# SpringBoot

## Mybatis-plus

### 网站介绍

[MyBatis-Plus](https://github.com/baomidou/mybatis-plus" \t "_blank)（简称 MP）是一个 [MyBatis](http://www.mybatis.org/mybatis-3/" \t "_blank) 的增强工具，在 MyBatis 的基础上只做增强不做改变，为简化开发、提高效率而生。

## 入门案例

### 导入jar包

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus-boot-starter</artifactId>

<version>3.0.6</version>

</dependency>

### 编辑yml配置文件

server:

port: 8091

servlet:

context-path: /

spring:

datasource:

#引入druid数据源

type: com.alibaba.druid.pool.DruidDataSource

driver-class-name: com.mysql.jdbc.Driver

url: jdbc:mysql://127.0.0.1:3306/jtdb?serverTimezone=GMT%2B8&useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true

username: root

password: root

#mybatis-plush配置

mybatis-plus:

type-aliases-package: com.jt.pojo

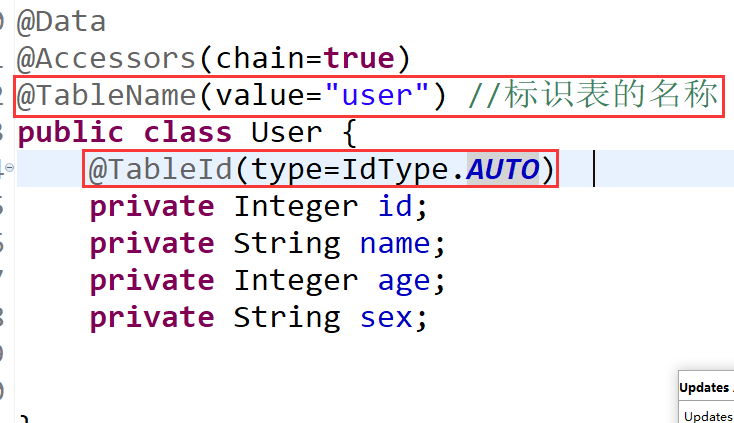
mapper-locations: classpath:/mybatis/mappers/\*.xml

configuration:

map-underscore-to-camel-case: true #自动开启驼峰规则映射

### 对象关系映射

1. 对象与数据库表的映射



1. 继承公共接口

**public** **interface** UserMapper **extends** BaseMapper<User>{

List<User> findAll();

}

### 数据库操作入门

@RunWith(SpringRunner.**class**)

@SpringBootTest

**public** **class** TestMybatis {

//该测试类,相当于直接获取了spring容器对象

@Autowired

**private** UserMapper userMapper;

@Test

**public** **void** testFind() {

List<User> userList = userMapper.findAll();

System.***out***.println(userList);

}

//实现查询操作 查询age=3000岁

//要求0<age<150 大于 gt 小于 lt

@Test

**public** **void** testFind2() {

//User user = new User().setAge(3000);

QueryWrapper<User> queryWrapper =

**new** QueryWrapper<>();

queryWrapper.gt("age", 0).lt("age", 150);

//queryWrapper.eq("age", 3000);

List<User> userList =

userMapper.selectList(queryWrapper);

System.***out***.println(userList);

}

//新增入库

@Test

**public** **void** testInster() {

User user = **new** User().setName("流浪地球").setAge(1).setSex("男");

**int** rows = userMapper.insert(user);

System.***out***.println("入库成功!影响了"+rows+"行");

}

//删除

@Test

**public** **void** testDel() {

QueryWrapper<User> queryWrapper =

**new** QueryWrapper<User>();

queryWrapper.eq("name","马化腾");

userMapper.delete(queryWrapper);

System.***out***.println("删除数据成功!!");

}

/\*\*

\* 修改操作

\* 参数介绍:

\* sql: update 表名 set 字段名=值 where id =100

\* 1.entity 修改后的值使用entity封装

\* 2.updateWrapper 定义where条件的值

\*案例:

\* 要求: 将id=172的数据,修改为汤圆/元宵 年龄20/ 性别女

\*/

@Test

**public** **void** testUpdate() {

User user = **new** User().setName("汤圆")

.setAge(20)

.setSex("女");

UpdateWrapper<User> updateWrapper =

**new** UpdateWrapper<>();

updateWrapper.eq("id", 172);

userMapper.update(user, updateWrapper);

System.***out***.println("更新数据成功!!");

}

}

## SpringBoot整合redis集群

### 配置类

说明:旧框架使,需要编辑很多的配置文件.新的框架一般都在采用配置类的形式管理项目.

@Configuration

**public** **class** UserConfig {

//通过spring容器管理User对象

@Bean

**public** User getUser() {

**return** **new** User().setId(1000)

.setName("配置类")

.setAge(100)

.setSex("男");

}

}

### 导入jar包

<!--spring整合redis -->

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-redis</artifactId>

</dependency>

### 编辑pro文件

redis.nodes=192.168.126.174:7000,192.168.126.174:7001,192.168.126.174:7002,192.168.126.174:7003,192.168.126.174:7004,192.168.126.174:7005,192.168.126.174:7006,192.168.126.174:7007,192.168.126.174:7008

### 编辑配置类整合redis集群

@Configuration

//引入指定的配置文件redis.properties

@PropertySource("classpath:/properties/redis.properties")

**public** **class** RedisClusterConfig {

@Value("${redis.nodes}")

**private** String redisNodes;

@Bean

**public** JedisCluster getJedisCluster() {

Set<HostAndPort> nodes = **new** HashSet<>();

String[] hostAndPorts = redisNodes.split(",");

**for** (String node : hostAndPorts) {

String[] args = node.split(":");

String host = args[0];

**int** port = Integer.*parseInt*(args[1]);

nodes.add(**new** HostAndPort(host, port));

}

**return** **new** JedisCluster(nodes);

}

}

### 编辑测试类

//测试redis集群是否正确

@Autowired

**private** JedisCluster jedisCluster;

@Test

**public** **void** testRedis() {

jedisCluster.set("1810","springboot整合redis集群成功");

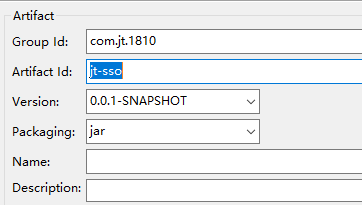
System.***out***.println(jedisCluster.get("1810"));

}

# 实现JT-SSO项目创建

## 搭建项目

### 创建项目



### 引入pom.xml文件

<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>com.jt.1810</groupId>

<artifactId>jt-sso</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.3.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--springBoot动态的引入springMVC全部的配置 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!--引入测试类 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<!--添加属性注入依赖 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

<!--支持热部署 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

<version>1.2.8.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<!--引入插件lombok 自动的set/get/构造方法插件 -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</dependency>

<!--引入数据库驱动 -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

<!--引入druid数据源 -->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.1.12</version>

</dependency>

<!--spring整合mybatis-plus -->

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus-boot-starter</artifactId>

<version>3.0.6</version>

</dependency>

<!--spring整合redis -->

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-redis</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<!--引入springBoot项目启动项 内置tomcat -->

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

### 编辑yml文件

server:

port: 8093

servlet:

context-path: /

spring:

datasource:

#引入druid数据源

type: com.alibaba.druid.pool.DruidDataSource

driver-class-name: com.mysql.jdbc.Driver

url: jdbc:mysql://127.0.0.1:3306/jtdb?serverTimezone=GMT%2B8&useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true

username: root

password: root

#mybatis-plush配置

mybatis-plus:

type-aliases-package: com.jt.sso.pojo

mapper-locations: classpath:/mybatis/mappers/\*.xml

configuration:

map-underscore-to-camel-case: true #自动开启驼峰规则映射

### 引入redis集群配置

@Configuration

//引入指定的配置文件redis.properties

@PropertySource("classpath:/properties/redis.properties")

**public** **class** RedisClusterConfig {

@Value("${redis.nodes}")

**private** String redisNodes;

@Bean

**public** JedisCluster getJedisCluster() {

Set<HostAndPort> nodes = **new** HashSet<>();

String[] hostAndPorts = redisNodes.split(",");

**for** (String node : hostAndPorts) {

String[] args = node.split(":");

String host = args[0];

**int** port = Integer.*parseInt*(args[1]);

nodes.add(**new** HostAndPort(host, port));

}

**return** **new** JedisCluster(nodes);

}

}

之后的配置按照下发代码自行完成

### 配置nginx实现反向代理

要求:用户通过sso.jt.com访问单点登录项目,编辑完成将nginx重启

#编辑单点登录服务器

server {

listen 80;

server\_name sso.jt.com;

location / {

#反向代理到url

proxy\_pass http://localhost:8093;

}

}

## 实现页面跳转

### 页面分析

登录: /user/login.html

注册: /user/register.html

### 跳转实现

@Controller

@RequestMapping("/user")

**public** **class** UserController {

//实现用户页面通用跳转

@RequestMapping("/{moduleName}")

**public** String index(@PathVariable String moduleName) {

**return** moduleName;

}

}

## JSONP实现原理

### 跨域访问测试

1. 在jt-manage中测试js

<script type="text/javascript" src="http://manage.jt.com/js/jquery-easyui-1.4.1/jquery.min.js"></script>

<script type="text/javascript">

$(**function**(){

$.get("http://manage.jt.com/test.json",**function**(data){

alert(data.name);

})

})

</script>

1. jt-manage后台测试



1. 用户前台访问

<script type="text/javascript" src="http://manage.jt.com/js/jquery-easyui-1.4.1/jquery.min.js"></script>

<script type="text/javascript">

//页面加载完成

$(**function**(){

alert("测试开始!!!!!!!!!!")

$.get("http://manage.jt.com/test.json",**function**(data){

alert(data.name);

})

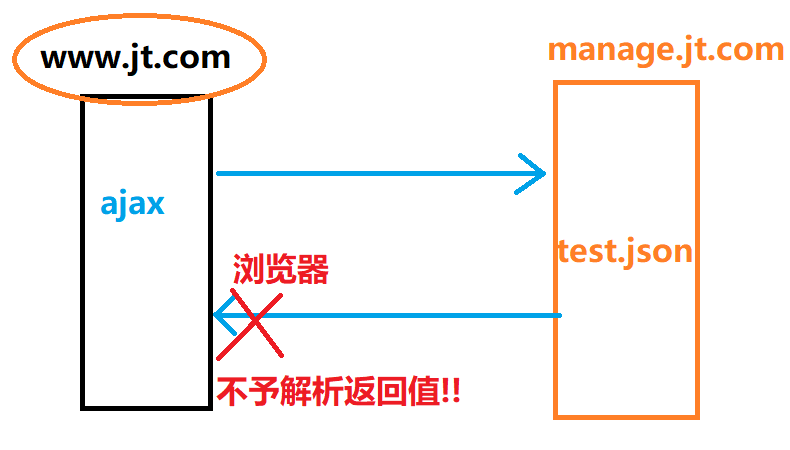
})

</script>

1. 访问结果

Ajax请求正确发送,并且浏览器已经成功接收返回值数据.但是数据没有显示.问:为什么???

### 浏览器不允许跨域访问



### 同源策略

核心: **访问协议://域名:端口**

**只有这三项都相同表示同源.同源浏览器可以正确解析.**



问题:

Localhost:8091/index

Manage.jt.com/index 改访问时跨域请求…

跨域的说明:

要素:

1. 浏览器发起的AJAX请求
2. 发起的是跨域的请求.

### 实现跨域底层原理

1. 利用javaScript中src属性可以实现跨域的访问形式

<script type="text/javascript" src="http://manage.jt.com/test.json"></script>

1. 定义回调函数

//2.定义回调函数

**function** hello(data){

alert(data.name);

}

1. 定义返回值

回调函数名称(JSON)

hello({"id":*"1"*,"name":*"tom"*})

### 跨域优化

1. 回调函数的名称应该是动态获取的.

url: <http://manage.jt.com?callback=hello>

2.发起跨域请求的方式,比较的负载,最好的方式是通过ajax直接发请求.

<script type="text/javascript">

//2.定义回调函数

**function** hello(data){

alert(data.name);

}

</script>

<!--1.利用src实现跨域 -->

<script type="text/javascript" src="http://manage.jt.com/test.json"></script>

## JSONP实现

### JSONP介绍

JSONP(JSON with Padding)是[JSON](https://baike.baidu.com/item/JSON" \t "_blank)的一种“使用模式”，可用于解决**主流浏览器的跨域数据访问的问题**。由于同源策略，一般来说位于 server1.example.com 的网页无法与不是 server1.example.com的服务器沟通，而 HTML 的<script> 元素是一个例外。利用 <script> 元素的这个**开放策略**，网页可以得到从其他来源动态产生的 JSON 资料，而这种使用模式就是所谓的 JSONP。用 JSONP 抓到的资料并不是 JSON，而是任意的JavaScript，用 JavaScript 直译器执行而不是用 JSON 解析器解析。

### jQuery实现JSONP请求

1. 编辑AJAX请求

<script type="text/javascript">

$(**function**(){

alert("测试访问开始!!!!!")

$.ajax({

url:"http://manage.jt.com/web/testJSONP",

type:"get", //只能发起get请求

dataType:"jsonp", //返回数据的类型

//jsonp: "callback", //指定参数名称

//jsonpCallback: "hello", //指定回调函数名称

success:**function** (data){

alert(data.id);

//转化为字符串使用

//var obj = eval("("+data+")");

//alert(obj.name);

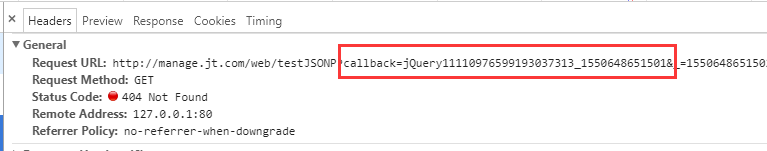
}

});

})

</script>

1. 请求路径



### 接收JSONP请求

@RestController

**public** **class** JSONPController {

/\*\*

\* url:

\* http://manage.jt.com/web/testJSONP?callback=jQuery11110976599193037313\_1550648651501&\_=1550648651502

\* 参数定义:

\* 1.返回值写法

\* 对象代替:JSONPObject

\* 函数名称(JSON)

\*

\*/

@RequestMapping("/web/testJSONP")

**public** JSONPObject jsonp(String callback) {

User user = **new** User();

user.setId(100);

user.setName("tomcat猫!!!!");

JSONPObject object =

**new** JSONPObject(callback, user);

**return** object;

}

}

## 实现用户校验

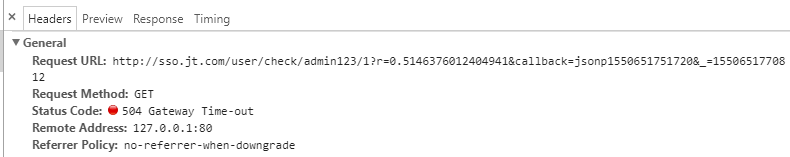
### 用户校验原理

1. 当用户输入用户名时,当鼠标离焦时发起ajax请求.
2. 应该通过ajax请求,最好 直接访问sso.jt.com
3. Sso.jt.com.接收用户请求后,进行后台的业务处理.返回json数据
4. 页面通过ajax请求获取数据后,解析返回值结果,如果用户名已经存在则给用户提示,如果用户名不存在则友好提示即可.

### 页面分析

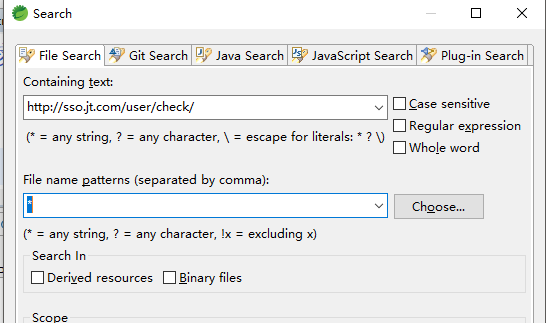
通过分析:该访问是一个跨域访问.使用JSONP实现

1.url分析



2.ajax分析

页面搜索:



$.ajax({

url : "http://sso.jt.com/user/check/"+escape(pin)+"/1?r=" + Math.random(),

dataType : "jsonp",

success : **function**(data) {

checkpin = data.data?"1":"0";

**if** (!data.data) {

validateSettings.succeed.run(option);

namestate = **true**;

}**else** {

validateSettings.error.run(option, "该用户名已占用！");

namestate = **false**;

}

}

});

### 业务接口规范

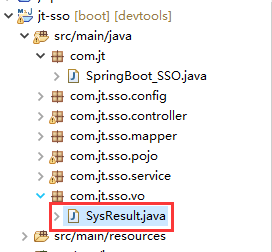
说明:该需求文档,是jt-web调用jt-sso时使用的

现在:通过JSONP的形式进行调用

|  |  |
| --- | --- |
| **请求方法** | **GET** |
| **URL** | http://sso.jt.com/user/check/{param}/{type} |
| **参数** | 格式如：chenchen/1  其中chenchen是校验的数据  Type为类型，可选参数1 username、2 phone、3 email |
| **示例** | http://sso.jt.com/user/check/chenchen/1 |
| **返回值** | {  status: 200 //200 成功，201 没有查到  msg: “OK” //返回信息消息  data: false //返回数据true用户已存在，false用户不存在，可以  } |
|  |  |

### 业务准备

封装一个vo对象



### 编辑SSO的Controller

@RestController

@RequestMapping("/user")

**public** **class** UserController {

@Autowired

**private** UserService userService;

/\*\*

\* 1.实现用户信息的校验

\* url:http://sso.jt.com/user/check/asdfasdfasdf/1?r=0.5283910956938842&callback=jsonp1550651751721&\_=1550653365532

\*/

@RequestMapping("/check/{param}/{type}")

**public** JSONPObject findCheckUser(

String callback,

@PathVariable String param,

@PathVariable Integer type) {

//获取数据库返回值结果

**boolean** flag =

userService.findCheckUser(param,type);

//封装返回值数据

SysResult result = SysResult.*oK*(flag);

**return** **new** JSONPObject(callback, result);

}

}

### 编辑SSO的Service

@Service

**public** **class** UserServiceImpl **implements** UserService {

@Autowired

**private** UserMapper userMapper;

/\*\*

\* 思路:根据用户传递的数据表和类型查询数据库

\* select count(\*) from 表名 where 字段名 = 值

\*/

@Override

**public** **boolean** findCheckUser(String param, Integer type) {

String column = **null**;

**switch** (type) {

**case** 1:

column = "username";

**break**;

**case** 2:

column = "phone";

**break**;

**case** 3:

column = "email";

**break**;

}

QueryWrapper<User> queryWrapper =

**new** QueryWrapper<>();

queryWrapper.eq(column, param);

**int** count = userMapper.selectCount(queryWrapper);

//count 0返回false !0返回true

**return** count==0?**false**:**true**;

}

}

### 页面效果

