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

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

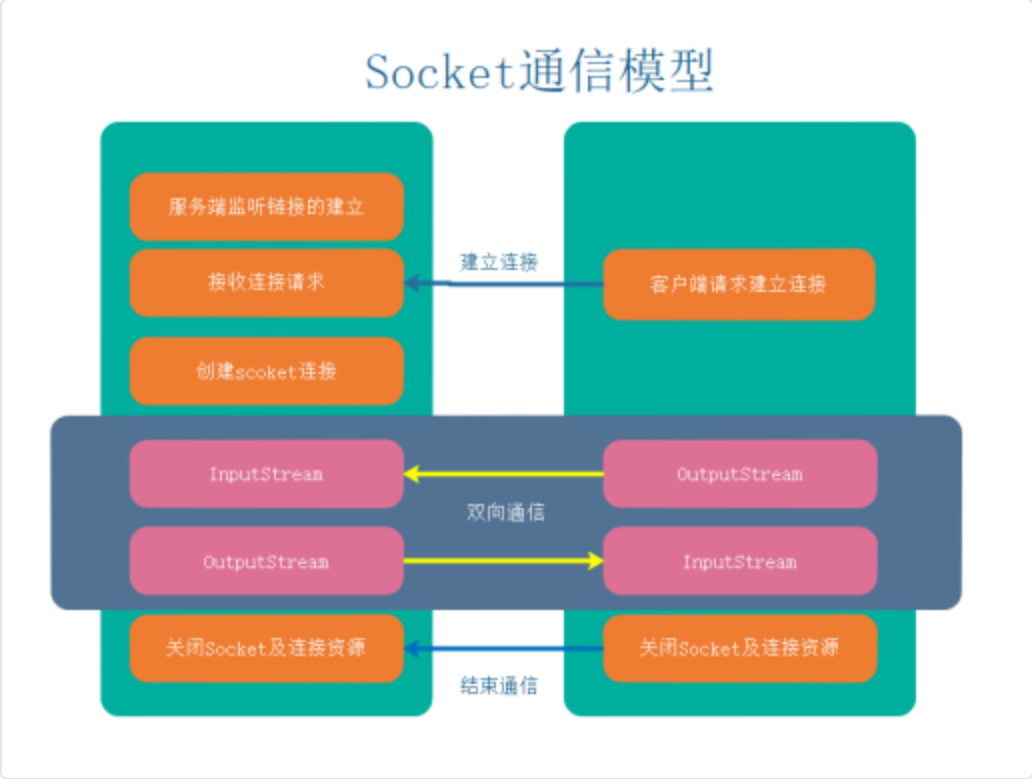
通常的做法就是需要以较小的间隔，频繁的向服务器建立http连接询问任务状态的更新，然后刷新页面显示状态。但这样做的后果就是浪费大量流量，对服务端造成了非常大的压力。

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。

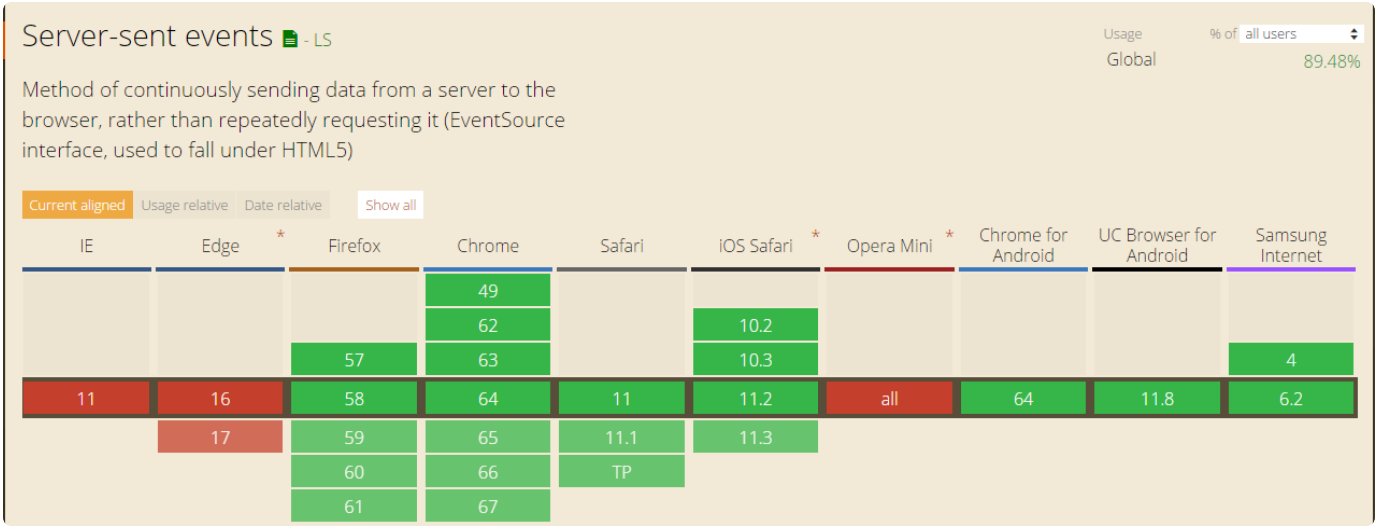
- 建立连接
- 服务端 –> 浏览器(连接保持)
- 关闭连接

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

Java | 复制代码

```java
package com.mqxu.boot.websocket.controller;

import lombok.extern.slf4j.Slf4j;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseBodyEmitter
;
import org.springframework.web.servlet.mvc.method.annotation.SseEmitter;

import java.io.IOException;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.List;

/**
 * @description: SSE练习
 * @author: mqxu
 * @date: 2022-04-18
 **/
@Slf4j
@Controller
public class SseController {

    @RequestMapping(value = "/server/info", method = {RequestMethod.GET},
produces = "text/event-stream;charset=UTF-8")
    public ResponseBodyEmitter pushMsg() {
        final SseEmitter emitter = new SseEmitter(0L);
        log.info("emitter push message .....");
        List<String> list = new ArrayList<>();
        list.add("aaa");
        list.add("bbb");
        list.add("ccc");
        try {
            emitter.send(list.toString(), MediaType.TEXT_EVENT_STREAM);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return emitter;
    }

    @RequestMapping(value = "/server/data", method = {RequestMethod.GET},
produces = "text/event-stream;charset=UTF-8")
```

```
    public ResponseBodyEmitter push() {
        final SseEmitter emitter = new SseEmitter(0L);
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            e.printStackTrace();
        }
        double money = Math.random() * 10;
        DecimalFormat df = new DecimalFormat(".00");
        String param = df.format(money);
        try {
            emitter.send("白菜价格行情:" + param + "元" + "\n\n",
MediaType.TEXT_EVENT_STREAM);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return emitter;
    }

}
```

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。

JavaScript | 复制代码

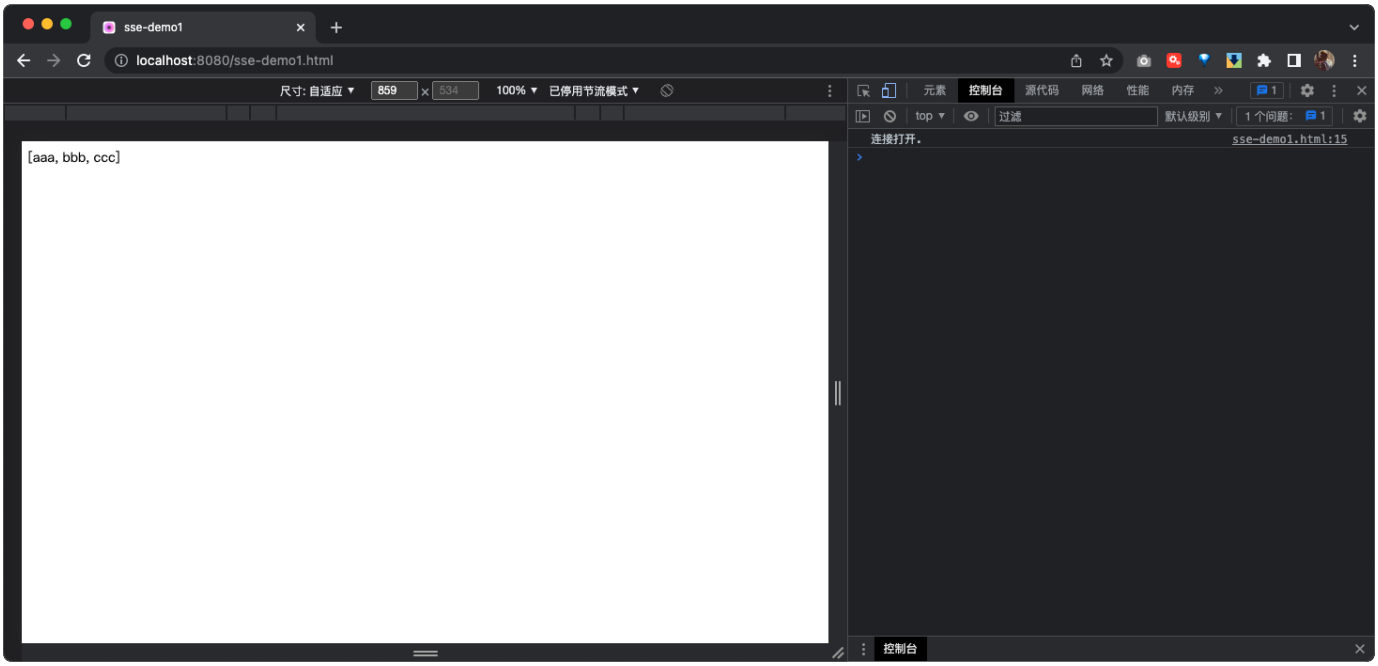
```javascript
var es = new EventSource('事件源名称') ;  //与事件源建立连接
//标准事件处理方法,还有onopen、onerror
es.onmessage = function(e) {
};
//可以监听自定义的事件名称
es.addEventListener('自定义事件名称', function(e) {
});
```

public/sse-demo1.html

HTML | 复制代码

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>sse-demo1</title>
</head>
<body>
<div id="msg_from_server">
</div>
<script>
    if (!!window.EventSource) {
        let source = new
EventSource('http://localhost:8080/server/info');
        let s = '';
        source.addEventListener('open', () => {
            console.log("连接打开.");
        }, false);
        source.addEventListener('message', (e) => {
            s += e.data + "<br/>"
            document.getElementById("msg_from_server").innerHTML = s;
        });
        source.addEventListener('error', (e) => {
            if (e.readyState === EventSource.CLOSED) {
                console.log("连接关闭");
            } else {
                console.log(e.readyState);
            }
        }, false);
    } else {
        alert(4);
        console.log("没有sse");
    }
</script>
</body>
</html>
```

运行效果

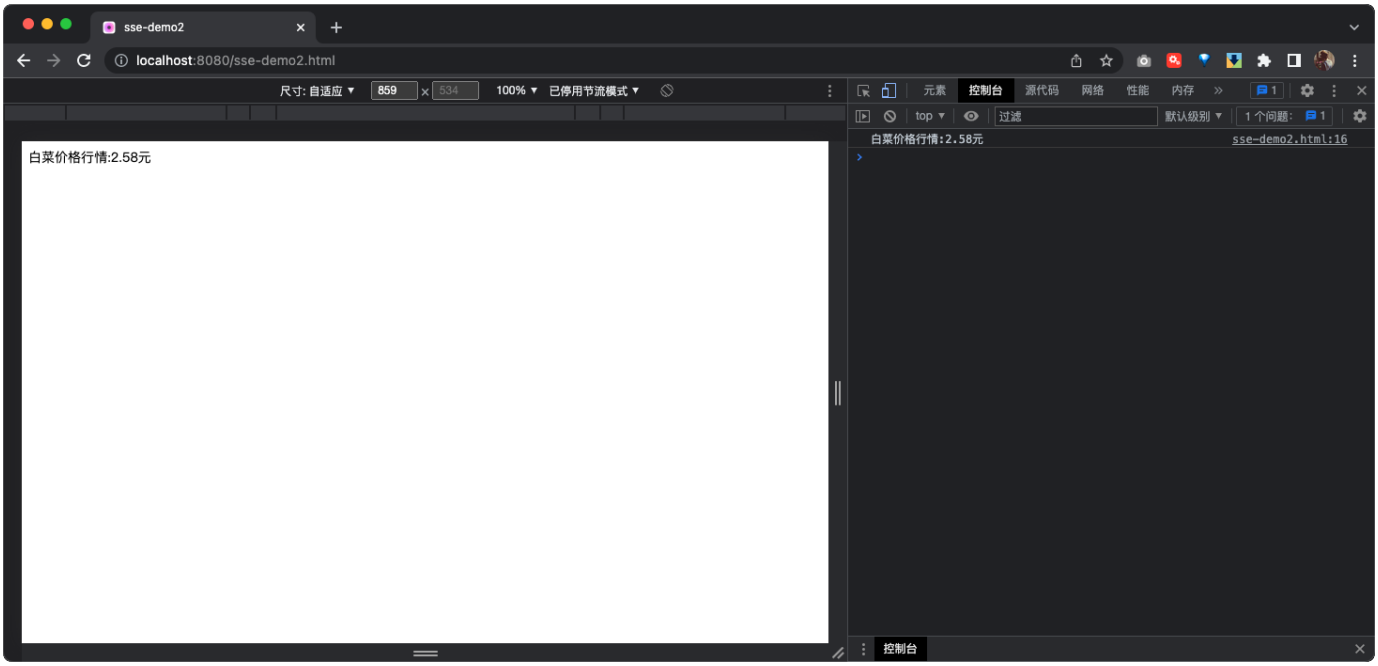


public/sse-demo2.html

HTML 复制代码

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>sse-demo2</title>
</head>

<body>
<div id="result">
</div>

<script type="text/javascript">
    //需要判断浏览器支不支持，可以去w3c进行查看
    const source = new EventSource('http://localhost:8080/server/data');
    source.onmessage = function (event) {
        console.info(event.data);
        document.getElementById('result').innerText = event.data
    };

</script>
</body>

</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

```xml
<!-- 引入websocket依赖 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```

- 编写配置类，开启websocket功能

```java
@Configuration
public class WebSocketConfig {
    @Bean
    public ServerEndpointExporter serverEndpointExporter() {
        return new ServerEndpointExporter();
    }
}
```

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。

```java
@Bean
public TomcatContextCustomizer tomcatContextCustomizer() {

    return new TomcatContextCustomizer() {
        @Override
        public void customize(Context context) {
            context.addServletContainerInitializer(new WsSci(), null);
        }

    };
}
```

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

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

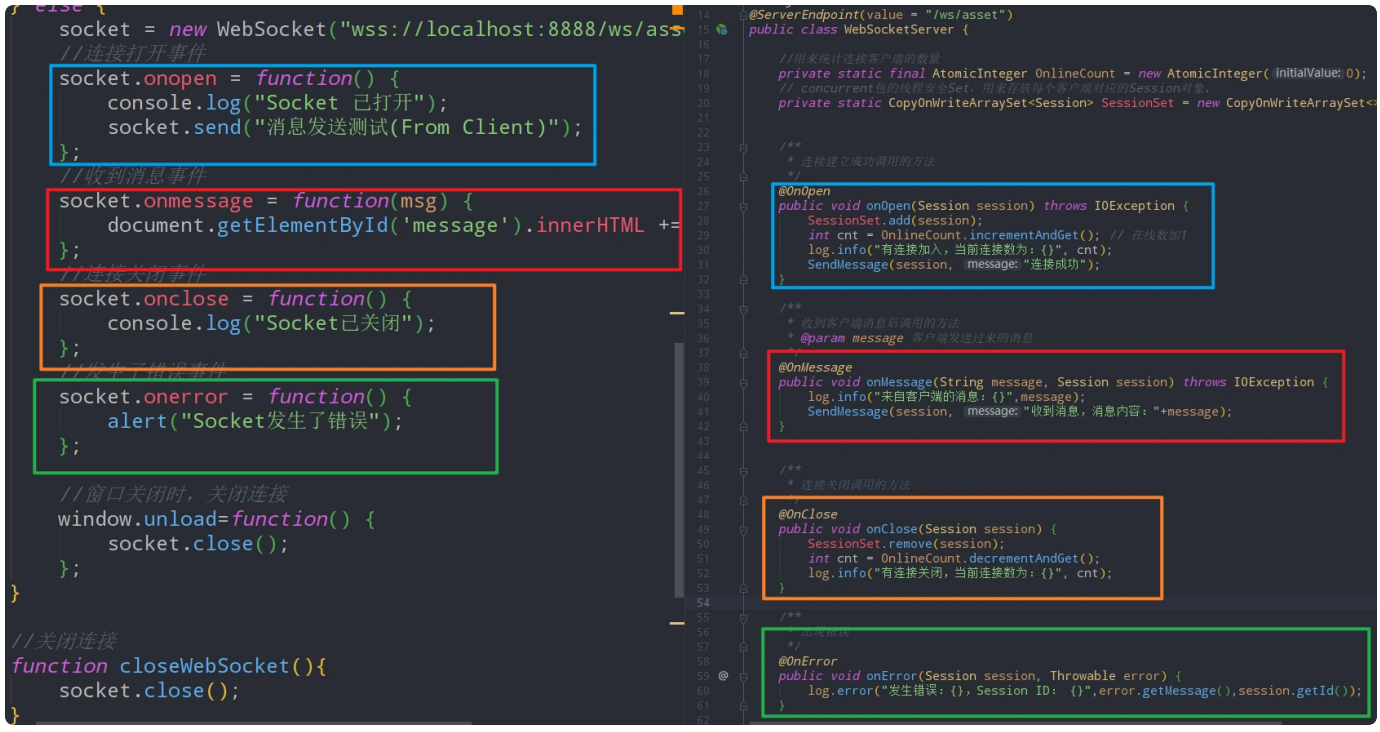
SpringBoot服务端WebSocket服务接收类定义如下：

```java
@Component
@Slf4j
@ServerEndpoint(value = "/ws/asset")
public class WebSocketServer {

}
```

3.2.全双工数据交互

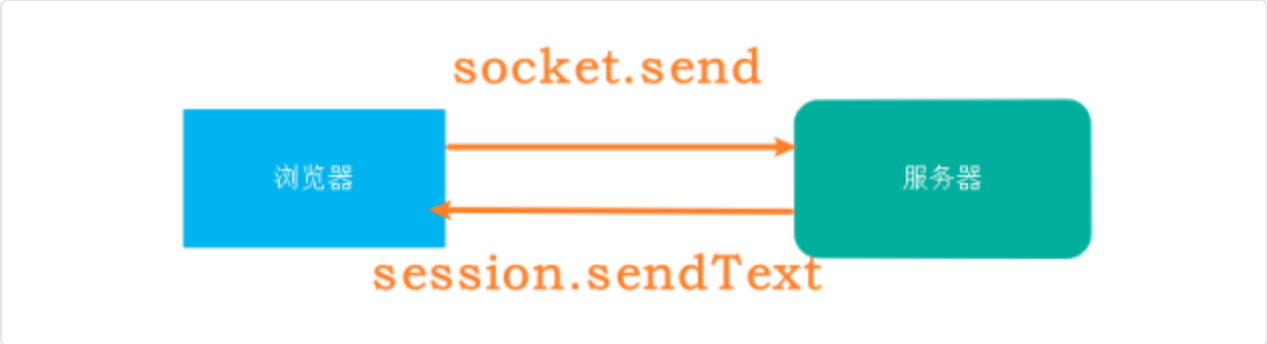
前端后端都有

- 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



3.3.数据发送

浏览器与服务器交换数据



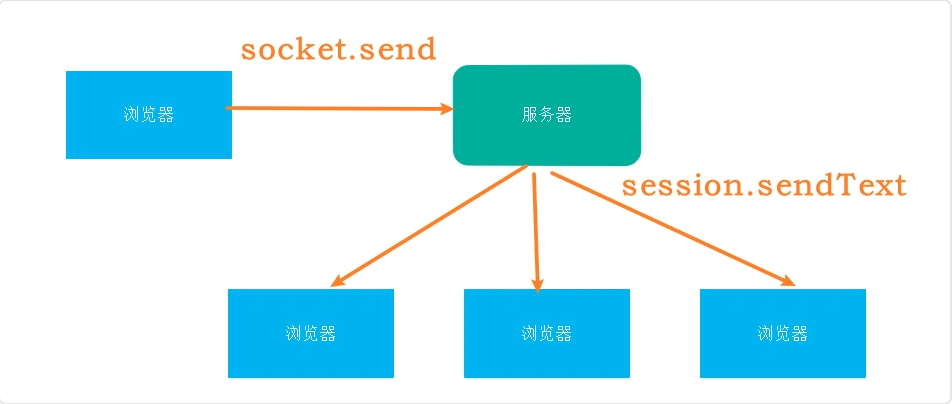
前端JS

```javascript
socket.send(message);
```

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

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

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

```java
/**
 * 群发消息
 * @param message  消息
 */
public static void broadCastInfo(String message) throws IOException {
    for (Session session : SessionSet) {
        if(session.isOpen()){
            sendMessage(session, message);
        }
    }
}
```

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

Java | 复制代码

```java
package com.mqxu.boot.websocket.server;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

import javax.websocket.*;
import javax.websocket.server.PathParam;
import javax.websocket.server.ServerEndpoint;
import java.io.IOException;
import java.util.concurrent.ConcurrentHashMap;

/**
 * @author mqxu
 * @description: webSocket服务层，连接webSocket的时候，路径中传一个参数值id，用来区分不同页面推送不同的数据
 * @date 2022-04-18
 */
@ServerEndpoint(value = "/socket/{id}")
@Component
@Slf4j
public class WebSocketServer {
    /**
     * 静态变量，用来记录当前在线连接数，线程安全
     */
    private static int onlineCount = 0;

    /**
     * concurrent包的线程安全Set，用来存放每个客户端对应的MyWebSocket对象
     */
    public static ConcurrentHashMap<Integer, WebSocketServer> webSocketSet = new ConcurrentHashMap<>();

    /**
     * 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
     */
    private Session session;

    /**
     * 传过来的id
     */
    private Integer id = 0;

    /**
     * 连接建立成功调用的方法
     */
```

```
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
```

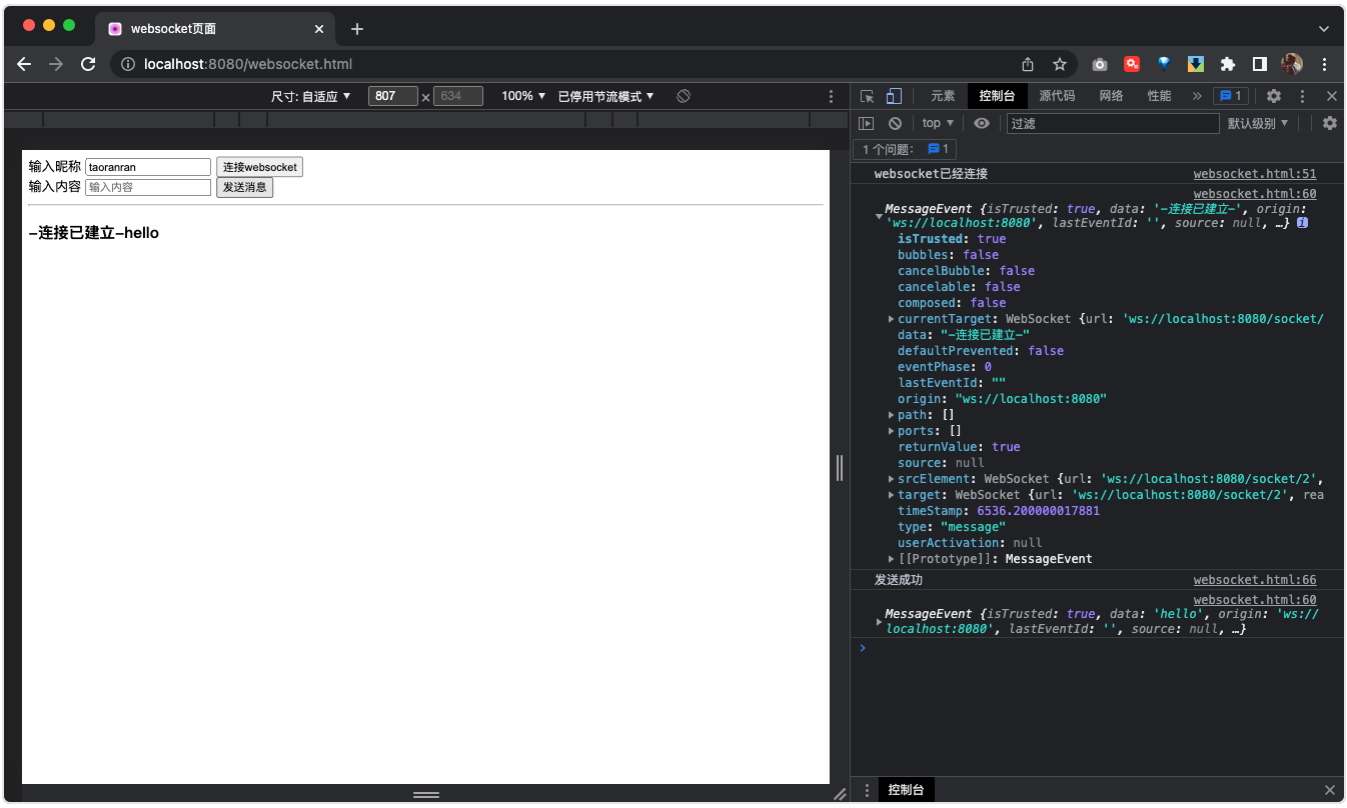
客户端代码，public/websocket.html

HTML | 复制代码

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>websocket页面</title>
</head>
<body>
<div id="app">
    <div>
        <label>
            输入昵称
            <input type="text" v-model="nickname" id="nickname" placeholder="输入昵称">
        </label>
        <button @click="open">连接websocket</button>
    </div>
    <div>
        <label>
            输入内容
            <input type="text" v-model="content" id="content" placeholder="输入内容">
        </label>
        <button @click="sendMsg">发送消息</button>
    </div>
    <hr>
    <div>
        <h3> {{message}}</h3>
    </div>

</div>

<script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
<script>
    const vm = new Vue({
        el: "#app",
        data() {
            return {
                ws: null,
                content: "",
                message: "",
                nickname: ""
            }
        },
        methods: {
```

```
            open() {
                if (this.nickname === "") {
                    alert("昵称不能为空");
                    return;
                }
                ws = new WebSocket(`ws://localhost:8888/websocket?
nickname=${this.nickname}`);

                ws.onopen = () => {
                    console.log("websocket已经连接");
                }
                ws.onclose = () => {
                    console.log("websocket已经关闭");
                }
                ws.onerror = () => {
                    console.log("websocket出现异常");
                }
                ws.onmessage = (msg) => {
                    console.log(msg);
                    this.message = this.message.concat(msg.data);
                }
            },
            sendMsg() {
                ws.send(this.content);
                console.log("发送成功");
                this.content = ""
            }
        }
    })
</script>
</body>
</html>
```

运行效果




```
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
```