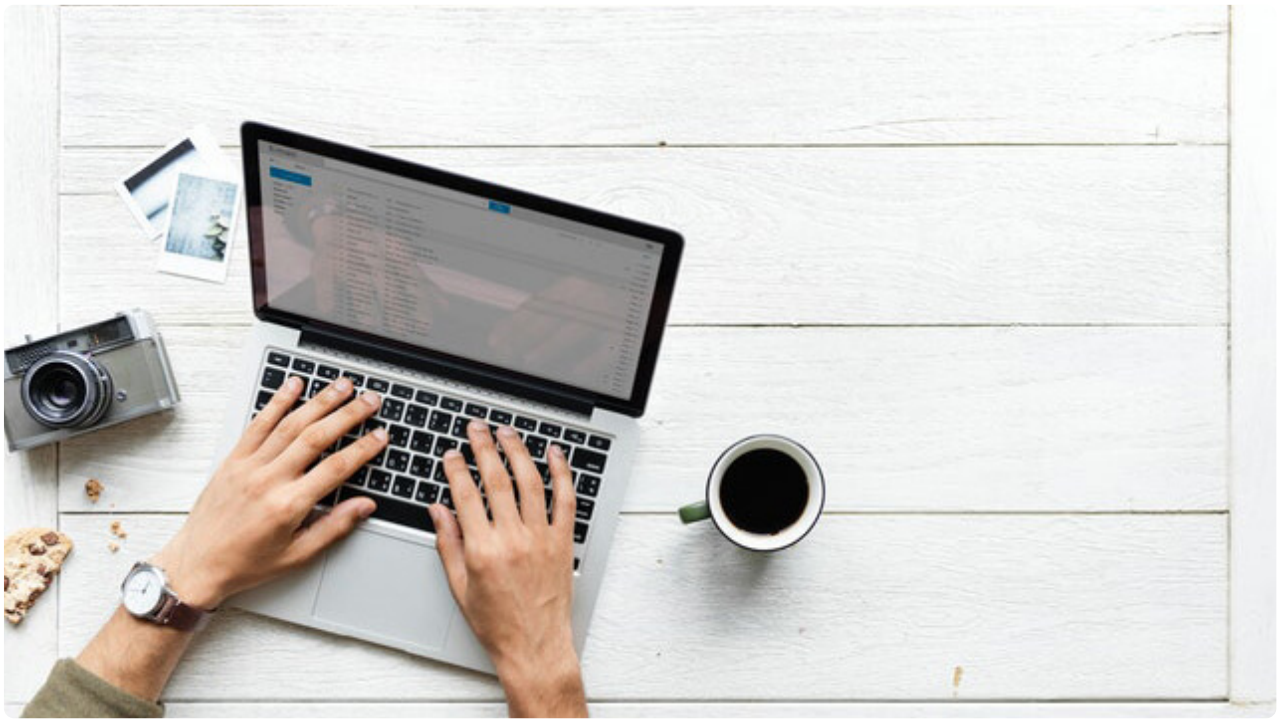
27 使用 Scoket 改造后端接口

更新时间：2019-09-20 10:38:13



> 先相信你自己，然后别人才会相信你。
>
> ——屠格涅夫

本章节，我们将会采用web-scoket来开发聊天私信聊天界面的后端逻辑，我们首先会了解一下聊天实现的基本原理，然后在理解原理的前提下，前端采用vue-socket.io库，后端采用Node.js的Socket.io库来开发聊天逻辑。

本章节完整源代码地址，大家可以事先浏览一下：

Github-message.js

Github-socket.js

Github-chat/index.vue

聊天场景分析

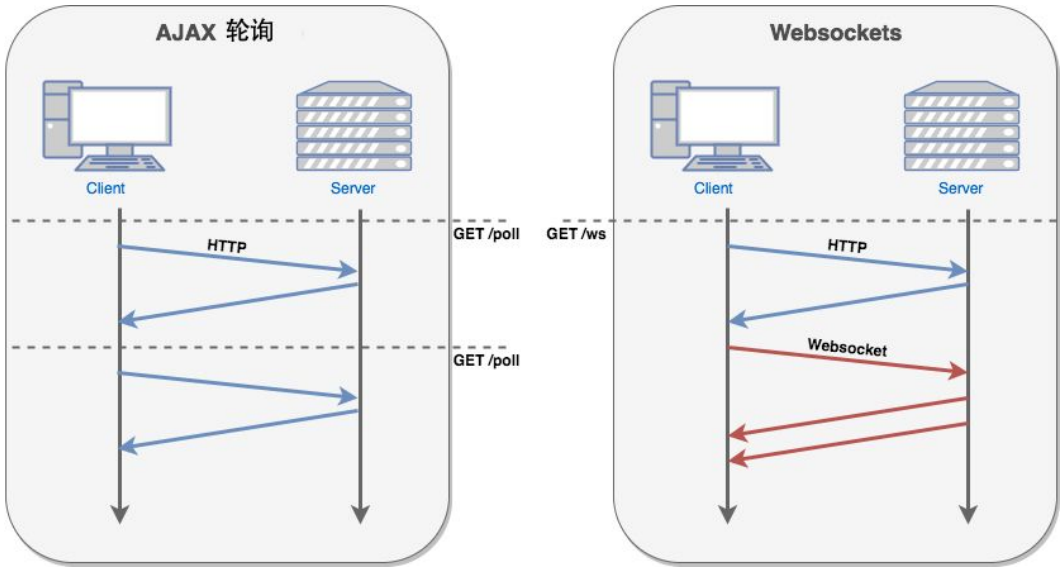
私信聊天本身就是一个对实时性要求比较高的场景，我们来总结一下各种实现的方案：

轮询（Polling）： 前端不断向后台发请求以，实时更新数据，不管后台数据是否有更新都会返回数据，后端程序编写比较容易。对服务器压力大。请求中有大半是无用，浪费带宽和服务器资源。适于小型应用。

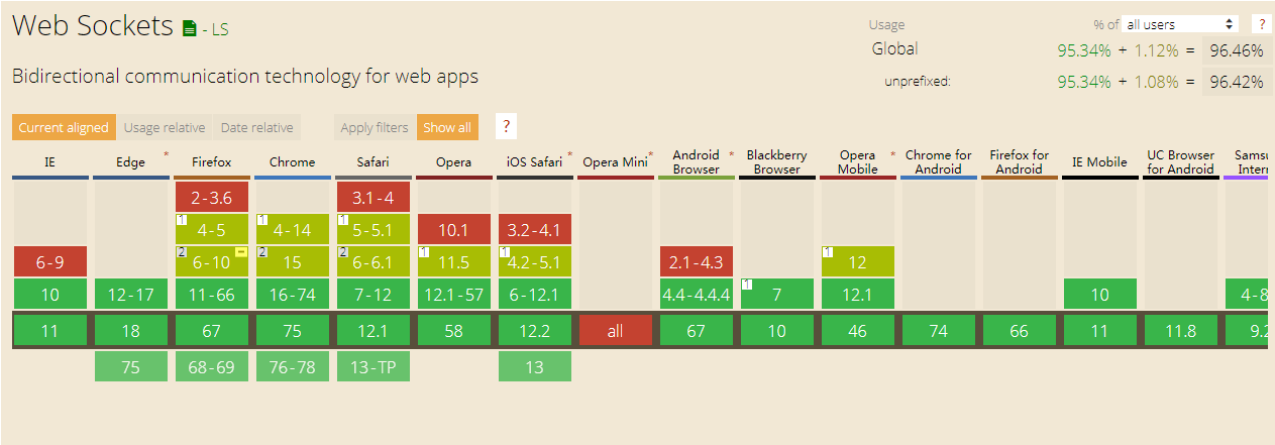
长轮询（Long Polling）：前端向后台发请求，后台数据如果还没有更新则不返回，直到后台数据更新了再返回给前端，前端收到后台返回的数据后才发下一次请求。在无消息的情况下不会频繁的请求。但是请求在后台一直悬挂，连接长时间保持，浪费资源。

WebSocket：是HTML5新增加的一种通信协议，目前流行的浏览器都支持这个协议，是基于TCP协议的协议。

WebSocket 解决的第一个问题是，通过第一个 HTTP request 建立了 TCP 连接之后，之后的交换数据都不需要再发 HTTP request 了，使得这个长连接变成了一个真–长连接。



每次使用一个新特性之前，我们都有确认兼容性：



基本上除了Android4.4之前的版本不支持以外，现在主流的手机终端都可以使用，可以欢快的使用了！

安装**Scoket.io**相关库

Socket.io是一个封装了 Websocket、基于 Node 的 JavaScript 框架，包含 client 端和 server 端。

在WebSocket没有出现之前，实现与服务端的实时通讯可以通过轮询来完成任务。Socket.io将WebSocket和轮询（Polling）机制以及其它的实时通信方式封装成了通用的接口，并且在服务端实现了这些实时机制的相应代码。

也就是说，Websocket仅仅是Socket.io实现实时通信的一个子集。另外，Socket.io 还有一个非常重要的好处。其不仅支持 WebSocket，还支持许多种轮询机制以及其他实时通信方式，并封装了通用的接口。这些方式包含 Adobe Flash Socket、Ajax 长轮询、Ajax multipart streaming 、持久 Iframe、JSONP轮询等。换句话说，当Socket.io检测到当前环境不支持WebSocket时，能够自动地选择最佳的方式来实现网络的实时通信。

前端安装 socket.io-client 和 vue-socket.io :

```
npm install socket.io-client --save
npm install vue-socket.io@2.1.1-a --save
```

socket.io-client 是 Socket.io 的客户端库，而 vue-socket.io 是将 vue 和 socket.io-client 结合起来的库，让我们在 vue 里更加方便的使用 scoket。

后端安装 socket.io ：

```
npm install socket.io --save
```

开发Scoket.io实时聊天逻辑

在后端启动时，添加 socket 逻辑，修改后端项目 bin 目录下的 server.js (原bin/www)：

```
var io = require('socket.io')(server);

io.on('connection', function(_socket){

  socket.setSocket(_socket)

});
```

简单3行代码，我们的后端 WebSocket 服务就启动了，在后端项目 utils 文件夹下新建一个 socket.js 将我们关于scoket的逻辑放在这里：

```
var socketPoll = {};//存储当前聊天用户的池子

module.exports = {

  setSocket(socket){

    //用户进入聊天页面代表登录
    socket.on('login', function(obj) {
      console.log('用户'+obj._id+'进入聊天页面')
      //将用户id和当前用户的socket存起来
      socketPoll[obj._id] = socket;

    });
    //用户离开聊天页面代表登出
    socket.on('loginout', function(obj) {
      console.log('用户'+obj._id+'离开聊天页面')
      //将该用户从用户池中删除
      delete socketPoll[obj._id];
    });
  },

  sendMsg(obj){
    //根据id，找到对应的socket
    var currentSocket = socketPoll[obj.id];
    console.log('向客户端推送消息')
    if (currentSocket) {
      //向客户端推送消息
      currentSocket.emit(obj.action, obj.data);
    }
  }

};
```

上面代码，总结一下逻辑：

`socket.on()` 表示监听客户端的事件，当接收到客户端的事件时，执行里面的逻辑。
`sendMsg()` 方法是对 `socket.emit()` 方法的封装， `socket.emit()` 表示发送了一个 action 命令，携带 data 参数，客户端采用 `socket.on('action',function(){...})` 便可以接收到。

我们在发送消息的接口采用的是 http 协议，在消息存储成功之后需要找到当前接收者的socket，所以我们需要一个 `socketPoll` 池来保存每一个进入聊天的用户的socket，这里我们采用了JavaScript的对象Object，借助key的唯一性来存储。

在后端项目 routes 文件夹下的 message.js 文件里的 `addmsg` 方法下，增加发送socket事件：

```
//socket实时消息
socket.sendMsg({
  id:toUserId,
  action: 'recieveMsg',
  data: {
    content: result.content,
    fromUser: user
  },

});
```

前端引入vue-socket.io

前端方面，在前端项目 views 文件夹下的 chat 文件夹下的 index.vue 里对 created 和 beforeDestroy 方法来进行登录和登出操作，即将后台用户池进行增加和减少：

```
created () {
  if (this.$store.state.currentUser && this.$store.state.currentUser._id) {
    this.$socket.emit('login',this.$store.state.currentUser);
  }

  this.fetchData()
},
beforeDestroy () {
  // window.clearInterval(this.loopFlag)
  if (this.$store.state.currentUser && this.$store.state.currentUser._id) {
    this.$socket.emit('loginout',this.$store.state.currentUser);
  }
},
```

借助了 vue-socket.io 我们就可以不需要写 socket.on('action',function(){...}) 接收事件，直接在一个vue的组件里写即可。

在前端项目 views 文件夹下的 chat 文件夹下的 index.vue 里添加上 sockets 对象的 recieveMsg 方法和 reconnect 方法，代码如下：

```
sockets: {
  /*
  * 接收到消息
  */
  recieveMsg: function(obj) {
    //将接收到的消息push到当前的list中
    if (obj.fromUser._id === this.toUserId) {
      this.afterCommit({
        content: obj.content,
        fromUser: obj.fromUser,
      })
    }

  },
  /*
  * 服务端掉线之后客户端会自动重连，此事件在重连成功时触发
  */
  reconnect: function(obj) {

    //后端重启之后需要重新登录一次
    if (this.$store.state.currentUser && this.$store.state.currentUser._id) {
      this.$socket.emit('login',this.$store.state.currentUser);
    }

  }
},
```

recieveMsg 就代表 socket.on('recieveMsg',function(){...})，这里我们在和对方聊天时，自己发的消息直接在本地push到dataList里面，而通过 recieveMsg 将接收到的对方发的消息，也不断的push进去。

reconnect 表示socket.io的一个内置事件，如果服务器因为发版或者上线需要重启，这里有一个事件来进行重连登录。

客户端其他内置事件：

```
connect：连接成功
connecting：正在连接
disconnect：断开连接
connect_failed：连接失败
error：错误发生，并且无法被其他事件类型所处理
message：同服务器端message事件
anything：同服务器端anything事件
reconnect_failed：重连失败
reconnect：成功重连
reconnecting：正在重连
```

到此，我们来总结一下，整个scoket聊天的流程：

1. 在后端项目`server.js`中，启动web-socket。
2. 在后端项目中，封装`socket.js`，来存储用户池，和发送事件逻辑。
3. 在后端项目中的`message.js`路由中，在发送节消息的接口中添加事件通知逻辑。
4. 在前端项目中的`chat`的`index.vue`中，处理后台的发送和接受逻辑。

小结

本章节主要讲解了WebSocket的基本知识，同时将聊天逻辑改造成了基于socket.io的逻辑。

相关知识点：

1. 轮询，长轮询，WebSocket直接的区别和联系。
2. 使用socket.io对聊天页面的前后端逻辑进行改造。

本章节完整源代码地址：

Github-message.js

Github-socket.js

Github-chat/index.vue

}

26 私信后端接口开发

28 聊天列表页面 UI 开发