SpringBoot系列 - 集成WebSocket实时通信

spring springboot

会飞的污熊 2018-01-20 1987 阅读

WebSocket是 HTML5 开始提供的一种浏览器与服务器间进行全双工通讯的网络技术。WebSocket 通信协议于2011年被IETF定为标准RFC 6455，WebSocketAPI 被W3C定为标准。在WebSocket API 中，浏览器和服务器只需要要做一个握手的动作，然后，浏览器和服务器之间就形成了一条快速通道。两者之间就直接可以数据互相传送。

注意特点：

- 为浏览器和服务端提供了双工异步通信的功能，即服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于服务器推送技术的一种。
- 建立在 TCP 协议之上，服务器端的实现比较容易。
- 与 HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。
- 数据格式比较轻量，性能开销小，通信高效。
- 可以发送文本，也可以发送二进制数据。
- 没有同源限制，客户端可以与任意服务器通信。
- 协议标识符是ws（如果加密，则为wss），服务器网址就是 URL。

有多种方式实现WebSocket协议，比如SpringBoot官方推荐的基于STOMP实现，实现起来非常简单，还有通过Socket.IO来实现。这里我讲一下后者实现方案。

概述

Socket.IO 主要使用WebSocket协议。但是如果需要的话，Socket.io可以回退到几种其它方法，例如Adobe Flash Sockets，JSONP拉取，或是传统的AJAX拉取，并且在同时提供完全相同的接口。尽管它可以被用作WebSocket的包装库，它还是提供了许多其它功能，比如广播至多个套接字，存储与不同客户有关的数据，和异步IO操作。

更多请参考 Socket.IO 官网： https://socket.io/

基于 socket.io 来说，采用 node 实现更加合适，本文使用两个后端的Java开源框架实现。

- 服务端使用 netty-socketio
- 客户端使用 socket.io-client-java

业务需求是将之前通过轮询方式调动RESTFul API改成使用WebSocket长连接方式，实现要服务器实时的推送消息，另外还要实时监控POS机的在线状态等。

引入依赖

```
<!-- netty-socketio-->
<dependency>
    <groupId>com.corundumstudio.socketio</groupId>
    <artifactId>netty-socketio</artifactId>
    <version>1.7.13</version>
</dependency>
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-resolver</artifactId>
    <version>4.1.15.Final</version>
</dependency>
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-transport</artifactId>
    <version>4.1.15.Final</version>
</dependency>
<dependency>
    <groupId>io.socket</groupId>
    <artifactId>socket.io-client</artifactId>
    <version>1.0.0</version>
</dependency>
```

服务器代码

先来服务端程序爽一把，话不多说，先上代码：

```
            }
        });

        /*
         * 添加监听事件，监听客户端的事件
         * 1.第一个参数eventName需要与客户端的事件要一致
         * 2.第二个参数eventClase是传输的数据类型
         * 3.第三个参数listener是用于接收客户端传的数据，数据类型需要与eventClass一致
         */
        server.addEventListener("login", LoginRequest.class, new DataListener<LoginRequest>() {
            @Override
            public void onData(SocketIOClient client, LoginRequest data, AckRequest ackRequest) {
                logger.info("接收到客户端login消息：code = " + data.getCode() + ",body = " + data.getBody());
                // check is ack requested by client, but it's not required check
                if (ackRequest.isAckRequested()) {
                    // send ack response with data to client
                    ackRequest.sendAckData("已成功收到客户端登录请求", "yeah");
                }
                // 向客户端发送消息
                List<String> list = new ArrayList<>();
                list.add("登录成功，sessionId=" + client.getSessionId());
                // 第一个参数必须与eventName一致，第二个参数data必须与eventClass一致
                String room = client.getHandshakeData().getSingleUrlParam("appid");
                server.getRoomOperations(room).sendEvent("login", list.toString());
            }
        });
        //启动服务
        server.start();
    }
}
```

Android客户端

老规矩，先上代码爽爽

```
            public void call(Object... args) {
                logger.info("Socket.EVENT_CONNECT_ERROR");
                socket.disconnect();
            }
        }).on(Socket.EVENT_CONNECT_TIMEOUT, new Emitter.Listener() {
            @Override
            public void call(Object... args) {
                logger.info("Socket.EVENT_CONNECT_TIMEOUT");
                socket.disconnect();
            }
        }).on(Socket.EVENT_PING, new Emitter.Listener() {
            @Override
            public void call(Object... args) {
                logger.info("Socket.EVENT_PING");
            }
        }).on(Socket.EVENT_PONG, new Emitter.Listener() {
            @Override
            public void call(Object... args) {
                logger.info("Socket.EVENT_PONG");
            }
        }).on(Socket.EVENT_MESSAGE, new Emitter.Listener() {
            @Override
            public void call(Object... args) {
                logger.info("-----------接受到消息啦--------" + Arrays.toString(args));
            }
        }).on(Socket.EVENT_DISCONNECT, new Emitter.Listener() {
            @Override
            public void call(Object... args) {
                logger.info("客户端断开连接啦。。。");
```

关于心跳机制

根据 Socket.IO文档 解释，客户端会定期发送心跳包，并触发一个ping事件和一个pong事件，如下：

- `ping` Fired when a ping packet is written out to the server.
- `pong` Fired when a pong is received from the server. Parameters:
  - **`Number`** number of ms elapsed since `ping` packet (i.e.: latency)

这里最重要的两个服务器参数如下：

1. pingTimeout (Number): how many ms without a pong packet to consider the connection closed (60000)
2. pingInterval (Number): how many ms before sending a new ping packet (25000).

也就是说握手协议的时候，客户端从服务器拿到这两个参数，一个是ping消息的发送间隔时间，一个是从服务器返回pong消息的超时时间，客户端会在超时后断开连接。心跳包发送方向是客户端向服务器端发送，以维持在线状态。

关于断线和超时

关闭浏览器、直接关闭客户端程序、kill进程、主动执行disconnect方法都会导致立刻产生断线事件。而客户端把网络断开，服务器端在 `pingTimeout` ms后产生断线事件、客户端在 `pingTimeout` ms后也产生断线事件。

实际上，超时后会产生一个断线事件，叫”disconnect”。客户端和服务器端都可以对这个事件作出应答，释放连接。

客户端代码：

```
.on(Socket.EVENT_DISCONNECT, new Emitter.Listener() {
    @Override
    public void call(Object... args) {
        logger.info("客户端断开连接啦。。。");
        socket.disconnect();
    }
});
```

连上服务器后，断开网络。超过了心跳超时时间后，产生断线事件。如果客户端不主动断开连接的话，会自动重连，这时候发现连接不上，又产生连接错误事件，然后重试2次，都失败后自动断开连接了。

下面是客户端日志：

```
SocketClient - 回执消息=服务器已成功收到客户端登录请求,yeah
SocketClient - Socket.EVENT_PING
SocketClient - Socket.EVENT_PONG
SocketClient - 客户端断开连接啦。。。
SocketClient - Socket.EVENT_CONNECT_ERROR
```

服务器端代码：

```
server.addDisconnectListener(new DisconnectListener() {
    @Override
    public void onDisconnect(SocketIOClient client) {
        System.out.println("服务器收到断线通知... sessionId=" + client.getSessionId());
    }
});
```

服务器逻辑是，如果在心跳超时后，就直接断开这个连接，并且产生一个断开连接事件。

服务器通过netty处理心跳包ping/pong的日志如下：

```
WebSocket08FrameDecoder - Decoding WebSocket Frame opCode=1
WebSocket08FrameDecoder - Decoding WebSocket Frame length=1
WebSocket08FrameEncoder - Encoding WebSocket Frame opCode=1 length=1
```

浏览器客户端演示

对于 `netty-socketio` 有一个demo工程，里面通过一个网页的聊天小程序演示了各种使用方法。

demo地址： netty-socketio-demo

SpringBoot集成

最后重点讲一下如何在SpringBoot中集成。

修改配置

首先maven依赖之前已经讲过了，先修改下 `application.yml` 配置文件来配置下几个参数，比如主机、端口、心跳时间等等。

```
##################  自定义项目配置 ##################
enzhico:
  socket-hostname: localhost
  socket-port: 9096
```

添加Bean配置

然后增加一个SocketServer的Bean配置：

```
@Configuration
public class NettySocketConfig {

    @Resource
    private MyProperties myProperties;

    @Resource
    private ApiService apiService;
    @Resource
    private ManagerInfoService managerInfoService;

    private static final Logger logger = LoggerFactory.getLogger(NettySocketConfig.class);

    @Bean
    public SocketIOServer socketIOServer() {
        /*
         * 创建Socket，并设置监听端口
         */
        com.corundumstudio.socketio.Configuration config = new com.corundumstudio.socketio.Configuration();
        // 设置主机名，默认是0.0.0.0
        // config.setHostname("localhost");
        // 设置监听端口
        config.setPort(9096);
        // 协议升级超时时间（毫秒），默认10000。HTTP握手升级为ws协议超时时间
        config.setUpgradeTimeout(10000);
        // Ping消息间隔（毫秒），默认25000。客户端向服务器发送一条心跳消息间隔
        config.setPingInterval(60000);
        // Ping消息超时时间（毫秒），默认60000，这个时间间隔内没有接收到心跳消息就会发送超时事件
        config.setPingTimeout(180000);
        // 这个版本0.9.0不能处理好namespace和query参数的问题。所以为了做认证必须使用全局默认命名空间
```

注意，我在 **AuthorizationListener** 里面通过调用service做了用户名和密码的认证。通过注解方式可以注入service，执行相应的连接授权动作。

后面还有个 **SpringAnnotationScanner** 的定义不能忘记。

添加消息结构类

预先定义好客户端和服务器端直接传递的消息类型，使用简单的JavaBean即可，比如

```
public class ReportParam {
    /**
     * IMEI码
     */
    private String imei;
    /**
     * 位置
     */
    private String location;

    public String getImei() {
        return imei;
    }

    public void setImei(String imei) {
        this.imei = imei;
    }

    public String getLocation() {
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }
}
```

添加消息处理类

这里才是最核心的接口处理类，所有接口处理逻辑都应该写在这里面，我只举了一个例子，就是POS上传位置接口：

```
/**
 * 消息事件处理器
 *
 * @author XiongNeng
 * @version 1.0
 * @since 2018/1/19
 */
@Component
public class MessageEventHandler {

    private final SocketIOServer server;
    private final ApiService apiService;

    private static final Logger logger = LoggerFactory.getLogger(MessageEventHandler.class);

    @Autowired
    public MessageEventHandler(SocketIOServer server, ApiService apiService) {
        this.server = server;
        this.apiService = apiService;
    }

    //添加connect事件，当客户端发起连接时调用
    @OnConnect
    public void onConnect(SocketIOClient client) {
        if (client != null) {
            String imei = client.getHandshakeData().getSingleUrlParam("imei");
            String applicationId = client.getHandshakeData().getSingleUrlParam("appid");
            logger.info("连接成功, applicationId=" + applicationId + ", imei=" + imei +
                    ", sessionId=" + client.getSessionId().toString() );
            client.joinRoom(applicationId);
```

添加ServerRunner

还有一个步骤就是添加启动器，在SpringBoot启动之后立马执行：

```
/**
 * SpringBoot启动之后执行
 *
 * @author XiongNeng
 * @version 1.0
 * @since 2017/7/15
 */
@Component
@Order(1)
public class ServerRunner implements CommandLineRunner {
    private final SocketIOServer server;
    private static final Logger logger = LoggerFactory.getLogger(ServerRunner.class);

    @Autowired
    public ServerRunner(SocketIOServer server) {
        this.server = server;
    }

    @Override
    public void run(String... args) throws Exception {
        logger.info("ServerRunner 开始启动啦...");
        server.start();
    }
}
```

参考文章

- Android端与Java服务端交互——SocketIO
- Spring Boot 集成 socket.io 后端实现消息实时通信
- Spring Boot实战之netty-socketio实现简单聊天室

WebSocket是 HTML5 开始提供的一种浏览器与服务器间进行全双工通讯的网络技术。 WebSocket 通信协议于2011年被IETF定为标准RFC 6455，WebSocketAPI 被W3C定为标准。 在WebSocket API 中，浏览器和服务器只需要要做一个握手的动作，然后，浏览器和服务器之间就形成了一条快速通道。 两者之间就直接可以数据互相传送。

注意特点：

- 为浏览器和服务端提供了双工异步通信的功能，即服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于服务器推送技术的一种。

- 建立在 TCP 协议之上，服务器端的实现比较容易。
- 与 HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。
- 数据格式比较轻量，性能开销小，通信高效。
- 可以发送文本，也可以发送二进制数据。
- 没有同源限制，客户端可以与任意服务器通信。
- 协议标识符是ws（如果加密，则为wss），服务器网址就是 URL。

有多种方式实现WebSocket协议，比如SpringBoot官方推荐的基于STOMP实现，实现起来非常简单，还有通过Socket.IO来实现。 这里我讲一下后者实现方案。

spring springboot

点赞

作者：会飞的污熊

August trip

原文地址：SpringBoot系列 - 集成WebSocket实时通信, 感谢原作者分享。

←HTTPS协议详解 →Java加解密算法

您可能感兴趣的博客

SpringBoot系列 - 自己写starter 博主 发表1年前
SpringBoot系列 - 集成WebSocket实时通信 博主 发表1年前
SpringBoot系列 - 异步线程池 博主 发表1年前
SpringBoot系列 - 使用AOP 博主 发表1年前
SpringBoot系列 - 缓存 博主 发表1年前
SpringBoot系列 - 集成JWT实现接口权限认证 博主 发表1年前
SpringBoot系列 - 集成Echarts导出图片 博主 发表1年前
spring boot处理定时任务方式 博主 发表2年前
SpringBoot系列 - 集成Swagger2 博主 发表1年前
SpringBoot系列 - 批处理 博主 发表1年前
SpringBoot系列 - 使用消息队列RabbitMQ 博主 发表1年前
SpringBoot系列 - 定时任务 博主 发表1年前

您可能感兴趣的代码

spring ibatis 配置多个sqlMapConfig.xml文件的方法 by 金背二郎	6年前
Spring 中设置依赖注入 by 金背二郎	7年前
Spring+iBatis+DBUnit 进行单元测试 by 疯子liu	6年前
Spring 依赖注入：自动注入properties文件中的配置 by 金背二郎	6年前
spring 配置文件中定义list引用一个已存在的bean by 甄码农	5年前
spring mvc velocity的模板引擎 by 甄码农	4年前
jdbctemplate 批量操作代码 by 廖钊权	2年前
spring+mybatis 多数据源切换 by Loli控	7年前
使用spring远程调用服务端接口实现WebService功能 by 梁方	6年前
Spring 中注入bean的properties配置文件位置问题解决 by 金背二郎	6年前
在spring mvc中使用flush先发送部分网页内容的方法 by demon	5年前
spring component-scan 扫描多个包的写法 by 甄码农	4年前