# 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：

```
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"}}
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"}}
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"}}
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"}}
```

```
mock.infosList1={"data":{"list":{"total":8,"data":[{"id":13,"info_title":"wwwwwwwwwwwww","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"}}
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"}}
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"}}
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。

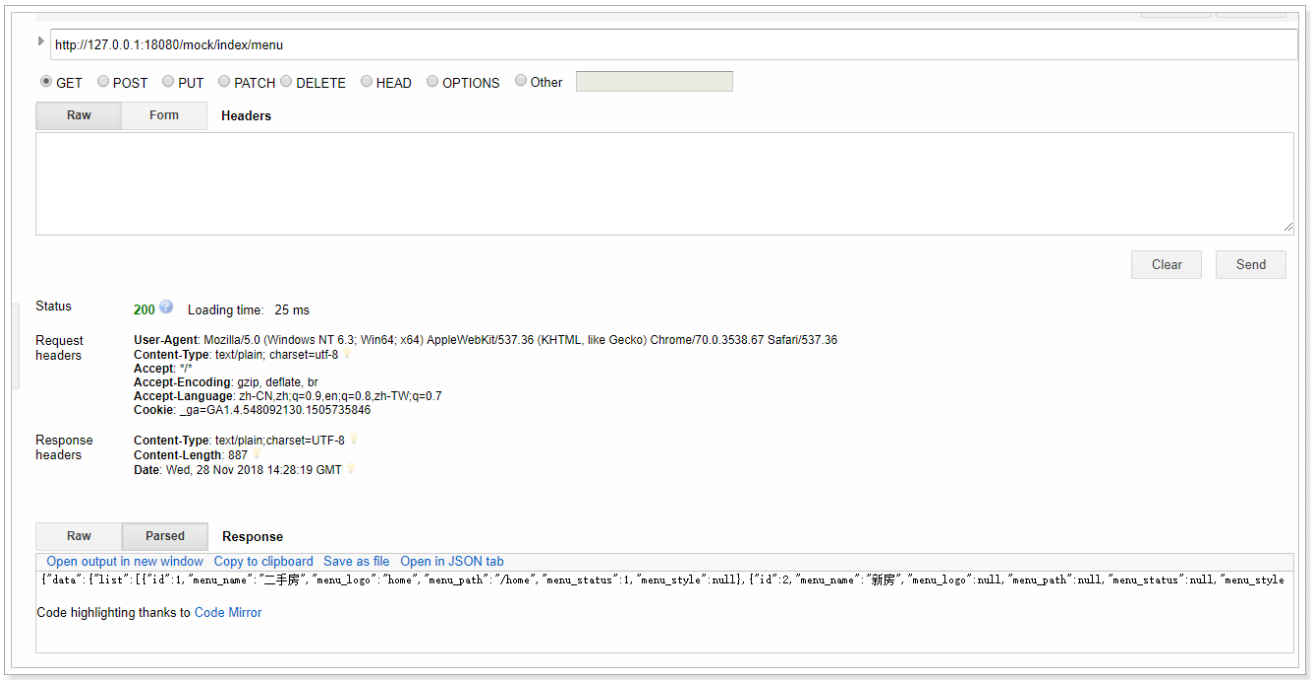
```java
package cn.itcast.haoke.dubbo.api.config;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;

@Configuration
@PropertySource("classpath:mock-data.properties")
@ConfigurationProperties(prefix = "mock")
@Data
public class MockConfig {

    private String indexMenu;
    private String indexInfo;
    private String indexFaq;
    private String indexHouse;
    private String infosList1;
    private String infosList2;
    private String infosList3;
    private String my;

}

```

## 1.3、创建MockController

```java
package cn.itcast.haoke.dubbo.api.controller;

import cn.itcast.haoke.dubbo.api.config.MockConfig;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.PropertySource;
import org.springframework.web.bind.annotation.*;

@RequestMapping("mock")
@RestController
@CrossOrigin
public class MockController {

    @Autowired
    private MockConfig mockConfig;

    /**
     * 菜单
     *
     * @return
     */
    @GetMapping("index/menu")
    public String indexMenu() {
        return this.mockConfig.getIndexMenu();
    }

    /**
     * 首页资讯
     * @return
     */
    @GetMapping("index/info")
    public String indexInfo() {
        return this.mockConfig.getIndexInfo();
    }

    /**
     * 首页问答
     * @return
     */
    @GetMapping("index/faq")
    public String indexFaq() {
        return this.mockConfig.getIndexFaq();
    }

    /**
     * 首页房源信息
     * @return
     */
    @GetMapping("index/house")
    public String indexHouse() {
        return this.mockConfig.getIndexHouse();
    }
```

```
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



## 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：

```
{
  HouseResources(id: 8) {
    id
    title
    estateId
    buildingUnit
    buildingFloorNum
    mobile
    useArea
    pic
  }
}
```

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：

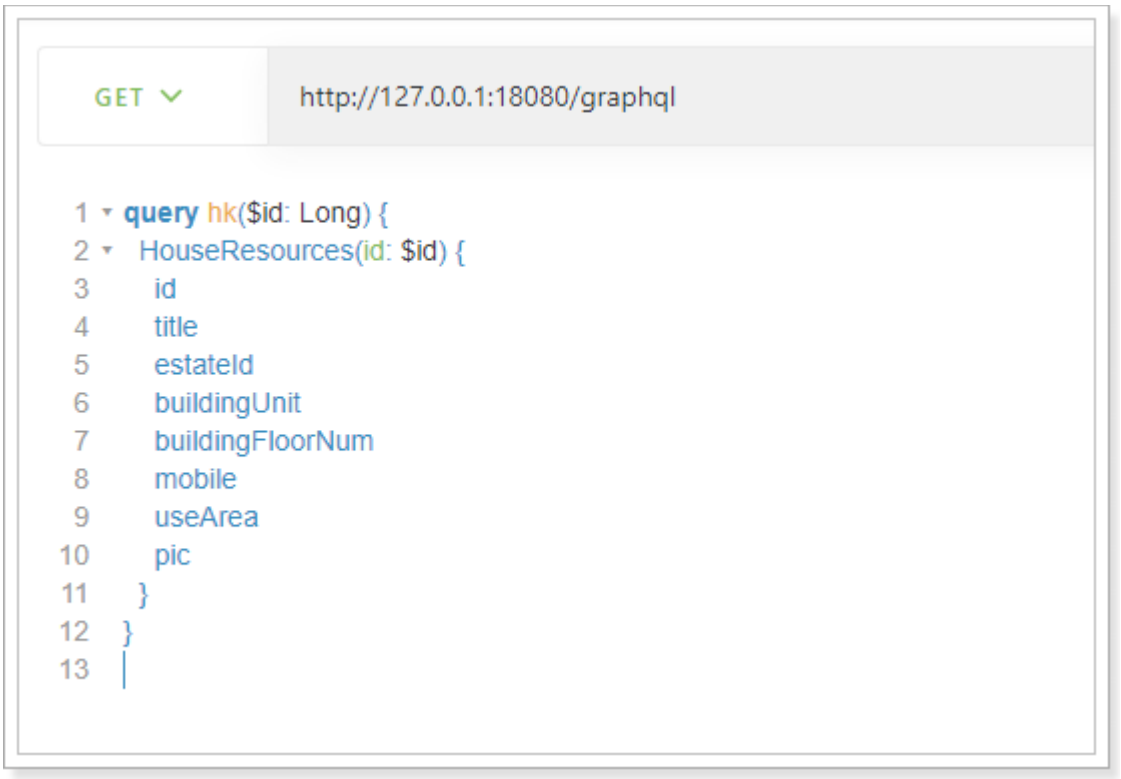
1. 使用 `$variableName` 替代查询中的静态值。
2. 声明 `$variableName` 为查询接受的变量之一。
3. 将 `variableName: value` 通过传输专用（通常是 JSON）的分离的变量字典中。

全部做完之后就像这个样子：

```
# { "graphiql": 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




```
query hk($id: Long) {
  HouseResources(id: $id) {
    id
    title
    estateId
    buildingUnit
    buildingFloorNum
    mobile
    useArea
    pic
  }
}
```

> 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

```
{
  "id":8
}
```

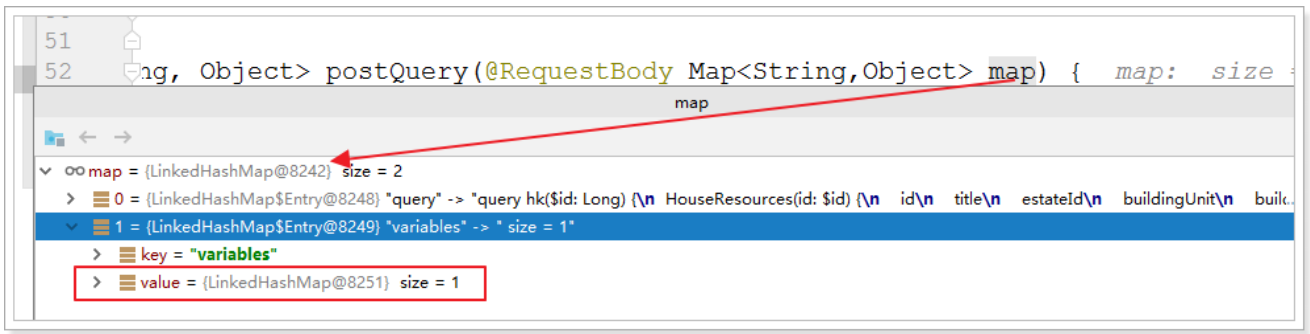

## 2.3、服务端处理参数

```
package cn.itcast.haoke.dubbo.api.controller;

import com.fasterxml.jackson.databind.ObjectMapper;
import graphql.ExecutionInput;
import graphql.GraphQL;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import java.io.IOException;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

@RequestMapping("graphql")
@Controller
@CrossOrigin
public class GraphQLController {

    @Autowired
    private GraphQL graphQL;

    private static final ObjectMapper MAPPER = new ObjectMapper();


    /**
     * 实现GraphQL查询
     *
     * @param query
     * @return
     */
    @GetMapping
    @ResponseBody
    public Map<String, Object> query(@RequestParam("query") String query,
                                     @RequestParam(value = "variables", required =
false) String variablesJson,
```

```
                                        @RequestParam(value = "operationName",
required = false) String operationName) {
        try {
            Map variables = MAPPER.readValue(variablesJson,
MAPPER.getTypeFactory().constructMapType(HashMap.class, String.class,
Object.class));
            return this.executeGraphqlQuery(query, operationName, variables);
        } catch (IOException e) {
            e.printStackTrace();
        }

        Map<String, Object> error = new HashMap<>();
        error.put("status", 500);
        error.put("msg", "查询出错");
        return error;
    }

    @PostMapping
    @ResponseBody
    public Map<String, Object> postQuery(@RequestBody Map<String,Object> map) {
        try {
            String query = (String) map.get("query");
            if(null == query){
                query = "";
            }
            String operationName = (String) map.get("operationName");
            Map variables = (Map) map.get("variables");
            if(variables == null){
                variables = Collections.EMPTY_MAP;
            }
            return this.executeGraphqlQuery(query, operationName, variables);
        } catch (Exception e) {
            e.printStackTrace();
        }

        Map<String, Object> error = new HashMap<>();
        error.put("status", 500);
        error.put("msg", "查询出错");
        return error;
    }

    private Map<String, Object> executeGraphqlQuery(String query, String
operationName, Map<String, Object> variables) {
        ExecutionInput executionInput = ExecutionInput.newExecutionInput()
                .query(query)
                .operationName(operationName)
                .variables(variables)
                .build();
        return this.graphQL.execute(executionInput).toSpecification();
    }

}

```

## 2.4、测试


POST
http://127.0.0.1:18080/graphql
query hk($id: Long) {
HouseResources(id: $id) {
id
title
estateId
buildingUnit
buildingFloorNum
mobile
useArea
pic
}
}
VARIABLES
{
"id":8
}


ng, Object> postQuery(@RequestBody Map<String,Object> map) {
map
map = {LinkedHashMap@8242} size = 2
0 = {LinkedHashMap$Entry@8248} "query" -> "query hk($id: Long) {\n HouseResources(id: $id) {\n id\n title\n estateId\n buildingUnit\n
1 = {LinkedHashMap$Entry@8249} "variables" -> " size = 1"
key = "variables"
value = {LinkedHashMap@8251} size = 1


# 3、实现房源列表查询

## 3.1、编写查询字符串

```
query HouseResourcesList($pageSize: Int, $page: Int) {
  HouseResourcesList(pageSize: $pageSize, page: $page) {
    list {
      id
      pic
      title
      coveredArea
      orientation
      floor
      rent
    }
  }
}
```

查询参数:

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

## 3.2、改造list.js页面

```
import ApolloClient from "apollo-boost";
import gql from "graphql-tag";


const client = new ApolloClient({
    uri: "http://127.0.0.1:18080/graphql"
});

//定义查询
const QUERY_LIST = gql`
    query HouseResourcesList($pageSize: Int, $page: Int) {
        HouseResourcesList(pageSize: $pageSize, page: $page) {
            list {
                id
                pic
                title
                coveredArea
                orientation
                floor
                rent
            }
        }
    }
`;

// 查询
client.query({query: QUERY_LIST, variables: {"pageSize":3,"page":1}}).then(result
=> {
               this.setState({
                   listData:result.data.HouseResourcesList.list,
                   loadFlag: true
               });
      });

//渲染
let list = null;
    if(this.state.loadFlag) {
      list = this.state.listData.map(item=>{
        return (
          <Item key={item.id}>
            <Item.Image src={item.pic.split(',')[0]}/>
            <Item.Content>
              <Item.Header>{item.title}</Item.Header>
              <Item.Meta>
                <span className='cinema'>{item.coveredArea}
㎡/{item.orientation}/{item.floor}</span>
              </Item.Meta>
              <Item.Description>
                上海
              </Item.Description>
              <Item.Description>{item.rent}</Item.Description>
            </Item.Content>
          </Item>
```

```
        )
      });
    }
```

### 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

```
/**
     * 修改房源
     *
     * @param houseResources json数据
     * @return
     */
```

```
@PutMapping
@ResponseBody
public ResponseEntity<Void> update(@RequestBody HouseResources houseResources) {
    try {
        boolean bool = this.houseResourcesService.update(houseResources);
        if (bool) {
            return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
}
```

### 4.1.2、修改service

> itcast-haoke-manage-api-server/HouseResourcesService

```
public boolean update(HouseResources houseResources) {
    return this.apiHouseResourcesService.updateHouseResources(houseResources);
}
```

### 4.1.3、修改dubbo服务

修改接口 ApiHouseResourcesService：

```
    /**
     * 修改房源
     *
     * @param houseResources
     * @return
     */
    boolean updateHouseResources(HouseResources houseResources);
```

修改实现类ApiHouseResourcesServiceImpl：

```
    @Override
    public boolean updateHouseResources(HouseResources houseResources) {
        return this.houseResourcesService.updateHouseResources(houseResources);
    }

```

修改业务Service：HouseResourcesServiceImpl

```
    @Override
    public boolean updateHouseResources(HouseResources houseResources) {
        return super.update(houseResources) == 1;
    }

```

## 4.2、编写EditResource.js

```
import React from 'react';
import {Checkbox, Form, Input, Modal} from "antd";
import PicturesWall from "../Utils/PicturesWall";
import {connect} from "dva";

const FormItem = Form.Item;
const InputGroup = Input.Group;
const CheckboxGroup = Checkbox.Group;

const formItemLayout = {
  labelCol: {
    xs: { span: 24 },
    sm: { span: 7 },
  },
  wrapperCol: {
    xs: { span: 24 },
    sm: { span: 12 },
    md: { span: 10 },
  },
};

@connect()
@Form.create()
class EditResource extends React.Component{

  constructor(props){
    super(props);

    this.state={
      visible:false,
      pics:new Set()
    };
  }

  showModal = () => {
    this.setState({
      visible: true
    });
  };

  handleCancel = () => {
    this.setState({
      visible: false,
    });
  };

  handleSave = () => {

    const { dispatch, form, record } = this.props;
    form.validateFieldsAndScroll((err, values) => {
      if (!err) {
```

```
        if(this.state.pics.size > 0){
          values.pic = [...this.state.pics].join(',');
        }
        values.id = record.id;

        dispatch({
          type: 'house/updateHouseForm',
           payload: values,
        });
        setTimeout(()=>{
          this.handleCancel();
          this.props.reload();
        },500)

      }
    });

  };

  handleFileList = (obj)=>{
    let pics = new Set();
    obj.forEach((v, k) => {
      if(v.response){
        pics.add(v.response.name);
      }
      if(v.url){
        pics.add(v.url);
      }
    });

    this.setState({
      pics : pics
    })
  }

  render(){

    const record = this.props.record;
    const {
      form: { getFieldDecorator }
    } = this.props;

    return (
      <React.Fragment>
        <a onClick={() => {this.showModal()}}>编辑</a>
        <Modal
          title={'编辑'}
          width={640}
          visible={this.state.visible}
          onOk={()=>{this.handleSave()}}
          onCancel={()=>{this.handleCancel()}}
          destroyOnClose={true}
        >
```

```
          <div style={{ overflowY:'auto'}}>
            <Form hideRequiredMark style={{ marginTop: 8 }}>
              <FormItem {...formItemLayout} label="房源标题">
                {getFieldDecorator('title',{initialValue:record.title  ,rules:[{ required: true, message:"此项为必填项" }]})(<Input style={{ width: '100%' }} disabled={false} />)}
              </FormItem>
              <FormItem {...formItemLayout} label="租金">
                <InputGroup compact>
                  {getFieldDecorator('rent',{initialValue:record.rent ,rules:[{ required: true, message:"此项为必填项" }]})(<Input style={{ width: '50%' }} addonAfter="元/月" />)}
                </InputGroup>
              </FormItem>
              <FormItem {...formItemLayout} label="建筑面积">
                <InputGroup compact>
                  {getFieldDecorator('coveredArea', {initialValue:record.coveredArea,rules:[{ required: true, message:"此项为必填项" }]})(<Input style={{ width: '40%' }} addonAfter="平米" />)}
                </InputGroup>
              </FormItem>
              <FormItem {...formItemLayout} label="上传室内图">
                <PicturesWall handleFileList={this.handleFileList.bind(this)} fileList={record.pic}/>
              </FormItem>
            </Form>
          </div>

        </Modal>
      </React.Fragment>
    );
  }
}



export default EditResource;

```

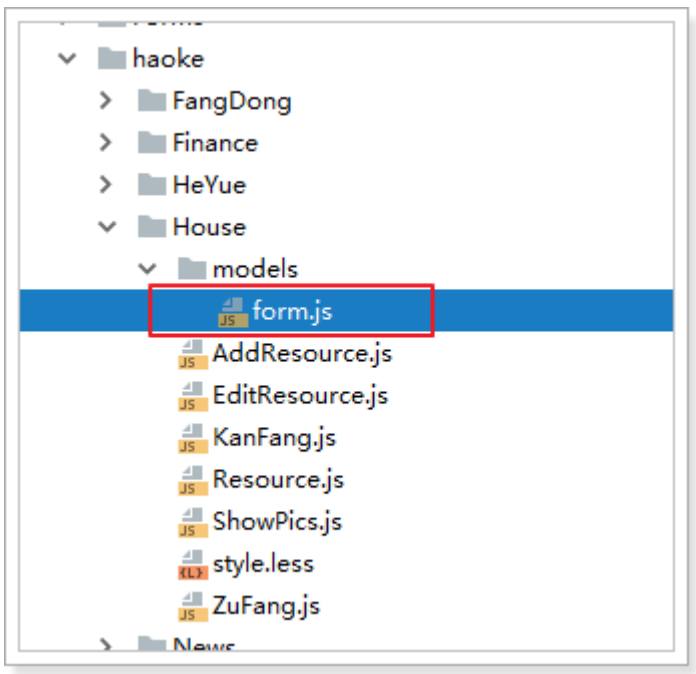
## 4.3、修改房源列表页

```
{
      title: '操作',
      render: (text, record) => (
        <Fragment>
          <a onClick={() => {}}>查看</a>
          <Divider type="vertical" />
          <EditResource record={record} reload={this.reload.bind(this)}/>
          <Divider type="vertical" />
          <a href="">删除</a>
        </Fragment>
      ),
```

```
    },

---------------------

reload(){
    const { dispatch } = this.props;
    dispatch({
      type: 'houseResource/fetch'
    });
  }
```

效果：

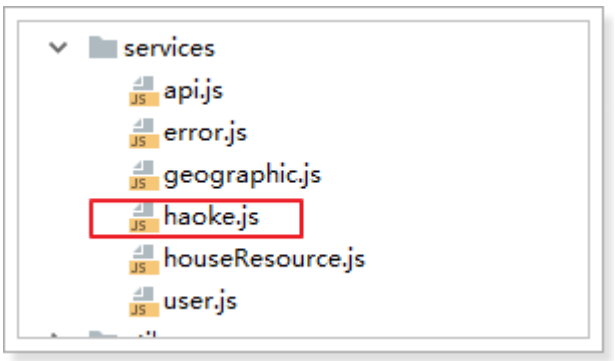

## 4.4、修改提交逻辑




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

```
import { message } from 'antd';
import { addHouseResource,updateHouseResource } from '@/services/haoke';

export default {
  namespace: 'house',

  state: {

  },

  effects: {
    *submitHouseForm({ payload }, { call }) {
      yield call(addHouseResource, payload);
      message.success('提交成功');
    },
    *updateHouseForm({ payload }, { call }) {
      yield call(updateHouseResource, payload);
      message.success('提交成功');
    }
  },

  reducers: {
    saveStepFormData(state, { payload }) {
      return {
        ...state
      };
    },
  },
};

```

## 4.5、修改service逻辑




```
import request from '@/utils/request';

export async function addHouseResource(params) {
  return request('/haoke/house/resources', {
    method: 'POST',
    body: params
  });
}

```

```
export async function updateHouseResource(params) {
  return request('/haoke/house/resources', {
    method: 'PUT',
    body: params
  });
}

```

## 4.6、测试


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

```
#拉取镜像
docker pull redis:5.0.2

#创建容器
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

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

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

#启动容器
docker start redis-node01 redis-node02 redis-node03

#开始组建集群

#进入redis-node01进行操作
docker exec -it redis-node01 /bin/bash

#组建集群
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: d328777729012693 75cb3e4993c05f0bb6356c84 172.17.0.1:6380
   slots:[5461-10922] (5462 slots) master
M: f558eacb2903c75667aa0a8d9afa0667242e0f66 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）：

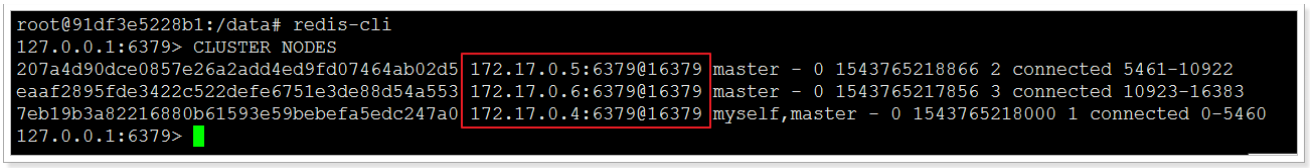
```
#查看容器的ip地址
```

```
docker inspect redis-node01  ->  172.17.0.4
docker inspect redis-node02  ->  172.17.0.5
docker inspect redis-node03  ->  172.17.0.6

#删除容器
docker stop redis-node01 redis-node02 redis-node03
docker rm redis-node01 redis-node02 redis-node03
rm -rf /data/redis-data

#进入redis-node01进行操作
docker exec -it redis-node01 /bin/bash

#组建集群(注意端口的变化)
redis-cli --cluster create 172.17.0.4:6379 172.17.0.5:6379 172.17.0.6:6379 --
cluster-replicas 0

```

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
```

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？

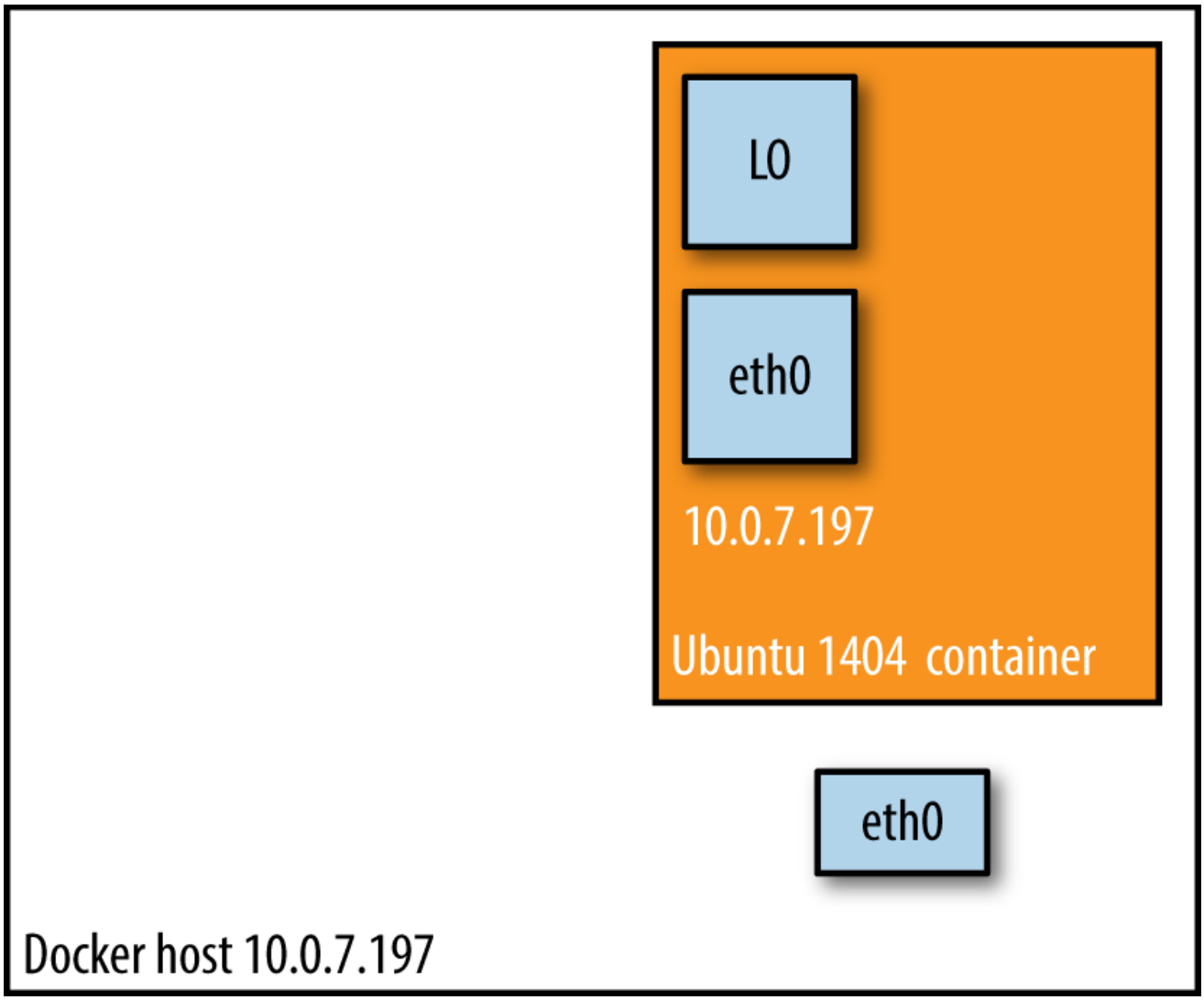
## 5.2、docker的网络类型

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

```
#创建容器
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

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

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

#启动容器
docker start redis-node01 redis-node02 redis-node03

#进入redis-node01容器进行操作
docker exec -it redis-node01 /bin/bash
#172.16.55.185是主机的ip地址
```

```
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：

```
root@itcast:/data# redis-cli
127.0.0.1:6379> CLUSTER NODES
46e5582cd2d96a506955cc08e7b08343037c91d9 172.16.55.185:6380@16380 master - 0
1543766975796 2 connected 5461-10922
b42d6ccc544094f1d8f35fa7a6d08b0962a6ac4a 172.16.55.185:6381@16381 master - 0
1543766974789 3 connected 10923-16383
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

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.6</version>
</dependency>
```

### 5.4.2、编写配置文件

```
# redis集群配置
spring.redis.jedis.pool.max-wait = 5000
spring.redis.jedis.pool.max-Idle = 100
spring.redis.jedis.pool.min-Idle = 10
spring.redis.timeout = 10
spring.redis.cluster.nodes = 172.16.55.185:6379,172.16.55.185:6380,172.16.55.185:6381
spring.redis.cluster.max-redirects=5
```

### 5.4.3、编写配置类

ClusterConfigurationProperties：

```
package cn.itcast.haoke.dubbo.api.config;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
@ConfigurationProperties(prefix = "spring.redis.cluster")
public class ClusterConfigurationProperties {


    private List<String> nodes;

    private Integer maxRedirects;

    public List<String> getNodes() {
        return nodes;
    }

    public void setNodes(List<String> nodes) {
```

```
        this.nodes = nodes;
    }

    public Integer getMaxRedirects() {
        return maxRedirects;
    }

    public void setMaxRedirects(Integer maxRedirects) {
        this.maxRedirects = maxRedirects;
    }
}
```

### 5.4.4、注册Redis连接工厂

RedisClusterConfig：

```
package cn.itcast.haoke.dubbo.api.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisClusterConfiguration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
public class RedisClusterConfig {

    @Autowired
    private ClusterConfigurationProperties clusterProperties;

    @Bean
    public RedisConnectionFactory connectionFactory() {
        RedisClusterConfiguration configuration = new
RedisClusterConfiguration(clusterProperties.getNodes());
        configuration.setMaxRedirects(clusterProperties.getMaxRedirects());
        return new JedisConnectionFactory(configuration);
    }

    @Bean
    public RedisTemplate<String, String> redisTemplate(RedisConnectionFactory
redisConnectionfactory) {
        RedisTemplate<String, String> redisTemplate = new RedisTemplate<>();
        redisTemplate.setConnectionFactory(redisConnectionfactory);
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        redisTemplate.setValueSerializer(new StringRedisSerializer());
        redisTemplate.afterPropertiesSet();
        return redisTemplate;
    }
}
```

### 5.4.5、编写测试用例

```
package cn.itcast.haoke.dubbo.api;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.Set;

@RunWith(SpringRunner.class)
@SpringBootTest
public class TestRedis {

    @Autowired
    private RedisTemplate<String,String> redisTemplate;

    @Test
    public void testSave(){
        for (int i = 0; i < 100; i++) {
            this.redisTemplate.opsForValue().set("key_" + i, "value_"+i);
        }

        Set<String> keys = this.redisTemplate.keys("key_*");
        for (String key : keys) {
            String value = this.redisTemplate.opsForValue().get(key);
            System.out.println(value);

            this.redisTemplate.delete(key);
        }

    }
}

```

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。

```java
package cn.itcast.haoke.dubbo.api.interceptor;

import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.commons.codec.digest.DigestUtils;
import org.apache.commons.io.IOUtils;
import org.apache.commons.lang3.StringUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Map;

@Component
public class RedisCacheInterceptor implements HandlerInterceptor {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    private static ObjectMapper mapper = new ObjectMapper();

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        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)) {
            // 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) throws Exception {
        String paramStr = request.getRequestURI();

        Map<String, String[]> parameterMap = request.getParameterMap();
        if (parameterMap.isEmpty()) {
            paramStr += IOUtils.toString(request.getInputStream(), "UTF-8");
```

```
        } else {
            paramStr += mapper.writeValueAsString(request.getParameterMap());
        }
        String redisKey = "WEB_DATA_" + DigestUtils.md5Hex(paramStr);

        return redisKey;
    }
}

```

注册拦截器到Spring容器：

```
package cn.itcast.haoke.dubbo.api.config;

import cn.itcast.haoke.dubbo.api.interceptor.RedisCacheInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Autowired
    private RedisCacheInterceptor redisCacheInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(this.redisCacheInterceptor).addPathPatterns("/**");
    }
}

```

### 5.5.2、测试拦截器

发起请求：

POST http://127.0.0.1:18080/graphql

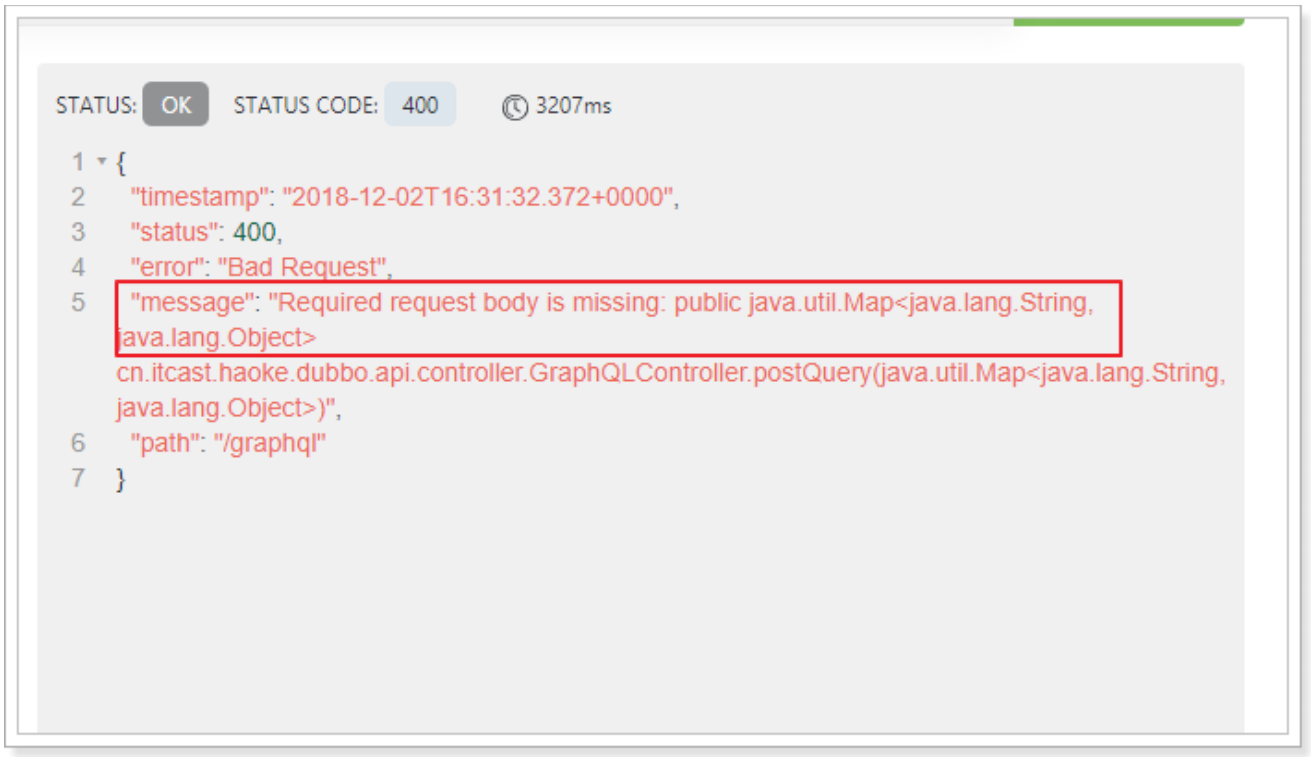
```
query HouseResourcesList($pageSize: Int, $page: Int) {
  HouseResourcesList(pageSize: $pageSize, page: $page) {
    list {
      id
      pic
      title
      coveredArea
      orientation
      floor
      rent
    }
  }
}
```

VARIABLES

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

```
@Override
public boolean preHandle(HttpServletRequest request, HttpServletResp
    if (!StringUtils.equalsIgnoreCase(request.getMethod(), "get")) {
        // 非get请求，如果不是graphql请求，放行
        if (!StringUtils.equalsIgnoreCase(request.getRequestURI(), "
            return true;
        }
    }

    String data = this.redisTemplate.opsForValue().get(createRedisKe
    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：




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：

```
package cn.itcast.haoke.dubbo.api.interceptor;

import org.apache.commons.io.IOUtils;

import javax.servlet.ReadListener;
import javax.servlet.ServletInputStream;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletRequestWrapper;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * 包装HttpServletRequest
 */
public class MyServletRequestWrapper extends HttpServletRequestWrapper {

    private final byte[] body;

    /**
     * Construct a wrapper for the specified request.
     *
     * @param request The request to be wrapped
```

```
     */
    public MyServletRequestWrapper(HttpServletRequest request) throws IOException
{
        super(request);
        body = IOUtils.toByteArray(super.getInputStream());
    }

    @Override
    public BufferedReader getReader() throws IOException {
        return new BufferedReader(new InputStreamReader(getInputStream()));
    }

    @Override
    public ServletInputStream getInputStream() throws IOException {
        return new RequestBodyCachingInputStream(body);
    }

    private class RequestBodyCachingInputStream extends ServletInputStream {
        private byte[] body;
        private int lastIndexRetrieved = -1;
        private ReadListener listener;

        public RequestBodyCachingInputStream(byte[] body) {
            this.body = body;
        }

        @Override
        public int read() throws IOException {
            if (isFinished()) {
                return -1;
            }
            int i = body[lastIndexRetrieved + 1];
            lastIndexRetrieved++;
            if (isFinished() && listener != null) {
                try {
                    listener.onAllDataRead();
                } catch (IOException e) {
                    listener.onError(e);
                    throw e;
                }
            }
            return i;
        }

        @Override
        public boolean isFinished() {
            return lastIndexRetrieved == body.length - 1;
        }

        @Override
        public boolean isReady() {
            // This implementation will never block
```

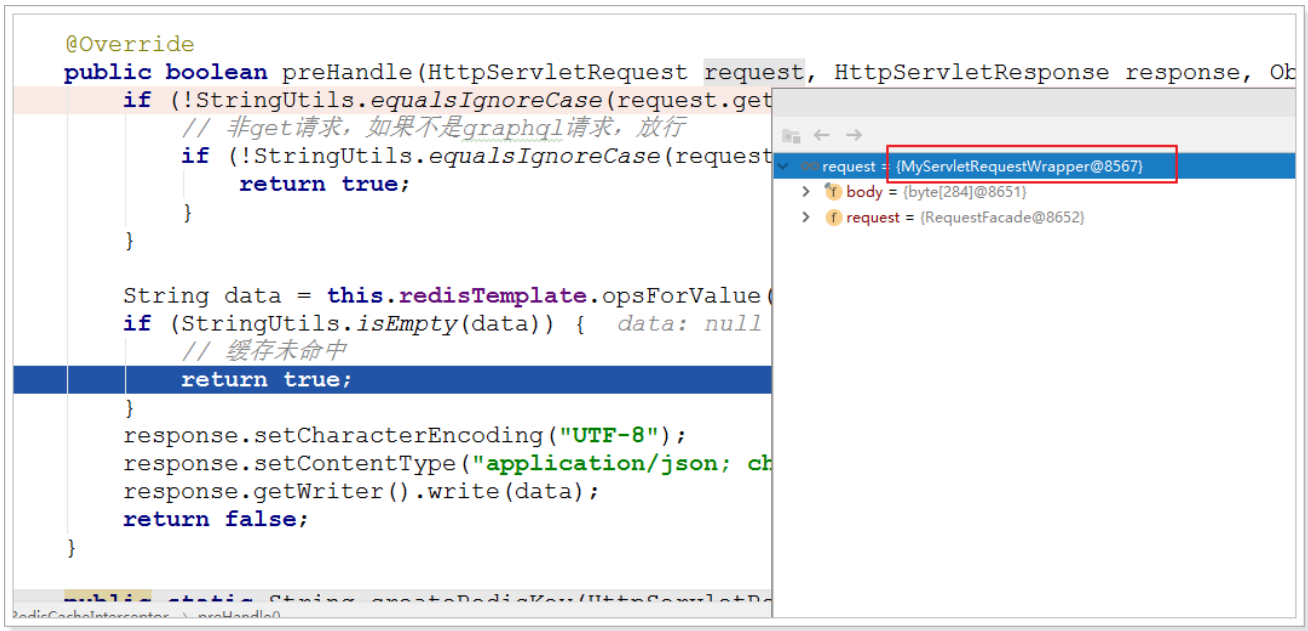
```
            // We also never need to call the readListener from this method, as
this method will never return false
            return isFinished();
        }

        @Override
        public void setReadListener(ReadListener listener) {
            if (listener == null) {
                throw new IllegalArgumentException("listener cann not be null");
            }
            if (this.listener != null) {
                throw new IllegalArgumentException("listener has been set");
            }
            this.listener = listener;
            if (!isFinished()) {
                try {
                    listener.onAllDataRead();
                } catch (IOException e) {
                    listener.onError(e);
                }
            } else {
                try {
                    listener.onAllDataRead();
                } catch (IOException e) {
                    listener.onError(e);
                }
            }
        }

        @Override
        public int available() throws IOException {
            return body.length - lastIndexRetrieved - 1;
        }

        @Override
        public void close() throws IOException {
            lastIndexRetrieved = body.length - 1;
            body = null;
        }
    }
}
```

通过过滤器进行包装request对象：

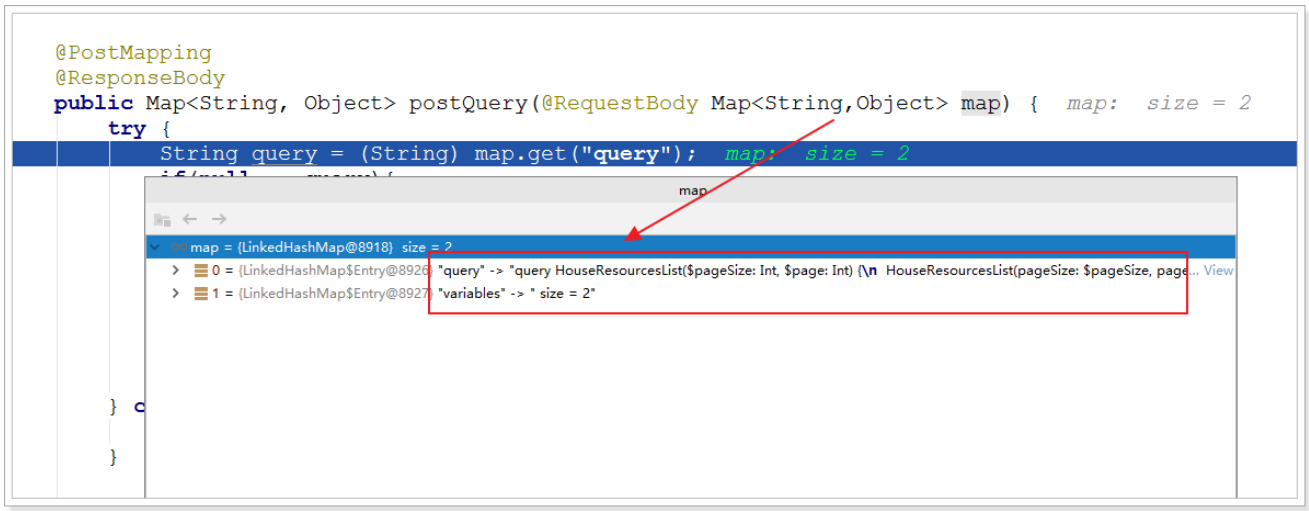
```
package cn.itcast.haoke.dubbo.api.interceptor;

import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
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
    response, FilterChain filterChain) throws ServletException, IOException {
20          if (!(request instanceof MyServletRequestWrapper)) {
21              request = new MyServletRequestWrapper(request);
22          }
23          filterChain.doFilter(request, response);
24      }
25  }
```

### 5.5.4、测试




可以看到，request对象已经经过了包装。



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。

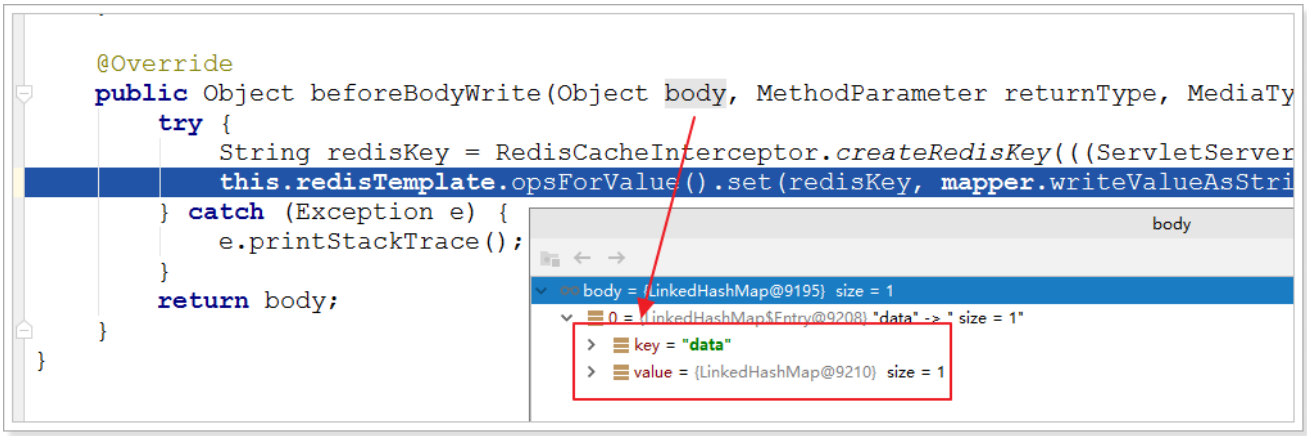
ResponseBodyAdvice是Spring提供的高级用法，会在结果被处理前进行拦截，拦截的逻辑自己实现，这样就可以实现拿到结果数据进行写入缓存的操作了。

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(),
returnType.getExecutable().getDeclaringClass().getName())) {
36             return true;
37         }
38
```

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

测试：




数据已经写入到redis：


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@8568
    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：

```
@Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        if(StringUtils.equalsIgnoreCase(request.getMethod(), "OPTIONS")){
            return true;
        }

        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)) {
            // 缓存未命中
            return true;
        }
        response.setCharacterEncoding("UTF-8");
        response.setContentType("application/json; charset=utf-8");

        // 支持跨域
        response.setHeader("Access-Control-Allow-Origin", "*");
        response.setHeader("Access-Control-Allow-Methods", "GET,POST,PUT,DELETE,OPTIONS");
        response.setHeader("Access-Control-Allow-Credentials", "true");
        response.setHeader("Access-Control-Allow-Headers", "Content-Type,X-Token");
        response.setHeader("Access-Control-Allow-Credentials", "true");
        response.getWriter().write(data);
        return false;
    }
```

# 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？

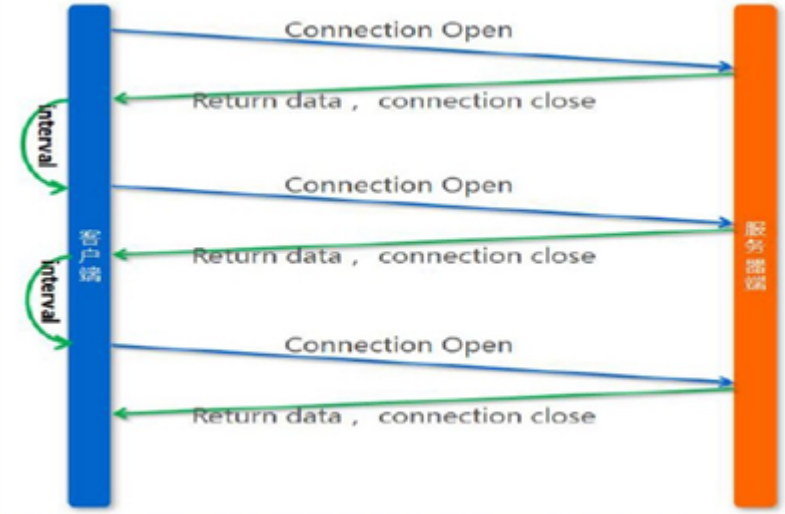
WebSocket 是HTML5一种新的协议。它实现了浏览器与服务器全双工通信(full-duplex)。一开始的握手需要借助HTTP请求完成。 WebSocket是真正实现了全双工通信的服务器向客户端推的互联网技术。 它是一种在单个TCP连接上进行全双工通讯协议。Websocket通信协议与2011年倍IETF定为标准RFC 6455，Websocket API被W3C定为标准。

> 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→B且B→A）。指A→B的同时B→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，当目前有两辆车对开，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

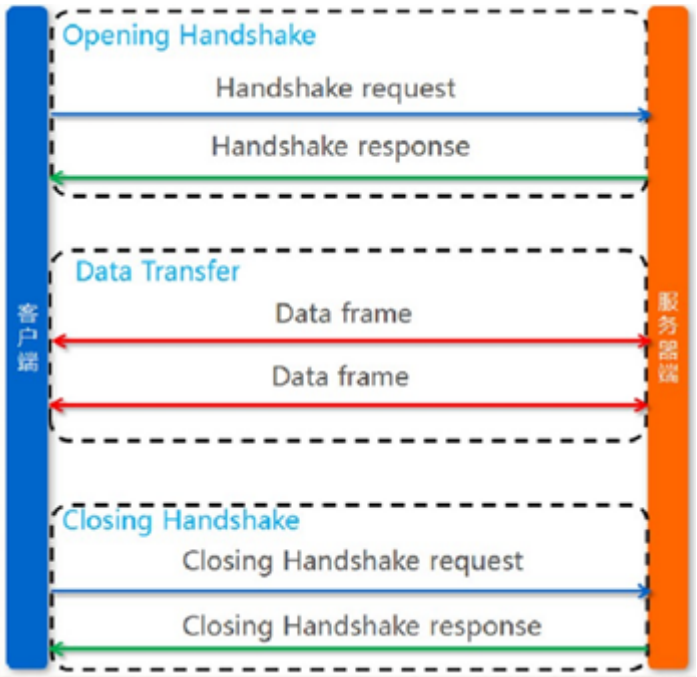
## 6.2.1、http

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。



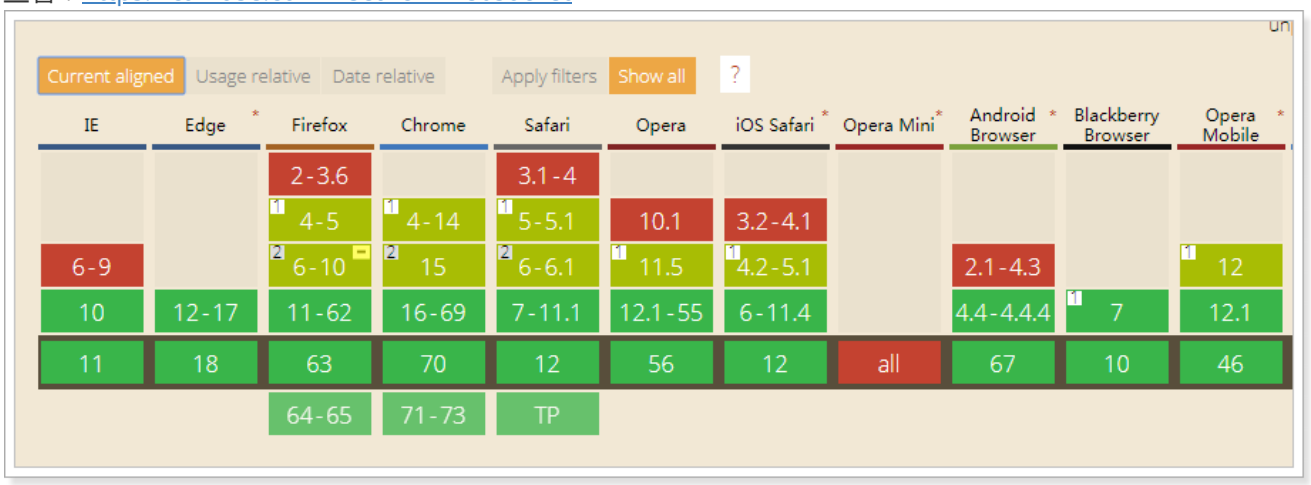

### 6.2.2、websocket

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




> 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

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>cn.itcast.websocket</groupId>
    <artifactId>itcast-websocket</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>7.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- java编译插件 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.2</version>
                <configuration>
                    <source>1.8</source>
```

```
                    <target>1.8</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
            <!-- 配置Tomcat插件 -->
            <plugin>
                <groupId>org.apache.tomcat.maven</groupId>
                <artifactId>tomcat7-maven-plugin</artifactId>
                <version>2.2</version>
                <configuration>
                    <port>8082</port>
                    <path>/</path>
                </configuration>
            </plugin>
        </plugins>
    </build>

</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

```
package cn.itcast.websocket;

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

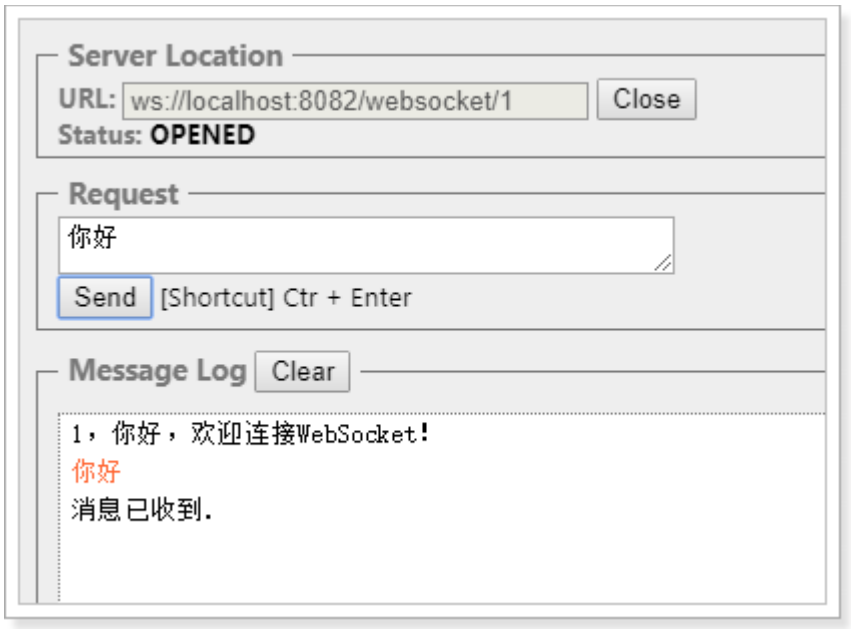
编写完成后，无需进额外的配置，直接启动tomcat即可。

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



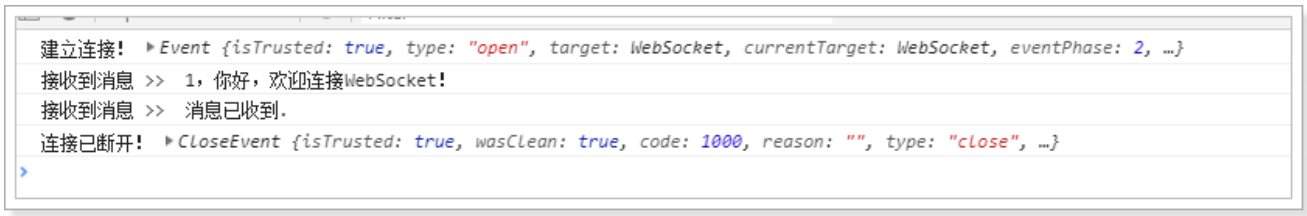

在线工具：https://easyswoole.com/wstool.html


## 6.4.4、编写js客户端

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<script>

    const socket = new WebSocket("ws://localhost:8082/websocket/1");
    socket.onopen = (ws) =>{
        console.log("建立连接！", ws);
    }
    socket.onmessage = (ws) =>{
        console.log("接收到消息 >> ",ws.data);
    }
```

```
    socket.onclose = (ws) =>{
        console.log("连接已断开！", ws);
    }
    socket.onerror = (ws) => {
        console.log("发送错误！", ws);
    }

    // 2秒后向服务端发送消息
    setTimeout(()=>{
        socket.send("发送一条消息试试");
    },2000);

    // 5秒后断开连接
    setTimeout(()=>{
        socket.close();
    },5000);

</script>
</body>
</html>
```

测试：




## 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

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!--spring boot的支持-->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.0.RELEASE</version>
    </parent>

    <groupId>cn.itcast.websocket</groupId>
    <artifactId>itcast-websocket</artifactId>
    <version>1.0-SNAPSHOT</version>
    <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
```

```
import java.io.IOException;

public class MyHandler extends TextWebSocketHandler {

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message)
throws IOException {
        System.out.println("获取到消息 >> " + message.getPayload());

        session.sendMessage(new TextMessage("消息已收到"));

        if(message.getPayload().equals("10")){
            for (int i = 0; i < 10; i++) {
                session.sendMessage(new TextMessage("消息 -> " + i));
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @Override
    public void afterConnectionEstablished(WebSocketSession session) throws
Exception {
        session.sendMessage(new TextMessage("欢迎连接到ws服务"));
    }

    @Override
    public void afterConnectionClosed(WebSocketSession session, CloseStatus status)
throws Exception {
        System.out.println("断开连接！");
    }

}
```

### 6.5.3、编写配置类

```
package cn.itcast.websocket.spring;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
```

```
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(), "/ws").setAllowedOrigins("*");
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

### 6.5.4、编写启动类

```
package cn.itcast.websocket;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}

```

### 6.5.5、测试


## 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：

```
package cn.itcast.websocket.spring;

import org.springframework.http.server.ServerHttpRequest;
import org.springframework.http.server.ServerHttpResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.server.HandshakeInterceptor;

import java.util.Map;

@Component
public class MyHandshakeInterceptor implements HandshakeInterceptor {

    /**
     * 握手之前，若返回false，则不建立链接
     *
     * @param request
     * @param response
     * @param wsHandler
     * @param attributes
     * @return
     * @throws Exception
     */
    @Override
    public boolean beforeHandshake(ServerHttpRequest request, ServerHttpResponse response, WebSocketHandler wsHandler, Map<String, Object> attributes) throws Exception {
        //将用户id放入socket处理器的会话(WebSocketSession)中
        attributes.put("uid", 1001);
        System.out.println("开始握手。。。。。。");
        return true;
    }

    @Override
    public void afterHandshake(ServerHttpRequest request, ServerHttpResponse response, WebSocketHandler wsHandler, Exception exception) {
        System.out.println("握手成功啦。。。。。");
    }
}

```

将拦截器添加到websocket服务中：

```
package cn.itcast.websocket.spring;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.WebSocketHandler;
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
   .setAllowedOrigins("*").addInterceptors(this.myHandshakeInterceptor);
22     }
23
24     @Bean
25     public WebSocketHandler myHandler() {
26         return new MyHandler();
27     }
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：