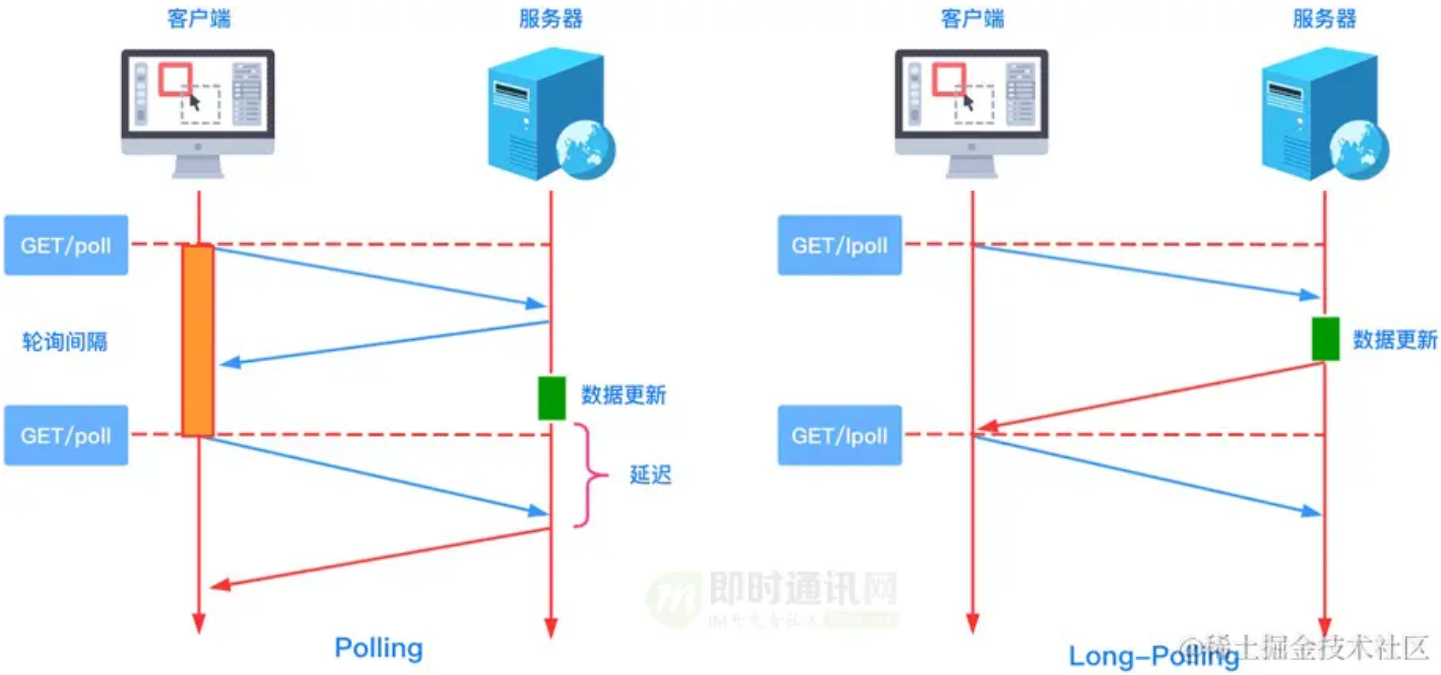
WebSocket

1. 背景

HTTP的局限性

HTTP 协议是 - 响应模式协议，客户端发起请求，服务器返回响应后连接关闭。

如果要实现数据变化时客户端马上收到的效果，比如在线聊天室，我们可以使用短轮询和长轮询。



这两种方法的缺点是服务器压力大，前者的延迟也很高。

WebSocket协议就是为了实现服务器主动向客户端发送消息而诞生的。

2. 应用

2.1 引入依赖

代码块

```
<!--        WebSocket -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-websocket</artifactId>
        </dependency>
<!--        json解析: gson-->
        <dependency>
```

```
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.10.1</version>
        </dependency>
```

2.2 Config

代码块

```
package com.gocile.websocketdemo2.websocket;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.server.standard.ServerEndpointExporter;

/**
 * 开启WebSocket支持
 * @author Gocile
 * @date 2025年05月13日 10:46
 */
@Configuration
public class WebSocketConfig {
    @Bean
    public ServerEndpointExporter serverEndpointExporter(){
        return new ServerEndpointExporter();
    }

}
```

2.3 Server

这里的server充当了controller的功能。

注解 @ServerEndpoint 的作用是定义一个服务端端点，表示将由这个类处理来自客户端的WebSocket连接请求。

其中 /im/{userId} 是路径模板，服务端通过不同的路径区分用户，比如路径为 /im/12 的连接会被视为id为12的用户拥有的连接。

在服务端端点类中，有四个核心的方法注解：

代码块

```
@OnOpen
//建立与客户端的连接时触发
@OnClose
```

```
4   //客户端关闭连接时触发
5   @OnMessage
6   //收到客户端消息时触发
7   @OnError
8   //发生错误时触发
```

代码块

```
1   package com.gocile.websocketdemo2.websocket;
2
3
4   import com.google.gson.Gson;
5   import io.micrometer.common.util.StringUtils;
6   import jakarta.websocket.*;
7   import jakarta.websocket.server.PathParam;
8   import jakarta.websocket.server.ServerEndpoint;
9   import org.springframework.stereotype.Component;
10  import java.io.IOException;
11  import java.util.concurrent.ConcurrentHashMap;
12
13
14  /**
15   * @author Gocile
16   * @date 2025年05月13日 11:01
17   */
18  @ServerEndpoint("/im/{userId}")//该注解定义一个服务端端点
19  @Component
20  public class WebSocketServer {
21
22      //当前在线连接数
23      private static int onlineCount = 0;
24      //用来存放每个客户端对应的WebSocket对象
25      private static ConcurrentHashMap<String,WebSocketServer> webSocketMap
26              = new ConcurrentHashMap<>();
27      //与某个客户端的连接会话
28      private Session session;
29      //用户id
30      private String userId="";
31
32
33      /**
34       * 连接建立成功时调用的方法
35       * */
36      @OnOpen
```

```
37      public void onOpen(Session session, @PathParam("userId")String userId){
38          this.session=session;
39          this.userId=userId;
40          //记录连接信息（已有则删除重新记录）
41          if(webSocketMap.containsKey(userId)){
42              webSocketMap.remove(userId);
43              webSocketMap.put(userId,this);
44          }else{
45              webSocketMap.put(userId,this);
46              //在线数+1
47              addOnlineCount();
48          }
49
50          System.out.println("用户"+userId+"连接,当前在线人数"+getOnlineCount());
51
52          try{
53              sendMessage("{\"msg\":\"连接成功\"}");
54          } catch (IOException e) {
55              System.out.println("用户"+userId+"连接状态异常，无法发送消息");
56          }
57      }
58
59      /**
60       * 连接关闭时调用的方法
61       * */
62      @OnClose
63      public void onClose(){
64          if(webSocketMap.containsKey(userId)){
65              webSocketMap.remove(userId);
66              //在线数-1
67              subOnlineCount();
68          }
69          System.out.println("用户"+userId+"断开连接，当前在线人
    数"+getOnlineCount());
70      }
71
72      /**
73       * 收到客户端消息后调用的方法
74       *
75       * @param message 客户端的消息
76       */
77      @OnMessage
78      public void onMessage(String message,Session session){
79          System.out.println("收到用户"+userId+"消息:"+message);
80          try {
81              //解析消息
82              Gson gson = new Gson();
```

```
83              Message message1 = gson.fromJson(message,Message.class);
84              String toUserId = message1.getToUserId();
85              //追加发送者id
86              message1.setFromUserId(this.userId);
```

2.4 Message实体（用于JSON解析）

代码块

```
1  package com.gocile.websocketdemo2.websocket;
2
3  /**
4   * @author Gocile
5   * @date 2025年05月13日 19:49
6   */
7  public class Message {
8      private String fromUserId;
9      private String toUserId;
10     private String contentText;
11
12     public String getFromUserId() {
13         return fromUserId;
14     }
15
16     public void setFromUserId(String fromUserId) {
17         this.fromUserId = fromUserId;
18     }
19
20     public String getToUserId() {
21         return toUserId;
22     }
23
24     public void setToUserId(String toUserId) {
25         this.toUserId = toUserId;
26     }
27
28     public String getContentText() {
29         return contentText;
30     }
31
32     public void setContentText(String contentText) {
33         this.contentText = contentText;
34     }
35 }
```

2.5 前端页面

填写信息后点击刷新信息，再开启socket或发送消息。更改输入框内容后必须再次点击刷新信息。

代码块

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>WebsocketDemo</title>
    <style>
        body{
            display: flex;
            justify-content: space-between;
            margin: 0 50px;
        }
    </style>
</head>
<body>
    <div class="left">
        <p>【userId】:
            <div><input id="userId" name="userId" type="text"></div>
            <p>【toUserId】:
            <div><input id="toUserId" name="toUserId" type="text"></div>
            <p>【contentText】:
            <div><input id="contentText" name="contentText" type="text"></div>
            <br><div><button onclick="refresh()">刷新信息</button></div>
            <br><div><button onclick="openSocket()">开启socket</button>
                   <button onclick="sendMessage()">发送消息
</button></div>
    </div>
    <div class="right">
        <h3>收到消息</h3>
        <p id="contentTextText">内容:</p>
        <p id="fromUserIdText">发送者:</p>
    </div>

</body>

<script>
    let socket;
    let userId;
    let toUserId;
    let contentText;
    console.log(userId);
```

```
41      function refresh() {
42          userId = document.getElementById("userId").value;
43          toUserId = document.getElementById("toUserId").value;
44          contentText = document.getElementById("contentText").value;
45      }
46      function openSocket() {
47          //获取WebSocket对象，指定要连接的服务器地址与端口并建立连接
48          //等同于socket = new WebSocket("ws://localhost:8888/xxxx/im/25");
49          let socketUrl = "http://localhost:8080/im/"+userId;
50          socketUrl = socketUrl.replace("https", "ws").replace("http", "ws");
51          console.log(socketUrl);
52          if (socket != null) {
53              socket.close();
54              socket = null;
55          }
56          socket = new WebSocket(socketUrl);
57
58          //websocket打开
59          socket.onopen = function () {
60              console.log("websocket已打开");
61          };
62
63          //获得消息
64          socket.onmessage = function (msg) {
65              console.log('msg:'+msg.data);
66              //处理并展示消息
67              document.getElementById("contentTextText").textContent = '内容: '+
    (JSON.parse(msg.data)?.contentText ?? '');
68              document.getElementById("fromUserIdText").textContent = '发送者: '+
    (JSON.parse(msg.data)?.fromUserId ?? '');
69          };
70
71          //关闭
72          socket.onclose = function () {
73              console.log("websocket已关闭");
74              console.log(socketUrl);
75          };
76
77          //异常
78          socket.onerror = function () {
79              console.log("websocket发生了错误");
80          }
81      }
82
83      //发送消息
84      function sendMessage() {
```

3. 拓展

3.1 心跳及重连机制

在使用websocket的过程中，遇到弱网或者网络暂时断连的情况时，服务端并没有触发onclose的事件，客户端也无法得知当前连接是否已经断开，服务端会继续向客户端发送数据，并且这些数据还会丢失。

为了保证连接的可持续性和稳定性，我们用心跳重连机制来检测客户端和服务端是否处于正常连接状态。

正式的项目会使用第三方websocket框架，稳定性、实时性有保证，并且包括一些心跳、重连机制。

> GoEasy专注于服务器与浏览器,浏览器与浏览器之间消息推送,完美兼容世界上的绝大多数浏览器,包括IE6, IE7之类的非常古老的浏览器。支持Uniapp,各种小程序，react，vue等所有主流Web前端技术。
>
> GoEasy采用 发布/订阅 的消息模式,帮助您非常轻松的实现一对一,一对多的通信。

3.1.1 心跳

客户端每隔一段时间（如30秒）向服务端发送特定格式的心跳消息（如 {"type": "ping"}）。

服务端收到心跳后立即回复（如{"type": "pong"}）。

客户端如果长时间（如60秒）未收到数据，则认为连接已断开，重新连接。

3.1.2 重连

指连接关闭后客户端自动尝试重新连接，而无需用户手动刷新页面。

需要在客户端的 onclose 事件中触发重连。

一般不会无限制频繁尝试重连（避免服务器压力过大），一方面重连间隔逐渐增加（1s, 2s, 4s, 8s...），另一方面重连失败几次后停止尝试并提示用户。

3.1.3 和轮询的性能对比

虽然心跳机制和轮询一样需要定时发送消息，但前者的性能开销小得多。

特性	WebSocket 心跳	HTTP 轮询
连接方式	长连接，一次建立后持续复用	短连接，每次轮询需重新建立 TCP 和 HTTP 连接
数据传输开销	极低（仅发送 Ping/Pong 帧或小数据包）	较高（HTTP 头 + 可能的冗余数据）
服务端压力	低（仅需回复轻量级响应）	高（每次轮询需完整处理 HTTP 请求）
实时性	高（支持双向实时推送）	低（依赖轮询间隔）
适用场景	高频交互场景（如聊天、实时协作）	低频更新场景（如天气预报）

3.2 对比EventSource

EventSource是另一种能实现实时通信的技术，但它仅支持服务端向客户端单向推送数据，客户端仍需要使用HTTP请求发送数据。

相较于WebSocket，EventSource的优点是：基于HTTP协议（WebSocket基于TCP协议），兼容性好，简单易用（如没有心跳机制，连接断开时会自动重连）。

在对双向通信能力要求不高的场景下（如新闻推送），EventSource是更好的选择。

4. 参考

CSDN：SpringBoot2.0集成WebSocket，实现后台向前端推送信息

稀土掘金：3分钟使用 WebSocket 搭建属于自己的聊天室（WebSocket 原理、应用解析）

CSDN：Websocket心跳检测、重连机制

公众号：ChatGPT 对话为什么不用 WebSocket 而使用 EventSource?