一文吃透WebSocket 原理

Gaby 計算機網絡工程師 2022-02-10 10:48

> **來自：掘金，作者：Gaby**
> **鏈接：https://juejin.cn/post/7020964728386093093**

一、前言

踩著年末的尾巴，提前佈局來年，為來年的工作做個好的鋪墊，所以就開始了面試歷程，因為項目中使用到了 `WebSocket` ，面試官在深挖項目經驗的時候，也難免提到 `WebSocket` 相關的知識點，因為之前並沒有考慮這麼深，所以，回答的還是有所欠缺，因此，趕緊趁熱再熟悉熟悉，也藉此機會，整理出來供大家咀嚼，每個項目都有其值得挖掘的閃光點，要用有愛的眼睛👁去發現。

二、什麼是WebSocket

`WebSocket` 是一種在單個TCP連接上進行全雙工通信的協議。 `WebSocket` 使得客戶端和服務器之間的數據交換變得更加簡單，允許服務端主動向客戶端推送數據。

在 `WebSocket API` 中，瀏覽器和服務器只需要完成一次握手，兩者之間就直接可以創建持久性的連接， 並進行雙向數據傳輸。（維基百科）

`WebSocket` 本質上一種計算機網絡應用層的協議，用來彌補 `http` 協議在持久通信能力上的不足。

WebSocket 協議在2008年誕生，2011年成為國際標準。現在最新版本瀏覽器都已經支持了。

它的最大特點就是，服務器可以主動向客戶端推送信息，客戶端也可以主動向服務器發送信息，是真正的雙向平等對話，屬於服務器推送技術的一種。

WebSocket 的其他特點包括：

- （1）建立在TCP 協議之上，服務器端的實現比較容易。
- （2）與HTTP 協議有著良好的兼容性。默認端口也是80和443，並且握手階段採用HTTP 協議，因此握手時不容易屏蔽，能通過各種HTTP 代理服務器。
- （3）數據格式比較輕量，性能開銷小，通信高效。
- （4）可以發送文本，也可以發送二進制數據。

- （5）沒有同源限制，客戶端可以與任意服務器通信。
- （6）協議標識符是ws（如果加密，則為wss），服務器網址就是URL。

```
ws://example.com:80/some/path
```


為什麼需要WebSocket？

我們已經有了HTTP 協議，為什麼還需要另一個協議？它能帶來什麼好處？

因為HTTP 協議有一個缺陷：通信只能由客戶端發起，不具備服務器推送能力。

舉例來說，我們想了解查詢今天的實時數據，只能是客戶端向服務器發出請求，服務器返回查詢結果。HTTP 協議做不到服務器主動向客戶端推送信息。

這種單向請求的特點，注定瞭如果服務器有連續的狀態變化，客戶端要獲知就非常麻煩。我們只能使用"輪詢"：每隔一段時候，就發出一個詢問，了解服務器有沒有新的信息。最典型的場景就是聊天室。輪詢的效率低，非常浪費資源（因為必須不停連接，或者HTTP 連接始終打開）。

在 `WebSocket` 協議出現以前，創建一個和服務端進雙通道通信的web 應用，需要依賴HTTP協議，進行不停的輪詢，這會導致一些問題：

- 服務端被迫維持來自每個客戶端的大量不同的連接
- 大量的輪詢請求會造成高開銷，比如會帶上多餘的header，造成了無用的數據傳輸。

`http` 协议本身是没有持久通信能力的，但是我们在实际的应用中，是很需要这种能力的，所以，为了解决这些问题， `WebSocket` 协议由此而生，于2011年被IETF定为标准RFC6455，并被RFC7936所补充规范。并且在 `HTML5` 标准中增加了有关 `WebSocket` 协议的相关 `api` ，所以只要实现了 `HTML5` 标准的客户端，就可以与支持 `WebSocket` 协议的服务器进行全双工的持久通信了。

WebSocket 与 HTTP 的区别

`WebSocket` 与 `HTTP` 的关系图：

- **相同点：** 都是一样基于TCP的，都是可靠性传输协议。都是应用层协议。
- **联系：** WebSocket在建立握手时，数据是通过HTTP传输的。但是建立之后，在真正传输时候是不需要HTTP协议的。

下面一张图说明了 HTTP 与 WebSocket 的主要区别：

不同点：

- 1、 `WebSocket` 是双向通信协议，模拟 `Socket` 协议，可以双向发送或接受信息，而 `HTTP` 是单向的；

- 2、 WebSocket 是需要浏览器和服务器握手进行建立连接的，而 http 是浏览器发起向服务器的连接。
- 3、 虽然 HTTP/2 也具备服务器推送功能，但 HTTP/2 只能推送静态资源，无法推送指定的信息。

三、WebSocket协议的原理

与http协议一样， WebSocket 协议也需要通过已建立的TCP连接来传输数据。具体实现上是通过http协议建立通道，然后在此基础上用真正 WebSocket 协议进行通信，所以WebSocket协议和http协议是有一定的交叉关系的。首先， WebSocket 是一个持久化的协议，相对于 HTTP 这种非持久的协议来说。简单的举个例子吧，用目前应用比较广泛的 PHP 生命周期来解释。

HTTP 的生命周期通过 Request 来界定，也就是一个 Request 一个 Response ，那么在 HTTP1.0 中，这次 HTTP 请求就结束了。

在 HTTP1.1 中进行了改进，使得有一个 keep-alive，也就是说，在一个 HTTP 连接中，可以发送多个 Request，接收多个 Response。但是请记住 Request = Response， 在 HTTP 中永远是这样，也就是说一个 Request 只能有一个 Response。而且这个 Response 也是被动的，不能主动发起。首先 WebSocket 是基于 HTTP 协议的，或者说借用了 HTTP 协议来完成一部分握手。

首先我们来看个典型的 WebSocket 握手

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

熟悉 HTTP 的童鞋可能发现了，这段类似 HTTP 协议的握手请求中，多了这么几个东西。

```
Upgrade: websocket
Connection: Upgrade
```

这个就是 WebSocket 的核心了，告诉 Apache 、 Nginx 等服务器：注意啦，我发起的请求要用 WebSocket 协议，快点帮我找到对应的助理处理~而不是那个老土的 HTTP 。

```
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
```

```
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

- 首先， Sec-WebSocket-Key 是一个 Base64 encode 的值，这个是浏览器随机生成的，告诉服务器：泥煤，不要忽悠我，我要验证你是不是真的是 WebSocket 助理。
- 然后， Sec_WebSocket-Protocol 是一个用户定义的字符串，用来区分同 URL 下，不同的服务所需要的协议。简单理解：今晚我要服务A，别搞错啦~
- 最后， Sec-WebSocket-Version 是告诉服务器所使用的 WebSocket Draft （协议版本），在最初的时候，WebSocket 协议还在 Draft 阶段，各种奇奇怪怪的协议都有，而且还有很多期奇奇怪怪不同的东西，什么 Firefox 和 Chrome 用的不是一个版本之类的，当初 WebSocket 协议太多可是一个大难题。。不过现在还好，已经定下来啦~大家都使用同一个版本：服务员，我要的是13岁的噢→_→ 然后服务器会返回下列东西，表示已经接受到请求，成功建立 WebSocket 啦！

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

这里开始就是 HTTP 最后负责的区域了，告诉客户，我已经成功切换协议啦~

```
Upgrade: websocket
Connection: Upgrade
```

依然是固定的，告诉客户端即将升级的是 WebSocket 协议，而不是 `mozillasocket` ，`lurnarsocket` 或者 `shitsocket` 。

然后， `Sec-WebSocket-Accept` 这个则是经过服务器确认，并且加密过后的 `Sec-WebSocket-Key` 。服务器：好啦好啦，知道啦，给你看我的 ID CARD 来证明行了吧。后面的， `Sec-WebSocket-Protocol` 则是表示最终使用的协议。至此，HTTP 已经完成它所有工作了，接下来就是完全按照 `WebSocket` 协议进行了。总结， `WebSocket` 连接的过程是：

- 首先，客户端发起http请求，经过3次握手后，建立起TCP连接； `http` 请求里存放 `WebSocket` 支持的版本号等信息，如：Upgrade、Connection、WebSocket-Version等；
- 然后，服务器收到客户端的握手请求后，同样采用HTTP协议回馈数据；

- 最后，客户端收到连接成功的消息后，开始借助于TCP传输信道进行全双工通信。

四、Websocket的优缺点

优点：

- WebSocket协议一旦建议后，互相沟通所消耗的请求头是很小的
- 服务器可以向客户端推送消息了

缺点：

- 少部分浏览器不支持，浏览器支持的程度与方式有区别（IE10）

五、WebSocket应用场景

- 即时聊天通信
- 多玩家游戏
- 在线协同编辑/编辑
- 实时数据流的拉取与推送
- 体育/游戏实况
- 实时地图位置
- 即时Web应用程序：即时Web应用程序使用一个Web套接字在客户端显示数据，这些数据由后端服务器连续发送。在WebSocket中，数据被连续推送/传输到已经打开的同一连接中，这就是为什么WebSocket更快并提高了应用程序性能的原因。例如在交易网站或比特币交易中，这是最不稳定的事情，它用于显示价格波动，数据被后端服务器使用Web套接字通道连续推送到客户端。
- 游戏应用程序：在游戏应用程序中，你可能会注意到，服务器会持续接收数据，而不会刷新用户界面。屏幕上的用户界面会自动刷新，而且不需要建立新的连接，因此在WebSocket游戏应用程序中非常有帮助。
- 聊天应用程序：聊天应用程序仅使用WebSocket建立一次连接，便能在订阅户之间交换，发布和广播消息。它重复使用相同的WebSocket连接，用于发送和接收消息以及一对一的消息传输。

六、websocket 断线重连

心跳就是客户端定时的给服务端发送消息，证明客户端是在线的， 如果超过一定的时间没有发送则就是离线了。

如何判断在线离线？

当客户端第一次发送请求至服务端时会携带唯一标识、以及时间戳，服务端到db或者缓存去查询改请求的唯一标识，如果不存在就存入db或者缓存中， 第二次客户端定时再次发送请求依旧携带唯一标识、以及时间戳，服务端到db或者缓存去查询改请求的唯一标识，如果存在就把上次的时间戳拿取出来，使用当前时间戳减去上次的时间， 得出的毫秒秒数判断是否大于指定的时间，若小于的话就是在线，否则就是离线；

如何解决断线问题

通过查阅资料了解到 nginx 代理的 websocket 转发，无消息连接会出现超时断开问题。网上资料提到解决方案两种，一种是修改nginx配置信息，第二种是 `websocket` 发送心跳包。下面就来总结一下本次项目实践中解决的 `websocket` 的断线 和 重连 这两个问题的解决方案。主动触发包括主动断开连接，客户端主动发送消息给后端

- 1 主动断开连接

```
ws.close();
```

主动断开连接，根据需要使用，基本很少用到。

- 2 主动发送消息

```
ws.send("hello world");
```

- 断线的可能原因1：websocket超时没有消息自动断开连接，应对措施：这时候我们就需要知道服务端设置的超时时长是多少，在小于超时时间内发送心跳包，有2中方案:一种是客户端主动发送上行心跳包，另一种方案是服务端主动发送下行心跳包。

下面主要讲一下客户端也就是前端如何实现心跳包：

首先了解一下心跳包机制

跳包之所以叫心跳包是因为：它像心跳一样每隔固定时间发一次，以此来告诉服务器，这个客户端还活着。事实上这是为了保持长连接，至于这个包的内容，是没有什么特别规定的，不过一般都是很小的包，或者只包含包头的一个空包。

在 TCP 的机制里面，本身是存在有心跳包的机制的，也就是 TCP 的选项： SO_KEEPALIVE 。系统默认是设置的2小时的心跳频率。但是它检查不到机器断电、网线拔出、防火墙这些断线。而且逻辑层处理断线可能也不是那么好处理。一般，如果只是用于保活还是可以的。

心跳包一般来说都是在逻辑层发送空的 echo 包来实现的。下一个定时器，在一定时间间隔下发送一个空包给客户端，然后客户端反馈一个同样的空包回来，服务器如果在一定时间内收不到客户端发送过来的反馈包，那就只有认定说掉线了。

在长连接下，有可能很长一段时间都没有数据往来。理论上说，这个连接是一直保持连接的，但是实际情况中，如果中间节点出现什么故障是难以知道的。更要命的是，有的节点(防火墙)会自动把一定时间之内没有数据交互的连接给断掉。在这个时候，就需要我们的心跳包了，用于维持长连接，保活。

心跳检测步骤：

- 客户端每隔一个时间间隔发生一个探测包给服务器
- 客户端发包时启动一个超时定时器
- 服务器端接收到检测包，应该回应一个包
- 如果客户机收到服务器的应答包，则说明服务器正常，删除超时定时器
- 如果客户端的超时定时器超时，依然没有收到应答包，则说明服务器挂了

```
// 前端解决方案：心跳检测
var heartCheck = {
    timeout: 30000, //30秒发一次心跳
    timeoutObj: null,
    serverTimeoutObj: null,
    reset: function(){
        clearTimeout(this.timeoutObj);
        clearTimeout(this.serverTimeoutObj);
        return this;
    },
    start: function(){
        var self = this;
        this.timeoutObj = setTimeout(function(){
            //这里发送一个心跳，后端收到后，返回一个心跳消息，
            //onmessage拿到返回的心跳就说明连接正常
            ws.send("ping");
            console.log("ping!")

            self.serverTimeoutObj = setTimeout(function(){//如果超过一定时间还没重置，说明后端主动断开
```

```
                ws.close(); //如果onclose会执行reconnect，我们执行ws.close()就行了.如果直接执行reconne
            }, self.timeout);
        }, this.timeout);
    }
}
```

- 断线的可能原因2： `websocket` 异常包括服务端出现中断，交互切屏等等客户端异常中断等等 当若服务端宕机了，客户端怎么做、服务端再次上线时怎么做？客户端则需要断开连接，通过 `onclose` 关闭连接，服务端再次上线时则需要清除之间存的数据，若不清除 则会造成只要请求到服务端的都会被视为离线。

针对这种异常的中断解决方案就是处理重连，下面我们给出的重连方案是使用js库处理：引入 `reconnecting-websocket.min.js` ，ws建立链接方法使用js库api方法：

```
var ws = new ReconnectingWebSocket(url);
// 断线重连：
reconnectSocket(){
    if ('ws' in window) {
        ws = new ReconnectingWebSocket(url);
    } else if ('MozWebSocket' in window) {
       ws = new MozWebSocket(url);
    } else {
      ws = new SockJS(url);
    }
}
```

断网监测支持使用js库： `offline.min.js`

```
onLineCheck(){
    Offline.check();
    console.log(Offline.state,'---Offline.state');
    console.log(this.socketStatus,'---this.socketStatus');

    if(!this.socketStatus){
        console.log('网络连接已断开！');
        if(Offline.state === 'up' && websocket.reconnectAttempts > websocket.maxReconnectInterval){
            window.location.reload();
        }
        reconnectSocket();
```

```
    }else{
        console.log('网络连接成功！');
        websocket.send("heartBeat");
    }
}

// 使用：在websocket断开链接时调用网络中断监测
websocket.onclose => () {
    onLineCheck();
};
```

以上方案，只是抛砖引玉，如果大家有更好的解决方案欢迎评论区分享交流。

七、总结

WebSocket 是为了在 web 应用上进行双通道通信而产生的协议，相比于轮询HTTP请求的方式，WebSocket 有节省服务器资源，效率高等优点。WebSocket 中的掩码是为了防止早期版本中存在中间缓存污染攻击等问题而设置的，客户端向服务端发送数据需要掩码，服务端向客户端发送数据不需要掩码。WebSocket 中 Sec-WebSocket-Key 的生成算法是拼接服务端和客户端生成的字符串，进行SHA1哈希算法，再用base64编码。WebSocket 协议握手是依靠 HTTP 协议的，依靠于 HTTP 响应101进行协议升级转换。

参考

- 阮一峰：WebSocket 教程
- 看完讓你徹底理解WebSocket 原理

--- EOF ---

閱讀原文

喜歡此內容的人還喜歡

用了HTTPS，沒想到還是被監控了！

計算機網絡工程師