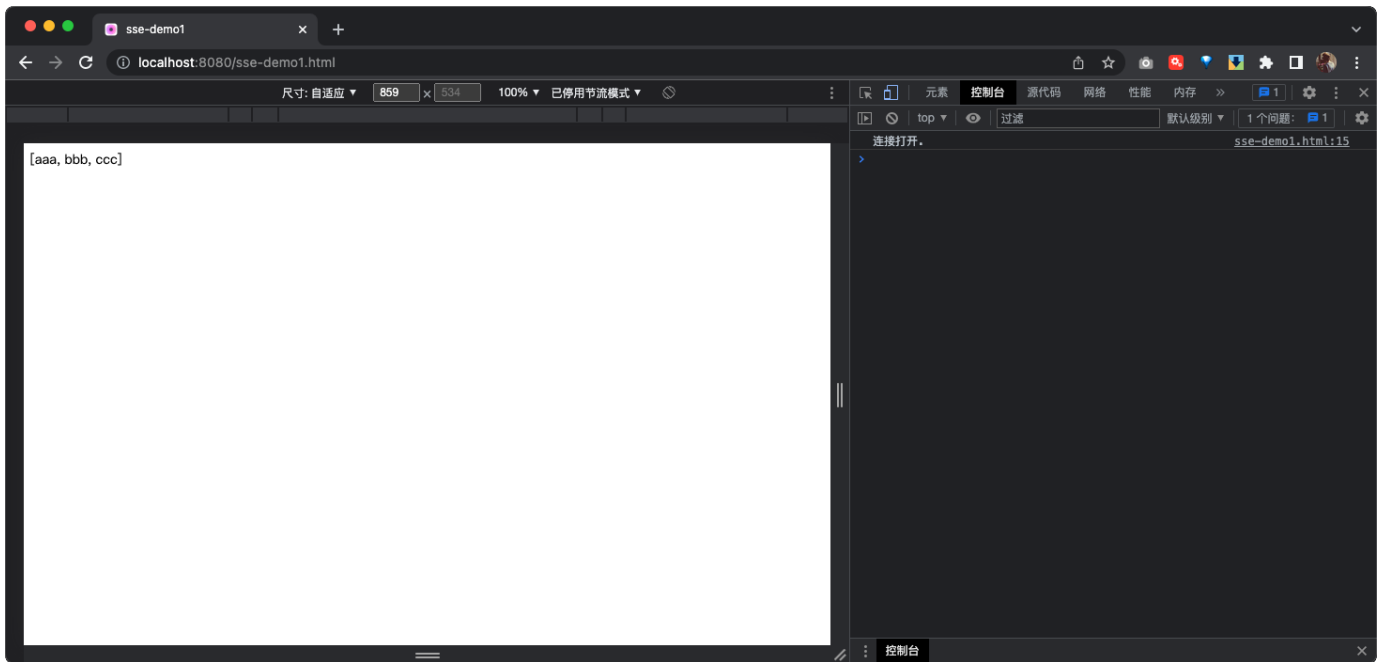
除了使用标准的事件处理方法，还可以使用 `addEventListener` 方法对事件进行监听。

```
1  var es = new EventSource('事件源名称') ; //与事件源建立连接
2  //标准事件处理方法,还有onopen、onerror
3  es.onmessage = function(e) {
4  };
5  //可以监听自定义的事件名称
6  es.addEventListener('自定义事件名称', function(e) {
7  });
```

public/sse-demo1.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>sse-demo1</title>
6  </head>
7  <body>
8      <div id="msg_from_server">
9      </div>
10 <script>
11     if (!!window.EventSource) {
12         let source = new
13         EventSource('http://localhost:8080/server/info');
14         let s = '';
15         source.addEventListener('open', () => {
16             console.log("连接打开.");
17         }, false);
18         source.addEventListener('message', (e) => {
19             s += e.data + "<br/>"
20             document.getElementById("msg_from_server").innerHTML = s;
21         });
22         source.addEventListener('error', (e) => {
23             if (e.readyState === EventSource.CLOSED) {
24                 console.log("连接关闭");
25             } else {
26                 console.log(e.readyState);
27             }
28         }, false);
29     } else {
30         alert(4);
31         console.log("没有sse");
32     }
33 </script>
34 </body>
35 </html>
```

运行效果

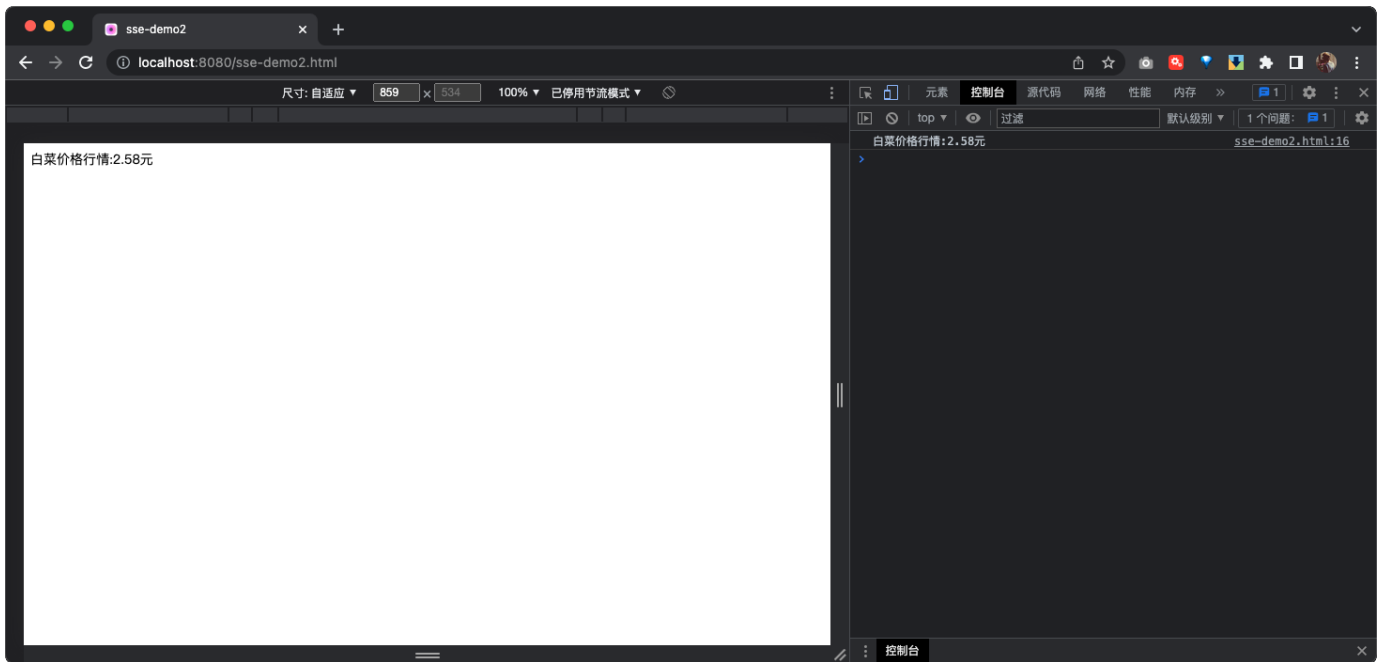


public/sse-demo2.html

```
HTML | 复制代码

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>sse-demo2</title>
6  </head>
7
8  <body>
9  <div id="result">
10 </div>
11
12 <script type="text/javascript">
13     //需要判断浏览器支不支持，可以去w3c进行查看
14     const source = new EventSource('http://localhost:8080/server/data');
15     source.onmessage = function (event) {
16         console.info(event.data);
17         document.getElementById('result').innerText = event.data
18     };
19
20 </script>
21 </body>
22
23 </html>
```

运行效果：1秒后出现服务端推送的数据



3.双向实时通信websocket

一、整合websocket

- 添加依赖

```
XML | 复制代码
1 <!-- 引入websocket依赖 -->
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-starter-websocket</artifactId>
5 </dependency>
```

- 编写配置类，开启websocket功能

```
1  @Configuration
2  public class WebSocketConfig {
3      @Bean
4      public ServerEndpointExporter serverEndpointExporter() {
5          return new ServerEndpointExporter();
6      }
7  }
```

二、兼容HTTPS协议

- WebSocket的ws协议是基于HTTP协议实现的
- WebSocket的wss协议是基于HTTPS协议实现的

一旦你的项目里面使用了https协议，websocket就要使用wss协议才可以。

在 TomcatCustomizer 配置的基础之上加上如下的代码，就可以支持wss协议。

```
1  @Bean
2  public TomcatContextCustomizer tomcatContextCustomizer() {
3
4      return new TomcatContextCustomizer() {
5          @Override
6          public void customize(Context context) {
7              context.addServletContainerInitializer(new WsSci(), null);
8          }
9
10     };
11 }
```

三、WebSocket编程基础

3.1.连接的建立

前端js向后端发送wss连接建立请求

```
1 socket = new WebSocket("wss://localhost:8888/ws/asset");
```

SpringBoot服务端WebSocket服务接收类定义如下：

```
1 @Component
2 @Slf4j
3 @ServerEndpoint(value = "/ws/asset")
4 public class WebSocketServer {
5
6 }
```

3.2.全双工数据交互

前端后端都有

- onopen事件监听，处理连接建立事件
- onmessage事件监听，处理对方发过来的消息数据
- onclose事件监听，处理连接关闭
- onerror事件监听，处理交互过程中的异常

```
// else {
socket = new WebSocket("wss://localhost:8888/ws/asset");
// 连接打开事件
socket.onopen = function() {
    console.log("Socket 已打开");
    socket.send("消息发送测试(From Client)");
};
// 收到消息事件
socket.onmessage = function(msg) {
    document.getElementById('message').innerHTML += msg.data;
};
// 连接关闭事件
socket.onclose = function() {
    console.log("Socket已关闭");
};
// 发生错误事件
socket.onerror = function() {
    alert("Socket发生了错误");
};

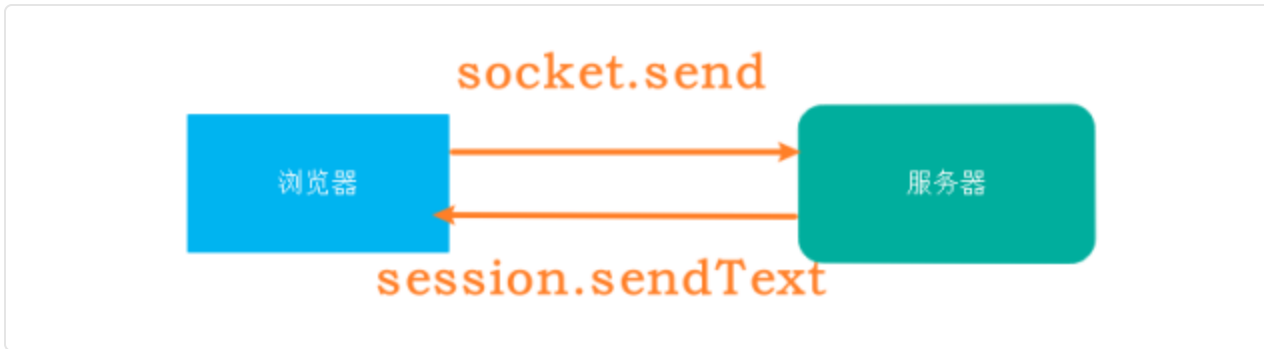
// 窗口关闭时，关闭连接
window.unload=function() {
    socket.close();
};

// 关闭连接
function closeWebSocket(){
    socket.close();
}
}
```

```
14 @ServerEndpoint(value = "/ws/asset")
15 public class WebSocketServer {
16
17     // 用来统计连接客户端的数量
18     private static final AtomicInteger OnlineCount = new AtomicInteger(0);
19     // concurrent包的线程安全Set，用来存放每个客户端对应的Session对象。
20     private static CopyOnWriteArraySet<Session> SessionSet = new CopyOnWriteArraySet<>();
21
22     /**
23      * 连接成功调用的方法
24      */
25     @OnOpen
26     public void onOpen(Session session) throws IOException {
27         SessionSet.add(session);
28         int cnt = OnlineCount.incrementAndGet(); // 在线数加1
29         log.info("有连接加入，当前连接数为：{}", cnt);
30         SendMessage(session, "连接成功");
31     }
32
33     /**
34      * 收到客户端消息后调用的方法
35      * @param message 客户端发过来的消息
36      */
37     @OnMessage
38     public void onMessage(String message, Session session) throws IOException {
39         log.info("来自客户端的消息：{}", message);
40         SendMessage(session, "收到消息，消息内容：" + message);
41     }
42
43     /**
44      * 连接关闭调用的方法
45      */
46     @OnClose
47     public void onClose(Session session) {
48         SessionSet.remove(session);
49         int cnt = OnlineCount.decrementAndGet();
50         log.info("有连接关闭，当前连接数为：{}", cnt);
51     }
52
53     /**
54      * 发生错误
55      */
56     @OnError
57     public void onError(Session session, Throwable error) {
58         log.error("发生错误：{}", error.getMessage(), session.getId());
59     }
60
61 }
62 }
```

3.3.数据发送

浏览器与服务器交换数据



前端JS

```
JavaScript | 复制代码

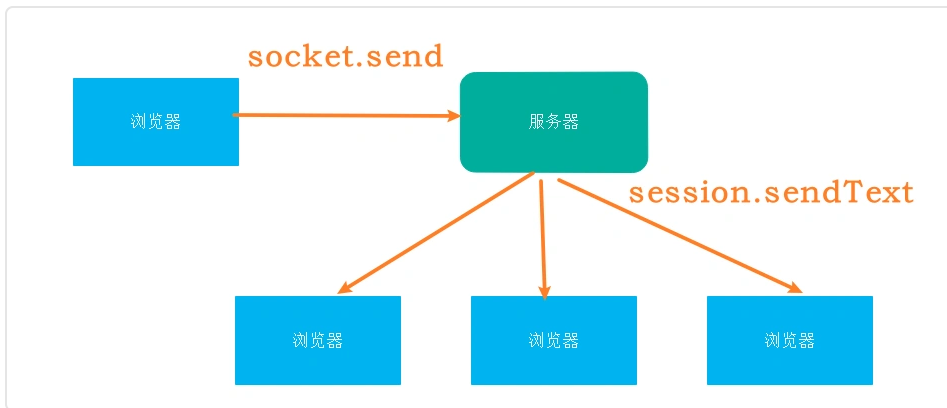
1 socket.send(message);
```

后端Java，向某一个javax.websocket.Session用户发送消息。

```
Java | 复制代码

1 /**
2  * 发送消息，每次浏览器刷新，session会发生变化。
3  * @param session session
4  * @param message 消息
5  */
6 private static void sendMessage(Session session, String message) throws
   IOException{
7     session.getBasicRemote().sendText(String.format("%s (From Server,
   Session ID=%s)",message,session.getId()));
8 }
9
```

一个用户向其他用户群发



服务器向所有在线的javax.websocket.Session用户发送消息。

```
Java | 复制代码

1  /**
2   * 群发消息
3   * @param message 消息
4   */
5  public static void broadCastInfo(String message) throws IOException {
6      for (Session session : SessionSet) {
7          if(session.isOpen()){
8              sendMessage(session, message);
9          }
10     }
11 }
```

四、websocket实现简单聊天

WebSocketServer核心代码

1. @ServerEndpoint(value = "/ws/asset")表示websocket的接口服务地址
2. @OnOpen注解的方法，为连接建立成功时调用的方法
3. @OnClose注解的方法，为连接关闭调用的方法
4. @OnMessage注解的方法，为收到客户端消息后调用的方法
5. @OnError注解的方法，为出现异常时调用的方法

```
1 package com.mqxu.boot.websocket.server;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.springframework.stereotype.Component;
5
6 import javax.websocket.*;
7 import javax.websocket.server.PathParam;
8 import javax.websocket.server.ServerEndpoint;
9 import java.io.IOException;
10 import java.util.concurrent.ConcurrentHashMap;
11
12 /**
13  * @author mqxu
14  * @description: websocket服务层, 连接websocket的时候, 路径中传一个参数值id, 用
15  * 来区分不同页面推送不同的数据
16  * @date 2022-04-18
17  */
18 @ServerEndpoint(value = "/socket/{id}")
19 @Component
20 @Slf4j
21 public class WebSocketServer {
22     /**
23      * 静态变量, 用来记录当前在线连接数, 线程安全
24      */
25     private static int onlineCount = 0;
26
27     /**
28      * concurrent包的线程安全Set, 用来存放每个客户端对应的MyWebSocket对象
29      */
30     public static ConcurrentHashMap<Integer, WebSocketServer>
31     websocketSet = new ConcurrentHashMap<>();
32
33     /**
34      * 与某个客户端的连接会话, 需要通过它来给客户端发送数据
35      */
36     private Session session;
37
38     /**
39      * 传过来的id
40      */
41     private Integer id = 0;
42
43     /**
44      * 连接建立成功调用的方法
45      */
46 }
```

```

44     @OnOpen
45     public void onOpen(@PathParam(value = "id") Integer param, Session
session) {
46         //接收到发送消息的人员编号
47         this.id = param;
48         this.session = session;
49         //加入set中
50         websocketSet.put(param, this);
51         //在线数加1
52         addOnlineCount();
53         log.info("有新连接加入! 当前在线人数为" + getOnlineCount());
54         sendMessage("-连接已建立-");
55
56     }
57
58     /**
59     * 连接关闭调用的方法
60     */
61     @OnClose
62     public void onClose() {
63         if (id != null && id != 0) {
64             //从set中删除
65             websocketSet.remove(id);
66             //在线数减1
67             subOnlineCount();
68             log.info("有一连接关闭! 当前在线人数为" + getOnlineCount());
69         }
70     }
71
72     /**
73     * 收到客户端消息后调用的方法
74     *
75     * @param message 客户端发送过来的消息
76     */
77     @OnMessage
78     public void onMessage(String message, Session session) {
79         log.info("来自客户端的消息:" + message);
80         this.sendMessage(message);
81     }
82
83     /**
84     * 发生错误时调用
85     */
86     @OnError
87     public void onError(Session session, Throwable error) {
88         log.info("发生错误");
89         error.printStackTrace();
90     }

```



```

91
92
93 ▼ public void sendMessage(String message) {
94 ▼     try {
95         getSession().getBasicRemote().sendText(message);
96 ▼     } catch (IOException e) {
97         log.info("发生错误");
98         e.printStackTrace();
99     }
100 }
101
102 /**
103  * 给指定的人发送消息
104  *
105  * @param id      id
106  * @param message message
107  */
108 ▼ public void sendToMessageById(Integer id, String message) {
109 ▼     if (websocketSet.get(id) != null) {
110         websocketSet.get(id).sendMessage(message);
111 ▼     } else {
112         log.info("websocketSet中没有此key, 不推送消息");
113     }
114 }
115
116 /**
117  * 群发自定义消息
118  */
119 ▼ public void broadcastInfo(String message) {
120 ▼     for (WebSocketServer item : websocketSet.values()) {
121         item.sendMessage(message);
122     }
123 }
124
125 ▼ public Session getSession() {
126     return session;
127 }
128
129 ▼ public static synchronized int getOnlineCount() {
130     return onlineCount;
131 }
132
133 ▼ public static synchronized void addOnlineCount() {
134     WebSocketServer.onlineCount++;
135 }
136
137 ▼ public static synchronized void subOnlineCount() {
138     WebSocketServer.onlineCount--;

```

```
139     }  
140 }  
141
```

客户端代码, <public/websocket.html>

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>websocket页面</title>
6  </head>
7  <body>
8  <div id="app">
9      <div>
10         <label>
11             输入昵称
12             <input type="text" v-model="nickname" id="nickname"
placeholder="输入昵称">
13         </label>
14         <button @click="open">连接websocket</button>
15     </div>
16     <div>
17         <label>
18             输入内容
19             <input type="text" v-model="content" id="content"
placeholder="输入内容">
20         </label>
21         <button @click="sendMsg">发送消息</button>
22     </div>
23     <hr>
24     <div>
25         <h3> {{message}}</h3>
26     </div>
27
28 </div>
29
30 <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js">
</script>
31 <script>
32     const vm = new Vue({
33         el: "#app",
34         data() {
35             return {
36                 ws: null,
37                 content: "",
38                 message: "",
39                 nickname: ""
40             }
41         },
42         methods: {
```

```

43 ▼      open() {
44 ▼          if (this.nickname === "") {
45              alert("昵称不能为空");
46              return;
47          }
48          ws = new WebSocket(`ws://localhost:8888/websocket?
nickname=${this.nickname}`);
49
50 ▼          ws.onopen = () => {
51              console.log("websocket已经连接");
52          }
53 ▼          ws.onclose = () => {
54              console.log("websocket已经关闭");
55          }
56 ▼          ws.onerror = () => {
57              console.log("websocket出现异常");
58          }
59 ▼          ws.onmessage = (msg) => {
60              console.log(msg);
61              this.message = this.message.concat(msg.data);
62          }
63      },
64 ▼      sendMsg() {
65          ws.send(this.content);
66          console.log("发送成功");
67          this.content = ""
68      }
69  }
70  })
71  </script>
72  </body>
73  </html>

```

运行效果

websocket页面

localhost:8080/websocket.html

尺寸: 自适应 807 x 634 100% 已停用节流模式

输入昵称: taoranran 连接websocket

输入内容: 输入内容 发送消息

-连接已建立-hello

websocket已经连接 websocket.html:51

```

MessageEvent {isTrusted: true, data: '-连接已建立-', origin: 'ws://localhost:8080', lastEventId: '', source: null, ...}
  isTrusted: true
  bubbles: false
  cancelBubble: false
  cancelable: false
  composed: false
  currentTarget: WebSocket {url: 'ws://localhost:8080/socket/'}
  data: "-连接已建立-"
  defaultPrevented: false
  eventPhase: 0
  lastEventId: ""
  origin: "ws://localhost:8080"
  path: []
  ports: []
  returnValue: true
  source: null
  srcElement: WebSocket {url: 'ws://localhost:8080/socket/2', ...}
  target: WebSocket {url: 'ws://localhost:8080/socket/2', ...}
  timeStamp: 6536.200000017881
  type: "message"
  userActivation: null
  [[Prototype]]: MessageEvent

```

发送成功 websocket.html:66

```

MessageEvent {isTrusted: true, data: 'hello', origin: 'ws://localhost:8080', lastEventId: '', source: null, ...}
  isTrusted: true
  bubbles: false
  cancelBubble: false
  cancelable: false
  composed: false
  currentTarget: WebSocket {url: 'ws://localhost:8080/socket/2', ...}
  data: "hello"
  defaultPrevented: false
  eventPhase: 0
  lastEventId: ""
  origin: "ws://localhost:8080"
  path: []
  ports: []
  returnValue: true
  source: null
  srcElement: WebSocket {url: 'ws://localhost:8080/socket/2', ...}
  target: WebSocket {url: 'ws://localhost:8080/socket/2', ...}
  timeStamp: 6536.200000017881
  type: "message"
  userActivation: null
  [[Prototype]]: MessageEvent

```

控制台

```

INFO 85043 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet
INFO 85043 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
INFO 85043 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
INFO 85043 --- [nio-8080-exec-1] c.m.b.w.controller.SseController : emitter push message ....
INFO 85043 --- [nio-8080-exec-5] c.m.b.websocket.server.WebSocketServer : 有新连接加入! 当前在线人数为1
INFO 85043 --- [nio-8080-exec-6] c.m.b.websocket.server.WebSocketServer : 来自客户端的消息:hello

```