WebSocket 从入门到入土

一.WebSocket 基本概念

1.WebSocket是什么?

> WebSocket 是基于 TCP 的一种新的应用层网络协议。它提供了一个全双工的通道，允许服务器和客户端之间实时双向通信。因此，在 WebSocket 中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输，客户端和服务器之间的数据交换变得更加简单。

2.与 HTTP 协议的区别

与 HTTP 协议相比，WebSocket 具有以下优点：

1. 更高的实时性能：WebSocket 允许服务器和客户端之间实时双向通信，从而提高了实时通信场景中的性能。
2. 更少的网络开销：HTTP 请求和响应之间需要额外的数据传输，而 WebSocket 通过在同一个连接上双向通信，减少了网络开销。
3. 更灵活的通信方式：HTTP 请求和响应通常是一一对应的，而 WebSocket 允许服务器和客户端之间以多种方式进行通信，例如消息 Push、事件推送等。
4. 更简洁的 API：WebSocket 提供了简洁的 API，使得客户端开发人员可以更轻松地进行实时通信。

当然肯定有缺点的：

1. 不支持无连接: WebSocket 是一种持久化的协议，这意味着连接不会在一次请求之后立即断开。这是有利的，因为它消除了建立连接的开销，但是也可能导致一些资源泄漏的问题。
2. 不支持广泛: WebSocket 是 HTML5 中的一种标准协议，虽然现代浏览器都支持，但是一些旧的浏览器可能不支持 WebSocket。
3. 需要特殊的服务器支持: WebSocket 需要服务端支持，只有特定的服务器才能够实现 WebSocket 协议。这可能会增加系统的复杂性和部署的难度。
4. 数据流不兼容: WebSocket 的数据流格式与 HTTP 不同，这意味着在不同的网络环境下，WebSocket 的表现可能会有所不同。

3.WebSocket工作原理

5. **握手阶段**

WebSocket在建立连接时需要进行握手阶段。握手阶段包括以下几个步骤：

- 客户端向服务端发送请求，请求建立WebSocket连接。请求中包含一个Sec-WebSocket-Key参数，用于生成WebSocket的随机密钥。
- 服务端接收到请求后，生成一个随机密钥，并使用随机密钥生成一个新的Sec-WebSocket-Accept参数。
- 客户端接收到服务端发送的新的Sec-WebSocket-Accept参数后，使用原来的随机密钥和新的Sec-WebSocket-Accept参数共同生成一个新的Sec-WebSocket-Key参数，用于加密数据传输。
- 客户端将新的Sec-WebSocket-Key参数发送给服务端，服务端接收到后，使用该参数加密数据传输。

6. **数据传输阶段**

建立连接后，客户端和服务端就可以通过WebSocket进行实时双向通信。数据传输阶段包括以下几个步骤：

- 客户端向服务端发送数据，服务端收到数据后将其转发给其他客户端。
- 服务端向客户端发送数据，客户端收到数据后进行处理。

双方如何进行相互传输数据的 具体的数据格式是怎么样的呢？WebSocket 的每条消息可能会被切分成多个数据帧（最小单位）。发送端会将消息切割成多个帧发送给接收端，接收端接收消息帧，并将关联的帧重新组装成完整的消息。

发送方 -> 接收方：ping。

接收方 -> 发送方：pong。

ping 、pong 的操作，对应的是 WebSocket 的两个控制帧

7. **关闭阶段**

当不再需要WebSocket连接时，需要进行关闭阶段。关闭阶段包括以下几个步骤：

- 客户端向服务端发送关闭请求，请求中包含一个WebSocket的随机密钥。
- 服务端接收到关闭请求后，向客户端发送关闭响应，关闭响应中包含服务端生成的随机密钥。
- 客户端收到关闭响应后，关闭WebSocket连接。

总的来说，WebSocket通过握手阶段、数据传输阶段和关闭阶段实现了服务器和客户端之间的实时双向通信。

二.WebSocket 数据帧结构和控制帧结构。

8. **数据帧结构**

WebSocket 数据帧主要包括两个部分：帧头和有效载荷。以下是 WebSocket 数据帧结构的简要介绍：

- 帧头：帧头包括四个部分：fin、rsv1、rsv2、rsv3、opcode、masked 和 payload_length。其中，fin 表示数据帧的结束标志，rsv1、rsv2、rsv3 表示保留字段，opcode 表示数据帧的类型，masked 表示是否进行掩码处理，payload_length 表示有效载荷的长度。

- 有效载荷：有效载荷是数据帧中实际的数据部分，它由客户端和服务端进行数据传输。

9. **控制帧结构**

除了数据帧之外，WebSocket 协议还包括一些控制帧，主要包括 Ping、Pong 和 Close 帧。以下是 WebSocket 控制帧结构的简要介绍：

- Ping 帧：Ping 帧用于测试客户端和服务端之间的连接状态，客户端向服务端发送 Ping 帧，服务端收到后需要向客户端发送 Pong 帧进行响应。
- Pong 帧：Pong 帧用于响应客户端的 Ping 帧，它用于测试客户端和服务端之间的连接状态。
- Close 帧：Close 帧用于关闭客户端和服务端之间的连接，它包括四个部分：fin、rsv1、rsv2、rsv3、opcode、masked 和 payload_length。其中，opcode 的值为 8，表示 Close 帧。

三. JavaScript 中 WebSocket 对象的属性和方法，以及如何创建和连接 WebSocket。

WebSocket 对象的属性和方法：

1. `WebSocket` 对象：WebSocket 对象表示一个新的 WebSocket 连接。
2. `WebSocket.onopen` 事件处理程序：当 WebSocket 连接打开时触发。
3. `WebSocket.onmessage` 事件处理程序：当接收到来自 WebSocket 的消息时触发。
4. `WebSocket.onerror` 事件处理程序：当 WebSocket 发生错误时触发。
5. `WebSocket.onclose` 事件处理程序：当 WebSocket 连接关闭时触发。
6. `WebSocket.send` 方法：向 WebSocket 发送数据。
7. `WebSocket.close` 方法：关闭 WebSocket 连接。

创建和连接 WebSocket：

1. 创建 WebSocket 对象：

```
1 javascript
2 复制代码
3 var socket = new WebSocket('ws://example.com');
```

其中，`ws://example.com` 是 WebSocket 的 URL，表示要连接的服务器。

1. 连接 WebSocket：

使用 `WebSocket.onopen` 事件处理程序检查 WebSocket 是否成功连接。

```
1 javascript
```

```
2 复制代码
3 socket.onopen = function() {    console.log('WebSocket connected');};
```

1. 接收来自 WebSocket 的消息：

使用 `WebSocket.onmessage` 事件处理程序接收来自 WebSocket 的消息。

```
1 javascript
2 复制代码
3 socket.onmessage = function(event) {    console.log('WebSocket message:',
  event.data);};
```

1. 向 WebSocket 发送消息：

使用 `WebSocket.send` 方法向 WebSocket 发送消息。

```
1 javascript
2 复制代码
3 socket.send('Hello, WebSocket!');
```

1. 关闭 WebSocket：

当需要关闭 WebSocket 时，使用 `WebSocket.close` 方法。

```
1 javascript
2 复制代码
3 socket.close();
```

注意：在 WebSocket 连接成功打开和关闭时，会分别触发 `WebSocket.onopen` 和 `WebSocket.onclose` 事件。在接收到来自 WebSocket 的消息时，会触发 `WebSocket.onmessage` 事件。当 WebSocket 发生错误时，会触发 `WebSocket.onerror` 事件。

四.webSocket简单示例

以下是一个简单的 WebSocket 编程示例，通过 WebSocket 向服务器发送数据，并接收服务器返回的数据：

1. 首先，创建一个 HTML 文件，添加一个按钮和一个用于显示消息的文本框：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>WebSocket 示例</title>
</head>
<body>
    <button id="sendBtn">发送消息</button>
    <textarea id="messageBox" readonly></textarea>
    <script src="main.js"></script>
</body>
</html>

```

1. 接下来，创建一个 JavaScript 文件（例如 `main.js`），并在其中编写以下代码：

```
// 获取按钮和文本框元素
const sendBtn = document.getElementById('sendBtn');
const messageBox = document.getElementById('messageBox');

// 创建 WebSocket 对象
const socket = new WebSocket('ws://echo.websocket.org'); // 使用一个 WebSocket 服务器进行测试

// 设置 WebSocket 连接打开时的回调函数
socket.onopen = function() {
    console.log('WebSocket 连接已打开');
};

// 设置 WebSocket 接收到消息时的回调函数
socket.onmessage = function(event) {
    console.log('WebSocket 接收到消息:', event.data);
    messageBox.value += event.data + '\n';
};

// 设置 WebSocket 发生错误时的回调函数
socket.onerror = function() {
    console.log('WebSocket 发生错误');
};

// 设置 WebSocket 连接关闭时的回调函数
socket.onclose = function() {
    console.log('WebSocket 连接已关闭');
};

```

```
// 点击按钮时发送消息
sendBtn.onclick = function() {
    const message = 'Hello, WebSocket!';
    socket.send(message);
    messageBox.value += '发送消息: ' + message + '\n';
};

```

五.webSocket应用场景

1. 实时通信：WebSocket 非常适合实时通信场景，例如聊天室、在线游戏、实时数据传输等。通过 WebSocket，客户端和服务器之间可以实时通信，无需依赖轮询，从而提高通信效率和减少网络延迟。
2. 监控数据传输：WebSocket 可以在监控系统中实现实时数据传输，例如通过 WebSocket，客户端可以实时接收和处理监控数据，而无需等待轮询数据。
3. 自动化控制：WebSocket 可以在自动化系统中实现远程控制，例如通过 WebSocket，客户端可以远程控制设备或系统，而无需直接操作。
4. 数据分析：WebSocket 可以在数据分析场景中实现实时数据传输和处理，例如通过 WebSocket，客户端可以实时接收和处理数据，而无需等待数据存储和分析。
5. 人工智能：WebSocket 可以在人工智能场景中实现实时数据传输和处理，例如通过 WebSocket，客户端可以实时接收和处理数据，而无需等待数据处理和分析。

六.WebSocket 错误处理

WebSocket 的错误处理

1. `WebSocket is not supported`：当浏览器不支持 WebSocket 时，会出现此错误。解决方法是在浏览器兼容性列表中检查是否支持 WebSocket。
2. `WebSocket connection closed`：当 WebSocket 连接被关闭时，会出现此错误。解决方法是在 `WebSocket.onclose` 事件处理程序中进行错误处理。
3. `WebSocket error`：当 WebSocket 发生错误时，会出现此错误。解决方法是在 `WebSocket.onerror` 事件处理程序中进行错误处理。
4. `WebSocket timeout`：当 WebSocket 连接超时时，会出现此错误。解决方法是在 `WebSocket.ontimeout` 事件处理程序中进行错误处理。
5. `WebSocket handshake error`：当 WebSocket 握手失败时，会出现此错误。解决方法是在 `WebSocket.onerror` 事件处理程序中进行错误处理。
6. `WebSocket closed by server`：当 WebSocket 连接被服务器关闭时，会出现此错误。解决方法是在 `WebSocket.onclose` 事件处理程序中进行错误处理。

7. `WebSocket closed by protocol`：当 WebSocket 连接被协议错误关闭时，会出现此错误。解决方法是在 `WebSocket.onclose` 事件处理程序中进行错误处理。
8. `WebSocket closed by network`：当 WebSocket 连接被网络错误关闭时，会出现此错误。解决方法是在 `WebSocket.onclose` 事件处理程序中进行错误处理。
9. `WebSocket closed by server`：当 WebSocket 连接被服务器错误关闭时，会出现此错误。解决方法是在 `WebSocket.onclose` 事件处理程序中进行错误处理。

通过为 `WebSocket` 对象的 `onclose`、`onerror` 和 `ontimeout` 事件添加处理程序，可以及时捕获和处理 WebSocket 错误，从而确保程序的稳定性和可靠性。

七.利用单例模式创建完整的wesocket连接

```
class webSocketClass {
    constructor(thatVue) {
      this.lockReconnect = false;
      this.localUrl = process.env.NODE_ENV === 'production' ? 你的websocket生产地址' : '测试地址';
      this.globalCallback = null;
      this.userClose = false;
      this.createWebSocket();
      this.webSocketState = false
      this.thatVue = thatVue
    }

    createWebSocket() {
      let that = this;
      // console.log('开始创建websocket新的实例', new Date().toLocaleString())
      if( typeof(WebSocket) != "function" ) {
        alert("您的浏览器不支持Websocket通信协议，请更换浏览器为Chrome或者Firefox再次使用! ")
      }
      try {
        that.ws = new WebSocket(that.localUrl);
        that.initEventHandle();
        that.startHeartBeat()
      } catch (e) {
        that.reconnect();
      }
    }

    //初始化
    initEventHandle() {
      let that = this;
      // //连接成功建立后响应
```

```
      that.ws.onopen = function() {
        console.log("连接成功");
      };
      //连接关闭后响应
      that.ws.onclose = function() {
        // console.log('websocket连接断开', new Date().toLocaleString())
        if (!that.userClose) {
          that.reconnect(); //重连
        }
      };
      that.ws.onerror = function() {
        // console.log('websocket连接发生错误', new Date().toLocaleString())
        if (!that.userClose) {
          that.reconnect(); //重连
        }
      };
      that.ws.onmessage = function(event) {
        that.getWebSocketMsg(that.globalCallback);
        // console.log('socket server return '+ event.data);
      };
    }
    startHeartBeat () {
      // console.log('心跳开始建立', new Date().toLocaleString())
      setTimeout(() => {
          let params = {
            request: 'ping',
          }
          this.webSocketSendMsg(JSON.stringify(params))
          this.webSocketState = true; // 在连接成功后设置为true
          this.waitingServer()
      }, 30000)
    }
    //延时等待服务端响应，通过webSocketState判断是否连线成功
    waitingServer () {
      this.webSocketState = false//在线状态
      setTimeout(() => {
          if(this.webSocketState) {
              this.startHeartBeat()
              return
          }
          // console.log('心跳无响应，已断线', new Date().toLocaleString())
          try {
            this.closeSocket()
          } catch(e) {
            console.log('连接已关闭，无需关闭', new Date().toLocaleString())
          }
          this.reconnect()
```

```
78            //重连操作
79        }, 5000)
80      }
81      reconnect() {
82        let that = this;
83        if (that.lockReconnect) return;
84        that.lockReconnect = true; //没连接上会一直重连，设置延迟避免请求过多
85        setTimeout(function() {
86          that.createWebSocket();
87          that.thatVue.openSuccess(that) //重连之后做一些事情
88          that.thatVue.getSocketMsg(that)
89          that.lockReconnect = false;
90        }, 15000);
91      }
92
93      webSocketSendMsg(msg) {
94        this.ws.send(msg);
95      }
96
97      getWebSocketMsg(callback) {
98        this.ws.onmessage = ev => {
99          callback && callback(ev);
100       };
101     }
102     onopenSuccess(callback) {
103       this.ws.onopen = () => {
104         // console.log("连接成功", new Date().toLocaleString())
105         callback && callback()
106       }
107     }
108     closeSocket() {
109       let that = this;
110       if (that.ws) {
111         that.userClose = true;
112         that.ws.close();
113       }
114     }
115   }
116   export default webSocketClass;
117
118
```