代码案例：<https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_049_webSocket>

HTML5给Web浏览器带来了全双工TCP连接websocket标准服务器的能力。

换句话说，浏览器能够与服务器建立连接，通过已建立的通信信道来发送和接收数据而不需要由HTTP协议引入额外其他的开销来实现。

包括jdk、spring boot实现、GoEasy、Pushlets、Pushlet等

# jdk

javaEE提供的API以及实现方案。

## WebSocket服务器端

@ServerEndpoint("/websocket")

public class WebSocketTest {

@OnMessage

public void onMessage(String message, Session session)

throws IOException, InterruptedException {

// Print the client message for testing purposes

System.out.println("Received: " + message);

// Send the first message to the client

session.getBasicRemote().sendText("This is the first server message");

// Send 3 messages to the client every 5 seconds

int sentMessages = 0;

while(sentMessages < 3){

Thread.sleep(5000);

session.getBasicRemote().

sendText("This is an intermediate server message. Count: "

+ sentMessages);

sentMessages++;

}

// Send a final message to the client

session.getBasicRemote().sendText("This is the last server message");

}

@OnOpen

public void onOpen() {

System.out.println("Client connected");

}

@OnClose

public void onClose() {

System.out.println("Connection closed");

}

}

@ServerEndpoint 注解是一个类层次的注解，它的功能主要是将目前的类定义成一个websocket服务器端。注解的值将被用于监听用户连接的终端访问URL地址。

onOpen 和 onClose 方法分别被@OnOpen和@OnClose 所注解。这两个注解的作用不言自明：他们定义了当一个新用户连接和断开的时候所调用的方法。

onMessage 方法被@OnMessage所注解。这个注解定义了当服务器接收到客户端发送的消息时所调用的方法。

注意：这个方法可能包含一个javax.websocket.Session可选参数（在我们的例子里就是session参数）。

如果有这个参数，容器将会把当前发送消息客户端的连接Session注入进去。

本例中我们仅仅是将客户端消息内容打印出来，然后首先我们将发送一条开始消息，之后间隔5秒向客户端发送1条测试消息，共发送3次，

最后向客户端发送最后一条结束消息。

## 客户端

var websocket = null;

//判断当前浏览器是否支持WebSocket

if ('WebSocket' in window) {

websocket = new WebSocket("ws://localhost:8080/20160910\_websocket/ws.do/${userSession.id }/${userSession.name }/${loginIp }");

} else {

alert('当前浏览器 Not support websocket')

}

//1.连接发生错误的回调方法

websocket.onerror = function () {

setMessageInnerHTML("WebSocket连接发生错误");

};

//2.连接成功建立的回调方法

websocket.onopen = function () {

setMessageInnerHTML("WebSocket连接成功");

}

//3.接收到消息的回调方法

websocket.onmessage = function (event) {

setMessageInnerHTML(event.data);

}

//4.连接关闭的回调方法

websocket.onclose = function () {

setMessageInnerHTML("WebSocket连接关闭");

}

//5.监听窗口关闭事件，当窗口关闭时，主动去关闭websocket连接，防止连接还没断开就关闭窗口，server端会抛异常。

window.onbeforeunload = function () {

closeWebSocket();

}

//6.将消息显示在网页上

function setMessageInnerHTML(innerHTML) {

document.getElementById('message').innerHTML += innerHTML + '<br/>';

}

//7.关闭WebSocket连接

function closeWebSocket() {

websocket.close();

}

//8.发送消息

function send() {

var friendId = document.getElementById('friend').value;

var message = document.getElementById('text').value;

websocket.send(friendId + ":" + message);

}

## WebSockets 握手（升级过程）

客户端和服务器端TCP连接建立在HTTP协议握手发生之后。通过HTTP流量调试，很容易观察到握手。客户端一创建一个 WebSocket实例，就会出现如下请求和服务器端响应：

注意: 我们只录入了WebSocket握手所用到的HTTP头。

请求:

GET /byteslounge/websocket HTTP/1.1

Connection: Upgrade

Upgrade: websocket

Sec-WebSocket-Key: wVlUJ/tu9g6EBZEh51iDvQ==

响应:

HTTP/1.1 101 Web Socket Protocol Handshake

Upgrade: websocket

Sec-WebSocket-Accept: 2TNh+0h5gTX019lci6mnvS66PSY=

注意：进行连接需要将通过Upgrade and Upgrade将协议升级到支持websocket HTTP头的Websocket协议。服务器响应表明请求被接受，协议将转换到WebSocket协议（HTTP状态码101）:

HTTP/1.1 101 Web Socket Protocol Handshake

# spring

## pom.xml引入依赖

<!-- webSocket 秒杀通知-->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-websocket</artifactId>

</dependency>

WebSocketConfig 配置：

/\*\*

\* WebSocket配置

\* 创建者 爪哇笔记

\* 创建时间 2018年5月29日

\*/

@Configuration

public class WebSocketConfig {

@Bean

public ServerEndpointExporter serverEndpointExporter() {

return new ServerEndpointExporter();

}

}

## WebSocketServer 配置

@ServerEndpoint("/websocket/{userId}")

@Component

public class WebSocketServer {

private final static Logger log = LoggerFactory.getLogger(WebSocketServer.class);

//静态变量，用来记录当前在线连接数。应该把它设计成线程安全的。

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。

private static CopyOnWriteArraySet<WebSocketServer> webSocketSet = new CopyOnWriteArraySet<WebSocketServer>();

//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;

//接收userId

private String userId="";

/\*\*

\* 连接建立成功调用的方法\*/

@OnOpen

public void onOpen(Session session,@PathParam("userId") String userId) {

this.session = session;

webSocketSet.add(this); //加入set中

addOnlineCount(); //在线数加1

log.info("有新窗口开始监听:"+userId+",当前在线人数为" + getOnlineCount());

this.userId=userId;

try {

sendMessage("连接成功");

} catch (IOException e) {

log.error("websocket IO异常");

}

}

/\*\*

\* 连接关闭调用的方法

\*/

@OnClose

public void onClose() {

webSocketSet.remove(this); //从set中删除

subOnlineCount(); //在线数减1

log.info("有一连接关闭！当前在线人数为" + getOnlineCount());

}

/\*\*

\* 收到客户端消息后调用的方法

\* @param message 客户端发送过来的消息\*/

@OnMessage

public void onMessage(String message, Session session) {

log.info("收到来自窗口"+userId+"的信息:"+message);

//群发消息

for (WebSocketServer item : webSocketSet) {

try {

item.sendMessage(message);

} catch (IOException e) {

e.printStackTrace();

}

}

}

/\*\*

\* @param session

\* @param error

\*/

@OnError

public void onError(Session session, Throwable error) {

log.error("发生错误");

error.printStackTrace();

}

/\*\*

\* 实现服务器主动推送

\*/

public void sendMessage(String message) throws IOException {

this.session.getBasicRemote().sendText(message);

}

/\*\*

\* 群发自定义消息

\* \*/

public static void sendInfo(String message,@PathParam("userId") String userId){

log.info("推送消息到窗口"+userId+"，推送内容:"+message);

for (WebSocketServer item : webSocketSet) {

try {

//这里可以设定只推送给这个userId的，为null则全部推送

if(userId==null) {

item.sendMessage(message);

}else if(item.userId.equals(userId)){

item.sendMessage(message);

}

} catch (IOException e) {

continue;

}

}

}

public static synchronized int getOnlineCount() {

return onlineCount;

}

public static synchronized void addOnlineCount() {

WebSocketServer.onlineCount++;

}

public static synchronized void subOnlineCount() {

WebSocketServer.onlineCount--;

}

}

## webSocket.js

$(function(){

socket.init();

});

var basePath = "ws://localhost:8080/seckill/";

socket = {

webSocket : "",

init : function() {

//userId：自行追加

if ('WebSocket' in window) {

webSocket = new WebSocket(basePath+'websocket/1');

}

else if ('MozWebSocket' in window) {

webSocket = new MozWebSocket(basePath+"websocket/1");

}

else {

webSocket = new SockJS(basePath+"sockjs/websocket");

}

webSocket.onerror = function(event) {

alert("websockt连接发生错误，请刷新页面重试!")

};

webSocket.onopen = function(event) {

};

webSocket.onmessage = function(event) {

var message = event.data;

alert(message)//判断秒杀是否成功、自行处理逻辑

};

}

}

## 客户端API

### 客户端与服务器通信

send() 向远程服务器发送数据

close() 关闭该websocket链接

### 监听函数

onopen 当网络连接建立时触发该事件

onerror 当网络发生错误时触发该事件

onclose 当websocket被关闭时触发该事件

onmessage 当websocket接收到服务器发来的消息的时触发的事件，也是通信中最重要的一个监听事件。msg.data

### readyState属性

这个属性可以返回websocket所处的状态。

CONNECTING(0) websocket正尝试与服务器建立连接

OPEN(1) websocket与服务器已经建立连接

CLOSING(2) websocket正在关闭与服务器的连接

CLOSED(3) websocket已经关闭了与服务器的连接

# goeasy

国产实现方案，需要收费。

GoEasy实时Web推送，支持后台推送和前台推送两种：后台推送可以选择Java SDK、 Restful API支持所有开发语言；前台推送：JS推送。

无论选择哪种方式推送代码都十分简单（10分钟可搞定）。

由于它支持websocket 和polling两种连接方式所以兼顾大多数主流浏览器，低版本的IE浏览器也是支持的。

地址：<http://goeasy.io/>

# Pushlets

Pushlets 是通过长连接方式实现“推”消息的。推送模式分为：Poll（轮询）、Pull（拉）。

地址：<http://www.pushlets.com/>

# Pushlet

Pushlet 是一个开源的 Comet 框架,Pushlet 使用了观察者模型：客户端发送请求，订阅感兴趣的事件；服务器端为每个客户端分配一个会话 ID 作为标记，事件源会把新产生的事件以多播的方式发送到订阅者的事件队列里。

地址：<https://github.com/wjw465150/Pushlet>

总结

其实前面有提过，尽管WebSocket有诸多优点，但是，如果服务端维护很多长连接也是挺耗费资源的，服务器集群以及览器或者客户端兼容性问题，

也会带来了一些不确定性因素。大体了解了一下各大厂的做法，大多数都还是基于轮询的方式实现的，比如：腾讯PC端微信扫码登录、京东商城支付成功通知等等。

有些小伙伴可能会问了，轮询岂不是会更耗费资源？其实在我看来，有些轮询是不可能穿透到后端数据库查询服务的，比如秒杀，一个缓存标记位就可以判定是否秒杀成功。

相对于WS的长连接以及其不确定因素，在秒杀场景下，轮询还是相对比较合适的。

# Dwr