

## 课程介绍

---

- 为前端系统提供mock服务
- 前端系统中通过graphql查询房源列表
- 实现后台系统的更新房源数据功能
- 为接口服务添加Redis缓存
- WebSocket入门

## 1、伪mock服务

---

前面完成了首页的轮播广告服务的支持，为力方便后面的项目开发，需要对前端所有的请求都进行支持。

暂时不实现的，先模拟数据返回。

### 1.1、构造数据

mock-data.properties :



```
1 mock.indexMenu={"data":{"list":[{"id":1,"menu_name":"二手房","menu_logo":"home","menu_path":"/home","menu_status":1,"menu_style":null},
{"id":2,"menu_name":"新房","menu_logo":null,"menu_path":null,"menu_status":null,"menu_style":null},
{"id":3,"menu_name":"租房","menu_logo":null,"menu_path":null,"menu_status":null,"menu_style":null},
{"id":4,"menu_name":"海外","menu_logo":null,"menu_path":null,"menu_status":null,"menu_style":null},
{"id":5,"menu_name":"地图找房","menu_logo":null,"menu_path":null,"menu_status":null,"menu_style":null},
{"id":6,"menu_name":"查公交","menu_logo":null,"menu_path":null,"menu_status":null,"menu_style":null},
{"id":7,"menu_name":"计算器","menu_logo":null,"menu_path":null,"menu_status":null,"menu_style":null},
{"id":8,"menu_name":"问答","menu_logo":null,"menu_path":null,"menu_status":null,"menu_style":null}]},"meta":{"status":200,"msg":"测试数据"}}
2 mock.indexInfo={"data":{"list":[{"id":1,"info_title":"房企半年销售业绩继","info_thumb":null,"info_time":null,"info_content":null,"user_id":null,"info_status":null,"info_type":1},{"id":2,"info_title":"上半年土地市场两重天：一线降温三四线量价齐升","info_thumb":null,"info_time":null,"info_content":null,"user_id":null,"info_status":null,"info_type":1}]},"meta":{"status":200,"msg":"测试数据"}}
3 mock.indexFaq={"data":{"list":[{"question_name":"在北京买房，需要支付的税费有哪些？","question_tag":"学区,海淀","answer_content":"各种费用","atime":33,"question_id":1,"qnum":2},{"question_name":"一般首付之后，贷款多久可以下来？","question_tag":"学区,昌平","answer_content":"大概1个月","atime":22,"question_id":2,"qnum":2}]},"meta":{"status":200,"msg":"测试数据"}}
4 mock.indexHouse={"data":{"list":[{"id":1,"home_name":"安贞西里123","home_price":"4511","home_desc":"72.32㎡/南北/低楼层","home_infos":null,"home_type":1,"home_tags":"海淀,昌平","home_address":null,"user_id":null,"home_status":null,"home_time":12,"group_id":1},{"id":8,"home_name":"安贞西里 三室一厅","home_price":"4500","home_desc":"72.32㎡/南北/低楼层","home_infos":null,"home_type":1,"home_tags":"海淀","home_address":null,"user_id":null,"home_status":null,"home_time":23,"group_id":2},{"id":3,"home_name":"安贞西里 三室一厅","home_price":"4220","home_desc":"72.32㎡/南北/低楼层","home_infos":null,"home_type":2,"home_tags":"海淀","home_address":null,"user_id":null,"home_status":null,"home_time":1,"group_id":1},{"id":4,"home_name":"安贞西里 三室一厅","home_price":"4500","home_desc":"72.32㎡/南北/低楼层","home_infos":"4500","home_type":2,"home_tags":"海淀","home_address":"","user_id":null,"home_status":null,"home_time":12,"group_id":2},{"id":5,"home_name":"安贞西里 三室一厅","home_price":"4522","home_desc":"72.32㎡/南北/低楼层","home_infos":null,"home_type":3,"home_tags":"海淀","home_address":null,"user_id":null,"home_status":null,"home_time":23,"group_id":1},{"id":6,"home_name":"安贞西里 三室一厅","home_price":"4500","home_desc":"72.32㎡/南北/低楼层","home_infos":null,"home_type":3,"home_tags":"海淀","home_address":null,"user_id":null,"home_status":null,"home_time":1221,"group_id":2},{"id":9,"home_name":"安贞西里 三室一厅","home_price":"4500","home_desc":"72.32㎡/南北/低楼层","home_infos":null,"home_type":4,"home_tags":"海淀","home_address":null,"user_id":null,"home_status":null,"home_time":23,"group_id":1}]},"meta":{"status":200,"msg":"测试数据"}}
```



```
5 mock.infosList1={"data":{"list":{"total":8,"data":
  [{"id":13,"info_title":"www", "info_thumb":null, "info_time":null, "info_content":null, "user_id":null, "info_status":null, "info_type":1}, {"id":12, "info_title":"房企
  半年销售业绩
  继", "info_thumb":null, "info_time":null, "info_content":null, "user_id":null, "info_status":null, "info_type":1}]}},"meta":{"status":200, "msg":"获取数据成功"}}
6 mock.infosList2={"data":{"list":{"total":4,"data":[{"id":9, "info_title":"房企半年销售业绩
  继续冲高三巨头销售额过
  亿", "info_thumb":null, "info_time":null, "info_content":null, "user_id":null, "info_status":null, "info_type":2}, {"id":7, "info_title":"房企半年销售业绩继续冲高三巨头销售额过
  亿", "info_thumb":null, "info_time":null, "info_content":null, "user_id":null, "info_status":null, "info_type":2}]}},"meta":{"status":200, "msg":"获取数据成功"}}
7 mock.infosList3={"data":{"list":{"total":10,"data":
  [{"username":"tom", "question_name":"在北京买房，需要支付的税费有哪些？", "question_tag":"学区, 海淀", "answer_content":"各种费用", "atime":33, "question_id":1, "qnum":2},
  {"username":"tom", "question_name":"一般首付之后，贷款多久可以下来？", "question_tag":"学区, 昌平", "answer_content":"大概1个月", "atime":22, "question_id":2, "qnum":2}]}},"meta":
  {"status":200, "msg":"获取数据成功"}}
8 mock.my={"data":
  {"id":1, "username":"tom", "password":"123", "mobile":"123", "type":null, "status":null, "avatar":"public/icon.png"}, "meta":{"status":200, "msg":"获取数据成功"}}
```

## 1.2、创建MockConfig

用于读取配置文件中的内容。

```
1 package cn.itcast.haoke.dubbo.api.config;
2
3 import lombok.Data;
4 import org.springframework.boot.context.properties.ConfigurationProperties;
5 import org.springframework.context.annotation.Configuration;
6 import org.springframework.context.annotation.PropertySource;
7
8 @Configuration
9 @PropertySource("classpath:mock-data.properties")
10 @ConfigurationProperties(prefix = "mock")
11 @Data
12 public class MockConfig {
13
14     private String indexMenu;
15     private String indexInfo;
16     private String indexFaq;
17     private String indexHouse;
18     private String infosList1;
19     private String infosList2;
20     private String infosList3;
21     private String my;
22
23 }
24
```

## 1.3、创建MockController



```
1 package cn.itcast.haoke.dubbo.api.controller;
2
3 import cn.itcast.haoke.dubbo.api.config.MockConfig;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.beans.factory.annotation.Value;
6 import org.springframework.boot.context.properties.ConfigurationProperties;
7 import org.springframework.context.annotation.PropertySource;
8 import org.springframework.web.bind.annotation.*;
9
10 @RequestMapping("mock")
11 @RestController
12 @CrossOrigin
13 public class MockController {
14
15     @Autowired
16     private MockConfig mockConfig;
17
18     /**
19      * 菜单
20      *
21      * @return
22      */
23     @GetMapping("index/menu")
24     public String indexMenu() {
25         return this.mockConfig.getIndexMenu();
26     }
27
28     /**
29      * 首页资讯
30      * @return
31      */
32     @GetMapping("index/info")
33     public String indexInfo() {
34         return this.mockConfig.getIndexInfo();
35     }
36
37     /**
38      * 首页问答
39      * @return
40      */
41     @GetMapping("index/faq")
42     public String indexFaq() {
43         return this.mockConfig.getIndexFaq();
44     }
45
46     /**
47      * 首页房源信息
48      * @return
49      */
50     @GetMapping("index/house")
51     public String indexHouse() {
52         return this.mockConfig.getIndexHouse();
53     }
54 }
```



```
54
55     /**
56     * 查询资讯
57     *
58     * @param type
59     * @return
60     */
61     @GetMapping("infos/list")
62     public String infosList(@RequestParam("type")Integer type) {
63         switch (type){
64             case 1:
65                 return this.mockConfig.getInfosList1();
66             case 2:
67                 return this.mockConfig.getInfosList2();
68             case 3:
69                 return this.mockConfig.getInfosList3();
70         }
71         return this.mockConfig.getInfosList1();
72     }
73
74     /**
75     * 我的中心
76     * @return
77     */
78     @GetMapping("my/info")
79     public String myInfo() {
80         return this.mockConfig.getMy();
81     }
82
83 }
84
```

## 1.4、测试

http://127.0.0.1:18080/mock/index/menu

GET POST PUT PATCH DELETE HEAD OPTIONS Other

Raw Form Headers

Status: 200 Loading time: 25 ms

Request headers: User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.67 Safari/537.36  
Content-Type: text/plain; charset=utf-8  
Accept: \*/\*  
Accept-Encoding: gzip, deflate, br  
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,zh-TW;q=0.7  
Cookie: \_\_ga=GA1.4.548092130.1505735846

Response headers: Content-Type: text/plain; charset=UTF-8  
Content-Length: 887  
Date: Wed, 28 Nov 2018 14:28:19 GMT

Raw Parsed Response

Open output in new window Copy to clipboard Save as file Open in JSON tab

```
[{"data": {"list": [{"id": 1, "menu_name": "二手房", "menu_logo": "home", "menu_path": "/home", "menu_status": 1, "menu_style": null}, {"id": 2, "menu_name": "新房", "menu_logo": null, "menu_path": null, "menu_status": null, "menu_style": null}]}}
```

Code highlighting thanks to [Code Mirror](#)



## 1.5、整合前端系统

```
let menu = new Promise((resolve, reject) => {
  axios.get('http://127.0.0.1:18080/mock/index/menu').then((data) => {
    resolve(data.data.list);
  });
})
let info = new Promise((resolve, reject) => {
  axios.get('http://127.0.0.1:18080/mock/index/info').then((data) => {
    resolve(data.data.list);
  });
})
let faq = new Promise((resolve, reject) => {
  axios.get('http://127.0.0.1:18080/mock/index/faq').then((data) => {
    resolve(data.data.list);
  });
})
let house = new Promise((resolve, reject) => {
  axios.get('http://127.0.0.1:18080/mock/index/house').then((data) => {
    resolve(data.data.list);
  });
})
Promise.all([swipe, menu, info, faq, house]).then((result) => {
```

实现效果一样。

## 2、GraphQL中的参数

在GraphQL查询中，往往是需要设置参数的，像这样：

```
1 {
2   HouseResources(id: 8) {
3     id
4     title
5     estateId
6     buildingUnit
7     buildingFloorNum
8     mobile
9     useArea
10    pic
11  }
12 }
```

其中，id: 8 就是在传递参数，但是这是静态参数，如果动态传递参数呢？

在GraphQL规范中，对参数的传递是有定义的：



目前为止，我们将参数写在了查询字符串内。但是在很多应用中，字段的参数可能是动态的：例如，可能是一个“下拉菜单”让你选择感兴趣的《星球大战》续集，或者是一个搜索区，或者是一组过滤器。

将这些动态参数直接传进查询字符串并不是好主意，因为这样我们的客户端就得动态地在运行时操作这些查询字符串了，再把它序列化成 GraphQL 专用的格式。其实，GraphQL 拥有一级方法将动态值提取到查询之外，然后作为分离的字典传进去。这些动态值即称为**变量**。

使用变量之前，我们得做三件事：

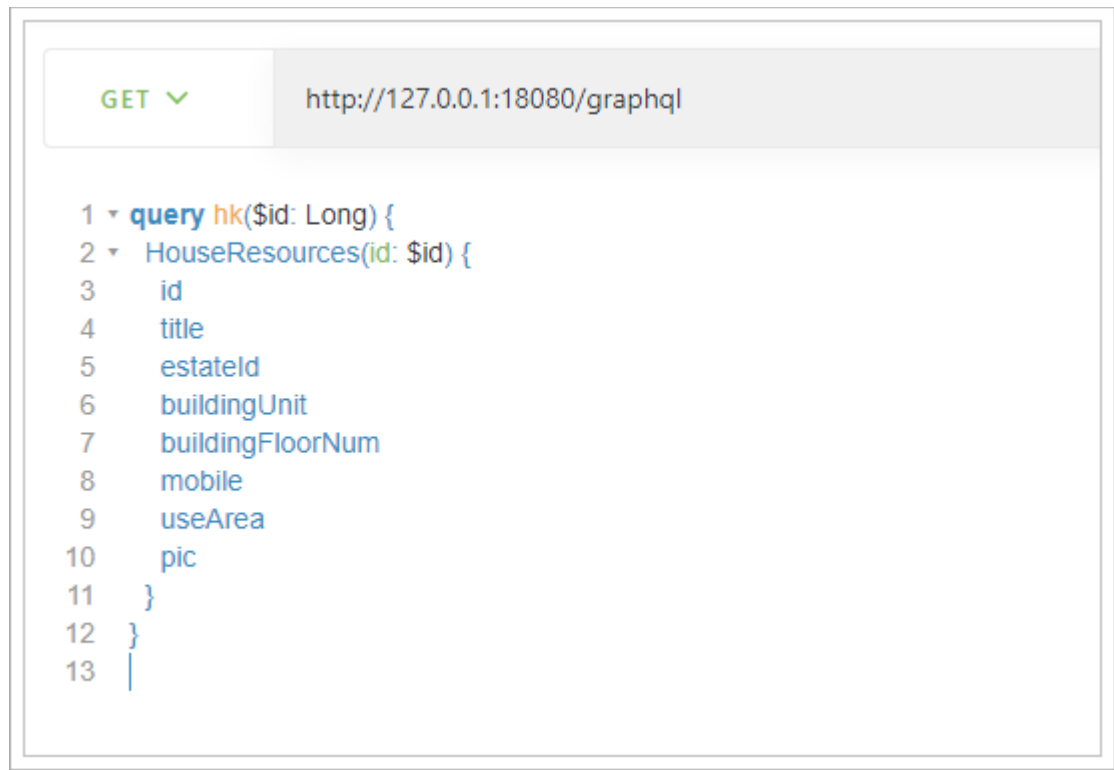
1. 使用 `$variableName` 替代查询中的静态值。
2. 声明 `$variableName` 为查询接受的变量之一。
3. 将 `variableName: value` 通过传输专用（通常是 JSON）的分离的变量字典中。

全部做完之后就像这个样子：

```
# { "graphql": true, "variables": { "episode": JEDI } }  
query HeroNameAndFriends($episode: Episode) {  
  hero(episode: $episode) {  
    name  
    friends {  
      name  
    }  
  }  
}
```

这样一来，我们的客户端代码就只需要传入不同的变量，而不用构建一个全新的查询了。这事实上也是一个良好实践，意味着查询的参数将是动态的——我们决不能使用用户提供的值来字符串插值以构建查询。

## 2.1、查询时传递参数



```
1 query hk($id: Long) {  
2   HouseResources(id: $id) {  
3     id  
4     title  
5     estateId  
6     buildingUnit  
7     buildingFloorNum  
8     mobile  
9     useArea  
10    pic  
11  }  
12 }  
13
```

说明：

hk -> 表示操作名称，这个名称随意。

\$id: Long -> 定义参数以及参数类型

(id: \$id) -> 通过变量使用参数

## 2.2、设置参数值

```
1 {  
2   "id":8  
3 }
```





#### VARIABLES

```
1 {  
2   "id":8  
3 }
```

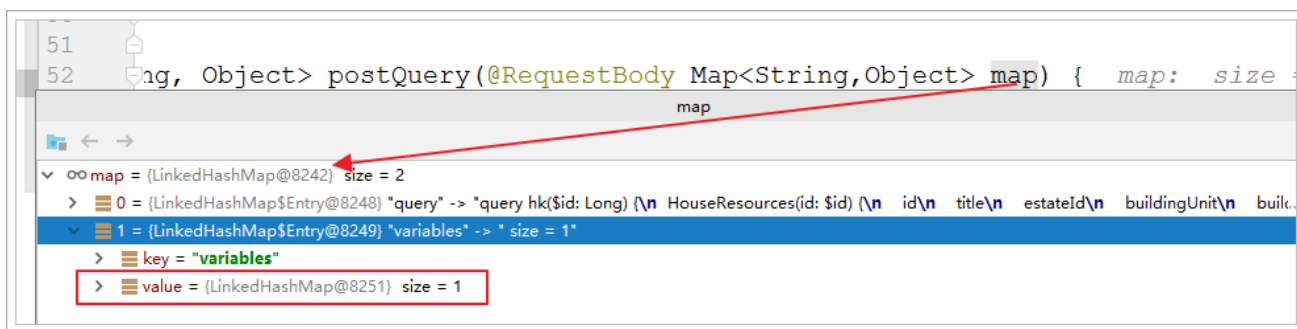
## 2.3、服务端处理参数

```
1 package cn.itcast.haoke.dubbo.api.controller;  
2  
3 import com.fasterxml.jackson.databind.ObjectMapper;  
4 import graphql.ExecutionInput;  
5 import graphql.GraphQL;  
6 import org.springframework.beans.factory.annotation.Autowired;  
7 import org.springframework.stereotype.Controller;  
8 import org.springframework.web.bind.annotation.*;  
9  
10 import java.io.IOException;  
11 import java.util.Collections;  
12 import java.util.HashMap;  
13 import java.util.Map;  
14  
15 @RequestMapping("graphql")  
16 @Controller  
17 @CrossOrigin  
18 public class GraphQLController {  
19  
20     @Autowired  
21     private GraphQL graphql;  
22  
23     private static final ObjectMapper MAPPER = new ObjectMapper();  
24  
25  
26     /**  
27      * 实现GraphQL查询  
28      *  
29      * @param query  
30      * @return  
31      */  
32     @GetMapping  
33     @ResponseBody  
34     public Map<String, Object> query(@RequestParam("query") String query,  
35                                     @RequestParam(value = "variables", required =  
false) String variablesJson,
```



```
36         @RequestParam(value = "operationName",  
required = false) String operationName) {  
37             try {  
38                 Map variables = MAPPER.readValue(variablesJson,  
MAPPER.getTypeFactory().constructMapType(HashMap.class, String.class,  
Object.class));  
39                 return this.executeGraphQLQuery(query, operationName, variables);  
40             } catch (IOException e) {  
41                 e.printStackTrace();  
42             }  
43  
44             Map<String, Object> error = new HashMap<>();  
45             error.put("status", 500);  
46             error.put("msg", "查询出错");  
47             return error;  
48         }  
49  
50         @PostMapping  
51         @ResponseBody  
52         public Map<String, Object> postQuery(@RequestBody Map<String, Object> map) {  
53             try {  
54                 String query = (String) map.get("query");  
55                 if(null == query){  
56                     query = "";  
57                 }  
58                 String operationName = (String) map.get("operationName");  
59                 Map variables = (Map) map.get("variables");  
60                 if(variables == null){  
61                     variables = Collections.EMPTY_MAP;  
62                 }  
63                 return this.executeGraphQLQuery(query, operationName, variables);  
64             } catch (Exception e) {  
65                 e.printStackTrace();  
66             }  
67  
68             Map<String, Object> error = new HashMap<>();  
69             error.put("status", 500);  
70             error.put("msg", "查询出错");  
71             return error;  
72         }  
73  
74         private Map<String, Object> executeGraphQLQuery(String query, String  
operationName, Map<String, Object> variables) {  
75             ExecutionInput executionInput = ExecutionInput.newExecutionInput()  
76                 .query(query)  
77                 .operationName(operationName)  
78                 .variables(variables)  
79                 .build();  
80             return this.graphQL.execute(executionInput).toSpecification();  
81         }  
82  
83     }  
84 }
```

## 2.4、测试





STATUS: OK STATUS CODE: 200 39ms

```
1 {
2   "data": {
3     "HouseResources": {
4       "id": 8,
5       "title": "最新房源",
6       "estateId": 1002,
7       "buildingUnit": "1",
8       "buildingFloorNum": "1",
9       "mobile": "1",
10      "useArea": "1",
11      "pic": "http://itcast-haoke.oss-cn-qingdao.aliyuncs.com/images/2018/11/17/15423896060254118.jpg,http://itcast-haoke.oss-cn-qingdao.aliyuncs.com/images/2018/11/17/15423896084306516.jpg"
12    }
13  }
14 }
```

## 3、实现房源列表查询

### 3.1、编写查询字符串

```
1 query HouseResourcesList($pageSize: Int, $page: Int) {
2   HouseResourcesList(pageSize: $pageSize, page: $page) {
3     list {
4       id
5       pic
6       title
7       coveredArea
8       orientation
9       floor
10      rent
11    }
12  }
13 }
14
```

查询参数:

```
1 {
2   "pageSize": 2,
3   "page": 1
4 }
```

### 3.2、改造list.js页面



```
1 import ApolloClient from "apollo-boost";
2 import gql from "graphql-tag";
3
4
5 const client = new ApolloClient({
6   uri: "http://127.0.0.1:18080/graphql"
7 });
8
9 //定义查询
10 const QUERY_LIST = gql`
11   query HouseResourcesList($pageSize: Int, $page: Int) {
12     HouseResourcesList(pageSize: $pageSize, page: $page) {
13       list {
14         id
15         pic
16         title
17         coveredArea
18         orientation
19         floor
20         rent
21       }
22     }
23   }
24 `;
25
26 // 查询
27 client.query({query: QUERY_LIST, variables: {"pageSize":3,"page":1}}).then(result
=> {
28
29   this.setState({
30     listData:result.data.HouseResourcesList.list,
31     loadFlag: true
32   });
33
34 //渲染
35 let list = null;
36 if(this.state.loadFlag) {
37   list = this.state.listData.map(item=>{
38     return (
39       <Item key={item.id}>
40         <Item.Image src={item.pic.split(',')[0]}/>
41         <Item.Content>
42           <Item.Header>{item.title}</Item.Header>
43           <Item.Meta>
44             <span className='cinema'>{item.coveredArea}
m²/{item.orientation}/{item.floor}</span>
45           </Item.Meta>
46           <Item.Description>
47             上海
48           </Item.Description>
49           <Item.Description>{item.rent}</Item.Description>
50         </Item.Content>
51       </Item>
```

```
52     )  
53     });  
54 }
```

### 3.3、测试



## 4、实现更新房源数据

前面已经将房源数据进行了展示，为了功能的相对完整，所以下面将实现房源数据的更新功能，在前端进行展示更新后的数据。

### 4.1、新增更新接口 ( RESTful )

#### 4.1.1、修改Controller

itcast-haoke-manage-api-server

```
1  /**  
2      * 修改房源  
3      *  
4      * @param houseResources json数据  
5      * @return  
6      */
```



```

7  @PostMapping
8  @ResponseBody
9  public ResponseEntity<Void> update(@RequestBody HouseResources houseResources) {
10     try {
11         boolean bool = this.houseResourcesService.update(houseResources);
12         if (bool) {
13             return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
14         }
15     } catch (Exception e) {
16         e.printStackTrace();
17     }
18     return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
19 }

```

#### 4.1.2、修改service

itcast-haoke-manage-api-server/HouseResourcesService

```

1  public boolean update(HouseResources houseResources) {
2      return this.apiHouseResourcesService.updateHouseResources(houseResources);
3  }

```

#### 4.1.3、修改dubbo服务

修改接口 ApiHouseResourcesService :

```

1  /**
2   * 修改房源
3   *
4   * @param houseResources
5   * @return
6   */
7  boolean updateHouseResources(HouseResources houseResources);

```

修改实现类 ApiHouseResourcesServiceImpl :

```

1  @Override
2  public boolean updateHouseResources(HouseResources houseResources) {
3      return this.houseResourcesService.updateHouseResources(houseResources);
4  }
5

```

修改业务Service : HouseResourcesServiceImpl

```

1  @Override
2  public boolean updateHouseResources(HouseResources houseResources) {
3      return super.update(houseResources) == 1;
4  }
5

```



## 4.2、编写EditResource.js

```
1  import React from 'react';
2  import {Checkbox, Form, Input, Modal} from "antd";
3  import Pictureswall from "../Utils/Pictureswall";
4  import {connect} from "dva";
5
6  const FormItem = Form.Item;
7  const InputGroup = Input.Group;
8  const CheckboxGroup = Checkbox.Group;
9
10 const formItemLayout = {
11   labelCol: {
12     xs: { span: 24 },
13     sm: { span: 7 },
14   },
15   wrapperCol: {
16     xs: { span: 24 },
17     sm: { span: 12 },
18     md: { span: 10 },
19   },
20 };
21
22 @connect()
23 @Form.create()
24 class EditResource extends React.Component{
25
26   constructor(props){
27     super(props);
28
29     this.state={
30       visible:false,
31       pics:new Set()
32     };
33   }
34
35   showModal = () => {
36     this.setState({
37       visible: true
38     });
39   };
40
41   handleCancel = () => {
42     this.setState({
43       visible: false,
44     });
45   };
46
47   handleSave = () => {
48
49     const { dispatch, form, record } = this.props;
50     form.validateFieldsAndScroll((err, values) => {
51       if (!err) {
```





```
52     if(this.state.pics.size > 0){
53         values.pic = [...this.state.pics].join(',');
54     }
55     values.id = record.id;
56
57     dispatch({
58         type: 'house/updateHouseForm',
59         payload: values,
60     });
61     setTimeout(()=>{
62         this.handleCancel();
63         this.props.reload();
64     },500)
65
66     }
67     });
68
69 };
70
71 handleFileList = (obj)=>{
72     let pics = new Set();
73     obj.forEach((v, k) => {
74         if(v.response){
75             pics.add(v.response.name);
76         }
77         if(v.url){
78             pics.add(v.url);
79         }
80     });
81
82     this.setState({
83         pics : pics
84     })
85 }
86
87 render(){
88
89     const record = this.props.record;
90     const {
91         form: { getFieldDecorator }
92     } = this.props;
93
94     return (
95         <React.Fragment>
96             <a onClick={() => {this.showModal()}}>编辑</a>
97             <Modal
98                 title={'编辑'}
99                 width={640}
100                 visible={this.state.visible}
101                 onOk={()=>{this.handleSave()}}
102                 onCancel={()=>{this.handleCancel()}}
103                 destroyOnClose={true}
104                 >
```



```

105     <div style={{ overflowY: 'auto' }}>
106       <Form hideRequiredMark style={{ marginTop: 8 }}>
107         <FormItem {...formItemLayout} label="房源标题">
108           {getFieldDecorator('title',{initialValue:record.title ,rules:[{
required: true, message:"此项为必填项" }]})(<Input style={{ width: '100%' }}
disabled={false} />)}
109         </FormItem>
110         <FormItem {...formItemLayout} label="租金">
111           <InputGroup compact>
112             {getFieldDecorator('rent',{initialValue:record.rent ,rules:[{
required: true, message:"此项为必填项" }]})(<Input style={{ width: '50%' }}
addonAfter="元/月" />)}
113           </InputGroup>
114         </FormItem>
115         <FormItem {...formItemLayout} label="建筑面积">
116           <InputGroup compact>
117             {getFieldDecorator('coveredArea',
{initialValue:record.coveredArea,rules:[{ required: true, message:"此项为必填项"
}]})(<Input style={{ width: '40%' }} addonAfter="平米" />)}
118           </InputGroup>
119         </FormItem>
120         <FormItem {...formItemLayout} label="上传室内图">
121           <PicturesWall handleFileList={this.handleFileList.bind(this)}
fileList={record.pic}/>
122         </FormItem>
123       </Form>
124     </div>
125
126     </Modal>
127   </React.Fragment>
128 );
129 }
130 }
131
132
133
134 export default EditResource;
135

```

### 4.3、修改房源列表页

```

1  {
2    title: '操作',
3    render: (text, record) => (
4      <Fragment>
5        <a onClick={() => {}}>查看</a>
6        <Divider type="vertical" />
7        <EditResource record={record} reload={this.reload.bind(this)} />
8        <Divider type="vertical" />
9        <a href="">删除</a>
10      </Fragment>
11    ),

```

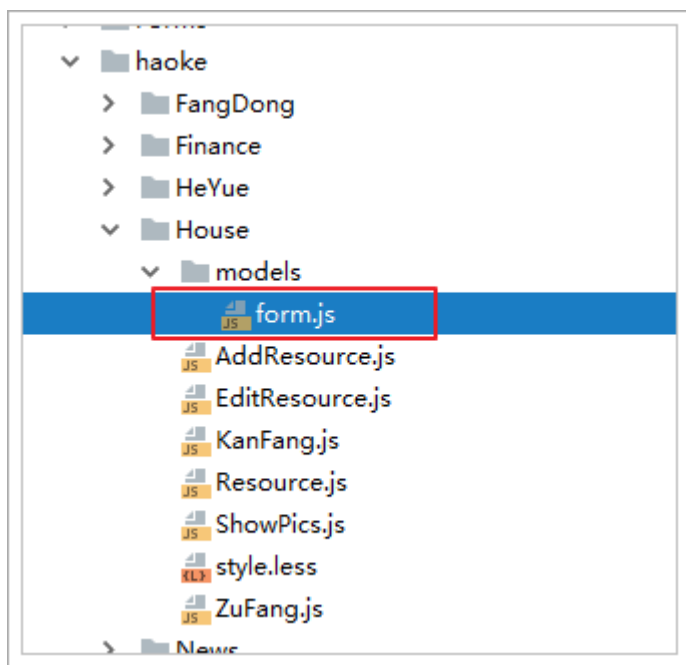
```

12     },
13
14     -----
15
16     reload(){
17       const { dispatch } = this.props;
18       dispatch({
19         type: 'houseResource/fetch'
20       });
21     }
  
```

效果：

面积	楼层	操作		
1平方	1/1	查看	编辑	删除
1平方	1/1	查看	编辑	删除
40平方	3/20	查看	编辑	删除
1平方	1/1	查看	编辑	删除

#### 4.4、修改提交逻辑

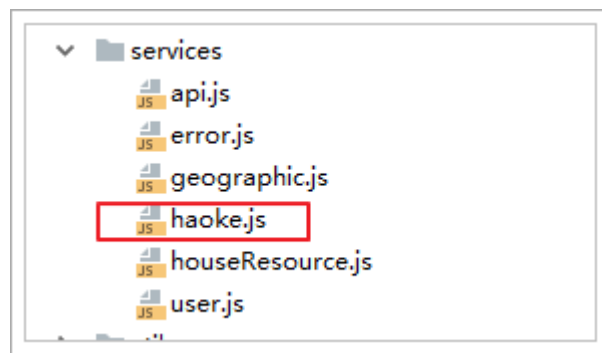


```
1 import { routerRedux } from 'dva/router';
```



```
2 import { message } from 'antd';
3 import { addHouseResource, updateHouseResource } from '@services/haoke';
4
5 export default {
6   namespace: 'house',
7
8   state: {
9
10 },
11
12 effects: {
13   *submitHouseForm({ payload }, { call }) {
14     yield call(addHouseResource, payload);
15     message.success('提交成功');
16   },
17   *updateHouseForm({ payload }, { call }) {
18     yield call(updateHouseResource, payload);
19     message.success('提交成功');
20   }
21 },
22
23 reducers: {
24   saveStepFormData(state, { payload }) {
25     return {
26       ...state
27     };
28   },
29 },
30 };
31
```

## 4.5、修改service逻辑



```
1 import request from '@utils/request';
2
3 export async function addHouseResource(params) {
4   return request('/haoke/house/resources', {
5     method: 'POST',
6     body: params
7   });
8 }
9
```

```
10 export async function updateHouseResource(params) {  
11   return request('/haoke/house/resources', {  
12     method: 'PUT',  
13     body: params  
14   });  
15 }  
16
```

## 4.6、测试

编辑

房源标题:

最新修改房源

租金:

1000

元/月

建筑面积:

100

平米

上传室内图:





+

Upload

取消

确定

实现了更新功能。

## 5、为接口添加缓存功能

在接口服务中，如果每一次都进行数据库查询，那么必然会给数据库造成很大的并发压力。所以需要为接口添加缓存，缓存技术选用Redis，并且使用Redis的集群，Api使用Spring-Data-Redis。

思考：缓存逻辑是加在api处还是dubbo服务处？



## 5.1、使用Docker搭建Redis集群

```
1 #拉取镜像
2 docker pull redis:5.0.2
3
4 #创建容器
5 docker create --name redis-node01 -v /data/redis-data/node01:/data -p 6379:6379
  redis:5.0.2 --cluster-enabled yes --cluster-config-file nodes-node-01.conf
6
7 docker create --name redis-node02 -v /data/redis-data/node02:/data -p 6380:6379
  redis:5.0.2 --cluster-enabled yes --cluster-config-file nodes-node-02.conf
8
9 docker create --name redis-node03 -v /data/redis-data/node03:/data -p 6381:6379
  redis:5.0.2 --cluster-enabled yes --cluster-config-file nodes-node-03.conf
10
11 #启动容器
12 docker start redis-node01 redis-node02 redis-node03
13
14 #开始组建集群
15
16 #进入redis-node01进行操作
17 docker exec -it redis-node01 /bin/bash
18
19 #组建集群
20 redis-cli --cluster create 172.17.0.1:6379 172.17.0.1:6380 172.17.0.1:6381 --
  cluster-replicas 0
```

出现连接不到redis节点的问题：

```
root@bdf613972619:/data# redis-cli --cluster create 172.17.0.1:63
>>> Performing hash slots allocation on 3 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
M: 8d63d7a336b4b729da350ff47d31fe9857ca7f77 172.17.0.1:6379
  slots:[0-5460] (5461 slots) master
M: d32877772901269375cb3e4993c05f0bb6356c84 172.17.0.1:6380
  slots:[5461-10922] (5462 slots) master
M: f558eachb2903c75667aa0a8d9afa0667242e0f66 172.17.0.1:6381
  slots:[10923-16383] (5461 slots) master
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
.....
```

尝试使用容器的ip地址（172.17.0.1这个地址是docker容器分配给主机的地址）：

```
1 #查看容器的ip地址
```



```
2 docker inspect redis-node01 -> 172.17.0.4
3 docker inspect redis-node02 -> 172.17.0.5
4 docker inspect redis-node03 -> 172.17.0.6
5
6 #删除容器
7 docker stop redis-node01 redis-node02 redis-node03
8 docker rm redis-node01 redis-node02 redis-node03
9 rm -rf /data/redis-data
10
11 #进入redis-node01进行操作
12 docker exec -it redis-node01 /bin/bash
13
14 #组建集群(注意端口的变化)
15 redis-cli --cluster create 172.17.0.4:6379 172.17.0.5:6379 172.17.0.6:6379 --
cluster-replicas 0
16
```

发现，搭建成功：

```
>>> Performing hash slots allocation on 3 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
M: 7eb19b3a82216880b61593e59bebefa5edc247a0 172.17.0.4:6379
slots:[0-5460] (5461 slots) master
M: 207a4d90dce0857e26a2add4ed9fd07464ab02d5 172.17.0.5:6379
slots:[5461-10922] (5462 slots) master
M: eaaf2895fde3422c522defe6751e3de88d54a553 172.17.0.6:6379
slots:[10923-16383] (5461 slots) master
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
..
>>> Performing Cluster Check (using node 172.17.0.4:6379)
M: 7eb19b3a82216880b61593e59bebefa5edc247a0 172.17.0.4:6379
slots:[0-5460] (5461 slots) master
M: 207a4d90dce0857e26a2add4ed9fd07464ab02d5 172.17.0.5:6379
slots:[5461-10922] (5462 slots) master
M: eaaf2895fde3422c522defe6751e3de88d54a553 172.17.0.6:6379
slots:[10923-16383] (5461 slots) master
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
root@91df3e5228b1:/data#
```

查看集群信息：



```
root@91df3e5228b1:/data# redis-cli
127.0.0.1:6379> CLUSTER NODES
207a4d90dce0857e26a2add4ed9fd07464ab02d5 172.17.0.5:6379@16379 master - 0 1543765218866 2 connected 5461-10922
eaaf2895fde3422c522defe6751e3de88d54a553 172.17.0.6:6379@16379 master - 0 1543765217856 3 connected 10923-16383
7eb19b3a82216880b61593e59bebefa5edc247a0 172.17.0.4:6379@16379 myself,master - 0 1543765218000 1 connected 0-5460
127.0.0.1:6379>
```

```
1 root@91df3e5228b1:/data# redis-cli
2 127.0.0.1:6379> CLUSTER NODES
3 207a4d90dce0857e26a2add4ed9fd07464ab02d5 172.17.0.5:6379@16379 master - 0
  1543765218866 2 connected 5461-10922
4 eaaf2895fde3422c522defe6751e3de88d54a553 172.17.0.6:6379@16379 master - 0
  1543765217856 3 connected 10923-16383
5 7eb19b3a82216880b61593e59bebefa5edc247a0 172.17.0.4:6379@16379 myself,master - 0
  1543765218000 1 connected 0-5460
```

可以看到，集群中节点的ip地址是docker分配的地址，那么在客户端（spring-data-redis）是没有办法访问的？如何解决？

## 5.2、docker的网络类型

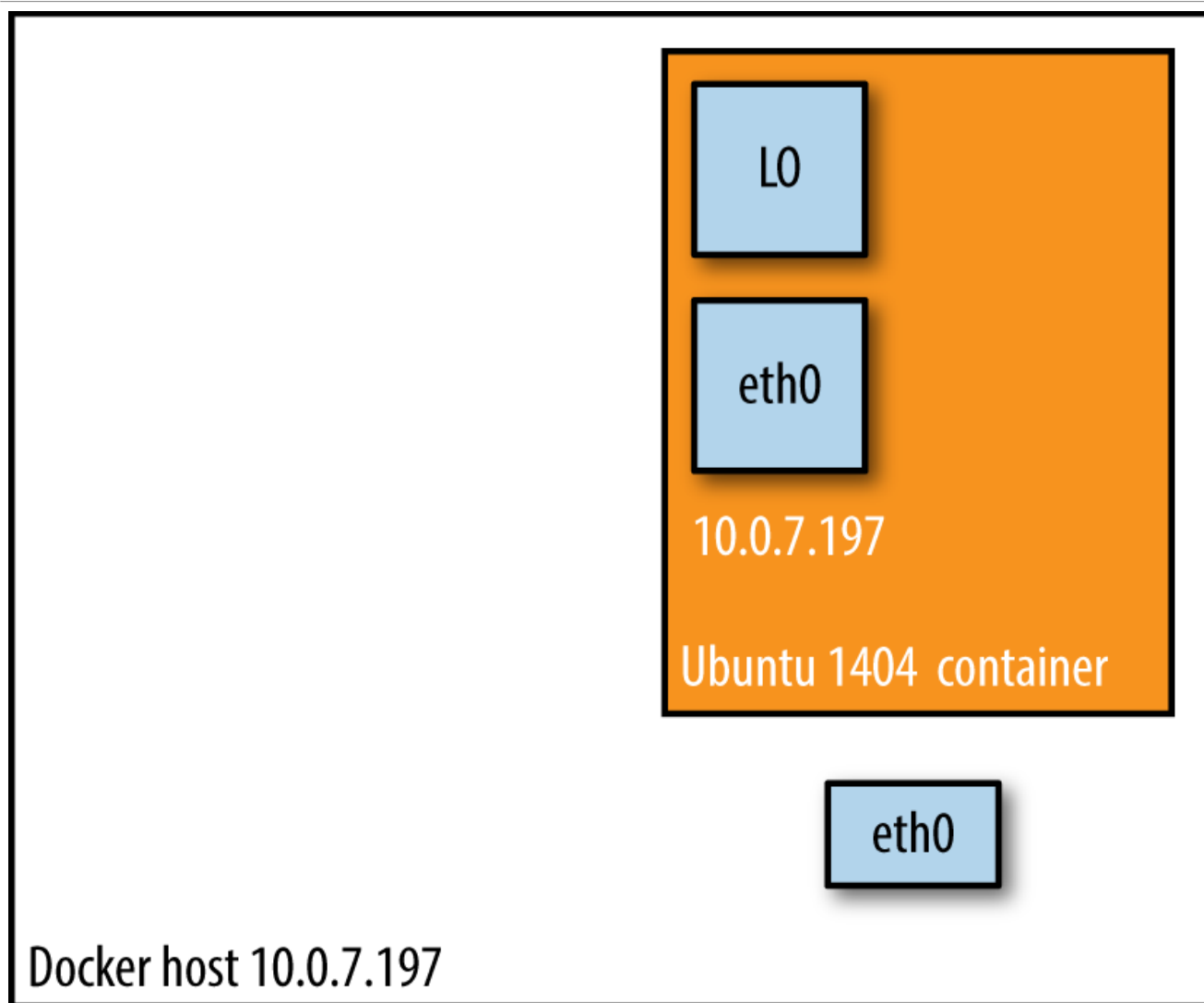
docker的网络类型有：

- None:不为容器配置任何网络功能，没有网络 --net=none
- Container:与另一个运行中的容器共享Network Namespace，--net=container:containerID
- Host:与主机共享Network Namespace，--net=host
- Bridge:Docker设计的NAT网络模型（默认类型）

重点关注下Host类型：

host模式创建的容器没有自己独立的网络命名空间，是和物理机共享一个Network Namespace，并且共享物理机的所有端口与IP。但是它将容器直接暴露在公共网络中，是有安全隐患的。





### 5.3、使用host网络进行搭建集群

```
1 #创建容器
2 docker create --name redis-node01 --net host -v /data/redis-data/node01:/data
  redis:5.0.2 --cluster-enabled yes --cluster-config-file nodes-node-01.conf --port
  6379
3
4 docker create --name redis-node02 --net host -v /data/redis-data/node02:/data
  redis:5.0.2 --cluster-enabled yes --cluster-config-file nodes-node-02.conf --port
  6380
5
6 docker create --name redis-node03 --net host -v /data/redis-data/node03:/data
  redis:5.0.2 --cluster-enabled yes --cluster-config-file nodes-node-03.conf --port
  6381
7
8 #启动容器
9 docker start redis-node01 redis-node02 redis-node03
10
11 #进入redis-node01容器进行操作
12 docker exec -it redis-node01 /bin/bash
13 #172.16.55.185是主机的ip地址
```



```
14 redis-cli --cluster create 172.16.55.185:6379 172.16.55.185:6380 172.16.55.185:6381  
--cluster-replicas 0
```

搭建成功：

```
>>> Performing hash slots allocation on 3 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
M: 4c60f45d1722f771831c64c66c141354f0e28d18 172.16.55.185:6379
slots:[0-5460] (5461 slots) master
M: 46e5582cd2d96a506955cc08e7b08343037c91d9 172.16.55.185:6380
slots:[5461-10922] (5462 slots) master
M: b42d6ccc544094f1d8f35fa7a6d08b0962a6ac4a 172.16.55.185:6381
slots:[10923-16383] (5461 slots) master
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
.
>>> Performing Cluster Check (using node 172.16.55.185:6379)
M: 4c60f45d1722f771831c64c66c141354f0e28d18 172.16.55.185:6379
slots:[0-5460] (5461 slots) master
M: 46e5582cd2d96a506955cc08e7b08343037c91d9 172.16.55.185:6380
slots:[5461-10922] (5462 slots) master
M: b42d6ccc544094f1d8f35fa7a6d08b0962a6ac4a 172.16.55.185:6381
slots:[10923-16383] (5461 slots) master
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
root@itcast:/data#
```

查看集群信息：

```
1 root@itcast:/data# redis-cli
2 127.0.0.1:6379> CLUSTER NODES
3 46e5582cd2d96a506955cc08e7b08343037c91d9 172.16.55.185:6380@16380 master - 0
1543766975796 2 connected 5461-10922
4 b42d6ccc544094f1d8f35fa7a6d08b0962a6ac4a 172.16.55.185:6381@16381 master - 0
1543766974789 3 connected 10923-16383
5 4c60f45d1722f771831c64c66c141354f0e28d18 172.16.55.185:6379@16379 myself,master - 0
1543766974000 1 connected 0-5460
```

## 5.4、编写代码进行测试集群



### 5.4.1、导入依赖

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-data-redis</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>redis.clients</groupId>
7   <artifactId>jedis</artifactId>
8   <version>2.9.0</version>
9 </dependency>
10 <dependency>
11   <groupId>commons-io</groupId>
12   <artifactId>commons-io</artifactId>
13   <version>2.6</version>
14 </dependency>
```

### 5.4.2、编写配置文件

```
1 # redis集群配置
2 spring.redis.jedis.pool.max-wait = 5000
3 spring.redis.jedis.pool.max-idle = 100
4 spring.redis.jedis.pool.min-idle = 10
5 spring.redis.timeout = 10
6 spring.redis.cluster.nodes = 172.16.55.185:6379,172.16.55.185:6380,172.16.55.185:6381
7 spring.redis.cluster.max-redirects=5
```

### 5.4.3、编写配置类

ClusterConfigurationProperties :

```
1 package cn.itcast.haoke.dubbo.api.config;
2
3 import org.springframework.boot.context.properties.ConfigurationProperties;
4 import org.springframework.stereotype.Component;
5
6 import java.util.List;
7
8 @Component
9 @ConfigurationProperties(prefix = "spring.redis.cluster")
10 public class ClusterConfigurationProperties {
11
12
13     private List<String> nodes;
14
15     private Integer maxRedirects;
16
17     public List<String> getNodes() {
18         return nodes;
19     }
20
21     public void setNodes(List<String> nodes) {
```



```
22         this.nodes = nodes;
23     }
24
25     public Integer getMaxRedirects() {
26         return maxRedirects;
27     }
28
29     public void setMaxRedirects(Integer maxRedirects) {
30         this.maxRedirects = maxRedirects;
31     }
32 }
```

#### 5.4.4、注册Redis连接工厂

RedisClusterConfig :

```
1  package cn.itcast.haoke.dubbo.api.config;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.context.annotation.Bean;
5  import org.springframework.context.annotation.Configuration;
6  import org.springframework.data.redis.connection.RedisClusterConfiguration;
7  import org.springframework.data.redis.connection.RedisConnectionFactory;
8  import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
9  import org.springframework.data.redis.core.RedisTemplate;
10 import org.springframework.data.redis.serializer.StringRedisSerializer;
11
12 @Configuration
13 public class RedisClusterConfig {
14
15     @Autowired
16     private ClusterConfigurationProperties clusterProperties;
17
18     @Bean
19     public RedisConnectionFactory connectionFactory() {
20         RedisClusterConfiguration configuration = new
RedisClusterConfiguration(clusterProperties.getNodes());
21         configuration.setMaxRedirects(clusterProperties.getMaxRedirects());
22         return new JedisConnectionFactory(configuration);
23     }
24
25     @Bean
26     public RedisTemplate<String, String> redisTemplate(RedisConnectionFactory
redisConnectionFactory) {
27         RedisTemplate<String, String> redisTemplate = new RedisTemplate<>();
28         redisTemplate.setConnectionFactory(redisConnectionFactory);
29         redisTemplate.setKeySerializer(new StringRedisSerializer());
30         redisTemplate.setValueSerializer(new StringRedisSerializer());
31         redisTemplate.afterPropertiesSet();
32         return redisTemplate;
33     }
34 }
```

### 5.4.5、编写测试用例

```
1 package cn.itcast.haoke.dubbo.api;
2
3 import org.junit.Test;
4 import org.junit.runner.RunWith;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.boot.test.context.SpringBootTest;
7 import org.springframework.data.redis.core.RedisTemplate;
8 import org.springframework.test.context.junit4.SpringRunner;
9
10 import java.util.Set;
11
12 @RunWith(SpringRunner.class)
13 @SpringBootTest
14 public class TestRedis {
15
16     @Autowired
17     private RedisTemplate<String,String> redisTemplate;
18
19     @Test
20     public void testSave(){
21         for (int i = 0; i < 100; i++) {
22             this.redisTemplate.opsForValue().set("key_" + i, "value_"+i);
23         }
24
25         Set<String> keys = this.redisTemplate.keys("key_*");
26         for (String key : keys) {
27             String value = this.redisTemplate.opsForValue().get(key);
28             System.out.println(value);
29
30             this.redisTemplate.delete(key);
31         }
32     }
33 }
34
35
```

测试结果：可以打印出结果，说明集群搭建成功！

## 5.5、添加缓存逻辑

实现缓存逻辑有2种方式：

1. 每个接口单独控制缓存逻辑
2. 统一控制缓存逻辑

我们采用第2种方式。

### 5.5.1、采用拦截器进行缓存命中

编写拦截器：RedisCacheInterceptor。



```
1 package cn.itcast.haoke.dubbo.api.interceptor;
2
3 import com.fasterxml.jackson.databind.ObjectMapper;
4 import org.apache.commons.codec.digest.DigestUtils;
5 import org.apache.commons.io.IOUtils;
6 import org.apache.commons.lang3.StringUtils;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.data.redis.core.RedisTemplate;
9 import org.springframework.stereotype.Component;
10 import org.springframework.web.servlet.HandlerInterceptor;
11
12 import javax.servlet.http.HttpServletRequest;
13 import javax.servlet.http.HttpServletResponse;
14 import java.util.Map;
15
16 @Component
17 public class RedisCacheInterceptor implements HandlerInterceptor {
18
19     @Autowired
20     private RedisTemplate<String, String> redisTemplate;
21
22     private static ObjectMapper mapper = new ObjectMapper();
23
24     @Override
25     public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
26         if (!StringUtils.equalsIgnoreCase(request.getMethod(), "get")) {
27             // 非get请求，如果不是graphql请求，放行
28             if (!StringUtils.equalsIgnoreCase(request.getRequestURI(), "/graphql"))
{
29                 return true;
30             }
31         }
32
33         String data =
this.redisTemplate.opsForValue().get(createRedisKey(request));
34         if (StringUtils.isEmpty(data)) {
35             // 缓存未命中
36             return true;
37         }
38         response.setCharacterEncoding("UTF-8");
39         response.setContentType("application/json; charset=utf-8");
40         response.getWriter().write(data);
41         return false;
42     }
43
44     public static String createRedisKey(HttpServletRequest request) throws
Exception {
45         String paramStr = request.getRequestURI();
46
47         Map<String, String[]> parameterMap = request.getParameterMap();
48         if (parameterMap.isEmpty()) {
49             paramStr += IOUtils.toString(request.getInputStream(), "UTF-8");
50         }
51     }
52 }
```



```
50         } else {
51             paramStr += mapper.writeValueAsString(request.getParameterMap());
52         }
53         String rediskey = "WEB_DATA_" + DigestUtils.md5Hex(paramStr);
54
55         return rediskey;
56     }
57 }
58
```

注册拦截器到Spring容器：

```
1 package cn.itcast.haoke.dubbo.api.config;
2
3 import cn.itcast.haoke.dubbo.api.interceptor.RedisCacheInterceptor;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.context.annotation.Configuration;
6 import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
7 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
8
9 @Configuration
10 public class WebConfig implements WebMvcConfigurer {
11
12     @Autowired
13     private RedisCacheInterceptor redisCacheInterceptor;
14
15     @Override
16     public void addInterceptors(InterceptorRegistry registry) {
17         registry.addInterceptor(this.redisCacheInterceptor).addPathPatterns("/**");
18     }
19 }
20
```

### 5.5.2、测试拦截器

发起请求：



POST ▼

http://127.0.0.1:18080/graphql

```
1 ▾ query HouseResourcesList($pageSize: Int, $page: Int) {  
2 ▾ HouseResourcesList(pageSize: $pageSize, page: $page) {  
3 ▾ list {  
4   id  
5   pic  
6   title  
7   coveredArea  
8   orientation  
9   floor  
10  rent  
11 }  
12 }  
13 }
```

#### VARIABLES

```
1 ▾ {  
2   "pageSize":2,  
3   "page":1  
4 }
```

```
@Override  
public boolean preHandle(HttpServletRequest request, HttpServletResponse response) {  
    if (!StringUtils.equalsIgnoreCase(request.getMethod(), "get")) {  
        // 非get请求, 如果不是graphql请求, 放行  
        if (!StringUtils.equalsIgnoreCase(request.getRequestURI(), "/graphql")) {  
            return true;  
        }  
    }  
  
    String data = this.redisTemplate.opsForValue().get(createRedisKey(request));  
    if (StringUtils.isEmpty(data)) { data: null }  
        // 缓存未命中  
        return true;  
    }  
    response.setCharacterEncoding("UTF-8");  
    response.setContentType("application/json; charset=utf-8");  
    response.getWriter().write(data);  
}
```

出现了错误：





```
STATUS: OK STATUS CODE: 400 ⌚ 3207ms
1 {
2   "timestamp": "2018-12-02T16:31:32.372+0000",
3   "status": 400,
4   "error": "Bad Request",
5   "message": "Required request body is missing: public java.util.Map<java.lang.String,
6   java.lang.Object>
7   cn.itcast.haoke.dubbo.api.controller.GraphQLController.postQuery(java.util.Map<java.lang.String,
8   java.lang.Object>)",
9   "path": "/graphql"
10 }
```

错误分析：由于在拦截器中读取了输入流的数据，在request中的输入流只能读取一次，请求进去Controller时，输入流中已经没有数据了，导致获取不到数据。

如何解决？

### 5.5.3、通过包装request解决

编写HttpServletRequest的包装类：

```
1 package cn.itcast.haoke.dubbo.api.interceptor;
2
3 import org.apache.commons.io.IOUtils;
4
5 import javax.servlet.ReadListener;
6 import javax.servlet.ServletInputStream;
7 import javax.servlet.http.HttpServletRequest;
8 import javax.servlet.http.HttpServletRequestWrapper;
9 import java.io.BufferedReader;
10 import java.io.IOException;
11 import java.io.InputStreamReader;
12
13 /**
14  * 包装HttpServletRequest
15  */
16 public class MyServletRequestWrapper extends HttpServletRequestWrapper {
17
18     private final byte[] body;
19
20     /**
21      * Construct a wrapper for the specified request.
22      *
23      * @param request The request to be wrapped
```



```
24     */
25     public MyServletRequestWrapper(HttpServletRequest request) throws IOException
26     {
27         super(request);
28         body = IOUtils.toByteArray(super.getInputStream());
29     }
30
31     @Override
32     public BufferedReader getReader() throws IOException {
33         return new BufferedReader(new InputStreamReader(getInputStream()));
34     }
35
36     @Override
37     public ServletInputStream getInputStream() throws IOException {
38         return new RequestBodyCachingInputStream(body);
39     }
40
41     private class RequestBodyCachingInputStream extends ServletInputStream {
42         private byte[] body;
43         private int lastIndexRetrieved = -1;
44         private ReadListener listener;
45
46         public RequestBodyCachingInputStream(byte[] body) {
47             this.body = body;
48         }
49
50         @Override
51         public int read() throws IOException {
52             if (isFinished()) {
53                 return -1;
54             }
55             int i = body[lastIndexRetrieved + 1];
56             lastIndexRetrieved++;
57             if (isFinished() && listener != null) {
58                 try {
59                     listener.onAllDataRead();
60                 } catch (IOException e) {
61                     listener.onError(e);
62                     throw e;
63                 }
64             }
65             return i;
66         }
67
68         @Override
69         public boolean isFinished() {
70             return lastIndexRetrieved == body.length - 1;
71         }
72
73         @Override
74         public boolean isReady() {
75             // This implementation will never block
76         }
77     }
78 }
```



```
75         // We also never need to call the readListener from this method, as
this method will never return false
76         return isFinished();
77     }
78
79     @Override
80     public void setReadListener(ReadListener listener) {
81         if (listener == null) {
82             throw new IllegalArgumentException("listener cann not be null");
83         }
84         if (this.listener != null) {
85             throw new IllegalArgumentException("listener has been set");
86         }
87         this.listener = listener;
88         if (!isFinished()) {
89             try {
90                 listener.onAllDataRead();
91             } catch (IOException e) {
92                 listener.onError(e);
93             }
94         } else {
95             try {
96                 listener.onAllDataRead();
97             } catch (IOException e) {
98                 listener.onError(e);
99             }
100         }
101     }
102
103     @Override
104     public int available() throws IOException {
105         return body.length - lastIndexRetrieved - 1;
106     }
107
108     @Override
109     public void close() throws IOException {
110         lastIndexRetrieved = body.length - 1;
111         body = null;
112     }
113 }
114 }
```

通过过滤器进行包装request对象：

```
1 package cn.itcast.haoke.dubbo.api.interceptor;
2
3 import org.springframework.stereotype.Component;
4 import org.springframework.web.filter.OncePerRequestFilter;
5
6 import javax.servlet.FilterChain;
7 import javax.servlet.ServletException;
8 import javax.servlet.http.HttpServletRequest;
9 import javax.servlet.http.HttpServletResponse;
```



```
10 import java.io.IOException;
11
12 /**
13  * 替换Request对象
14  */
15 @Component
16 public class RequestReplaceFilter extends OncePerRequestFilter {
17
18     @Override
19     protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
20 response, FilterChain filterChain) throws ServletException, IOException {
21         if (!(request instanceof MyServletRequestWrapper)) {
22             request = new MyServletRequestWrapper(request);
23         }
24         filterChain.doFilter(request, response);
25     }
26 }
```

#### 5.5.4、测试

```
@Override
public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Ob
    if (!StringUtils.equalsIgnoreCase(request.getMethod(), "GET")) {
        // 非GET请求，如果不是graphql请求，放行
        if (!StringUtils.equalsIgnoreCase(request.getMethod(), "POST")) {
            return true;
        }
    }

    String data = this.redisTemplate.opsForValue().get(request.getRequestURI());
    if (StringUtils.isEmpty(data)) {
        // 缓存未命中
        return true;
    }
    response.setCharacterEncoding("UTF-8");
    response.setContentType("application/json; charset=utf-8");
    response.getWriter().write(data);
    return false;
}

public static String createRedisKey(HttpServletRequest request) {
    return RedisCacheInterceptor.createRedisKey(request.getRequestURI());
}
```

Debugger View:

- request: (MyServletRequestWrapper@8567)
- body: (byte[284]@8651)
- request: (RequestFacade@8652)

可以看到，request对象已经经过了包装。

```
@PostMapping
@ResponseBody
public Map<String, Object> postQuery(@RequestBody Map<String, Object> map) {
    try {
        String query = (String) map.get("query");
        if (StringUtils.isEmpty(query)) {
            return null;
        }
    } catch (Exception e) {
        return null;
    }
}
```

Debugger View:

- map: (LinkedHashMap@8918) size = 2
- 0 = (LinkedHashMap\$Entry@8926) "query" -> "query HouseResourcesList(\$pageSize: Int, \$page: Int) { HouseResourcesList(pageSize: \$pageSize, page: \$page) { ... } }
- 1 = (LinkedHashMap\$Entry@8927) "variables" -> "size = 2"

并且在Controller中也可以获取到数据，问题解决。

## 5.6、响应结果写入到缓存

前面已经完成了缓存命中的逻辑，那么在查询到数据后，如果将结果写入到缓存呢？

思考：通过拦截器可以实现吗？

通过ResponseBodyAdvice进行实现。

ResponseBodyAdvice是Spring提供的高级用法，会在结果被处理前进行拦截，拦截的逻辑自己实现，这样就可以实现拿到结果数据进行写入缓存的操作了。

具体实现：

```
1 package cn.itcast.haoke.dubbo.api.interceptor;
2
3 import cn.itcast.haoke.dubbo.api.controller.GraphQLController;
4 import com.fasterxml.jackson.databind.ObjectMapper;
5 import org.apache.commons.lang3.StringUtils;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.core.MethodParameter;
8 import org.springframework.data.redis.core.RedisTemplate;
9 import org.springframework.http.MediaType;
10 import org.springframework.http.server.ServerHttpRequest;
11 import org.springframework.http.server.ServerHttpResponse;
12 import org.springframework.http.server.ServletServerHttpRequest;
13 import org.springframework.web.bind.annotation.ControllerAdvice;
14 import org.springframework.web.bind.annotation.GetMapping;
15 import org.springframework.web.bind.annotation.PostMapping;
16 import org.springframework.web.servlet.mvc.method.annotation.ResponseBodyAdvice;
17
18 import java.time.Duration;
19
20 @ControllerAdvice
21 public class MyResponseBodyAdvice implements ResponseBodyAdvice {
22
23     @Autowired
24     private RedisTemplate<String, String> redisTemplate;
25
26     private ObjectMapper mapper = new ObjectMapper();
27
28     @Override
29     public boolean supports(MethodParameter returnType, Class converterType) {
30         if (returnType.hasMethodAnnotation(GetMapping.class)) {
31             return true;
32         }
33
34         if (returnType.hasMethodAnnotation(PostMapping.class) &&
35             StringUtils.equals(GraphQLController.class.getName(),
36                 returnType.getExecutable().getDeclaringClass().getName())) {
37             return true;
38         }
39     }
40 }
```



```

39         return false;
40     }
41
42     @Override
43     public Object beforeBodyWrite(Object body, MethodParameter returnType,
44     MediaType selectedContentType, Class selectedConverterType, ServerHttpRequest
45     request, ServerHttpResponse response) {
46         try {
47             String redisKey =
48             RedisCacheInterceptor.createRedisKey(((ServletServerHttpRequest)
49             request).getServletRequest());
50             String redisValue;
51             if(body instanceof String){
52                 redisValue = (String)body;
53             }else{
54                 redisValue = mapper.writeValueAsString(body);
55             }
56             this.redisTemplate.opsForValue().set(redisKey,redisValue ,
57             Duration.ofHours(1));
58         } catch (Exception e) {
59             e.printStackTrace();
60         }
61         return body;
62     }
63 }

```

测试：

```

@Override
public Object beforeBodyWrite(Object body, MethodParameter returnType, MediaType
    try {
        String redisKey = RedisCacheInterceptor.createRedisKey(((ServletServer
        this.redisTemplate.opsForValue().set(redisKey, mapper.writeValueAsStri
    } catch (Exception e) {
        e.printStackTrace();
    }
    return body;
}

```

body

body = {LinkedHashMap@9195} size = 1

0 = {LinkedHashMap\$Entry@9208} "data" -> " size = 1"

key = "data"

value = {LinkedHashMap@9210} size = 1

数据已经写入到redis：

```

root@itcast:~# redis-cli -p 6379 -c
127.0.0.1:6379> keys *
1) "WEB_DATA_ffalbf0166720a09ec474990ec80d97"
127.0.0.1:6379> get WEB_DATA_ffalbf0166720a09ec474990ec80d97
"{\"data\":{\"HouseResourcesList\":{\"list\":[{\"id\":\"5\",\"pic\":\"http://itcast-haoke.oss-cn-qingdao.aliyuncs.com/images/2018/11/30/15435107495167066.jpg\",\"http://itcast-haoke.oss-cn-qingdao.aliyuncs.com/images/2018/11/30/15435118101831737.jpg\",\"title\":\"\\xe6\\x9c\\x80\\xe6\\x96\\xb0\\xe4\\xbf\\xae\\xe6\\x94\\xb9\\xe6\\x88\\xbf\\xe6\\xba\\x90\\\",\\\"coveredArea\\\":\\\"100\\\",\\\"orientation\\\":\\\"\\xe5\\x8d\\x97\\\",\\\"floor\\\":\\\"1/1\\\",\\\"rent\\\":1000},{\"id\":\"8\",\"pic\":\"http://itcast-haoke.oss-cn-qingdao.aliyuncs.com/images/2018/11/17/154238960254118.jpg\",\"http://itcast-haoke.oss-cn-qingdao.aliyuncs.com/images/2018/11/17/154238960254118.jpg\",\"title\":\"\\\"222\\\",\\\"coveredArea\\\":\\\"1\\\",\\\"orientation\\\":\\\"\\xe5\\x8d\\x97\\\",\\\"floor\\\":\\\"1/1\\\",\\\"rent\\\":1}]]}"
127.0.0.1:6379>

```

测试命中：



```
}  
  
String data = this.redisTemplate.opsForValue().get(createRedisKey(request)); data: "{\"data\":{\"Hou  
if (StringUtils.isEmpty(data)) { data: \"{"data":{"HouseResourcesList":{"list":[{"id":5,"pic":"htt  
    // 缓存未命中  
    return true;  
}  
}  
response.setCharacterEncoding("UTF-8"); response: ResponseFacade@8560  
response.setContentType("application/json; charset=utf-8");  
response.getWriter().write(data);  
return false;  
}
```

可以看到，数据已经从Redis中命中，进行返回，就不再由Controller处理了。从而达到了缓存的目的。

## 5.7、增加CORS的支持

整合前端系统测试会发现，前面实现的拦截器中并没有对跨域进行支持，需要对CORS跨域支持：

```
1  @Override  
2      public boolean preHandle(HttpServletRequest request, HttpServletResponse  
response, Object handler) throws Exception {  
3          if(StringUtils.equalsIgnoreCase(request.getMethod(), "OPTIONS")){  
4              return true;  
5          }  
6  
7          if (!StringUtils.equalsIgnoreCase(request.getMethod(), "get")) {  
8              // 非get请求，如果不是graphql请求，放行  
9              if (!StringUtils.equalsIgnoreCase(request.getRequestURI(), "/graphql"))  
10             {  
11                 return true;  
12             }  
13  
14             String data =  
this.redisTemplate.opsForValue().get(createRedisKey(request));  
15             if (StringUtils.isEmpty(data)) {  
16                 // 缓存未命中  
17                 return true;  
18             }  
19             response.setCharacterEncoding("UTF-8");  
20             response.setContentType("application/json; charset=utf-8");  
21  
22             // 支持跨域  
23             response.setHeader("Access-Control-Allow-Origin", "*");  
24             response.setHeader("Access-Control-Allow-Methods",  
"GET,POST,PUT,DELETE,OPTIONS");  
25             response.setHeader("Access-Control-Allow-Credentials", "true");  
26             response.setHeader("Access-Control-Allow-Headers", "Content-Type,X-Token");  
27             response.setHeader("Access-Control-Allow-Credentials", "true");  
28             response.getWriter().write(data);  
29             return false;  
30         }  
}
```

## 6、WebSocket

## 6.1、网站中的消息功能如何实现？



思考：像这样的消息功能怎么实现？如果网页不刷新，服务端有新消息如何推送到浏览器？

解决方案，采用轮询的方式。即：通过js不断的请求服务器，查看是否有新数据，如果有，就获取到新数据。

这种解决方法是否存在问题呢？

当然是有的，如果服务端一直没有新的数据，那么js也是需要一直的轮询查询数据，这就是一种资源的浪费。

那么，有没有更好的解决方案？有！那就是采用WebSocket技术来解决。

## 6.1、什么是WebSocket？

WebSocket 是HTML5一种新的协议。它实现了浏览器与服务器全双工通信(full-duplex)。一开始的握手需要借助HTTP请求完成。WebSocket是真正实现了全双工通信的服务器向客户端推的互联网技术。它是一种在单个TCP连接上进行全双工通讯协议。Websocket通信协议与2011年IETF定为标准RFC 6455，Websocket API被W3C定为标准。

全双工和单工的区别？

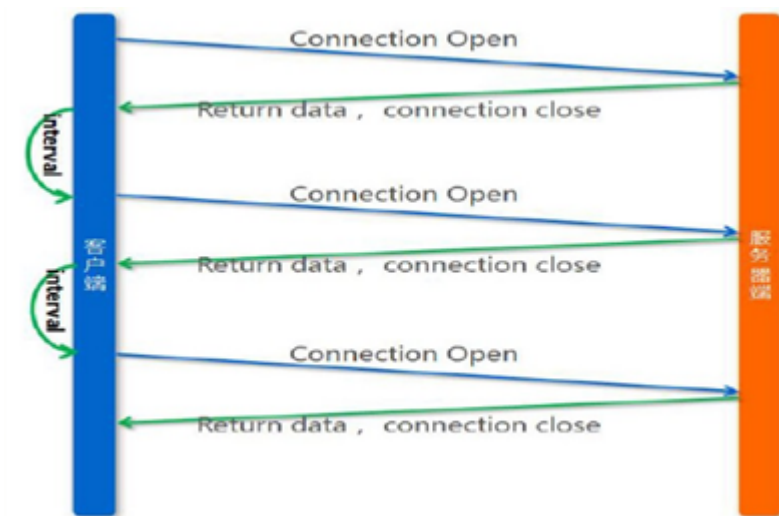
- 全双工（Full Duplex）是通讯传输的一个术语。通信允许数据在两个方向上同时传输，它在能力上相当于两个单工通信方式的结合。全双工指可以同时（瞬时）进行信号的双向传输（ $A \rightarrow B$ 且 $B \rightarrow A$ ）。指 $A \rightarrow B$ 的同时 $B \rightarrow A$ ，是瞬时同步的。
- 单工、半双工（Half Duplex），所谓半双工就是指一个时间段内只有一个动作发生，举个简单例子，一条窄窄的马路，同时只能有一辆车通过，当前有两辆车对开，这种情况下就只能一辆先过，等到头儿后另一辆再开，这个例子就形象的说明了半双工的原理。早期的对讲机、以及早期集线器等设备都是基于半双工的产品。随着技术的不断进步，半双工会逐渐退出历史舞台。

## 6.2、http与websocket的区别

### 6.2.1、http

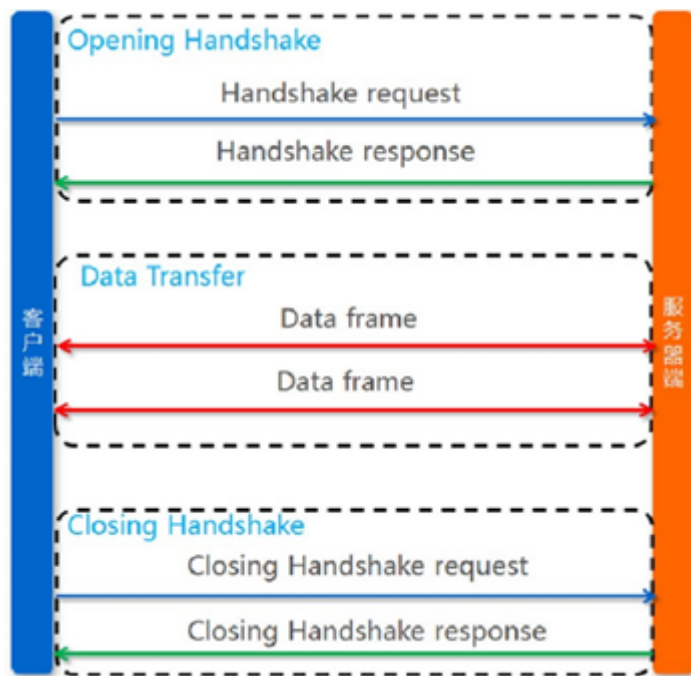
http协议是短连接，因为请求之后，都会关闭连接，下次重新请求数据，需要再次打开链接。





### 6.2.2、websocket

WebSocket协议是一种长链接，只需要通过一次请求来初始化链接，然后所有的请求和响应都是通过这个TCP链接进行通讯。



### 6.3、浏览器支持情况



查看：<https://caniuse.com/#search=websocket>

Current aligned Usage relative Date relative Apply filters Show all ?										
IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Blackberry Browser	Opera Mobile *
		2-3.6		3.1-4						
		4-5	4-14	5-5.1	10.1	3.2-4.1				
6-9		6-10	15	6-6.1	11.5	4.2-5.1		2.1-4.3		12
10	12-17	11-62	16-69	7-11.1	12.1-55	6-11.4		4.4-4.4.4	7	12.1
11	18	63	70	12	56	12	all	67	10	46
		64-65	71-73	TP						

服务器支持情况：Tomcat 7.0.47+以上才支持。

## 6.4、快速入门

### 6.4.1、创建itcast-websocket工程

pom.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5         http://maven.apache.org/xsd/maven-4.0.0.xsd">
6
7     <groupId>cn.itcast.websocket</groupId>
8     <artifactId>itcast-websocket</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <packaging>war</packaging>
11
12    <dependencies>
13        <dependency>
14            <groupId>javax</groupId>
15            <artifactId>javaee-api</artifactId>
16            <version>7.0</version>
17            <scope>provided</scope>
18        </dependency>
19    </dependencies>
20
21    <build>
22        <plugins>
23            <!-- java编译插件 -->
24            <plugin>
25                <groupId>org.apache.maven.plugins</groupId>
26                <artifactId>maven-compiler-plugin</artifactId>
27                <version>3.2</version>
28                <configuration>
29                    <source>1.8</source>
```



```
30         <target>1.8</target>
31         <encoding>UTF-8</encoding>
32     </configuration>
33 </plugin>
34 <!-- 配置Tomcat插件 -->
35 <plugin>
36     <groupId>org.apache.tomcat.maven</groupId>
37     <artifactId>tomcat7-maven-plugin</artifactId>
38     <version>2.2</version>
39     <configuration>
40         <port>8082</port>
41         <path>/</path>
42     </configuration>
43 </plugin>
44 </plugins>
45 </build>
46
47 </project>
```

### 6.4.2、websocket的相关注解说明

- @ServerEndpoint("/websocket/{uid}")
  - 申明这是一个websocket服务
  - 需要指定访问该服务的地址，在地址中可以指定参数，需要通过{}进行占位
- @OnOpen
  - 用法：public void onOpen(Session session, @PathParam("uid") String uid) throws IOException {}
  - 该方法将在建立连接后执行，会传入session对象，就是客户端与服务端建立的长连接通道
  - 通过@PathParam获取url申明中的参数
- @OnClose
  - 用法：public void onClose() {}
  - 该方法是在连接关闭后执行
- @OnMessage
  - 用法：public void onMessage(String message, Session session) throws IOException {}
  - 该方法用于接收客户端发来的消息
  - message：发来的消息数据
  - session：会话对象（也是通道）
- 发送消息到客户端
  - 用法：session.getBasicRemote().sendText("你好");
  - 通过session进行发送。

### 6.4.3、实现websocket服务

```
1 package cn.itcast.websocket;
2
3 import javax.websocket.*;
4 import javax.websocket.server.PathParam;
5 import javax.websocket.server.ServerEndpoint;
6 import java.io.IOException;
```



```
7
8 @ServerEndpoint("/websocket/{uid}")
9 public class MyWebSocket {
10
11     @OnOpen
12     public void onOpen(Session session, @PathParam("uid") String uid) throws
13     IOException {
14         // 连接成功
15         session.getBasicRemote().sendText(uid + ", 你好, 欢迎连接WebSocket!");
16     }
17
18     @OnClose
19     public void onClose() {
20         System.out.println(this + "关闭连接");
21     }
22
23     @OnMessage
24     public void onMessage(String message, Session session) throws IOException {
25         System.out.println("接收到消息:" + message);
26         session.getBasicRemote().sendText("消息已收到.");
27     }
28
29     @OnError
30     public void onError(Session session, Throwable error) {
31         System.out.println("发生错误");
32         error.printStackTrace();
33     }
34 }
35
```

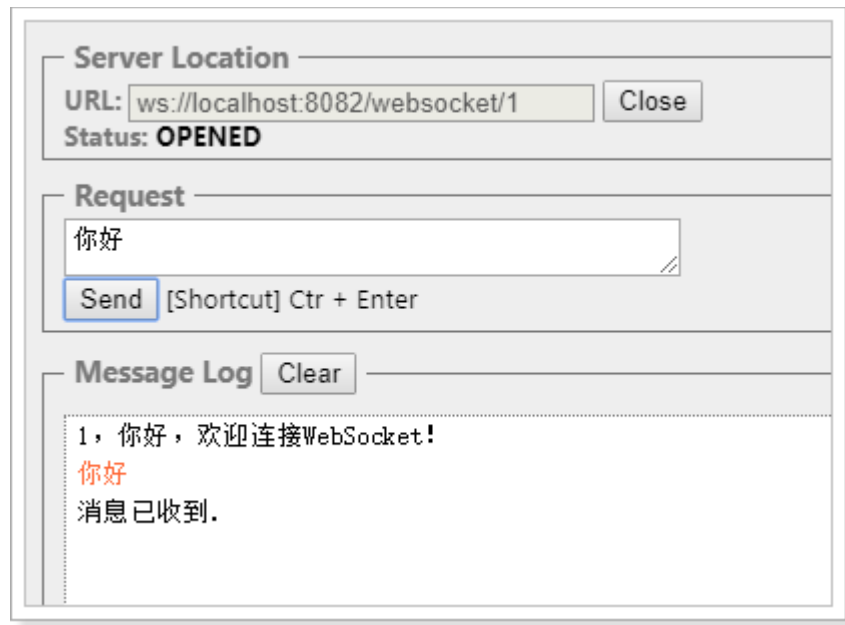
编写完成后，无需进额外的配置，直接启动tomcat即可。

#### 6.4.4、测试

可以通过安装chrome插件或者通过在线工具进行测试：

chrome插件，Simple WebSocket Client：

<https://chrome.google.com/webstore/detail/simple-websocket-client/pfdhoblngboilpfeibdedpjgfnlcodoo>



在线工具：<https://easyswoole.com/wstool.html>



#### 6.4.4、编写js客户端

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Title</title>
6 </head>
7 <body>
8 <script>
9
10   const socket = new WebSocket("ws://localhost:8082/websocket/1");
11   socket.onopen = (ws) =>{
12     console.log("建立连接!", ws);
13   }
14   socket.onmessage = (ws) =>{
15     console.log("接收到消息 >> ",ws.data);
16   }
```



```
17     socket.onclose = (ws) =>{
18         console.log("连接已断开！", ws);
19     }
20     socket.onerror = (ws) => {
21         console.log("发送错误！", ws);
22     }
23
24     // 2秒后向服务端发送消息
25     setTimeout(()=>{
26         socket.send("发送一条消息试试");
27     },2000);
28
29     // 5秒后断开连接
30     setTimeout(()=>{
31         socket.close();
32     },5000);
33
34 </script>
35 </body>
36 </html>
```

测试：

```
建立连接! ▶Event {isTrusted: true, type: "open", target: WebSocket, currentTarget: WebSocket, eventPhase: 2, ...}
接收到消息 >> 1, 你好, 欢迎连接WebSocket!
接收到消息 >> 消息已收到.
连接已断开! ▶CloseEvent {isTrusted: true, wasClean: true, code: 1000, reason: "", type: "close", ...}
```

## 6.5、SpringBoot整合WebSocket

Spring对WebSocket做了支持，下面我们看下在springboot中如何使用。

### 6.5.1、导入依赖

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5         http://maven.apache.org/xsd/maven-4.0.0.xsd">
6     <modelVersion>4.0.0</modelVersion>
7
8     <!--spring boot的支持-->
9     <parent>
10         <groupId>org.springframework.boot</groupId>
11         <artifactId>spring-boot-starter-parent</artifactId>
12         <version>2.1.0.RELEASE</version>
13     </parent>
14
15     <groupId>cn.itcast.websocket</groupId>
16     <artifactId>itcast-websocket</artifactId>
17     <version>1.0-SNAPSHOT</version>
18     <packaging>war</packaging>
```



```
18
19     <dependencies>
20         <!--<dependency>-->
21             <!--<groupId>javax</groupId>-->
22             <!--<artifactId>javaee-api</artifactId>-->
23             <!--<version>7.0</version>-->
24             <!--<scope>provided</scope>-->
25         <!--</dependency>-->
26         <dependency>
27             <groupId>org.springframework.boot</groupId>
28             <artifactId>spring-boot-starter-websocket</artifactId>
29         </dependency>
30     </dependencies>
31
32     <build>
33         <plugins>
34             <!-- java编译插件 -->
35             <plugin>
36                 <groupId>org.apache.maven.plugins</groupId>
37                 <artifactId>maven-compiler-plugin</artifactId>
38                 <version>3.2</version>
39                 <configuration>
40                     <source>1.8</source>
41                     <target>1.8</target>
42                     <encoding>UTF-8</encoding>
43                 </configuration>
44             </plugin>
45             <!-- 配置Tomcat插件 -->
46             <plugin>
47                 <groupId>org.apache.tomcat.maven</groupId>
48                 <artifactId>tomcat7-maven-plugin</artifactId>
49                 <version>2.2</version>
50                 <configuration>
51                     <port>8082</port>
52                     <path>/</path>
53                 </configuration>
54             </plugin>
55         </plugins>
56     </build>
57
58 </project>
```

## 6.5.2、编写WebSocketHandler

在Spring中，处理消息的具体业务逻辑需要实现WebSocketHandler接口。

```
1 package cn.itcast.websocket.spring;
2
3 import org.springframework.web.socket.CloseStatus;
4 import org.springframework.web.socket.TextMessage;
5 import org.springframework.web.socket.WebSocketSession;
6 import org.springframework.web.socket.handler.TextWebSocketHandler;
7
```



```
8 import java.io.IOException;
9
10 public class MyHandler extends TextWebSocketHandler {
11
12     @Override
13     public void handleTextMessage(WebSocketSession session, TextMessage message)
14     throws IOException {
15         System.out.println("获取到消息 >> " + message.getPayload());
16
17         session.sendMessage(new TextMessage("消息已收到"));
18
19         if(message.getPayload().equals("10")){
20             for (int i = 0; i < 10; i++) {
21                 session.sendMessage(new TextMessage("消息 -> " + i));
22                 try {
23                     Thread.sleep(100);
24                 } catch (InterruptedException e) {
25                     e.printStackTrace();
26                 }
27             }
28         }
29
30         @Override
31         public void afterConnectionEstablished(WebSocketSession session) throws
32         Exception {
33             session.sendMessage(new TextMessage("欢迎连接到ws服务"));
34         }
35
36         @Override
37         public void afterConnectionClosed(WebSocketSession session, CloseStatus status)
38         throws Exception {
39             System.out.println("断开连接!");
40         }
41     }
42 }
```

### 6.5.3、编写配置类

```
1 package cn.itcast.websocket.spring;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.web.socket.WebSocketHandler;
6 import org.springframework.web.socket.config.annotation.EnableWebSocket;
7 import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
8 import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;
9
10 @Configuration
11 @EnableWebSocket
12 public class WebSocketConfig implements WebSocketConfigurer {
13
14     @Override
```



```
15     public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
16         registry.addHandler(myHandler(), "/ws").setAllowedOrigins("*");
17     }
18
19     @Bean
20     public WebSocketHandler myHandler() {
21         return new MyHandler();
22     }
23
24 }
```

#### 6.5.4、编写启动类

```
1 package cn.itcast.websocket;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class MyApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(MyApplication.class, args);
11     }
12 }
13
```

#### 6.5.5、测试

服务器配置 状态：连接成功

服务地址

ws://localhost:8083/ws

关闭连接

发包设置

每隔

1

秒发送内容

PING

开始发送

10

☐ 发包清空输入

发送到服务端

调试消息

02:23:04 => 初始化完成

02:23:05 => OPENED => localhost:8083/ws

消息记录

☐ 收包清空记录 ☐ 收包JSON解码 ☐ 暂停接收

收到消息 02:23:05

欢迎连接到ws服务

发送消息 02:23:09

11

收到消息 02:23:09

消息已收到

发送消息 02:23:14

10

收到消息 02:23:14

消息已收到

收到消息 02:23:14

### 6.6、websocket拦截器

在Spring中提供了websocket拦截器，可以在建立连接之前写些业务逻辑，比如校验登录等。



实现：

```
1 package cn.itcast.websocket.spring;
2
3 import org.springframework.http.server.ServerHttpRequest;
4 import org.springframework.http.server.ServerHttpResponse;
5 import org.springframework.stereotype.Component;
6 import org.springframework.web.socket.WebSocketHandler;
7 import org.springframework.web.socket.server.HandshakeInterceptor;
8
9 import java.util.Map;
10
11 @Component
12 public class MyHandshakeInterceptor implements HandshakeInterceptor {
13
14     /**
15      * 握手之前，若返回false，则不建立链接
16      *
17      * @param request
18      * @param response
19      * @param wsHandler
20      * @param attributes
21      * @return
22      * @throws Exception
23      */
24     @Override
25     public boolean beforeHandshake(ServerHttpRequest request, ServerHttpResponse
response, WebSocketHandler wsHandler, Map<String, Object> attributes) throws
Exception {
26         //将用户id放入socket处理器的会话(WebSocketSession)中
27         attributes.put("uid", 1001);
28         System.out.println("开始握手。。。。。。");
29         return true;
30     }
31
32     @Override
33     public void afterHandshake(ServerHttpRequest request, ServerHttpResponse
response, WebSocketHandler wsHandler, Exception exception) {
34         System.out.println("握手成功啦。。。。。。");
35     }
36 }
37
```

将拦截器添加到websocket服务中：

```
1 package cn.itcast.websocket.spring;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6 import org.springframework.web.socket.WebSocketHandler;
7 import org.springframework.web.socket.config.annotation.EnableWebSocket;
```



```
8 import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
9 import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;
10
11 @Configuration
12 @EnableWebSocket
13 public class WebSocketConfig implements WebSocketConfigurer {
14
15     @Autowired
16     private MyHandshakeInterceptor myHandshakeInterceptor;
17
18     @Override
19     public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
20         registry.addHandler(myHandler(), "/ws")
21
22         .setAllowedOrigins("*").addInterceptors(this.myHandshakeInterceptor);
23     }
24
25     @Bean
26     public WebSocketHandler myHandler() {
27         return new MyHandler();
28     }
29 }
```

获取uid：

```
@Override
public void afterConnectionEstablished(WebSocketSession session) throws Exception {
    System.out.println("uid => " + session.getAttributes().get("uid"));
    session.sendMessage(new TextMessage(payload: "欢迎连接到ws服务"));
}
```

测试：

```
开始握手。。。。。。
握手成功啦。。。。。。
uid => 1001
```