# 淘商城案例

## 课程计划

共12天的内容

第一天：了解电商行业、了解淘淘商城。后台工程搭建。框架整合ssm。

第二天：商品管理🡪商品列表展示🡪商品添加🡪商品类目选择。

第三天：商品添加：上传图片。Nginx、FastDFS。

第四天：商品添加实现：父文本编辑器的使用、商品的规格参数实现。

第五天：前台系统搭建。展示首页、商品的类目选择。

第六天、第七天：首页大广告位的展示。CMS(内容定制)系统实现。Redis缓存，Redis缓存(集群)，首页大广告位展示。

第八天：搜索功能的实现。实现solr实现(solr集群)

第九天：商品详情页面实现，网页静态化freemaker实现

第十天：单点登录系统实现，session共享。

第十一天：购物车、订单系统。

第十二天：Quartz任务调度框架。项目部署、项目总结、面试中的问题。

第一天的课程内容：

1. 了解电商行业
2. 项目介绍
   1. 功能介绍
   2. 架构介绍
3. 后台工程搭建
   1. 使用maven搭建。
   2. 搭建聚合工程
4. 代码上传到git
5. 整合ssm框架整合

# 了解电商行业

## 电商行业技术特点：

* 技术新
* 技术范围广
* 分布式
* 高并发、集群、负载均衡、高可用
* 海量数据
* 业务复杂
* 系统安全

## 淘淘商城介绍

淘淘网上商城是一个综合性的B2C平台，类似京东商城、天猫商城。会员可用在商城浏览商品、下订单，以及参加各种活动。

管理员、运营可用在平台后台管理商品、订单、会员等。

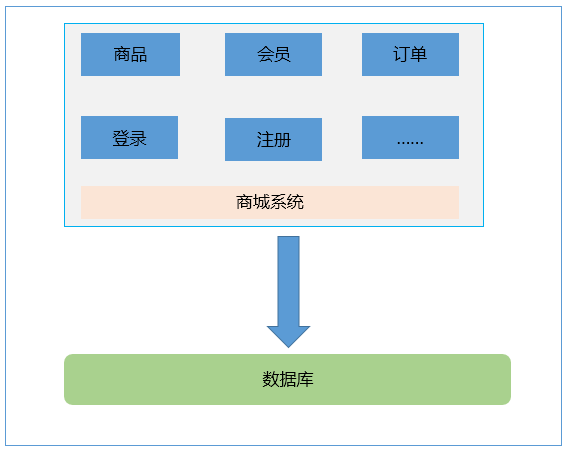
客服可用在后台管理系统中处理用户询问以及投诉。

## 功能架构



## 技术架构

### 传统架构



有什么问题

1. 块之间耦合度太高，其中一个升级其他都得升级
2. 开发困难，各个团队开发最后都要整合在一起
3. 系统的扩展性差
4. 不能灵活的进行分布式部署

### 分布式系统架构



分布式架构：

把系统按照模块拆分成多个子系统。

优点：

1. 把模块拆分，使用接口通信，降低莫魁岸之间的耦合度。
2. 把项目拆封城若干个子项目，不同的团队负责不同的子项目。
3. 增加功能时只需要再增加一个子项目，调用其他系统的接口就可以。
4. 可以灵活的进行分布式部署。

缺点：

系统之间交互需要使用远程通信，接口开发增加工作量。

## 技术选型

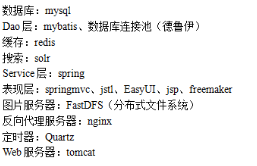
* Spring、SpringMVC、Mybatis
* 图片服务器：FastDFS(分布式文件系统)
* 数据库连接池(德鲁伊)
* freemaker
* JSP、JSTL、jQuery plugin、EasyUI、KindEditor(父文本编辑器)、CSS+DIV
* redis(缓存服务器)
* Solr(搜索)
* httpclient(调用系统服务)
* MySql
* Nginx(web服务器)
* 定时器：Quartz

## 开发环境

IDEA 2016.1.1

Maven3

Tomcat 8



冗余信息，保证高可用。

## 后台管理系统搭建

网站一般分为前台和后台。前台是给用户看的后台是管理人员使用。

使用maven管理工程。

1. 依赖管理
2. 项目构建

后台创建一个独立的web工程。创建一个maven工程，打包方式为war包。

可以创建一个聚合工程，每个模块都是一个jar包，可以被其他系统依赖。

创建

Taotao-parent(父工程管理jar包的版本)

|--Taotao-common(把通用的工具类打包)

|--Taotao-manager

|--Taotao-manager-pojo

|--Taotao-manager-dao

|--Taotao-manager-service

|--Taotao-manager-web(war包)

# taotao-parent工程

## 修改pom文件，加载对应jar包

|  |
| --- |
| <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.caxins.taotao</groupId>  <artifactId>taotao-parent</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>pom</packaging>  <modules>  <module>../taotao-common</module>  <module>../taotao-manager</module>  </modules>  <!-- 集中定义依赖版本号 -->  <properties>  <junit.version>4.12</junit.version>  <spring.version>4.1.3.RELEASE</spring.version>  <mybatis.version>3.2.8</mybatis.version>  <mybatis.spring.version>1.2.2</mybatis.spring.version>  <mybatis.paginator.version>1.2.15</mybatis.paginator.version>  <mysql.version>5.1.32</mysql.version>  <slf4j.version>1.6.4</slf4j.version>  <jackson.version>2.4.2</jackson.version>  <druid.version>1.0.9</druid.version>  <httpclient.version>4.3.5</httpclient.version>  <jstl.version>1.2</jstl.version>  <servlet-api.version>2.5</servlet-api.version>  <jsp-api.version>2.0</jsp-api.version>  <joda-time.version>2.5</joda-time.version>  <commons-lang3.version>3.3.2</commons-lang3.version>  <commons-io.version>1.3.2</commons-io.version>  <commons-net.version>3.3</commons-net.version>  <pagehelper.version>4.1.6</pagehelper.version>  <jsqlparser.version>0.9.6</jsqlparser.version>  <commons-fileupload.version>1.3.1</commons-fileupload.version>  <jedis.version>2.7.2</jedis.version>  <solrj.version>4.10.3</solrj.version>  <freemarker.version>2.3.23</freemarker.version>  <quartz.version>2.2.2</quartz.version>  </properties>  <dependencyManagement>  <dependencies>  <!-- 时间操作组件 -->  <dependency>  <groupId>joda-time</groupId>  <artifactId>joda-time</artifactId>  <version>${joda-time.version}</version>  </dependency>  <!-- Apache工具组件 -->  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-lang3</artifactId>  <version>${commons-lang3.version}</version>  </dependency>  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-io</artifactId>  <version>${commons-io.version}</version>  </dependency>  <dependency>  <groupId>commons-net</groupId>  <artifactId>commons-net</artifactId>  <version>${commons-net.version}</version>  </dependency>  <!-- Jackson Json处理工具包 -->  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>${jackson.version}</version>  </dependency>  <!-- httpclient -->  <dependency>  <groupId>org.apache.httpcomponents</groupId>  <artifactId>httpclient</artifactId>  <version>${httpclient.version}</version>  </dependency>  <!-- 单元测试 -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>${junit.version}</version>  <scope>test</scope>  </dependency>  <!-- 日志处理 -->  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>${slf4j.version}</version>  </dependency>  <!-- Mybatis -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>${mybatis.version}</version>  </dependency>  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis-spring</artifactId>  <version>${mybatis.spring.version}</version>  </dependency>  <dependency>  <groupId>com.github.miemiedev</groupId>  <artifactId>mybatis-paginator</artifactId>  <version>${mybatis.paginator.version}</version>  </dependency>  <dependency>  <groupId>com.github.pagehelper</groupId>  <artifactId>pagehelper</artifactId>  <version>${pagehelper.version}</version>  </dependency>  <!-- MySql -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>${mysql.version}</version>  </dependency>  <!-- 连接池 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>${druid.version}</version>  </dependency>  <!-- Spring -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- JSP相关 -->  <dependency>  <groupId>jstl</groupId>  <artifactId>jstl</artifactId>  <version>${jstl.version}</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>servlet-api</artifactId>  <version>${servlet-api.version}</version>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jsp-api</artifactId>  <version>${jsp-api.version}</version>  <scope>provided</scope>  </dependency>  <!-- 文件上传组件 -->  <dependency>  <groupId>commons-fileupload</groupId>  <artifactId>commons-fileupload</artifactId>  <version>${commons-fileupload.version}</version>  </dependency>  <!-- Redis客户端 -->  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>${jedis.version}</version>  </dependency>  <!-- solr客户端 -->  <dependency>  <groupId>org.apache.solr</groupId>  <artifactId>solr-solrj</artifactId>  <version>${solrj.version}</version>  </dependency>  <dependency>  <groupId>org.freemarker</groupId>  <artifactId>freemarker</artifactId>  <version>${freemarker.version}</version>  </dependency>  <dependency>  <groupId>org.quartz-scheduler</groupId>  <artifactId>quartz</artifactId>  <version>${quartz.version}</version>  </dependency>  </dependencies>  </dependencyManagement>   <build>  <finalName>${project.artifactId}</finalName>  <plugins>  <!-- 资源文件拷贝插件 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-resources-plugin</artifactId>  <version>2.7</version>  <configuration>  <encoding>UTF-8</encoding>  </configuration>  </plugin>  <!-- java编译插件 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.2</version>  <configuration>  <source>1.7</source>  <target>1.7</target>  <encoding>UTF-8</encoding>  </configuration>  </plugin>  </plugins>  <pluginManagement>  <plugins>  <!-- 配置Tomcat插件 -->  <plugin>  <groupId>org.apache.tomcat.maven</groupId>  <artifactId>tomcat7-maven-plugin</artifactId>  <version>2.2</version>  </plugin>  </plugins>  </pluginManagement>  </build> </project> |

## 创建taotao-common

保存的是用到的通用的工具类以及通用的pojo

### pom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>  <parent>  <groupId>com.caxins.taotao</groupId>  <artifactId>taotao-parent</artifactId>  <version>0.0.1-SNAPSHOT</version>  </parent>  <groupId>com.caxins.taotao</groupId>  <artifactId>taotao-common</artifactId>  <version>0.0.1-SNAPSHOT</version>  <!-- 添加依赖 -->  <dependencies>  <!-- 时间操作组件 -->  <dependency>  <groupId>joda-time</groupId>  <artifactId>joda-time</artifactId>  </dependency>  <!-- Apache工具组件 -->  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-lang3</artifactId>  </dependency>  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-io</artifactId>  </dependency>  <dependency>  <groupId>commons-net</groupId>  <artifactId>commons-net</artifactId>  </dependency>  <!-- Jackson Json处理工具包 -->  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  </dependency>  <!-- httpclient -->  <dependency>  <groupId>org.apache.httpcomponents</groupId>  <artifactId>httpclient</artifactId>  </dependency>  <!-- 单元测试 -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  </dependency>  <!-- 日志处理 -->  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  </dependency>  </dependencies> </project> |

## 创建taotao-manager

为聚合工程。打包方式是pom

### pom文件

|  |
| --- |
| <?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">  <parent>  <artifactId>taotao-parent</artifactId>  <groupId>com.caxins.taotao</groupId>  <version>0.0.1-SNAPSHOT</version>  <relativePath>../taotao-parent/pom.xml</relativePath>  </parent>  <modelVersion>4.0.0</modelVersion>  <artifactId>taotao-manager</artifactId>  <packaging>pom</packaging>  <modules>  <module>../taotao-pojo</module>  <module>../taotao-dao</module>  <module>../taotao-service</module>  <module>../taotao-web</module>  </modules>  <!-- 添加依赖 -->  <dependencies>  <dependency>  <groupId>com.caxins.taotao</groupId>  <artifactId>taotao-common</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  </dependencies>  <!-- 添加tomcat插件 -->  <build>  <plugins>  <plugin>  <groupId>org.apache.tomcat.maven</groupId>  <artifactId>tomcat7-maven-plugin</artifactId>  <configuration>  <port>8888</port>  <path>/</path>  </configuration>  </plugin>  </plugins>  </build> </project> |

### 添加模块taotao-pojo

#### pom文件

|  |
| --- |
| <?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">  <parent>  <artifactId>taotao-manager</artifactId>  <groupId>com.caxins.taotao</groupId>  <version>0.0.1-SNAPSHOT</version>  <relativePath>../taotao-manager/pom.xml</relativePath>  </parent>  <packaging>jar</packaging>  <modelVersion>4.0.0</modelVersion>  <artifactId>taotao-pojo</artifactId> </project> |

### 添加模块taotao-dao

#### pom文件

|  |
| --- |
| <?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">  <parent>  <artifactId>taotao-parent</artifactId>  <groupId>com.caxins.taotao</groupId>  <version>0.0.1-SNAPSHOT</version>  <relativePath>../taotao-manager/pom.xml</relativePath>  </parent>  <packaging>jar</packaging>  <modelVersion>4.0.0</modelVersion>  <artifactId>taotao-dao</artifactId>  <dependencies>  <dependency>  <groupId>com.caxins.taotao</groupId>  <artifactId>taotao-pojo</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <!-- Mybatis -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  </dependency>  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis-spring</artifactId>  </dependency>  <dependency>  <groupId>com.github.miemiedev</groupId>  <artifactId>mybatis-paginator</artifactId>  </dependency>  <dependency>  <groupId>com.github.pagehelper</groupId>  <artifactId>pagehelper</artifactId>  </dependency>  <!-- MySql -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <!-- 连接池 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  </dependency>  </dependencies> </project> |

### 添加模块taotao-service

#### pom文件

|  |
| --- |
| <?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">  <parent>  <artifactId>taotao-manager</artifactId>  <groupId>com.caxins.taotao</groupId>  <version>0.0.1-SNAPSHOT</version>  <relativePath>../taotao-manager/pom.xml</relativePath>  </parent>  <modelVersion>4.0.0</modelVersion>  <artifactId>taotao-service</artifactId>  <dependencies>  <dependency>  <groupId>com.caxins.taotao</groupId>  <artifactId>taotao-dao</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  </dependency>  </dependencies> </project> |

### 添加模块taotao-web

#### pom文件

|  |
| --- |
| <?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">  <parent>  <artifactId>taotao-manager</artifactId>  <groupId>com.caxins.taotao</groupId>  <version>0.0.1-SNAPSHOT</version>  <relativePath>../taotao-manager/pom.xml</relativePath>  </parent>  <modelVersion>4.0.0</modelVersion>  <artifactId>taotao-web</artifactId>  <packaging>war</packaging>  <!-- 添加依赖 -->  <dependencies>  <dependency>  <groupId>com.caxins.taotao</groupId>  <artifactId>taotao-service</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <!-- JSP相关 -->  <dependency>  <groupId>jstl</groupId>  <artifactId>jstl</artifactId>  <version>${jstl.version}</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>servlet-api</artifactId>  <version>${servlet-api.version}</version>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jsp-api</artifactId>  <version>${jsp-api.version}</version>  <scope>provided</scope>  </dependency>  <!-- 文件上传组件 -->  <dependency>  <groupId>commons-fileupload</groupId>  <artifactId>commons-fileupload</artifactId>  <version>${commons-fileupload.version}</version>  </dependency>  </dependencies> </project> |

# 整合SSM框架分析

## dao层

整合mybatis+spring

### 需要的jar包

1. mybatis的jar包
2. mysql数据库驱动
3. 数据库连接池
4. mybatis和spring的整合包
5. spring的jar包

### 配置文件

1. mybatis的配置文件：SqlMapConfig.xml
2. spring的配置文件：applicatrionContext-dao.xml
   1. 配置数据源
   2. 配置数据库连接池
   3. 配置SqlSessionFactory(需要使用mybatis整合包中的)
   4. 配置mapper文件的扫描器

## Service层

### 需要的jar包

Spring的jar包

### 配置文件

1. Spring配置文件：applicationContext-service.xml
   1. 配置包扫描器，扫描所有带@Service注解的类
2. 事务配置：applicationContext-trans.xml
   1. 配置事务管理器
   2. 配置tx(事务通知)
   3. 配置aop(切面)

## 表现层

### 需要的jar包

使用springmvc，需要使用springmvc和spring的jar包

### 配置文件

1. springmvc的配置：springmvc.xml
   1. 配置注解驱动(处理器映射器和处理器适配器)
   2. 配置视图解析器
   3. 包扫描器，扫描@Controller注解

## web.xml配置

1. 配置springmvc的前端控制器
2. Spring容器初始化的listener

# SSM实际整合

## 框架整合dao层

### SqlMapConfig.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"> <configuration> </configuration> |

### applicationContext-dao.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:p="http://www.springframework.org/schema/p"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-4.0.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop-4.0.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx-4.0.xsd  http://www.springframework.org/schema/util  http://www.springframework.org/schema/util/spring-util-4.0.xsd">   <!-- 数据库连接池 -->  <!-- 加载配置文件 -->  <context:property-placeholder location="classpath:properties/\*.properties"/>  <!-- 数据库连接池 -->  <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"  destroy-method="close">  <property name="url" value="${jdbc.url}"/>  <property name="username" value="${jdbc.username}"/>  <property name="password" value="${jdbc.password}"/>  <property name="driverClassName" value="${jdbc.driver}"/>  <property name="maxActive" value="10"/>  <property name="minIdle" value="5"/>  </bean>  <!-- 让spring管理sqlsessionfactory 使用mybatis和spring整合包中的 -->  <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  <!-- 数据库连接池 -->  <property name="dataSource" ref="dataSource"/>  <!-- 加载mybatis的全局配置文件 -->  <property name="configLocation" value="classpath:mybatis/SqlMapConfig.xml"/>  </bean>  <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  <property name="basePackage" value="com.taotao.mapper"/>  </bean> </beans> |

## 整合service层

### 配置applicationContext-service

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:p="http://www.springframework.org/schema/p"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-4.0.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop-4.0.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx-4.0.xsd  http://www.springframework.org/schema/util  http://www.springframework.org/schema/util/spring-util-4.0.xsd">  <!-- 配置包扫描器 -->  <context:component-scan base-package="com.taotao.service" /> </beans> |

### 配置applicationContext-trans.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:p="http://www.springframework.org/schema/p"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-4.0.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop-4.0.xsd   http://www.springframework.org/schema/tx   http://www.springframework.org/schema/tx/spring-tx-4.0.xsd  http://www.springframework.org/schema/util  http://www.springframework.org/schema/util/spring-util-4.0.xsd">  <!-- 事务管理器 -->  <bean id="transactionManager"  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <!-- 数据源 -->  <property name="dataSource" ref="dataSource" />  </bean>  <!-- 通知 -->  <tx:advice id="txAdvice" transaction-manager="transactionManager">  <tx:attributes>  <!-- 传播行为 -->  <tx:method name="save\*" propagation="REQUIRED" />  <tx:method name="insert\*" propagation="REQUIRED" />  <tx:method name="add\*" propagation="REQUIRED" />  <tx:method name="create\*" propagation="REQUIRED" />  <tx:method name="delete\*" propagation="REQUIRED" />  <tx:method name="update\*" propagation="REQUIRED" />  <tx:method name="find\*" propagation="SUPPORTS" read-only="true" />  <tx:method name="select\*" propagation="SUPPORTS" read-only="true" />  <tx:method name="get\*" propagation="SUPPORTS" read-only="true" />  </tx:attributes>  </tx:advice>  <!-- 切面 -->  <aop:config>  <aop:advisor advice-ref="txAdvice" pointcut="execution(\* com.taotao.service.\*.\*(..))" />  </aop:config> </beans> |

## 整合表现层

### 配置springmvc

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:p="http://www.springframework.org/schema/p"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:mvc="http://www.springframework.org/schema/mvc"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/mvc  http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd">  <!-- 配置包扫描器 -->  <context:component-scan base-package="com.taotao.controller"></context:component-scan>  <!-- 配置注解驱动 -->  <mvc:annotation-driven/>  <!-- 视图解析器 -->  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix" value="/WEB-INF/jsp/" />  <property name="suffix" value=".jsp" />  </bean> </beans> |

### 配置web.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns="http://java.sun.com/xml/ns/javaee"  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" id="taotao" version="2.5"> <display-name>taotao-manager</display-name> <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>index.htm</welcome-file>  <welcome-file>index.jsp</welcome-file>  <welcome-file>default.html</welcome-file>  <welcome-file>default.htm</welcome-file>  <welcome-file>default.jsp</welcome-file> </welcome-file-list> <!-- 初始化spring容器 --> <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring/applicationContext-\*.xml</param-value> </context-param> <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class> </listener> <!-- 解决post乱码 --> <filter>  <filter-name>CharacterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param> </filter> <filter-mapping>  <filter-name>CharacterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern> </filter-mapping> <!-- springmvc的前端控制器 --> <servlet>  <servlet-name>taotao-manager</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <!-- contextConfigLocation不是必须的， 如果不配置contextConfigLocation， springmvc的配置文件默认在：WEB-INF/servlet的name+"-servlet.xml" -->  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring/springmvc.xml</param-value>  </init-param>  <!-- 在启动tomcat初始化servlet,否则，第一次访问时才初始化该servlet -->  <load-on-startup>1</load-on-startup> </servlet> <servlet-mapping>  <servlet-name>taotao-manager</servlet-name>  <url-pattern>/</url-pattern> </servlet-mapping> </web-app> |

# 测试整合框架

## 需求

根据商品ID查询商品信息，返回json数据

## Dao层

查询的表tb\_item，根据商品id查询。可以使用逆向工程生成的代码

## service层

接受一个商品ID，调用Mapper查询商品信息，返回一个商品的POJO

参数：Long itemId

返回TbItem

|  |
| --- |
| */\*\*  \* 商品查询Service  \* Created by lenovo on 2018/1/30.  \*/* @Service public class ItemServiceImpl implements ItemService {   @Autowired  private TbItemMapper itemMapper;  @Override  public TbItem getItemById(Long itemId) {  return itemMapper.selectByPrimaryKey(itemId);  } } |

## Controller层

接收商品ID，调用Service返回一个商品的pojo，直接相应pojp。需要返回json数据，使用@ResponseBody

注意：使用@RespinseBody的时候一定要把Jackson包添加到工程中。

Url：/item/{itemId}

相应：TbItem json数据

|  |
| --- |
| */\*\*  \* 商品查询Controller  \*  \** ***@author*** *lenovo  \** ***@date*** *2018/1/30  \*/* @RequestMapping("/item") @Controller public class ItemController {  @Autowired  ItemService itemService;  @RequestMapping("/selectOne/{itemId}")  @ResponseBody  public TbItem selectOne(@PathVariable Long itemId){  return itemService.getItemById(itemId);  } } |

## 解决mapper映射文件不拷贝的问题

### 需要修改taotao-dao工程pom文件，添加以下内容

|  |
| --- |
| <build>  <resources>  <resource>  <directory>src/main/java</directory>  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  <resource>  <directory>src/main/resources</directory>  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  </resources> </build> |

# 淘淘商城第二天

## 回顾

1. 电商行业的了解
2. 淘淘商城项目介绍
3. 后台系统的搭建
   1. 使用maven管理
   2. 聚合工程
   3. 根据层级，创建maven模块
4. SSM框架的整合

## 课程计划

1. 商品列表的查询功能
   1. easyui
   2. 分页处理，分页插件
2. 商品的添加
   1. 商品类目选择-easyui异步tree控件的使用
   2. 图片上传，如何保存图片，图片服务器(开头，Nginx)
   3. 父文本编辑器的使用
   4. 添加的实现

# 展示首页

创建一个Controller，做一个页面跳转。展示index。

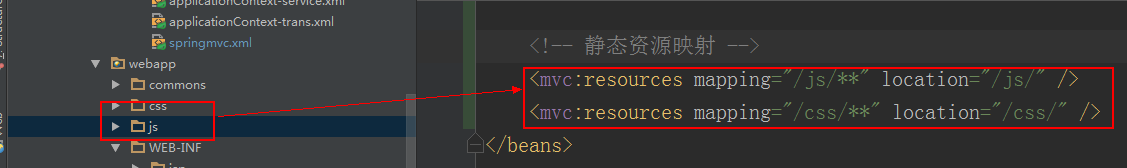
## 分析

请求的Url：/

返回的结果：返回一个String类型，代表逻辑视图

## 静态资源被拦截

使用静态资源映射

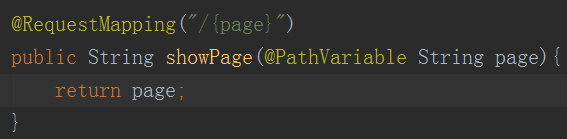


使用静态资源映射，通过springmvc进行资源获取，即便是放在WEB-INF中，也会产生这种效果，可以访问

# 商品列表展示

## 分析

打开商品列表页面，应该创建一个Handler接受请求，跳转到对应的页面。请求和页面的名称一致，可以统一处理。



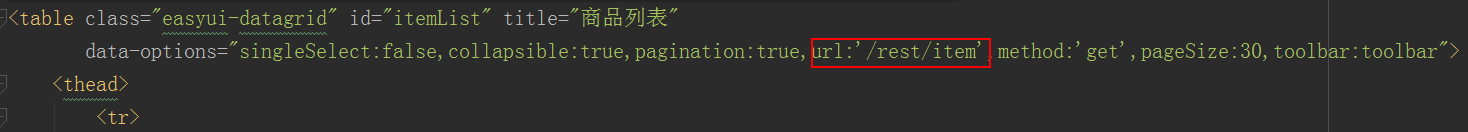
查询的表：tb\_item

单表查询SQL：select \* from tb\_item limit 0,10。

需要实现分页，使用逆向工程，可以使用mybatis的分页插件。

### 页面分析

easyUI会在页面跳转时，通过url实现ajax请求，从而接受后台数据。



#### ajax请求

请求的参数：

#### 相应的Json数据

相应的数据必须包含total、rows两个属性

Total：查询结果的总记录数。

Rows：集合，包含显示的所有的数据。其中集合中每个元素的key应该和dategrid的field对应。



## 代码实现

### Dao层

可以使用逆向工程生成的代码。

## Service层

接受参数，调用Dao查询列表。

参数：page(显示的页码)、rows(每页显示的记录数)

返回值：创建一个pojo表示返回值。应该包含total、rows两个属性。

应该放到taotao-common工程中，和其他系统共用。

### pojo类

|  |
| --- |
| public class EasyUIDataGridResult {  // 总记录数  private long total;  // 每页数据  private List<?> rows;  public long getTotal() {  return total;  }   public void setTotal(long total) {  this.total = total;  }  public List<?> getRows() {  return rows;  }  public void setRows(List<?> rows) {  this.rows = rows;  } } |

### service类

|  |
| --- |
| @Override public EasyUIDataGridResult getItemList(int page, int rows) {  // 分页处理  PageHelper.*startPage*(page,rows);  // 执行查询  TbItemExample example = new TbItemExample();  List<TbItem> itemList = itemMapper.selectByExample(example);  // 取分页信息  PageInfo<TbItem> pageInfo = new PageInfo<>(itemList);  // 返回处理结果  EasyUIDataGridResult result = new EasyUIDataGridResult();  result.setTotal(pageInfo.getTotal());  result.setRows(itemList);  return result; } |

## Controller层

接受两个参数：page、rows

调用Service查询商品列表。返回EasyUIDataGridResult对象(json数据)，需要使用@ResponseBody

### Handler

|  |
| --- |
| @RequestMapping("/rest/item") @ResponseBody public EasyUIDataGridResult getItemList(Integer page, Integer rows){  return itemService.getItemList(page,rows); } |

# 新增商品功能

## 功能分析

### 选择类目功能

点击该按钮，弹出对话框，并可选择商品类目信息等。

### 上传图片功能

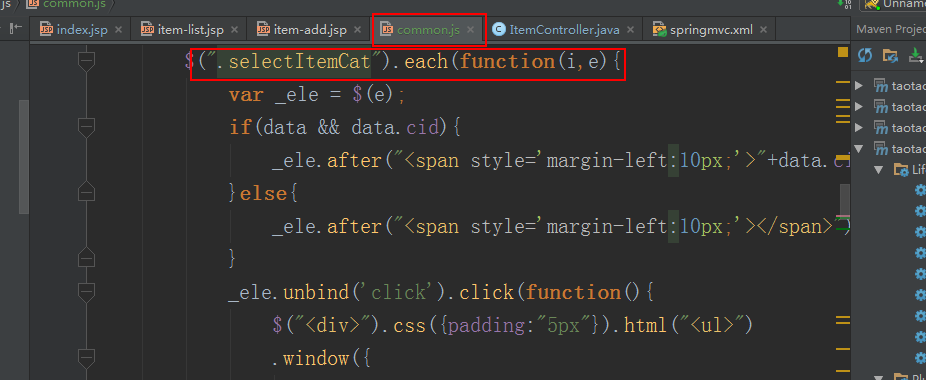
可批量选择图片，并一起上传。

## 添加商品-类目选择

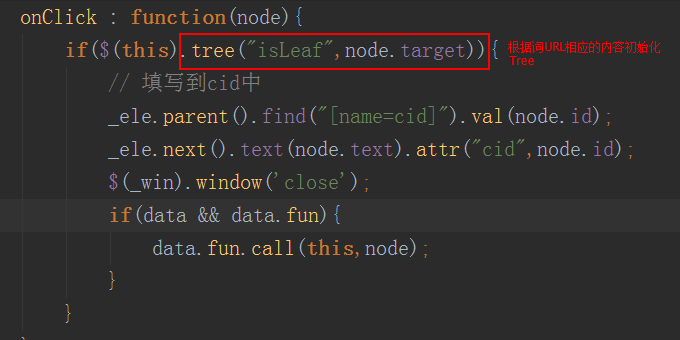
### 分析

商品类目使用的表：tb\_item\_cat

#### 初始化类目选择



#### EasyUI的异步Tree控件：



请求的url：/rest/item/cat/list

请求的参数：id(点击的节点的id)

响应的结果：json数据。

必须包含的属性：

id text state

如果当前节点为父节点，state应为closed、如果是叶子节点为open

### 异步树控件

树控件内建异步加载模式的支持，用户先创建一个空的树，然后指定一个服务器端，执行检索后动态返回JSON数据来填充树并完成异步请求。

树控件读取URL。子节点的加载依赖于父节点的状态。当展开一个封闭的节点，如果节点没有加载子节点，它将会把节点ID的值作为http请求参数并命名为id，通过URL发送到服务器上面检索子节点。

## dao层

查询tb\_item\_cat表，根据ID查询商品分类列表。可以使用逆向工程生成代码。

## Service层

接受参数parentId，根据parentId查询分类列表。返回一个POJO的列表。

POJO应该包含三个属性：

Id、text、state

POJO应该放在taotao-common中

### POJO

|  |
| --- |
| public class EasyUITreeNode {  private long id;  private String text;  private String state;} |

### 代码

|  |
| --- |
| public List<EasyUITreeNode> getItemCatList(long parentId) {  TbItemCatExample itemCatExample = new TbItemCatExample();  TbItemCatExample.Criteria criteria = itemCatExample.createCriteria();  criteria.andParentIdEqualTo(parentId);  // 执行查询  List<TbItemCat> list = itemCatMapper.selectByExample(itemCatExample);  // 转换成树形EasyUITreeNode列表  List<EasyUITreeNode> resultList = new ArrayList<>();  for (TbItemCat tbItemCat : list){  EasyUITreeNode node = new EasyUITreeNode();  node.setId(tbItemCat.getId());  node.setText(tbItemCat.getName());  node.setState(tbItemCat.getIsParent()?"closed":"open");  // 添加到列表中  resultList.add(node);  }  return resultList; } |

## Controller层

接受参数，parentId。调用Service查询分类类别，返回列表(jsoin数据)，需要使用@RequestBody。

请求的Url：/rest/item/cat/list

|  |
| --- |
| @Controller @RequestMapping("/rest/item/cat") public class ItemCatController {  @Autowired  private ItemCatService itemCatService;  @RequestMapping("/list")  @ResponseBody  private List<EasyUITreeNode> getItemCatList(@RequestParam(value = "id",defaultValue = "0") Long parentId){  return itemCatService.getItemCatList(parentId);  } } |

# Mybatis的分页插件 – pageHelper说明

## 原理分析

Executor执行器对象

SqlSessionFactory

SqlSession

提供很多用户方法

MappedStatement

Sql语句的封装

数据库

拦截器，需要实现mybatis的拦截器接口。

对sql语句进行编辑。把修改后的sql语句设置回去。

## 简介

如果你在在使用MyBatis，建议尝试该分页插件，这个一定是最方便使用的分页插件。该插件目前支持Oracle,MySql,MariaDB,SQLite,Hsqldb,PostgreSql六中数据库分页。

其实际来说，Oracle实际上是Java语言开发的一款Sql数据库，其中涉及很多的算法结构，使其数据存储与读取速度增快。

## 使用方法

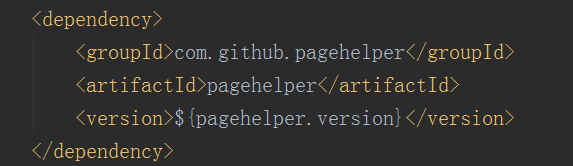
第一步：需要在SqlMapConfig.xml，配置一个plugin。将PageHelper添加到工程中。

第二步：在sql语句执行之前，添加一个PageHelper.startPage(page,rows)。

第三步：取分页的结果。创建一个PageInfo对象，需要一个参数，查询结果返回的list。

## 代码方法

### 将jar包添加进入子项目



### 配置SqlSessionConfig.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"> <configuration>  <!-- 配置分页插件 -->  <plugins>  <plugin interceptor="com.github.pagehelper.PageHelper">  <!-- 指定使用的数据库是什么 -->  <property name="dialect" value="mysql" />  </plugin>  </plugins> </configuration> |

## 分页插件测试

|  |
| --- |
| public class TestPageHelper {  @Test  public void testPageHelper() throws Exception {  // 1.获取mapper代理对象 ApplicationContext applicationContext = new ClassPathXmlApplicationContext ("classpath:spring/applicationContext-\*.xml");  TbItemMapper itemMapper = applicationContext.getBean(TbItemMapper.class);  // 2.设置分页  PageHelper.*startPage*(1, 30);  // 3.执行查询  TbItemExample example = new TbItemExample();  List<TbItem> tbItems = itemMapper.selectByExample(example);  // 4.取分页之后的结果  PageInfo<TbItem> pageInfo = new PageInfo<>(tbItems);  long total = pageInfo.getTotal();  System.*out*.println("total:" + total);  System.*out*.println("pages:" + pageInfo.getPages());  System.*out*.println("pageSize:" + pageInfo.getPageSize());  } } |

Spring实际上是使用反射机制与XML文件实现的类加载机制。

使用日志文件监控，可以监控表结构变化，只要是对数据库进行的操作两边都会进行同步，但是还有种情况是，突然之间大批量数据导入，可能会导致数据日志文件突然之间的急剧增加。所以，应该限制日志文件的大小，使用单线程操作，保证数据的平稳读取与传送。

在并发量在可控制范围之内时，没有太大的必要去操作分布式系统，配置集群即可

# 图片上传功能

## 图片上传分析

### 传统方式

Tomcat

访问图片

http://ip:port/project/imahes/xxx.jpg

上传图片

上传到images目录

Webapp

|-index.html

|-WEB-INF

|-images

并发量小，用户少

Tomcat

Tomcat

并发量高，用户多

配置集群，使负载均衡

Webapp

|-index.html

|-WEB-INF

|-images

Webapp

|-index.html

|-WEB-INF

|-images

当用户上传图片至服务器时

请求图片

上传图片

存储在一个服务器中，但不

能保证下一次访问的还是该

服务器。

### 互联网项目

解决方案，新建一个图片服务器。专门保存图片，不管是那个服务器接受到图片，都把图片上传到图片服务器。

图片服务器

有独立IP

Webapp

|-index.html

|-WEB-INF

~~|-images~~

Tomcat

Tomcat

Webapp

|-index.html

|-WEB-INF

~~|-images~~

图片服务器上需要安装一个http服务器，可以使用tomcat、apache、nginx

上传图片

请求图片

http://imageserver-ip:port/project/image.a,jpg

nginx性能非常不错。

# Nginx

## 课程目标

掌握nginx+tomcat反向代理的使用方法。

掌握nginx作为负载均衡器的使用方法。

掌握nginx实现web缓存的方法。

## nginx介绍

nginx是一款高性能http服务器/反向代理服务器及电子邮件(IMAP/POP3)代理服务器。由俄罗斯的程序设计器Igor Sysoev所开发，官方测试nginx能够支撑5W并发链接，并且cpu、内存等资源想好非常低，运行非常稳定。

## Nginx的应用场景

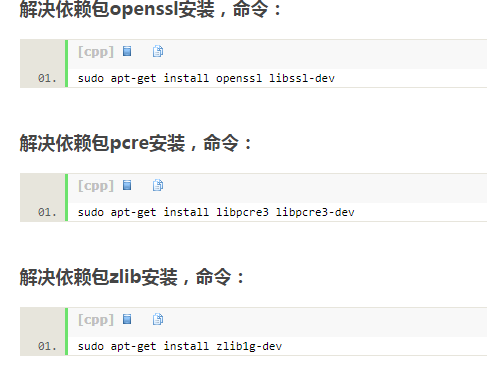
1. http服务器。Nginx是一个http服务可以独立提供http服务。可以做网页静态服务器。
2. 虚拟主机。可以实现在一台服务器上模拟出多个网站。例如个人网站使用的虚拟主机。
3. 反向代理，负载均衡。当网站的访问量达到一定程度后，单台服务器不能满足用户的请求时，需要多台服务器集群可以使用nginx做反向代理。并且多台服务器可以平均分担负载，不会因为某台服务器负载高而宕机而某台服务器闲置的情况。

使用C语言开发的，高性能的HTTP服务器，及反向代理服务器。

nginx是C语言开发的，建议在Linux上运行，要安装C语言的编译环境gcc。

### Nginx安装

1. 下载并安装gcc
2. 下载并安装nginx的安装包PCRE zlib openssl



### nginx的启动以及停止

启动./nginx即可，可以直接访问80端口，如果访问不到，首先查看防火墙是否关闭，80端口是否并禁。

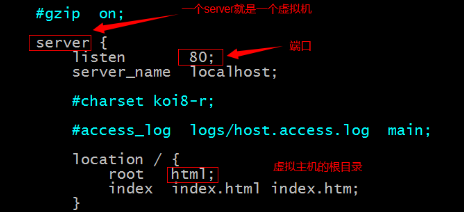
停止nginx -s stop

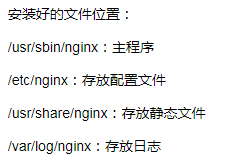
刷新配置：nginx -s reload 不关闭nginx服务的前提下刷新配置文件。

## nginx的配置

/etc/nginx/nginx.conf为其配置文件。

一个server就是一个虚拟主机





### 使用nginx配置虚拟主机

通过端口区分虚拟机。

在http标签下配置如下内容



要求根目录必须存在，根目录在/usr/share/nginx下

## 根据域名区分虚拟机

DNS服务器，将域名转换成IP地址，然后提供给用户访问，直接的域名是没有办法进行访问的。

可以通过修改host文件指定域名的ip地址。可以进行host劫持。

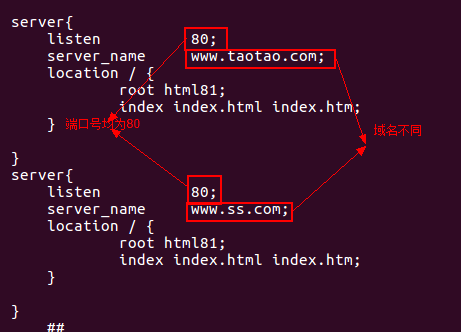
### host文件的位置

window的hosts文件

C：\window\System32\drivers\etc

## 配置基于域名的虚拟主机

### 修改nginx.conf配置文件



# 淘淘商城第三天

## 第二天内容回顾

1. 展示商品列表。
   1. 分页插件的使用PageHelper。
   2. easyUIDateGrid的使用方法。
2. 商品的类目选择
   1. easyUI异步tree控件的使用
3. 图片上传
   1. 为什么要有图片服务器。
   2. fastDFS保存文件，分布式文件系统。
   3. http服务器访问图片。nginx。
4. nginx
   1. http服务器，虚拟主机，反向代理及负载均衡。
   2. 虚拟主机
      1. 基于端口
      2. 基于域名

## 课程计划

1. nginx的反向代理与负载均衡。
2. FastDFS的使用。保证图片服务器的高可用，分布式文件系统。
3. 实现图片上传功能。

## nginx的反向代理

### 什么是反向代理

#### 正向代理

代

理

服

务

器

PC

PC

PC

#### 反向代理

淘宝首页

反向代理服务器

www.taobao.com

PC

淘宝首页

淘宝首页

将域名与反向代理服务器的IP进行绑定。

功能就是请求的转发。

反向代理服务器是引用在服务端。决定是哪台服务器提供服务。

反向代理类似于控制器，与控制器不同的是，控制器是转发给不同的后台处理代码，而反向代理是将数据转向集群类服务器。反向代理服务器是将所有的数据接收，并返回给按照一定的规则给与服务器处理。

## 反向代理的模拟

应该有一个nginx服务器，有多个应用服务器(可以是tomcat)

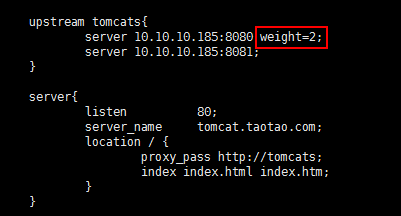
可以使用一台虚拟机，安装一个nginx，多个tomcat来进行



### 负载均衡

只需要再upstream的server后面添加一个weight即可，代表权重。

权重越高，分配的请求的数量就越多。默认权重是1。



nginx是一个代理服务器，对于nginx的访问，是通过nginx所在的ip地址进行的。

因为域名是绑定ip地址的，所以，通过hosts的配置，在电脑本地将域名与ip地址访问，访问的就是ip地址。

当nginx收到客户发来的访问请求时，会自动解析内容，比如访问的域名，再将域名与server中的server\_name进行匹配，从而确定真实的服务器地址。

# 图片服务器的搭建

## 搭建图片服务器

搭建图片服务器使用FastDFS。

### 什么是FastDFS

FastDFS是用C语言编写的一款开源的分布式文件系统。FastDFS为互联网量身定制，重复考虑了冗余备份、负载均衡、线性扩容等机制，并注重高可用、高性能等指标，使用FastDFS很容易搭建一套高性能的文件服务器集群提供文件上传、下载等服务。

### FastDFS架构

FastDFS架构包括Tracker server和Storage server。客户端请求Tracker server进行文件上传、下载，通过Tracker server调度最终由Storage server完成文件上传和下载。

Tracker server作用是负载均衡和调度，通过Tracker server在文件上传时可用根据一些策略找到Storager server提供文件上传服务。可用将tracker称为追踪服务器或调度服务器。

Storage server作用是文件存储，客户端上传的文件最终存储在Stroage服务器上，Storage server没有实现自己的文件系统而是利用操作系统的文件系统来管理文件。可用将storage称为存储服务器。



### Tracker集群

FastDFS集群中的Tracker server可用有多台，Tracker server之间相互平等关系同时提供服务，Tracker server不存在单点故障。客户端请求Tracker server采用轮行方式，如果请求的tracker无法提供服务则换另一个tracker。

### Stortage集群

Storage集群采用了分组存储方式。storage集群由一个或多个组构成，集群存储总容量为集群中所有组的存储容量之和。一个组由一台或多台存储服务器组成，组内的Storage server之间是平等关系，不同组的Storage server之前不会相互通信，同组内的Storage server之间会相互连接进行文件同步，从而保证同组内每个stgorage上的文件完全一致的。一个组的存储容量为改组内存存储服务器容量最小的那个，由此可见组内存储服务器的软硬件配置最好是一致的。

采用分组存储方式的好处是灵活、可控性较强。比如上传文件时，可以由客户端直接指定上传到的组也可以由tracker进行调度选择。一个分组的存储服务器访问压力较大时，可以(纵向扩容)。当系统容量不足时，可以增加组来扩种存储容量(横向扩容)。

### 文件上传的流程



客户端上传文件后，存储服务器将文件ID返回给客户端，此文件ID用户以后访问该文件的索引信息。文件索引信息包括：组名，虚拟磁盘路径，数据两级目录，文件名。

* 组名：文件上传后所在的storage组名称，在文件上传成功之后，由storage服务器返回，交由客户端，客户端自行进行保存。
* 虚拟磁盘路径：storage配置的虚拟路径，与磁盘选
* 项store\_path\*对应。如果配置了store\_path0则是M00，如何配置了store\_path1则是M01。
* 数据两级目录：storage服务器在每个虚拟磁盘路径下创建的两级目录，用户存储数据文件。
* 文件名：与文件上传时不同。是由存储服务器根据特定的信息生成的，文件名包含：源存储服务器IP地址、文件创建时间戳、文件大小、随机数和文件扩展名等信息。

### 文件下载流程



* 通过组名tracker server能很快的定位到客户端需要访问的存储服务器名，并选择合适的存储服务器提供客户端访问。
* 存储服务器根据“文件存储虚拟磁盘路径”和“数据文件两级目录”可以很快丁文到文件所在的目录，并根据文件名找到客户端需要访问的文件。

# FastDFS搭建

架构图

Storage Server Group1

Http

TCP/IP

Http

clientAPI

storage 1

192.168.101.3

部署程序：

FastDFS

nginx

Tracker

192.168.101.3

部署程序：

FastDFS

client

可以使用一台虚拟机来模拟，只有一个Treacker、一个Storage服务。

配合nginx访问图片。

## 搭建步骤

### 所需要的文件



### 搭建步骤

第一步：把FastDFS文件上传到linux系统。

第二步：安装FastDFS之前，先安装libevent工具包



第三步：安装libfastcommon



* + 解压缩
    - 
  + 运行./make
    - 
  + 运行./make install
    - 

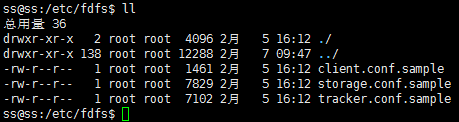
第四步：安装tracker服务



* + 解压缩
  + ./make.sh
  + ./make.sh install

安装后在/usr/bin目录下有以fdfs\_开头的文件都是编译出来的。

配置文件都放到/etc/fdfs文件夹。



将所有配置文件全部都复制到/etc/fdfs/下



第五步：配置tracker服务，修改/etc/fdfs/tracker.conf



用来存储日志文件，在服务启动时，该文件必须存在，改成自己的路径，路径必须存在。

第六步：启动tracker。

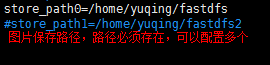


/usr/bin/fdfs\_trackerd /etc/fdfs/tracker.conf

重启：

第七步：安装Storage服务

如果是在不同的服务器安装，第四步要重新执行

* 配置storage服务器，修改/etc/fdfs/storage.conf
* 配置storage服务日志存放路径，路径必须存在
* 请注意此路径，该路径需要与
* 

第八步：启动storage服务器

，重启加入restart

第九步：测试服务

修改配置文件





启动测试



第十步：搭建nginx提供http服务器

可以使用官方提供的nginx插件。要使用nginx插件需要重新编译。

1. nginx插件

修改nginx插件配置信息



将其中的所有local去掉

1. 对nginx重新配置，在nginx下配置如下内容，最后一条数据需要指定配置的nginx插件src路径。

./configure \

--prefix=/usr/local/nginx \

--pid-path=/var/run/nginx.pid \

--lock-path=/var/lock/nginx.lock \

--error-log-path=/var/log/nginx/error.log \

--http-log-path=/var/log/nginx/access.log \

--with-http\_gzip\_static\_module \

--http-client-body-temp-path=/var/temp/nginx/client \

--http-proxy-temp-path=/var/temp/nginx/proxy \

--http-fastcgi-temp-path=/var/temp/nginx/fastcgi \

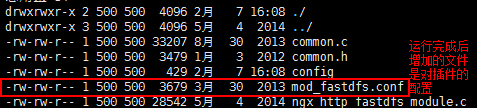
--http-uwsgi-temp-path=/var/temp/nginx/uwsgi \

--http-scgi-temp-path=/var/temp/nginx/scgi \

--add-module=/usr/fastdfs/fastdfs-nginx-module/src

3. 在nginx目录下运行make命令

4. 在nginx目录下运行make install命令



1. 将/usr/fastdfs/fastdfs-nginx-module/src/mod\_fastdfs.conf文件复制到/etc/fdfs目录下



1. 修改mod\_fastdfs.conf文件

该路径必须存在

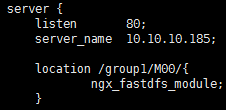




配置图片保存路径，是storage的文件路径

第十一步：配置nginx

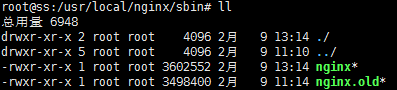
在nginx,http中增加server



第十二步：将libfdfsclient.so拷贝到/usr/lib中



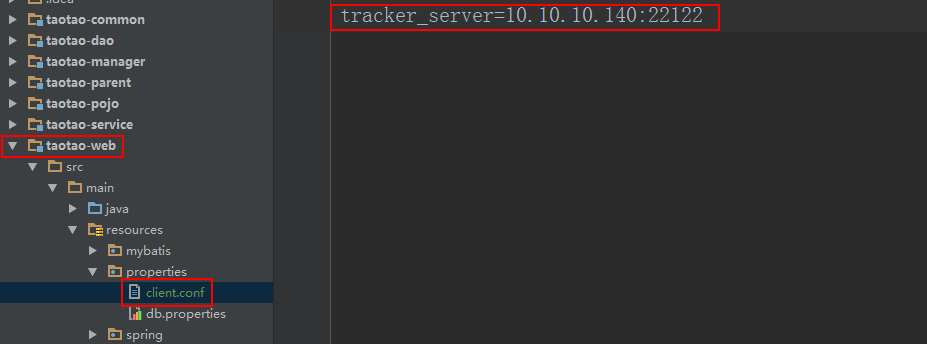
第十三步：启动nginx



## 图片服务器的使用

官方提供一个jar包

### 新建一个配置文件client.conf



### 使用方法：

1. 将FastDFS提供的jar包添加到工程中
2. 初始化全局配置。加载一个配置文件
3. 创建TrackerClient对象
4. 通过TrackerClient对象，创建一个TrackerServer对象
5. 声明一个StorgerServer对象，null
6. 获得StorageClient对象
7. 调用StorageClient对象的方法上传文件即可，返回组名与

|  |
| --- |
| public class FastdfsTest {  public static void main(String[] args) throws Exception {  //1.将FastDFS提供的jar包添加到工程中  //2.初始化全局配置。加载一个配置文件  ClientGlobal.*init*("E:\\taotao\\taotao-web\\src\\main\\resources\\properties\\client.conf");  //3.创建TrackerClient对象  TrackerClient trackerClient = new TrackerClient();  //4.通过TrackerClient对象，创建一个TrackerServer对象  TrackerServer trackerServer = trackerClient.getConnection();  //5.声明一个StorgerServer对象，null  StorageServer storageServer = null;  //6.获得StorageClient对象  StorageClient storageClient = new StorageClient(trackerServer,storageServer);  //7.调用StorageClient对象的方法上传文件即可  String[] strings = storageClient.upload\_file("C:\\Users\\lenovo\\Downloads\\a316d468f885805347bea69700f1cda8.jpg", "jpg", null);  for (String str : strings){  System.*out*.println(str);  }  } } |

# FastDFS工具类的使用

## FastDFS工具类

|  |
| --- |
| private TrackerClient trackerClient = null; private TrackerServer trackerServer = null; private StorageServer storageServer = null; private StorageClient storageClient = null; public FastDFSClient(String conf) throws Exception{  if(conf.contains("classpath:")){  conf = conf.replace("classpath:",this.getClass().getResource("/").getPath());  }  ClientGlobal.*init*(conf);  trackerClient = new TrackerClient();  trackerServer = trackerClient.getConnection();  storageServer = null;  storageClient = new StorageClient(trackerServer,storageServer); }  */\*\*  \* 文件上传方法  \** ***@param*** *fileName 文件全路径  \** ***@param*** *extName 文件扩展名  \** ***@param*** *metas 文件扩展信息  \** ***@return*** *\** ***@throws*** *Exception  \*/* public String uploadFile(String fileName, String extName, NameValuePair[] metas) throws Exception{  String[] result = storageClient.upload\_file(fileName,extName,metas);  return result[0]+"/"+result[1]; } public String uploadFile(String fileName,String extName) throws Exception{  return uploadFile(fileName,extName,null); } public String uploadFile(String fileName) throws Exception{  return uploadFile(fileName,null,null); } */\*\*  \* 文件上传方法  \** ***@param*** *fileContent 文件byte数组  \** ***@param*** *extName 文件扩展名  \** ***@param*** *metas 文件扩展信息  \** ***@return*** *\** ***@throws*** *Exception  \*/* public String uploadFile(byte[] fileContent,String extName,NameValuePair[] metas) throws Exception{  String[] result = storageClient.upload\_file(fileContent,extName,metas);  return result[0]+"/"+result[1]; } public String uploadFile(byte[] fileContext,String extName) throws Exception{  return uploadFile(fileContext,extName,null); } public String uploadFile(byte[] fileContext) throws Exception{  return uploadFile(fileContext,null,null); } |

## FastDFS工具类测试

|  |
| --- |
| @Test public void testFastdfs() throws Exception{  FastDFSClient fastDFSClient = new FastDFSClient("E:\\taotao\\taotao-web\\src\\main\\resources\\properties\\client.conf");  String s = fastDFSClient.uploadFile("C:\\Users\\lenovo\\Downloads\\u=4030449382,3054740177&fm=27&gp=0.jpg");  System.*out*.println(s); } |

# 图片上传功能

## 功能分析—页面功能实现步骤

### 初始化

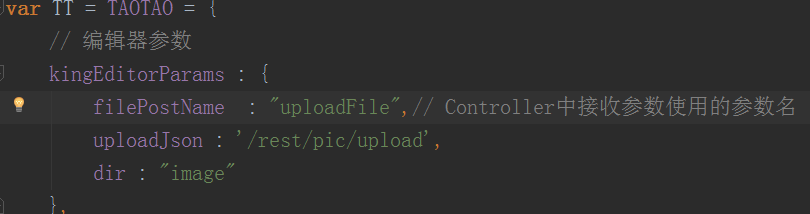


### 绑定Click事件

使用的是富文本编辑器的图片上传插件



### 请求的url



富文本编辑器参考文档：

<http://kindeditor.net/docs/upload.html>

### 添加jar包：

Commons-io, Commons-fileupload

### 在springmvc.xml中配置多媒体解析器

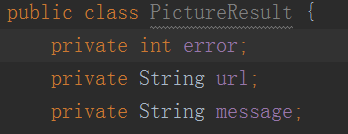
|  |
| --- |
| <!-- 多媒体文件解析器 --> <bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  <!-- 设置默认编码 -->  <property name="defaultEncoding" value="UTF-8" />  <!-- 设置文件上传的最大值 -->  <property name="maxUploadSize" value="5242880"></property> </bean> |

### 响应的内容



### 创建一个pojo描述返回值

pojo中有三个属性:error、url、message，可以放到taotao-common中



## 功能分析--service层

接受图片数据，把图片上传到图片服务器，返回PictureResult。需要使用FastDFSClientg工具类。

参数：MulitiPartFile pictureFile。

返回值：PictureResult。

|  |
| --- |
| @Override public PictureResult uploadPic(MultipartFile picFile) {  PictureResult result = new PictureResult();   // 判断图片是否为空  if(picFile.isEmpty()){  result.setError(1);  result.setMessage("图片为空");  return result;  }  // 上传到图片服务器  try {  // 取图片的扩展名  String originalFilename = picFile.getOriginalFilename();  String ext = originalFilename.substring(originalFilename.lastIndexOf(".") + 1);  FastDFSClient fastDFSClient = new FastDFSClient("classpath:properties/client.conf");  String url = fastDFSClient.uploadFile(picFile.getBytes(), ext);  // 拼接图片服务器的ip地址  url = IMAGE\_SERVER\_BASE\_URL+url;  // 将url相应给客户端  result.setError(0);  result.setUrl(url);  } catch (Exception e) {  result.setError(1);  result.setMessage("图片上传失败");  e.printStackTrace();  }  return result; } |

## 功能分析—Controller层

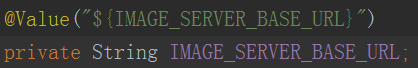
接受上传的图片信息，调用servirce将图片上传到图片服务器。但会json数据。需要使用@ResponseBody数据。

|  |
| --- |
| @RequestMapping("/rest/pic/upload") @ResponseBody public PictureResult uploadFile(MultipartFile uploadFile){  return pictureService.uploadPic(uploadFile); } |

## 加载属性文件

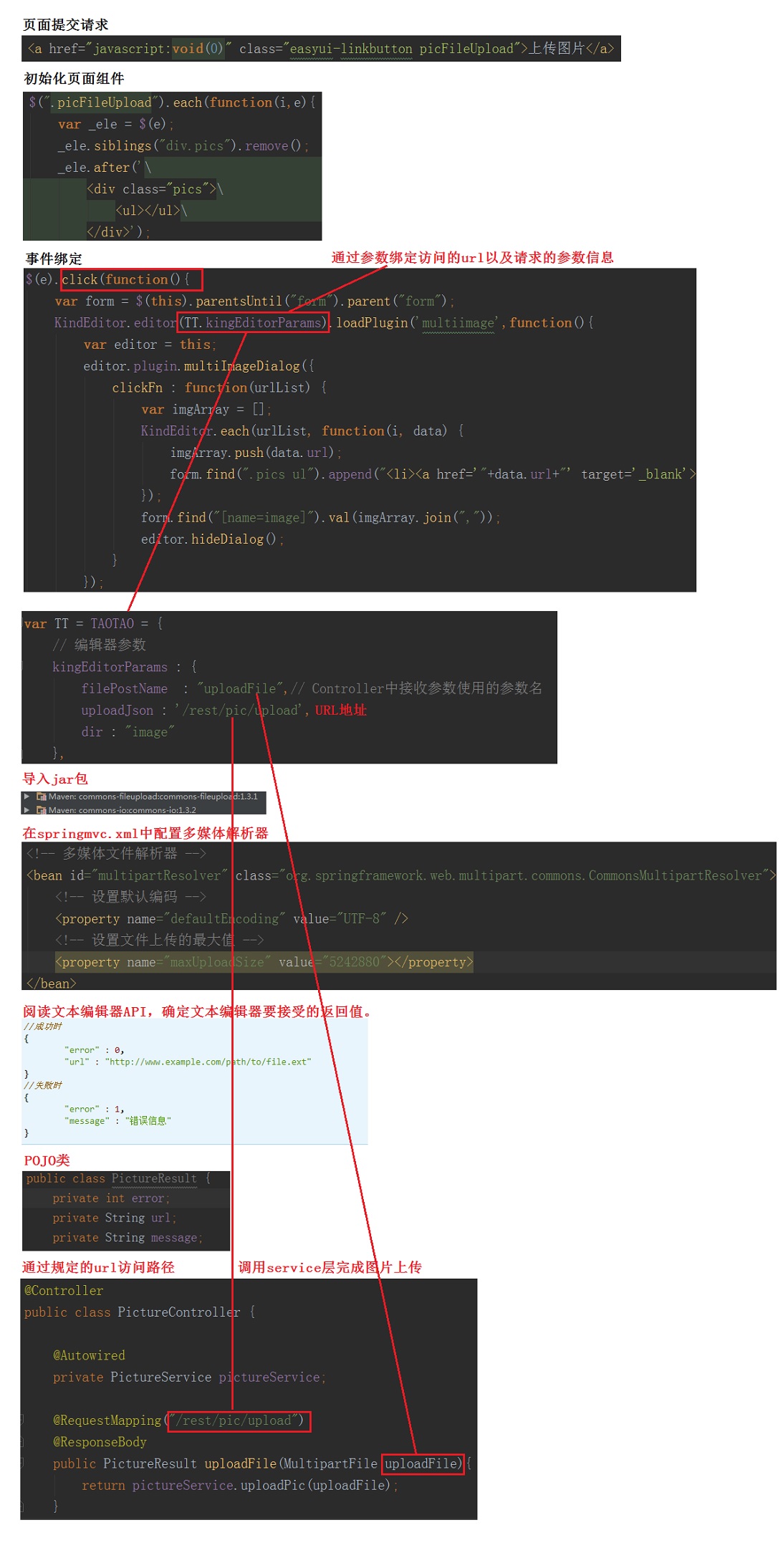
1. 创建一个属性文件
2. 使用spring容器扫描属性文件
3. 使用Value注解获取Value值

加载进入Spring容器的值，可以通过该注解获取并赋予其属性



### 主要过程

如下图：



## 解决浏览器兼容性问题

要求返回的数据是一个文本类型，要求content-type为text/plan

即Controller返回的不再是json数据，而是原始的String数据



# spring的父子容器问题

子容器可以访问父容器的对象，所以Controller可以注入Service对象。

但是父容器不能访问子容器中的对象。

子容器优先使用子容器自己的对象，找不到时才会访问父容器的对象，当子容器使用自己的service对象时，父容器所配置的事务管理不会生效。

springmvc容器

加载Controller

Spring容器

加载mapper接口与service

# 淘淘商城第四天

## 第三天复习

1. nginx的反向代理、负载均衡(权重配置)
2. FastDFS，分布式文件系统。图片服务器。访问图片使用nginx+fastDFS的插件。
3. 使用FastDFS的客户端来上传图片。
4. 图片上传功能实现。
5. springmvc和spring父子容器。

## 课程计划

1. 富文本编辑器的使用KindEditor。
2. 商品添加功能的实现。
3. 规格参数的实现。

# 富文本编辑器

## 使用方法

第一步：从KindEditor的官方网站下载源码http://kindeditor.net/down.php

第二步：将源码放入工程中

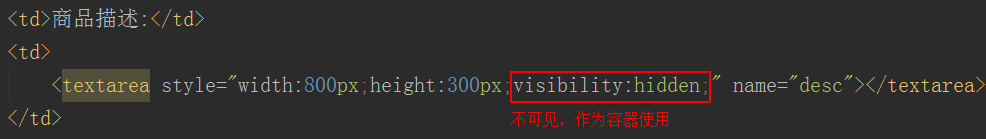
第三步：引入资源包

<script type="text/javascript" charset="utf-8" src="/js/kindeditor-4.1.10/kindeditor-all-min.js"></script>

第四步：将KindEditor的语言包引入jsp中

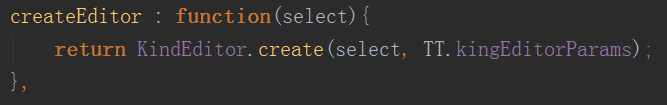
<script type="text/javascript" charset="utf-8" src="/js/kindeditor-4.1.10/lang/zh\_CN.js"></script>

第五步：创建一个textArea控件，作为富文本编辑器的数据源

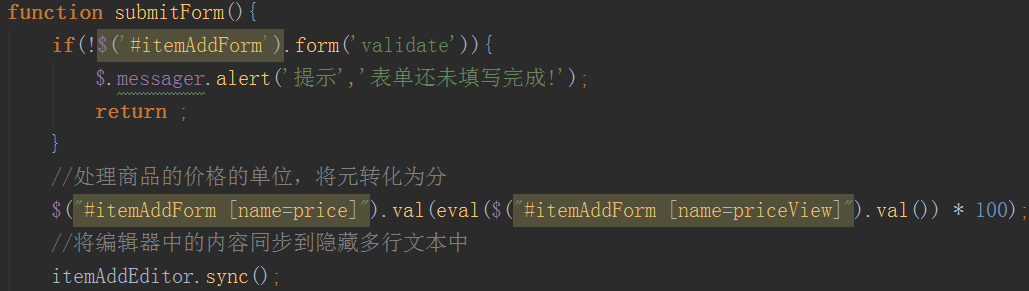


第六步：编写js代码，初始化KindEditor控件。需要指定TextArea控件。





第七步：在提交表单之前，先把富文本编辑器的内容同步到textArea控件中



## 流行的富文本编辑器

1. KindEditor
2. UEditor
3. CKEditor

# 商品添加的实现

## 分析

### 数据库

相关的表：tb\_item、tb\_item\_desc

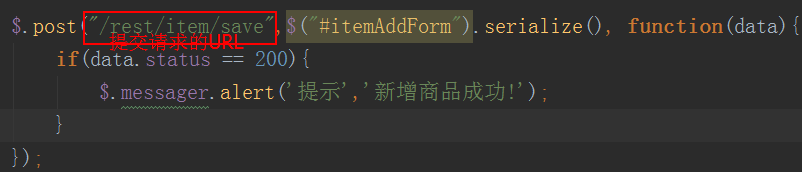
执行insert操作，可以使用逆向工程生成的代码

商品id应该是一串数字。redis自增数列。从redis中获取主键。

使用工具类IDUtils生成ID

### 页面

提交请求的url



表单.serialize()键值序列化，会生成的格式，作为表单提交的参数。

第三个参数代表新增商品成功。返回值应该是一个json数据。应该包含一个state属性，这个属性代表状态码。返回值为TaoTaoResult。

## DAO层实现

可以使用逆向工程生成的代码。

## Service层实现

Controller

数据库水平拆分。

数据库水平拆分，通过一定的规则将数据进行分库存储，分库存储的时候，连接的数据库配置信息进行规划，可以配置多数据源进行数据库的水平拆分。

1. 接受TbItem对象，String desc接受商品描述这两个参数。
2. 在TbItem中不全属性，包括：id、status、create、update。
3. 调用Mapper的insert方法插入数据。
4. 创建一个TbItemDesc对象，补全属性。
5. 调用Mapper方法插入到商品描述表。
6. 返回TaotaoResult对象。

参数：TbItem、String

返回值：TaotaoResult对象。

|  |
| --- |
| public TaotaoResult createItem(TbItem item, String desc) {  // 生成商品ID  long itemId = IDUtils.*genItemId*();  // 不全TbItem属性  item.setId(itemId);  // 商品状态，1-正常，2-下架，3-删除  item.setStatus((byte) 1);  // 创建时间和更新时间  Date date = new Date();  item.setCreated(date);  item.setUpdated(date);  // 插入商品表  itemMapper.insert(item);  // 商品描述  TbItemDesc itemDesc = new TbItemDesc();  itemDesc.setItemId(itemId);  itemDesc.setItemDesc(desc);  itemDesc.setUpdated(date);  itemDesc.setCreated(date);  // 插入商品描述信息  itemDescMapper.insert(itemDesc);  return TaotaoResult.*ok*(); } |

## Controller实现

1. 接受页面提交的内容，接受表单中的内容，使用POJO的格式接受参数。可以使用TbItem接受表单中提交的内容。还需要使用一个字符串类型的参数接受商品描述
2. 调用Service插入商品信息。
3. 返回TaotaoResult。(json格式)

|  |
| --- |
| @RequestMapping(value="/rest/item/save",method = RequestMethod.*POST*) @ResponseBody public TaotaoResult createItem(TbItem item,String desc){  return itemService.createItem(item,desc); } |

# 商品的规格参数

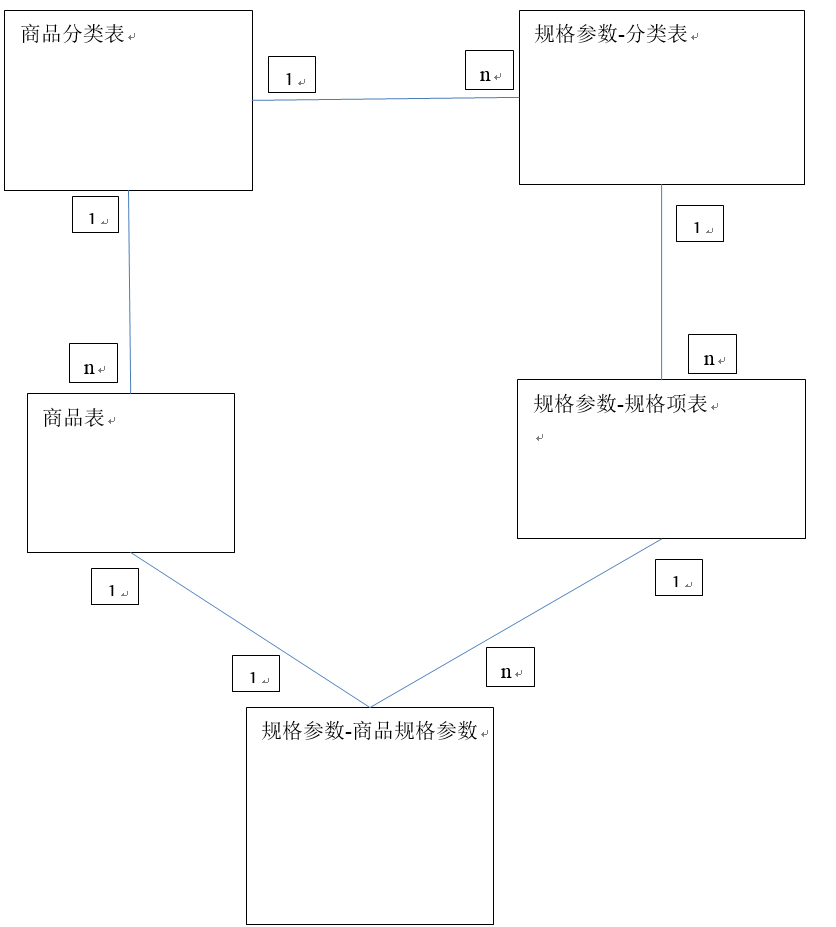
## 分析

规格参数：

1. 同一类商品的规格分类相同。
2. 同一类商品的规格项一致。
3. 不同商品的规格参数不同。

### 思路一：

第一种方式：将数据存储到表中。



当使用这种方式的时候，需要关联的表很多，会给数据库造成很大的压力。不推荐使用此方法。

### 思路二：

将分类表与规格项表采取数据冗余的方式进行数据存储。

1. 可以将规格分组和规格项使用json数据来表示，保存到一个字段中。形成规格参数的模板。
2. 在添加商品(或修改商品)时，读取规格参数的json数据，每添加一个商品根据模板生成表单，供商家输入规格参数。
3. 将动态表单中的内容读取出来生成json数据，保存到数据库中。
4. 展示规格参数时，根据商品id读取规格参数json数据生成html展示到jsp页面。



## 实现方式

### 数据库

1. 数据库参数模板表(商品分类ID) tb\_item\_param
2. 规格参数表(商品ID) tb\_item\_param\_item

## 规格参数模板管理

功能列表

1. 规格参数模板列表
2. 添加规格参数模板
3. 修改规格参数模板
4. 删除规格参数模板

### 规格参数列表展示

从tb\_item\_param中查询数据展示到jsp，单标查询，实现分页。

#### dao层

使用逆向工程生成的代码，新增代码，使其进行多表链接查询。

|  |
| --- |
| <select id="selectItemParam" resultMap="ResultMapWithBLOBs">  SELECT a.id,a.item\_cat\_id,b.name,a.created,a.updated,a.param\_data  FROM tb\_item\_param a,tb\_item\_cat b  WHERE a.id = b.id </select> |

修改resultMap.xml，新增列name，与其对应的为itemCatName。

|  |
| --- |
| <resultMap id="BaseResultMap" type="com.taotao.pojo.TbItemParam" >  <id column="id" property="id" jdbcType="BIGINT" />  <result column="item\_cat\_id" property="itemCatId" jdbcType="BIGINT" />  <result column="created" property="created" jdbcType="TIMESTAMP" />  <result column="updated" property="updated" jdbcType="TIMESTAMP" />  <result column="name" property="itemCatName" /> </resultMap> <resultMap id="ResultMapWithBLOBs" type="com.taotao.pojo.TbItemParam" extends="BaseResultMap" >  <result column="param\_data" property="paramData" jdbcType="LONGVARCHAR" /> </resultMap> |

修改resultMap.java，新增selectItemParam抽象方法。

|  |
| --- |
| List<TbItemParam> selectItemParam(); |

在TbItemParam中，

#### service层

调用dao层，不接受参数，进行分页数据处理，返回EasyUIDataGridResult

对象

#### Controller层

调用service层，不接受参数，以json格式返回List<TbItemParam>对象

### 新增规格参数模板

新增规格参数模板分为两部分

#### 第一部分：避免重复录入

进行catId查询，避免同一类商品多次录入规格参数

##### dao层

调用逆向工程，根据商品类目catId查询是否存在数据，接受商品类目Id，返回商品类目信息

|  |
| --- |
| <select id="selectByCatId" resultMap="ResultMapWithBLOBs" parameterType="java.lang.Long">  select  id, item\_cat\_id, created, updated  from tb\_item\_param  where item\_cat\_id = #{itemCatId,jdbcType=BIGINT} </select> |

##### service层

调用dao层，接受商品类目id，进行数据判断，接受商品类目CatId，返回TaotaoResult

|  |
| --- |
| @Override public TaotaoResult getItemParamById(Long id) {  TbItemParam tbItemParam = itemParamMapper.selectByCatId(id);  if(tbItemParam == null){  return TaotaoResult.*ok*();  }  return TaotaoResult.*build*(500, "商品已存在规格参数"); } |

##### Controller层

接受商品类目CatId，返回TaotaoResult对象

|  |
| --- |
| @RequestMapping("/rest/item/param/{id}") @ResponseBody public TaotaoResult addItemParamList(@PathVariable Long id, String paramData){  System.*out*.println(paramData);  if(paramData == null || paramData.isEmpty()) {  return itemParamService.getItemParamById(id);  }  else{  return itemParamService.saveItemParam(id,paramData);  } } |

#### 第二部分：新增数据

##### dao层

调用逆向工程对应代码

##### service层

|  |
| --- |
| @Override public TaotaoResult saveItemParam(Long id, String paramData) {  Date date = new Date();  TbItemParam itemParam = new TbItemParam();  itemParam.setItemCatId(id);  itemParam.setParamData(paramData);  itemParam.setCreated(date);  itemParam.setUpdated(date);  int insert = itemParamMapper.insert(itemParam);  if(insert == 1){  return TaotaoResult.*ok*();  }  return TaotaoResult.*build*(500,"添加失败"); } |

##### Controller层

@RequestMapping("/rest/item/param/{id}")  
@ResponseBody  
public TaotaoResult addItemParamList(@PathVariable Long id, String paramData){  
 System.*out*.println(paramData);  
 if(paramData == null || paramData.isEmpty()) {  
 return itemParamService.getItemParamById(id);  
 }  
 else{  
 return itemParamService.saveItemParam(id,paramData);  
 }  
}

### 删除规格参数模板

#### dao层

调用逆向工程自动生成的代码即可

#### service层

接受要删除的规格参数字符串数组，调用dao层，删除对应数据

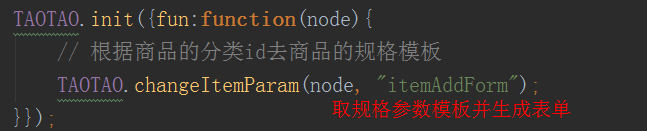
|  |
| --- |
| @Override public TaotaoResult deleteItemParam(Long[] ids) {  // 逆向工程查询条件对象  TbItemParamExample itemParamExample = new TbItemParamExample();  // 条件对象  TbItemParamExample.Criteria criteria = itemParamExample.createCriteria();  // 添加对应条件  criteria.andIdIn(Arrays.*asList*(ids));  // 执行删除  int total = itemParamMapper.deleteByExample(itemParamExample);  // 返回对应信息  return TaotaoResult.*ok*(total); } |

#### Controller层

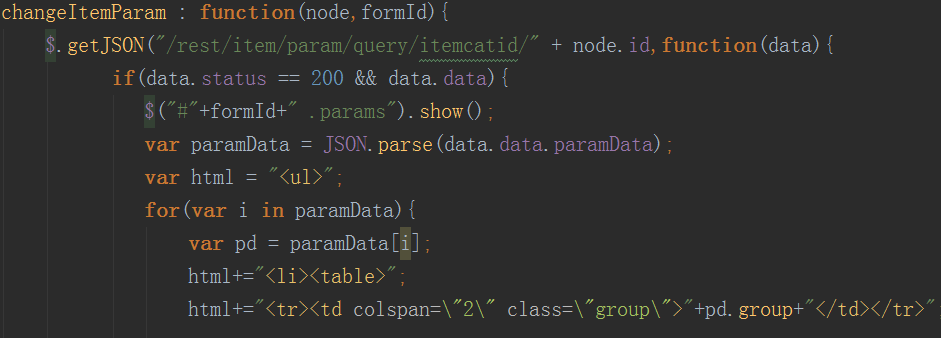
|  |
| --- |
| @RequestMapping("/rest/item/param/delete") @ResponseBody public TaotaoResult removeItemParam(Long[] ids){  return itemParamService.deleteItemParam(ids); } |

# 规格参数模板的使用

分析：应该在商品添加时或者商品修改时，根据商品的分类id查询此商品分类对应的规格参数模板。根据规格参数模板生成一个表单，供用户使用。



当商品类目选择完成后，调用以下方法



## 展示规格参数

从表中把规格参数json数据取处理，可以在java代码中解析json数据，生成html展示到jsp页面。

根据商品id查询规格参数

请求的[url:/item/{itemId}](file:///C:\item\%7bitemId%7d)

返回结果：返回String(逻辑视图)

### Dao层

单表查询，可以使用逆向工程。

### Service层

接受商品ID，根据商品id取规格参数。可以将json转换成java对象。JsonUtils可以完成该步骤。遍历java对象，生成html，返回。

参数：商品id

返回值：字符串(html字符串)。

### Controller层

接受商品id，调用Service查询规格参数，返回html片段。把html片段传递给jsp。

参数：商品id，Model

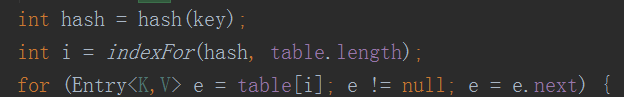
|  |
| --- |
| @RequestMapping("/select/item/{itemId}") @ResponseBody public String showItemParam(@PathVariable Long itemId, Model model){  String html = itemService.getItemParamHtml(itemId);  model.addAttribute("html",html);  return "itemparam"; } |

html事件

要想触发，不外乎，触发事件，触发数据。

一般来说，当程序运行时，会自动加载Map集合，干什么的不知道

但是对于HashMap来说，当HashMap初始化时，会自动生成一个Hash表，这个Hash表中存储着key值经过加密的hash值



这使用了for循环的特性进行了一步简化，相当于在此进行了一个if判断，i值是取hash值，与table长度的位运算

当Map存储数据的时候，会判断Hash表中是否存在着对应的值，若是存在，则将值进行替换，并返回被替换的值，若是不存在，则重新生成一个Entry节点，并在生成时，对Hash表长度进行判断，查看长度是否允许存储数据，若是不足够存储数据，则对hash表进行扩容，扩容是采取遍历

# 淘淘商城第五天

## 第四天内容回顾

商品添加功能的实现

1. 富文本编辑器的使用KindEditor
   1. 引入js
   2. 使用js代码初始化kindEditor，需要一个textarea控件。
   3. 提交表单之前调用sync()方法，同步数据。是富文本编辑器textArea控件同步数据
2. 商品添加功能实现。
   1. 向商品表插入数据
   2. 向商品描述表插入数据
3. 商品的规格参数
   1. 使用模板的参数
   2. 为每类商品创建一个参数模板。
   3. 添加商品时，读取模板，基于模板，生成一个表单，供商家输入规格参数。
   4. 使用js把规格参数转换成json数据，保存到数据库
   5. 读取规格参数，将规格参数转换成html展示到页面。

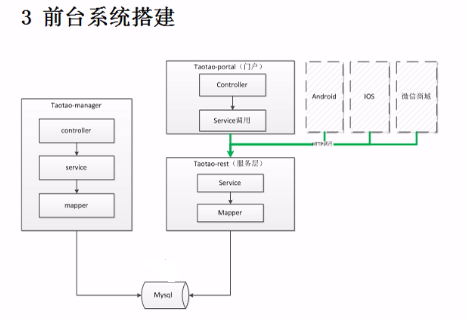
## 课程计划

* 前台系统搭建
  + 服务层
  + 表现层
* 商城首页展示
* 实现商品分类展示功能
* 首页内容管理思路及cms系统

# 前台系统搭建

分布式架构，把不同的模块，组成不同的子系统

## 系统架构



查询商品信息，pc端和移动端有相同的功能模块

可以把业务逻辑提取出来，发布服务，供pc端、移动端公用。

好处：

* 提高代码复用度
* 可以灵活的进行分布式部署
* 系统之间耦合度低
* 团队开发不冲突

缺点：

* 需要发布webService，系统之间需要远程调用
* 工作量增加
* 需要系统之间协作才能完成整个业务流程，系统之间调用变得复杂

需要创建两个工程

服务层：taotao-rest：没有表现层，只有业务逻辑。需要发布服务。

表现层：taotao-portal：只有表现层，没有业务逻辑，不需要连接数据库。

表现层与服务层通信，使用webservice进行通信。Restful形式的webservice。http+json数据

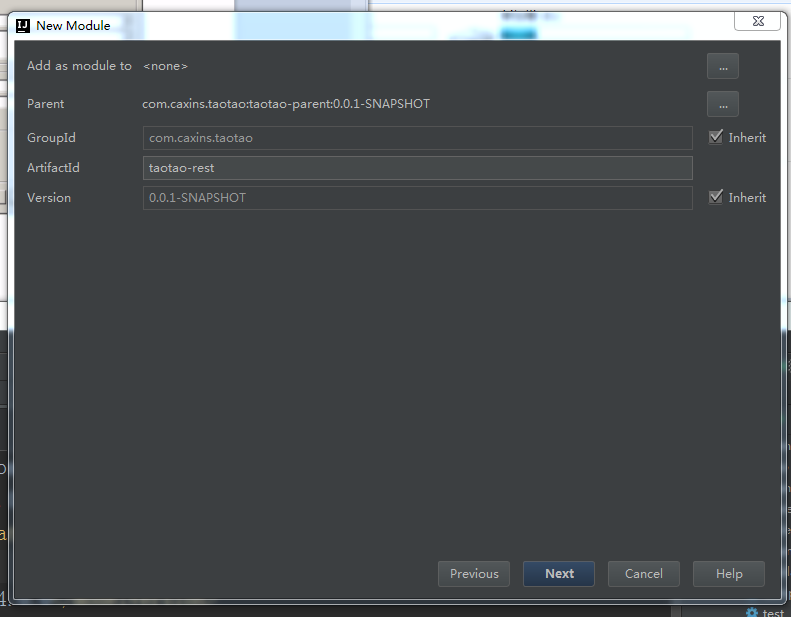
## taotao-rest工程

### 服务层taotao-rest

使用maven管理工程。war包。

使用的技术：

* Mybatis
* Spring
* 发布服务：~~cxf~~、springmvc



### 框架整合

整合ssm框架。参考taotao-manager功能。

Pom文件

添加对taotao-dao的依赖。

### pom文件

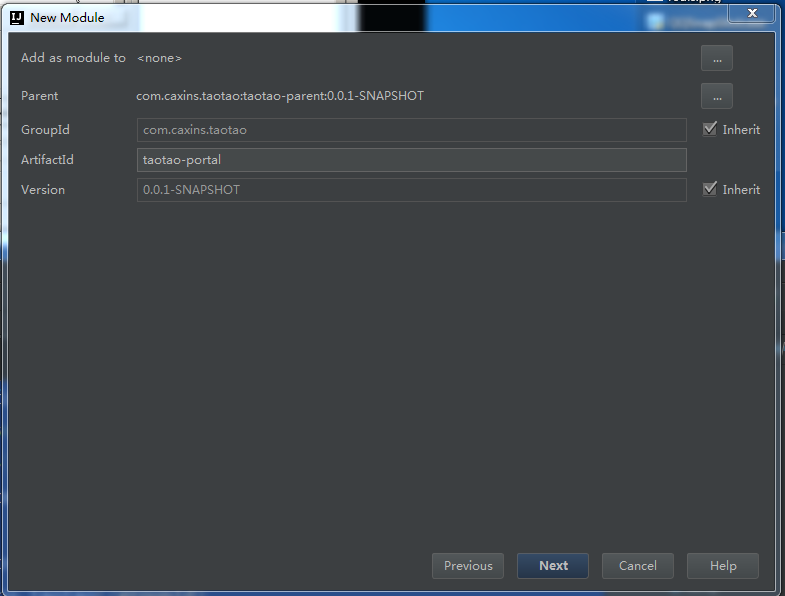
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <parent>  <artifactId>taotao-parent</artifactId>  <groupId>com.caxins.taotao</groupId>  <version>0.0.1-SNAPSHOT</version>  <relativePath>../taotao-parent/pom.xml</relativePath>  </parent>  <modelVersion>4.0.0</modelVersion>   <artifactId>taotao-rest</artifactId>  <packaging>war</packaging>   <dependencies>  <dependency>  <groupId>com.caxins.taotao</groupId>  <artifactId>taotao-dao</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <!-- 添加spring的依赖jar包 -->  <!-- Spring -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  </dependency>  <!-- Servlet依赖，编译时期使用 -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>servlet-api</artifactId>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jsp-api</artifactId>  <scope>provided</scope>  </dependency>  </dependencies> </project> |

## taotao-portal工程

### 使用的技术

* spring
* springmvc

### 创建工程



#### 依赖的jar包

* taotao-pojo
* spring

### pom文件

|  |
| --- |
| <?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">  <parent>  <artifactId>taotao-parent</artifactId>  <groupId>com.caxins.taotao</groupId>  <version>0.0.1-SNAPSHOT</version>  <relativePath>../taotao-parent/pom.xml</relativePath>  </parent>  <modelVersion>4.0.0</modelVersion>   <artifactId>taotao-portal</artifactId>  <packaging>war</packaging>   <dependencies>  <dependency>  <groupId>com.caxins.taotao</groupId>  <artifactId>taotao-pojo</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <!-- Spring -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  </dependency>  <!-- JSP相关 -->  <dependency>  <groupId>jstl</groupId>  <artifactId>jstl</artifactId>  <version>${jstl.version}</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>servlet-api</artifactId>  <version>${servlet-api.version}</version>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jsp-api</artifactId>  <version>${jsp-api.version}</version>  <scope>provided</scope>  </dependency>  </dependencies> </project> |

### 框架整合

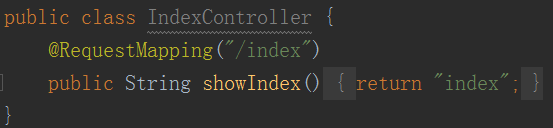
spring和springmvc

## 访问首页

创建一个Controller，当访问<http://localhost:8082/taotao-portal/>时跳转到首页

请求的url:/

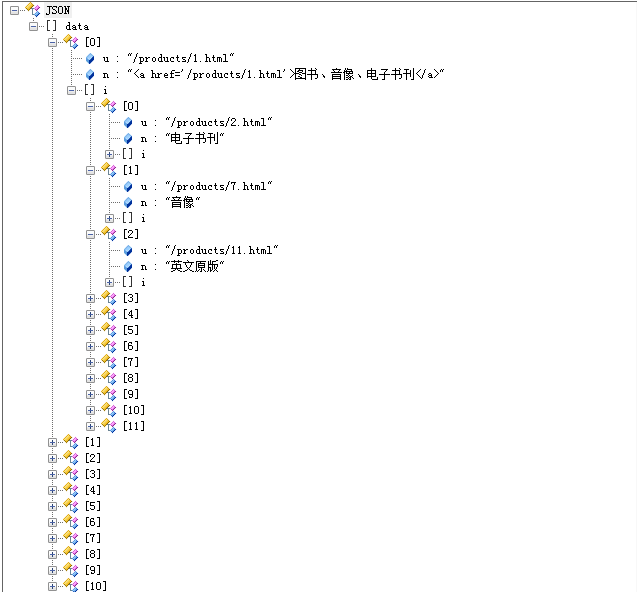
当用户访问“/”时，会跳转至web.xml配置的欢迎页index.html，这时springmvc会将请求拦截。



## 首页商品类目加载

* 类目展示需要异步加载。
* 商品类目从数据库中获得，调用taotao-rest发布的服务获得数据。
* 可以在首页中使用js直接调用taotao-rest发布的服务来获得数据。

需要使用到的json数据格式



## JS跨域

Js为了安全有一个限制，不允许跨域访问

### 跨域情况

1. 两个url的域名不同
2. 域名相同端口不同
3. ip不同也是跨域

js在taotao-portal不能使用ajax直接调用taotao-rest的服务。

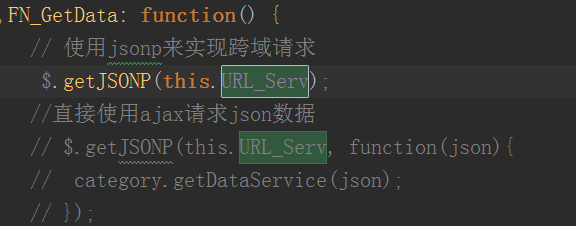
## 解决跨域问题

在js中，跨域使用jsonp解决跨域问题

1. 在js中是不能进行跨域访问数据，但是js是可以跨域请求js片段的。
2. 可以把数据包装成js片段。可以把数据使用js的方法进行包装，形成一条方法的调用语句。
3. 可以使用ajax请求js片段，当js片段一旦浏览器会被立即执行。
4. 可以在浏览器端先创建好回调方法，在回调方法中通过参数就可以获得请求的数据。

### 前期准备

1. 需要把js的回调方法先写好。
2. 做ajax请求时，需要把回调方法的方法名传递给服务端。
3. 服务端结合搜回调方法名，把数据包装好，响应给客户端。



使用getJSONP()，调用URL进行数据调用。

该事件触发由其他事件管理，当事件触发时，会调用getJSONP方法进行跨域请求。

请求的URL中拼接了对应的函数调用，所以未在getJSONP中再拼接函数，也可以是Url中可以删除？与其之后的部分。

方式一



方式二



# 发布商品类目列表查询服务

## 分析

获得一个分类列表的数据。从tb\_item\_cat表中获取数据。

可以创建一个pojo描述树形列表的节点，其中包含三个属性u\n\i。

u:url String

n:name String

i:items List

## 实现

### POJO

在taotao-rest工程里面新建pojo来保存data中的数据

|  |
| --- |
| public class CatNode {   @JsonProperty("u")  private String url;  @JsonProperty("n")  private String name;  @JsonProperty("i")  private List items;  } |

使用JsonProperty注解可以使该类变成json数据时，使用u/n/i进行显示。

生成封装类，将数据封装进入ItemCat中。

|  |
| --- |
| public class ItemCatResult {  private List date;  public List getDate() {return date;}  public void setDate(List date) {this.date = date;} } |

## Dao层

查询表tb\_item\_cat表中数据，单表查询，可以使用逆向工程生成的代码

## Service层

处理业务逻辑

返回结果：ItemCatResult

参数：没有参数

业务逻辑：根据parentid查询子节点列表，并生成递归调用。

## Controller层

没有业务参数，相应一个json数据。需要判断callBack是否为null，如果为空正常返回json数据，如果不为空，支持jsonp的调用。

### Controller支持jsonp的两种方式

第一种方式：直接相应字符串

|  |
| --- |
| @RequestMapping(value="/list",produces = MediaType.*APPLICATION\_JSON\_VALUE*+";charset=utf-8") @ResponseBody public String getItemCatList(String callback){  ItemCatResult result = itemCatService.getItemCatList();  if(StringUtils.*isBlank*(callback)){  // 将result转换成字符串  String json = JsonUtils.*objectToJson*(result);  return json;  }  // 如果字符串不为空，需要直接jsonp调用  String json = JsonUtils.*objectToJson*(result);  return "coallback"+"("+json+");"; } |

第二种方法：

要求springmvc必须是4.1以上版本。

MappingJacksonValue

|  |
| --- |
| @RequestMapping("/list") @ResponseBody public Object getItemCatList(String callback){  ItemCatResult result = itemCatService.getItemCatList();  if(StringUtils.*isBlank*(callback)){  // 将result转换成字符串  return result;  }  // 如果字符串不为空，需要直接jsonp调用  MappingJacksonValue mappingJacksonValue = new MappingJacksonValue(result);  mappingJacksonValue.setJsonpFunction(callback);  return mappingJacksonValue; } |

# 淘淘商城第六天

## 内容复习

1. 前台系统的搭建 处理高并发，进行分布式系统
   1. 服务层 进行数据处理
   2. 表现层 调用服务，展示数据
2. 首页的展示
3. 商品类目展示
   1. 使用ajax跨域调用服务。
   2. 使用jsonp实现。

## 课程计划

1. 首页的内容展示 轮播图展示。
2. cms系统 -- 内容管理系统实现。
3. taotao-rest发布服务。
4. 表现层调用服务，展示轮播图。

## 首页轮播图展示

首页内容管理思路

网页的所有内容要随着后台的变化而变化。

1. 内容的分组
2. 分组下还可以有小组
3. 广告项目中有图片，有标题，价格，链接，内容

第一块：内容的分组管理，树形结构。

第二块：内容管理。

内容下。有图片、链接、标题、价格、内容。

使用两个表来保存内容数据。

1. 内容分类表
2. 内容表



## 内容分类管理

### 展示分类列表

#### 分析

初始化easyUI的tree控件，url：/content/content/category

请求的参数: id，父节点ID

返回结果：json数据(EasyUITreeNode)

是一个列表，列表中每个元素包含三个属性：

1. id
2. state
3. state

### Dao层

从tb\_conten\_category表中取数据。根据parentid查询分类列表。可以使用逆向工程生成的代码。

### Service层

接受一个parentid，根据parentID查询节点列表。创建一个EasyUITreeNode列表。返回。

参数：Long parentId

返回值：List<EasyUITreeNode>

### Controller层

接受parentId调用Service查询节点列表，返回节点列表。返回json数据，需要使用ResponseBody

## 新增节点

分析：





请求的url: /content/category/create

请求的参数：parentId、name

返回的结果：返回TaotaoResult，包含新添加记录的ID

### Dao层

插入数据可以使用逆向工程。

### Service层

1. 接受两个参数：parentId、name
2. 创建一个对应tb\_content\_category表pojo对象的属性。
3. 补全pojo对象的数据
4. 插入数据
5. 需要mybatis返回主键
6. 使用TaotaoResult包装id，返回。

### Controller

接受两个参数，parentId、name。调用Service插入数据，返回TaotaoResult

## 修改节点

### Dao层

调用逆向工程生成的代码集合

### Service层

参数：当前节点的ID，与修改后的Name值

返回值：TaotaoResult

### Controller层

接受两个参数，调用service层，返回TaotaoResult

## 删除节点

### Dao层

直接使用逆向工程生成的代码

### Service层

需要判断是否是叶子节点，若是叶子节点，则需要判断父节点是否还存在子节点，若是不存在子节点，则需要将其设置为子节点。

需要判断是否是子节点，若是是子节点，需要使用递归的方式删除其下所有节点。

### Controller层

接受参数，调用service层

# 内容管理

## 内容列表

### Dao层

使用逆向工程生成的代码。

### Service层

接受三个参数：page,rows, categoryId

返回：TaotaoResult

进行分页查询

### Controller层

接受三个参数：page,rows, categoryId

返回：TaotaoResult

## 内容新增

### Dao层

可以使用tbContent生成的逆向工程代码

### Service层

接受一个TbContent对象并将其保存到数据库，返回一个EasyUIResult

### Controller层

接受一个TbContent对象，并调用Service

# 轮播图的展示

可以使用jsonp获得数据。做网站不推荐大面积使用ajax请求。

不使用ajax，如何获得数据？

只需要再portal的service调用taotao-rest服务，获得数据，将数据传递jsp即可。

要调用服务，需要使用java代码模拟浏览器调用服务请求数据。可以使用HttpClient。

## HttpClient

在java代码中模拟http请求。

### get请求方式

第一步：把HttpClient使用的jar包添加到工程中。

第二步：创建一个HttpClient的测试类。

第三步：创建测试方法。

第四步：创建一个HttpClient对象。

第五步：创建一个HttpGet对象，需要指定一个请求的url。

第六步：执行请求。

第七步：接受返回结果。HttpEntity对象。

第八步：获取的响应的内容。

第九步：关闭HttpGet、HttpClient

### post请求方式

实现步骤：

第一步：创建一个httpClient对象

第二步：创建一个HttpPost对象。需要指定一个URL

第三步：创建一个list模拟表单，list中每个元素都是一个NameValuePair对象

第四步：需要把表单包装到Entity对象中。StringEntity

第五步：执行请求。

第六步：接受返回结果

第七步：关闭流

### 提交表单和提交json数据的区别

content-Type不同

提交表单的content-Type：application/x-www-form-urlencoded

提交json的content-Type：application/json

实际使用时，需要把HttpClient封装成一个工具类。可以将工具类放到taotao-common中

## 在rest中发布服务

### 发布内容查询服务。

#### 分析

根据内容分类id查询tb\_content表。得到此分类下的内容列表，返回即可。

请求的url：/rest/content/{cid}

返回的数据格式：json数据

1. 包含查询的内容列表
2. 包含状态
3. 包含错误信息

使用TaotaoResult包装一个内容列表即可。

#### dao层

可以使用逆向工程。

#### Service层

接受内容分类id作为参数，使用mapper根据分类id查询内容列表，返回一个内容列表。

参数：Long cid

返回：内容列表

## 在taotao-portal中使用HttpClient调用服务获得数据

## 展示轮播图

# 淘淘商城第七天

## 第六天内容复习

### 轮播图的展示

* 内容管理的后台。CMS系统
  + 内容分类的管理。
  + 内容管理，需要指定内容的分类。
* 前台展示内容
  + 需要服务层发布服务。
  + Portal调用服务获得数据。
  + 使用HttpClient调用服务获得数据。
  + 获得数据后，把数据转换成需要的数据格式传递给jsp。

## 课程计划

* 在业务逻辑中，添加缓存
  + 缓存可以使用redis作为缓存。
  + redis集群。
  + 如何在java代码中使用redis单机版、集群版。
  + 在业务逻辑中添加缓存。
* 进行搜索系统的实现
  + 创建搜索的工程。
  + solr实现搜索。
  + 搭建solr集群

## 缓存的添加

搭建redis。

### redis的单机版

第一步：安装gcc编译环境

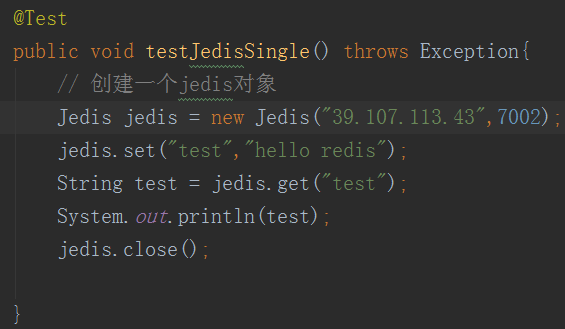
第二步：将redis的源码上传到linux服务器。

## 如何使用redis的java客户端

需要使用jedis连接redis服务器。

### 单机版

#### 直接连接



#### 连接池版

|  |
| --- |
| @Test public void testJedisPool() throws Exception{  // 创建一个连接池对象  // 系统中应该是单例的。  JedisPool jedisPool = new JedisPool("39.107.113.43",7001);  // 从连接池中获得一个连接  Jedis jedis = jedisPool.getResource();  String result = jedis.get("test");  System.*out*.println(result);  // jedis必须要关闭  jedis.close();  // 当系统关闭时关闭连接池  jedisPool.close(); } |

### 集群版

|  |
| --- |
| @Test public void testJedisCluster() throws Exception{  // 创建一个JedisCluster对象  Set<HostAndPort> nodes = new HashSet<>();  nodes.add(new HostAndPort("39.107.113.43",7001));  nodes.add(new HostAndPort("39.107.113.43",7002));  nodes.add(new HostAndPort("39.107.113.43",7003));  nodes.add(new HostAndPort("39.107.113.43",7004));  nodes.add(new HostAndPort("39.107.113.43",7005));  nodes.add(new HostAndPort("39.107.113.43",7006));  // 在nodes中指定每个节点的地址  // jedisCluster在系统中是单例的  JedisCluster cluster = new JedisCluster(nodes);  String set = cluster.set("name", "zhangsan");  String name = cluster.get("name");  System.*out*.println(name);  // 系统关闭时关闭连接池  cluster.close(); } |

# jedis与spring的整合

## 在项目中使用jedis

思路：创建一个redis操作的接口。分别创建两个实现类对应redis的单机版和集群版。当使用单机版redis时，就配置值单机版的实现类，当使用集群版的时候，配置集群版的实现类。

## spring配置

单机版与集群版

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:p="http://www.springframework.org/schema/p"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-4.0.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop-4.0.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx-4.0.xsd  http://www.springframework.org/schema/util  http://www.springframework.org/schema/util/spring-util-4.0.xsd">  <!-- 配置包扫描器 -->  <context:component-scan base-package="com.taotao.rest.service" />   <!-- 配置redis客户端单机版 -->  <!-- 创建一个jedisPool -->  <bean id="jedisPool" class="redis.clients.jedis.JedisPool">  <constructor-arg name="host" value="39.107.113.43"></constructor-arg>  <constructor-arg name="port" value="6379"></constructor-arg>  </bean>   <!-- 配置redis客户端实现类 -->  <bean id="jedisClientSingle" class="com.taotao.rest.compoent.impl.JedisClientSingle"/>   <!-- 配置redis集群版实现类 -->  <!--<bean id="jedisCluster" class="redis.clients.jedis.JedisCluster">  <constructor-arg>  <set>  <bean class="redis.clients.jedis.HostAndPort">  <constructor-arg name="host" value="39.107.113.43"></constructor-arg>  <constructor-arg name="port" value="7001"></constructor-arg>  </bean>  <bean class="redis.clients.jedis.HostAndPort">  <constructor-arg name="host" value="39.107.113.43"></constructor-arg>  <constructor-arg name="port" value="7002"></constructor-arg>  </bean>  <bean class="redis.clients.jedis.HostAndPort">  <constructor-arg name="host" value="39.107.113.43"></constructor-arg>  <constructor-arg name="port" value="7003"></constructor-arg>  </bean>  <bean class="redis.clients.jedis.HostAndPort">  <constructor-arg name="host" value="39.107.113.43"></constructor-arg>  <constructor-arg name="port" value="7004"></constructor-arg>  </bean>  <bean class="redis.clients.jedis.HostAndPort">  <constructor-arg name="host" value="39.107.113.43"></constructor-arg>  <constructor-arg name="port" value="7005"></constructor-arg>  </bean>  <bean class="redis.clients.jedis.HostAndPort">  <constructor-arg name="host" value="39.107.113.43"></constructor-arg>  <constructor-arg name="port" value="7006"></constructor-arg>  </bean>  </set>  </constructor-arg>  </bean>  <bean id="jedisClientCluster" class="com.taotao.rest.compoent.impl.JedisClientCluster"/> --> </beans> |

# 业务逻辑中添加缓存

注意：添加缓存的时候不要影响正常的业务逻辑。

添加缓存

|  |
| --- |
| @Service public class ContentServiceImpl implements ContentService {   @Autowired  private TbContentMapper contentMapper;  @Autowired  private JedisClient jedisClient;   @Value("${REDIS\_CONTENT\_KEY}")  private String REDIS\_CONTENT\_KEY;   @Override  public List<TbContent> getContentList(Long cid) {  try {  // 添加缓存  // 查询数据库之前先查询缓存，如果有直接返回  String json = jedisClient.hget(REDIS\_CONTENT\_KEY, cid + "");  if(!StringUtils.*isBlank*(json)){  List<TbContent> contentList = JsonUtils.*jsonToList*(json, TbContent.class);  return contentList;  }  } catch (Exception e){  e.printStackTrace();  }  // 根据cid查询内容列表  TbContentExample example = new TbContentExample();  TbContentExample.Criteria criteria = example.createCriteria();  criteria.andCategoryIdEqualTo(cid);  List<TbContent> contentList = contentMapper.selectByExampleWithBLOBs(example);  // 返回结果之前，向缓存中添加数据  try{  // 向缓存中添加数据  // 为了规范key,可以使用hash  // 定义一个保存内容的key，hash中每个项就是cid  // value是list，需要把list转换成json数据  jedisClient.hset(REDIS\_CONTENT\_KEY,cid+"", JsonUtils.*objectToJson*(contentList));  }catch (Exception e){  e.printStackTrace();  }  return contentList;  } } |

## 缓存同步

当后台修改内容信息后，就应该同步缓存，只需要把redis中缓存的数据删除即可。

后台系统不直接操作redis数据库。

可以在taotao-rest中发布一个服务，当后台对内容信息修改后，调用服务即可。

服务的功能就是根据cid删除redis中缓存数据。

## 发布服务

根据cid删除redis。

### Dao层

使用JedisClient进行实现。

### service层

接受cid，根据cid调用JedisClient删除redis中缓存的数据。

返回结果：TaotaoResult。

### Controller层

发布服务，接受参数cid，返回结果TaotaoResult

# 搜索系统

搜索功能需要发布服务，供PC端、移动端使用，根据关键词进行搜索，得到json格式的搜索结果。

搭建一个搜索服务系统，发布搜索服务。

系统架构：



* 创建一个搜索工程：taotao-search
* 搭建solr服务

## 创建搜索系统

可以参考taotao-rest创建

使用的技术：

* MyBatis
* Spring
* Spring-mvc(发布服务)
* SolrJ(solr服务的客户端)

ssm整合

# Solr服务搭建

在Liunx系统下搭建solr服务。

* 需要安装tomcat
* 安装jdk

## Linux单机版安装步骤

第一步：安装jdk、安装tomcat

第二步：解压solr的压缩包

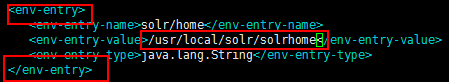
第三步：把dist/solr-4.10.3.war部署到tomcat下。

第四步：启动tomcat，将war包解压缩

第五步：需要/solr-4.10.3/example/lib/ext目录下所有的jar包添加到/usr/local/solr/tomcat/webapps/solr/WEB-INF/lib/下

第六步：创建solr-home，将/solr-4.10.3/example/solr复制一份作为solrhome

第七步：告诉solr服务solrhome的位置。需要修改web.xml，修改以下部分



第八步：启动tomcat。访问地址即可。

配置中文分析器、自定义业务域

## 配置中文分析器

分析器使用IKAnalyzer。

使用方法：

第一步：把IKAnalyzer依赖的jar包添加到solr工程中。把分析器使用的扩展词典添加到classpath中。

第二步：需要自定义一个FieldType，schema.xml中定义。可以在FieldType中指定中文分析器。

第三步：自定义域。指定域的类型为自定义的FieldType。



|  |
| --- |
| <fieldType name="text\_ik" class="solr.TextField">  <analyzer class="org.wltea.analyzer.lucene.IKAnalyzer"/>  </fieldType> |

第四步：自定义域。指定域的类型为自定义的FieldType。

设置业务系统Filed

### SQL语句

SELECT

a.id,

a.title,

a.sell\_point,

a.price,

a.image,

b.`name` category\_name,

c.item\_desc

FROM

tb\_item a

LEFT JOIN tb\_item\_cat b ON a.cid = b.id

LEFT JOIN tb\_item\_desc c ON a.id = c.item\_id

WHERE

a.`status` = 1

### Field域设置

|  |
| --- |
| <field name="item\_title" type="text\_ik" indexed="true" stored="true"/>  <field name="item\_sell\_point" type="text\_ik" indexed="true" stored="true"/>  <field name="item\_price" type="long" indexed="true" stored="true"/>  <field name="item\_image" type="string" indexed="false" stored="true" />  <field name="item\_category\_name" type="string" indexed="true" stored="true" />  <field name="item\_desc" type="text\_ik" indexed="true" stored="false" />  <field name="item\_keywords" type="text\_ik" indexed="true" stored="false" multiValued="true"/>  <copyField source="item\_title" dest="item\_keywords"/>  <copyField source="item\_sell\_point" dest="item\_keywords"/>  <copyField source="item\_category\_name" dest="item\_keywords"/>  <copyField source="item\_desc" dest="item\_keywords"/> |

第四步：从新启动tomcat

## 索引库中导入数据

### SolrJ的使用

|  |
| --- |
| @Test public void testSolrJ() throws Exception {  // 创建连接  SolrServer solrServer = new HttpSolrServer("http://39.107.113.43:8080/solr");  // 创建一个文档对象  SolrInputDocument document = new SolrInputDocument();  // 添加域  document.addField("id", "solrtest01");  document.addField("item\_title", "测试商品");  document.addField("item\_sell\_point", "卖点");  // 添加到索引库  solrServer.add(document);  // 提交  solrServer.commit(); }  @Test public void testQuery() throws Exception{  // 创建连接  SolrServer solrServer = new HttpSolrServer("http://39.107.113.43:8080/solr");  SolrQuery query = new SolrQuery();  query.setQuery("\*:\*");  QueryResponse response = solrServer.query(query);  SolrDocumentList results = response.getResults();  for(SolrDocument solrDocument : results){  System.*out*.println(solrDocument.get("id"));  System.*out*.println(solrDocument.get("item\_title"));  System.*out*.println(solrDocument.get("item\_sell\_point"));  } } |

### 分析

从数据库中根据SQL语句查询数据，遍历数据创建文档对象，将文档对象写入索引库。

### DAO层

SQL语句

SELECT

a.id,

a.title,

a.sell\_point,

a.price,

a.image,

b.`name` category\_name,

c.item\_desc

FROM

tb\_item a

LEFT JOIN tb\_item\_cat b ON a.cid = b.id

LEFT JOIN tb\_item\_desc c ON a.id = c.item\_id

WHERE

a.`status` = 1

需要创建一个mapper文件

使用redis进行session缓存，session在服务器中有sessionid存在，以此为键并在设置其生效时间，在指定时间过后，session自动销毁，数据不再存在。

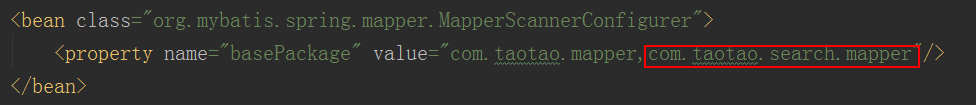
#### 创建Mapper接口

|  |
| --- |
| public interface ItemMapper {  List<SearchItem> getItemList(); } |

#### 创建Mapper.xml文件

|  |
| --- |
| <mapper namespace="com.taotao.search.mapper.ItemMapper">  <select id="getItemList" resultType="com.taotao.search.ItemPojo.SearchItem">  SELECT  a.id,  a.title,  a.sell\_point,  a.price,  a.image,  b.`name` category\_name,  c.item\_desc  FROM  tb\_item a  LEFT JOIN tb\_item\_cat b ON a.cid = b.id  LEFT JOIN tb\_item\_desc c ON a.id = c.item\_id  WHERE  a.`status` = 1  </select> </mapper> |

#### 修改spring-dao中的mapper扫描文件地址



### Service层

取商品列表，遍历列表，创建文档对象，把文档对象写入索引库。

要操作索引库需要SolrService对象，可以将SolrService放到spring容器中，注入到Service。

### Controller

请求一个url，返回TaotaoResult

|  |
| --- |
| @Controller public class ItemController {   @Autowired  private ItemService itemService;   @RequestMapping("/importall")  @ResponseBody  public TaotaoResult importAll(){  try {  TaotaoResult result = itemService.importItems();  return result;  } catch (Exception e) {  e.printStackTrace();  return TaotaoResult.*build*(500,ExceptionUtil.*getStackTrace*(e));  }  } } |

# 淘淘商城第八天

## 第七天内容回顾

* 向业务逻辑中添加缓存
  + Redis做缓存工具
  + 添加缓存不能影响正常的业务逻辑
  + 查询流程：
    - 从缓存中获取内容，如果存在直接返回
    - 如果没有查询数据库
    - 将查询结果添加到缓存
  + 缓存同步的问题
    - 发布服务
    - 当商品在后台系统中修改了内容信息后，直接调用服务，清空缓存
* 搜索系统的搭建
* 使用solr来实现搜索。
  + Linux系统下安装solr
  + 安装中文分析器及添加业务域
  + 将数据库的内容导入到索引库中

## 课程计划

* 搜索系统的实现
  + 发布搜索的服务
  + 在portal中调用搜索服务实现搜索功能
* 搭建solr集群

# 搜索系统的实现

## 搜索服务发布

调用服务传递过来一个查询条件，根据查询条件进行查询。返回查询结果。参数中应该包括分页条件。

参数：String queryString int page int rows;

返回结果：返回json数据。

包含查询结果的列表。使用商品的pojo来描述。SearchItem来描述商品的列表。

包含查询结果的总记录数。

包含查询结果的总页数。

包含当前的页码。

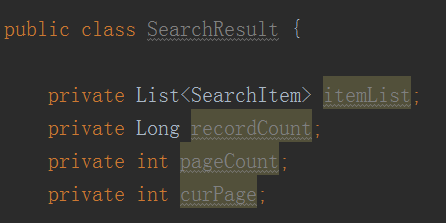
包含查询结果的状态。

包含错误信息。

## 创建一个SearchResult

包含三个属性：

* 商品列表
* 查询结果的总记录数据
* 查询结果的总页数
* 当前页码



使用TaotaoResult包装一个SearchResult返回。

## Dao层

根据查询条件进行查询，返回查询结果。

参数：SolrQuery对象

返回结果：

* 查询结果的商品列表
* 查询结果的总记录数

返回SearchResult

|  |
| --- |
| @Override public SearchResult search(SolrQuery solrQuery) throws SolrServerException {  // 执行查询  QueryResponse response = solrServer.query(solrQuery);  // 获取查询结果列表  SolrDocumentList solrDocuments = response.getResults();  List<SearchItem> itemList = new ArrayList<>();  for (SolrDocument solrDocument : solrDocuments){  SearchItem item = new SearchItem();  item.setCategoryName(solrDocument.get("item\_category\_name").toString());  item.setId(solrDocument.get("id").toString());  item.setImage(solrDocument.get("item\_image").toString());  item.setPrice(Long.*parseLong*(solrDocument.get("item\_price").toString()));  item.setSellPoint(solrDocument.get("item\_sell\_point").toString());  // 获取高亮显示 Map<String, Map<String, List<String>>> highlighting = response.getHighlighting(); List<String> list = highlighting.get(solrDocument.get("id")).get("item\_title");  String itemTitle = "";  if(list != null && list.size() > 0){  // 获取高亮后的结果  itemTitle = list.get(0);  }else{  itemTitle = solrDocument.get("item\_title").toString();  }  item.setTitle(itemTitle);  // 添加到列表  itemList.add(item);  }  SearchResult result = new SearchResult();  result.setItemList(itemList);  // 查询结果总数数量  result.setRecordCount(solrDocuments.getNumFound());  return result; } |

## Service层

* 接受查询条件、分页条件。
* 创建SolrQuery对象，设置查询条件和分页条件。
* 调用dao进行搜索
* 计算总页数，把总页数设置到SearchResult对象，设置当前页属性
* 返回SearchResult

参数：

* + 1. 查询条件
    2. Page
    3. Rows

返回结果SearchResult

|  |
| --- |
| @Override public SearchResult search(String queryString, int page, int rows) throws SolrServerException {  // 创建查询条件  SolrQuery query = new SolrQuery();  // 设置查询条件  query.setQuery(queryString);  // 设置分页条件  query.setStart((page-1)\*rows);  query.setRows(rows);  // 设置默认搜索域  query.set("df","item\_title");  // 设置高亮  query.setHighlight(true);  query.addHighlightField("item\_title");  query.setHighlightSimplePre("<font class=\"skcolor\_ljg\">");  query.setHighlightSimplePost("</font>");  // 执行查询  SearchResult result = searchDao.search(query);  // 计算总页数  Long recordCount = result.getRecordCount();  int pageCount = (int)((recordCount+rows-1) / rows);  // 设置分页条件  result.setPageCount(pageCount);  result.setCurPage(page);  return result; } |

## Controller层

使用Controller发布服务

搜索服务的url：/search/q?keyword=xxx&page=1&rows=30

参数：keyword、page、rows

返回的结果：json数据，使用TaotaoResult包装SearchResult。

|  |
| --- |
| @Autowired private SearchService searchService;  @RequestMapping("/q") @ResponseBody public TaotaoResult search(@RequestParam(defaultValue = "") String keyword,  @RequestParam(defaultValue = "1") Integer page,  @RequestParam(defaultValue = "30") Integer rows){  try {  // 转换字符集  keyword = new String(keyword.getBytes("ISO-8859-1"), "UTF-8");  SearchResult result = searchService.search(keyword, page, rows);  return TaotaoResult.*ok*(result);  } catch (Exception e) {  e.printStackTrace();  return TaotaoResult.*build*(500, ExceptionUtil.*getStackTrace*(e));  } } |

# 在Portal中实现

调用taotao-search发布的服务，实现搜索。使用HttpClient调用服务。返回json数据。需要把json转换成java对象。把Java对象传递给页面。

## 分析

请求的URL：<http://www.taotao.com/search.html>

参数：q:查询的条件

返回的结果：jsp页面

Search.jsp分析：

数据

Query：查询条件

totalPage：总页数

itemList：商品列表(每个元素可以是SearchItem)

page:当前页

## Service层

参数：查询条件、page、rows

根据查询调用taotao-search发布的服务，查询服务列表。得到json数据，需要把json数据转换成Java对象，将Java对象，返回SearchResult对象

|  |
| --- |
| @Override public SearchResult search(String keyword, int page, int rows) {  // 调用服务查询商品列表  HashMap<String, String> param = new HashMap<>();  param.put("keyword",keyword);  param.put("page",page+"");  param.put("rows",rows+"");  // 调用服务  String json = HttpClientUtil.*doGet*(SEARCH\_BASE\_URL, param);  // 转换成java对象  TaotaoResult taotaoResult = TaotaoResult.*formatToPojo*(json, SearchResult.class);  // 获取返回的结果  SearchResult searchResult = (SearchResult) taotaoResult.getData();  return searchResult; } |

## Controller层

接受三个参数：查询条件、page、rows

调用服务查询商品列表。

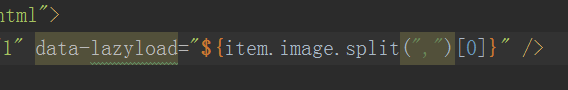
把商品列表传递给jsp、参数回显。

返回逻辑视图(search.jsp)

请求的Url：/search

|  |
| --- |
| @RequestMapping("/q") @ResponseBody public TaotaoResult search(@RequestParam(defaultValue = "") String keyword,  @RequestParam(defaultValue = "1") Integer page,  @RequestParam(defaultValue = "30") Integer rows){  try {  // 转换字符集  keyword = new String(keyword.getBytes("ISO-8859-1"), "UTF-8");  SearchResult result = searchService.search(keyword, page, rows);  return TaotaoResult.*ok*(result);  } catch (Exception e) {  e.printStackTrace();  return TaotaoResult.*build*(500, ExceptionUtil.*getStackTrace*(e));  } } |

## 解决图片显示不出来的问题



Http服务是最好跨平台服务，使用Http请求可以使多个界面之间进行数据共享。

对于不同平台的通讯，如Redis，感觉实际上也是服务器的一种变形。Tomcat平台使用的是C语言开发，使用Tomcat服务器实际上也是跨平台开发的一种。

索引是将数据库表中的某个字段就进行hash算法转换之后存储成为一个二叉树，在查找时更为方便，但是当频繁插入与删除时，二叉树结构被频繁破坏，需要重新构建，从而影响性能。

数据库存储不再使用平衡二叉树，而直接生成一颗指定节点数量的平衡二叉树，本二叉树极少变动，从而提高插入和查询效率。应对数据量的不同，会对二叉树进行扩展，在这种基础上，节点进行类索引式二叉树建立

# Solr集群(SolrCloud)

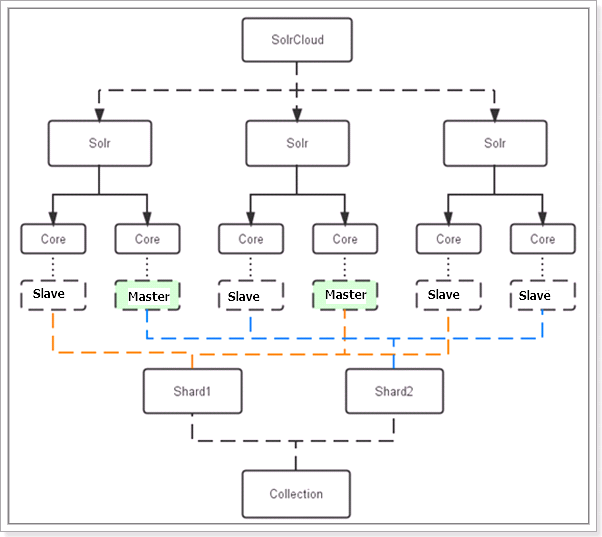
## Solr集群的架构

* SolrCloud(Solr云)是Solr提供的分布式搜索方案，当需要大规模、容错、分布式索引和检索能力时使用SolrCloud。
* SolrCloud是基于Solr和Zookeeper的分布式搜索方案，它的主要思想是使用Zookeeper作为集群的配置信息中心。
* 特点如下：
  + 集中式的配置信息
  + 自动容错
  + 近实时搜索
  + 查询时自动负载均衡
* Zookeeper是一个分布式的、开源的程序协调服务，是hadoop项目下的一个子项目。

## SolrCloud结构

SolrCloud为了降低单机的处理压力，需要由多台服务器共同来完成索引和搜索任务。实现的思路是将索引数据进行Shard(分片)拆分，每个分片由多个服务器共同完成，当一个索引或搜索轻轻过来时，会分别从不同的Shard的服务器中操作索引。

SolrCloud需要Solr基于Zookeeper你是，Zookeeper是一个集群管理软件，由于SolrCloud需要由多台服务器组成 ，由zookeeper进行协调管理。





## 要完成的集群结构



## Zookeeper

1. 集群管理
   1. 主从管理
   2. 负载均衡
   3. 高可用管理
   4. 集群的入口
   5. Zookeeper必须是集群才能保证高可用。
   6. Zookeeper有选举和投票的机制。集群中至少应该有三个节点。
2. 配置文件的集中管理
   1. 搭建Solr集群时，需要把Solr的配置文件上传到Zookeeper，让Zookeeper统一管理。每个节点都到zookeeper上读取配置文件
3. 分布式锁

## 集群需要的服务器

Zookeeper：3台

Solr：4台

伪分布式，zookeeper三个实例、tomcat(Solr)需要4个实例

Zookeeper需要安装jdk。

## 集群搭建的步骤

### 第一部分：Zookeeper集群

* 需要把zookeeper的安装包上传到服务器。
* 第二步解压缩



* 创建Zookeeper文件夹



* 复制三份Zookeeper



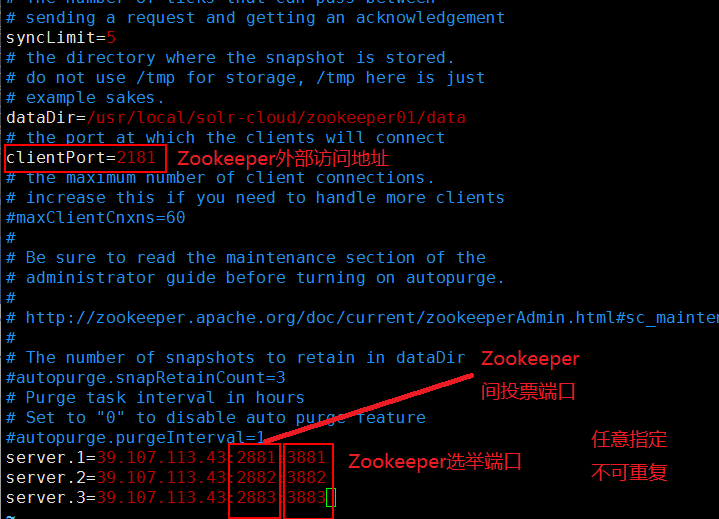
* 配置zookeeper
  + 在Zookeeper目录下创建一个data文件夹
  + 在data目录下创建一个myid的文件
  + myid的内容为1(02对应“2”，03对应3)
  + Zookeeper02、Zookeeper03以此类推
  + 进入conf文件夹 将zoo\_sample.cfg文件改名为zoo.cfg
  + 编辑zoo.cfg文件，将dataDir属性指定为刚创建的data文件夹
  + 编辑zoo.cfg文件，将clientPort指定为不冲突的端口号(01:2181:02:2182:03:2183)
  + 在zoo.cfgz中添加如下内容：

server.1=39.107.113.43:2881:3881

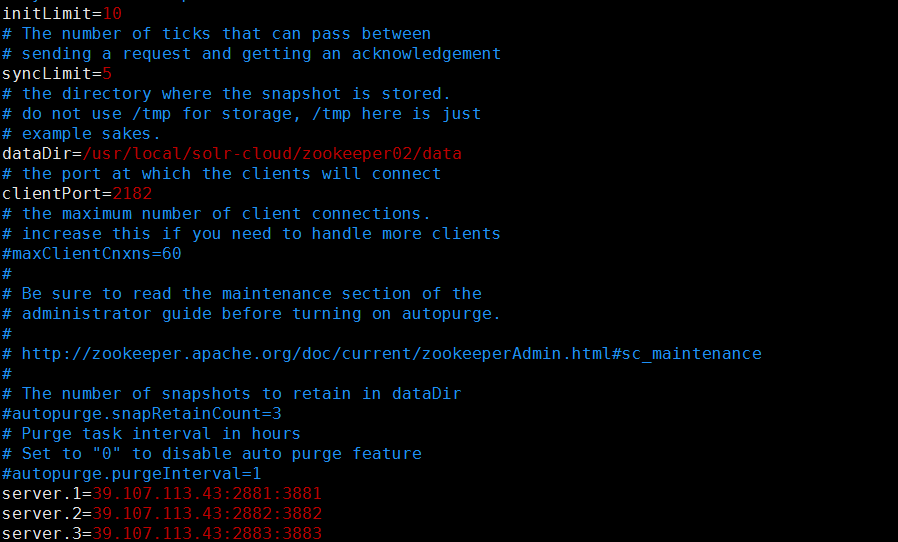
server.2=39.107.113.43:2882:3882

server.3=39.107.113.43:2883:3883

Zookeeper01下conf/zoo.cfg文件如下



Zookeeper02下conf/zoo.cfg文件如下

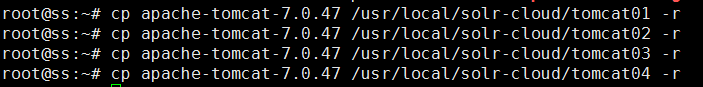


zookeeper03配置文件同理

* + 启动zookeeper
    - Zookeeper的目录下有一个bin目录。使用zkServer.sh来启动zookeeper服务器。
    - 启动 ./zkServer.sh start
    - 关闭 ./zkServer.sh stop
    - 查看服务状态 ./zkServer/sj status
    - 编辑文件用来一次启动所有zookeeper
      * 

### 第二部分：搭建Solr集群吧

* 第一步：安装四个Tomcat，修改器端口号不能冲突。9000-9003

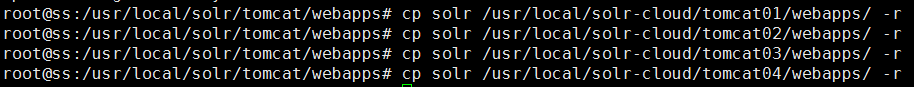


* 分别修改其端口号

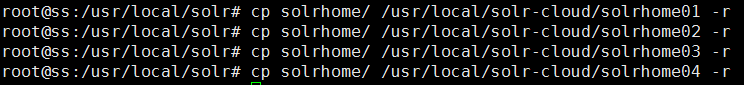
共修改3个port接口

9000-9002 9003-9005 9006-9008 9009-9011

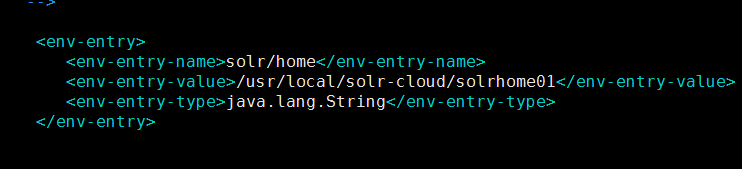
* 向Tomcat下部署Solr。将单机版的Solr工程复制到web下即可



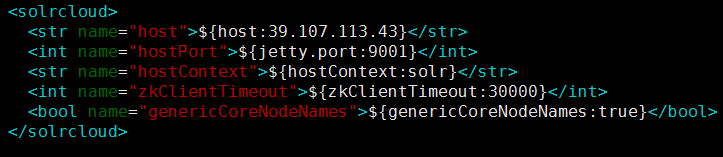
* 为每个solr实例创建一个solrHome



* 为每个solr实例关联对应的solrhome，修改web.xml



* 修改每个solrHome下的solr.xml文件。修改host与hostPort两个属性。分别是对应的ip和端口号



* 将配置文件上传到Zookeeper。需要使用solr原始文件夹中/root/solr-4.10.3/example/scripts/cloud-scripts下的zkcli.sh上传配置文件传递
* 将solrhome01下的/usr/local/solr-cloud/solrhome01/collection1/conf目录上传到zookeeper。需要Zookeeper集群已经启动
* 执行以下命令

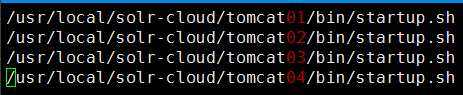
|  |
| --- |
| ./zkcli.sh -zkhost 172.17.90.96:2181 172.17.90.96:2182 172.17.90.96:2183 -cmd upconfig -confdir /usr/local/solr-cloud/solrhome01/collection1/conf -confname myconf |

* 查看是否上传成功，使用zookeeper的zkClient命令
  + /usr/local/solr-cloud/zookeeper01/bin/zkCli.sh ll /configs/myconf
* 告诉solr实例zookeeper的位置。需要修改tomcat的catalina.sh添加

JAVA\_OPTS=”-DzkHost=39.107.113.43:2181,39.107.113.43:2182,39.107.113.43:2183”

每个solr节点都需要添加。在JAVA\_OPTS第一次使用之前

* 启动每个solr实例。设置启动的sh脚本。并赋予权限

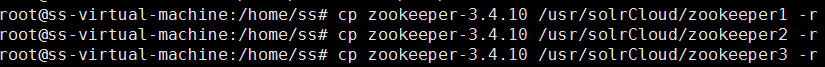




启动tomcat实例

# Solr集群搭建

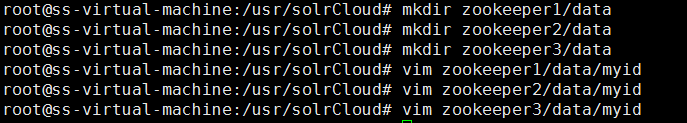
复制zookeeper



复制zoo.cfg文件

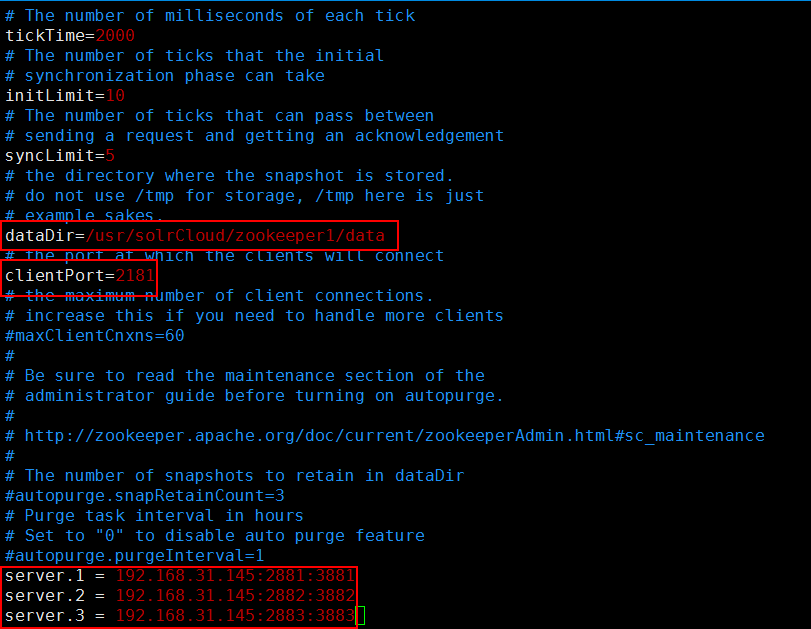


创建data文件夹并生成myid文件

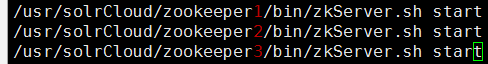


修改zoo.cfg文件

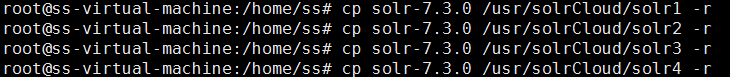
dataDir与Port均需要不同，但是server.n均相同



设置一个文件启动zookeeper集群的.sh文件，并赋予权限内容如下

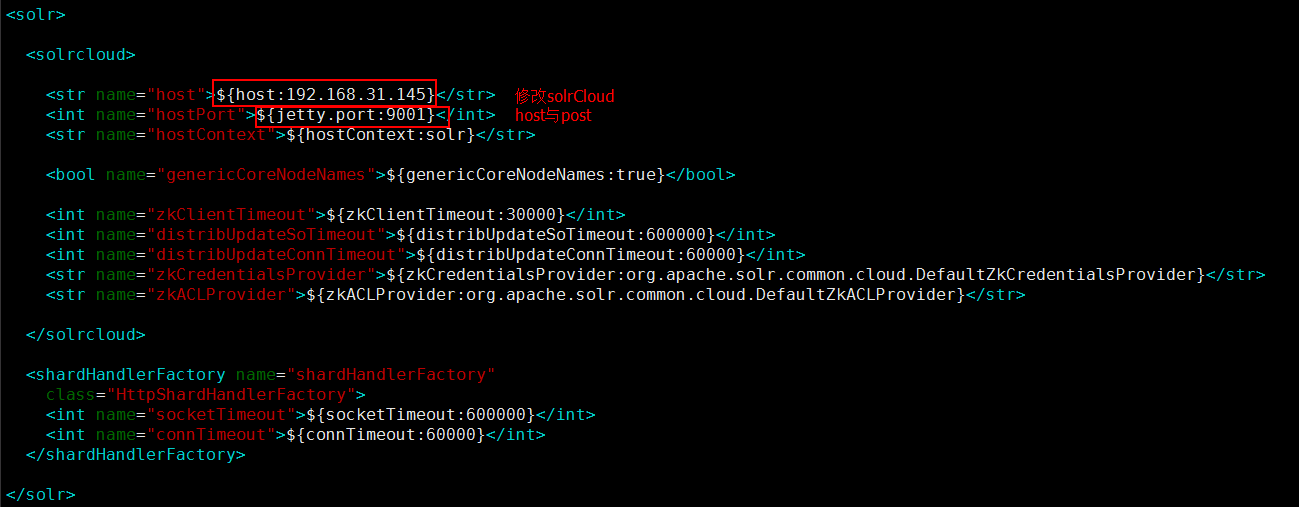


复制四份solr服务器

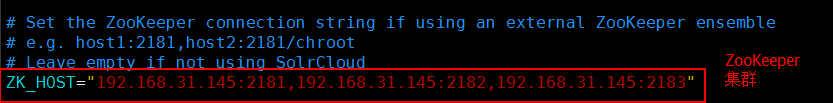


修改solr.xml文件





修改solr.in.sh文件



向zookeeper上传配置文件



|  |
| --- |
| /usr/solrCloud/solr1/bin/solr zk upconfig -d /usr/solrCloud/solr1/server/solr/configsets/\_default//conf -n myconf -z 192.168.31.185:2181 |

启动solr服务器

新建集合



|  |
| --- |
| /usr/solrCloud/solr1/bin/solr create\_collection -c myCollection -n myconf -shards 2 -replicationFactor 2 -force |

# 使用SolrJ连接集群

|  |
| --- |
| @Test public void testSolrClout() throws Exception{ HttpSolrClient solr = new HttpSolrClient.Builder("http://192.168.31.145:9001/solr/newCollection").withConnectionTimeout(10000).withSocketTimeout(60000).build();  SolrInputDocument document = new SolrInputDocument();  document.addField("id","test01");  document.addField("item\_title","长安新生二二三三");  solr.add(document);  solr.commit();  solr.close(); } |

## 项目切换到集群

只需要在Spring容器中配置集群版SolrServer对象即可

|  |
| --- |
| <bean id="httpSolrClient" class="org.apache.solr.client.solrj.impl.HttpSolrClient"> <constructor-arg name="builder" value="http://192.168.31.145:9001/solr/newCollection" /> </bean> |

## 商品同步

修改商品信息后，需要把商品信息同步到索引库。需要Serarch工程发布服务，在后台系统，如果商品信息发生修改后，可以在后台Controller中调用发布的服务。把商品id传递给服务，服务进行索引库的内容同步。

# 淘淘商城第九天

## 第八天内容回顾

* 搜索系统的实现
  + 发布搜索服务，PC端与移动端均可以调用
  + 在Portal中调用服务，实现搜索功能
* Solr提供的搜索服务
  + Solr的集群
  + Solr集群的使用

在Solr7版本的时候，Solr只需要配置访问地址即可，其实际上也是基于了分布式服务器系统，对外发布一个访问的网址，可以通过该网址进行数据传递与获取。不过，在查看的时候，未发现solr中存在java代码。但是lib中又存在jar包依赖，其中还包含hadoop的依赖jar，不明觉厉

# 课程计划

## 课程计划

### 商品详情页面的展示

* 伪静态化(可以支持1000左右的访问)
  + 使用JSP动态展示
  + 添加缓存，redis

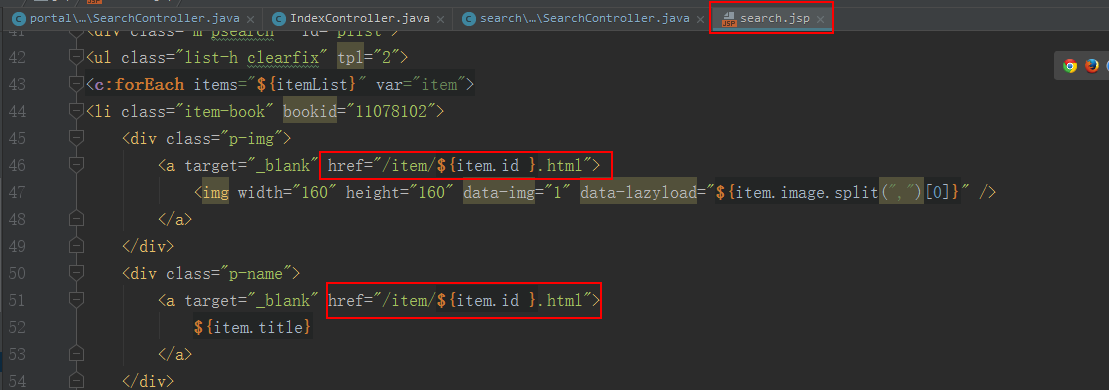
### 网页静态化

freemaker+nginx展示网页

# 商品详情页面展示

在搜索结果中点商品图片或者title打开商品详情页面。

商品详情页面展示的链接



在Collection中从url把商品id取出来，根据商品id，查询对应的商品详细信息。

商品信息应该是调用服务来获取。需要在taotao-rest中发布服务。

taotao-rest服务端也是根据商品id查询商品信息。

为了减轻数据库的压力，需要添加缓存。由于商品数据比较多，不能将所有商品设置进入redis，需要把用户访问的数据添加到缓存中，同时需要设置商品数据的过期时间，根据实际情况而定。需要把过期时间放到配置文件中。

需要用到的表：

商品基本信息：tb\_item，商品详情信息：tb\_item\_desc，规格参数：tb\_item\_param\_item。

应该发布三个服务：取商品基本信息、取商品详细信息，取规格参数。

要设置过期时间，redis只能在key上设置。所以不能使用hset这种方式。

所以采用存储String数据类型，同时为了便于分组可以使用“：”分隔的命名方式。

REDIS\_ITEM:BASE\_INFO:{ITEM\_ID} 作为一个KEY值。

## 后台程序(rest)

### Dao层

单表查询，直接调用逆向工程代码即可

### Service层

接收商品id查询数据库信息，在Service层添加缓存以提高效率

|  |
| --- |
| @Override public TbItem getItemById(Long id) {  // 查询缓存，如果有缓存，直接返回  try{  String json = jedisClient.get(REDIS\_ITEM\_KEY+":"+ITEM\_BASE\_INFO\_KEY+":"+id);  // 判断数据是否存在  if(StringUtils.*isNotBlank*(json)){  // 把json数据转换成java对象  TbItem tbItem = JsonUtils.*jsonToPojo*(json, TbItem.class);  return tbItem;  }  }catch (Exception e){  e.printStackTrace();  }  // 根据商品id查询商品基本信息  TbItem item = itemMapper.selectByPrimaryKey(id);   // 向redis中添加缓存。添加缓存的原则是不能影响正常的业务逻辑  try {  // 向Redis添加缓存  jedisClient.set(REDIS\_ITEM\_KEY+":"+ITEM\_BASE\_INFO\_KEY+":"+id,  JsonUtils.*objectToJson*(item));  // 设置key的过期时间  jedisClient.expire(REDIS\_ITEM\_KEY+":"+ITEM\_BASE\_INFO\_KEY+":"+id,ITEM\_EXPIRE\_SECOND);  }catch (Exception e){  e.printStackTrace();  }  return item; } |

### Collection层

接收商品id，根据商品id查询商品基本信息，调用Service查询，直接相应json数据。返回结果：使用TaotaoResult保障TbItem对象

请求的url：、rest/item/base/{itemId}

|  |
| --- |
| */\*\*  \* 查询商品基本信息  \** ***@param*** *itemId  \** ***@return*** *\*/* @RequestMapping("/base/{itemId}") @ResponseBody public TaotaoResult getItemById(@PathVariable Long itemId){  try {  TbItem item = itemCatService.getItemById(itemId);  return TaotaoResult.*ok*(item);  }catch (Exception e){  e.printStackTrace();  return TaotaoResult.*build*(500,ExceptionUtil.*getStackTrace*(e));  } } |

## 商品详情信息查询服务

根据商品id查询商品详情。tb\_item\_desc表

### Dao层

使用逆向工程代码。

### Service层

接收商品ID，根据商品id查询商品详情，返回TbItemDesc对象。

需要添加缓存。需要设置过期时间。

### Controller层

接受商品id，調用Service查詢商品傷情。返回json数据。使用TaotaoResuly保障一个商品详情pojo返回。

请求的url：/rest/item/desc/{itemId}

## 页面程序(portal)

### 分析

请求的Url：/item/{itemId}.html

根据商品id查询商品的信息，需要调用taotao-rest发布的服务。

将获得商品数据传递给jsp。

1. 先获取商品的基本信息展示给用户。
2. 当页面展示完成后，使用ajax调用服务来查询商品的详情。延迟1秒加载。
3. 商品规格参数，按需加载，当用户点击规格参数tab页的时候加载。

### Service层

根据商品id，调用服务，获得商品基本信息数据。需要把数据转换成Java对象。返回TbItem对象。

参数：itemId

返回值：TbItem

|  |
| --- |
| @Override public TbItem getItemById(Long itemId) {  // 根据商品id查询商品基本信息  String json = HttpClientUtil.*doGet*(REST\_BASE\_URL + REST\_ITEM\_BASE\_URL + itemId);  TaotaoResult taotaoResult = TaotaoResult.*formatToPojo*(json, TbItem.class);  TbItem item = (TbItem)taotaoResult.getData();  return item; } |

### Controller层

接收商品id，从url取。调用Sevice，把查询结果传递给jsp。

请求的url：[http://localhost:8082/item/{itemId}.html](http://localhost:8082/item/%7bitemId%7d.html)

把TbItem对象传递给jsp。Key就是item

因为需要images属性，所以继承TbItem类，添加getImage()方法

|  |
| --- |
| public class PortalItem extends TbItem {   public String[] getImages(){  String image = this.getImage();  if(image != null && !image.equals("")){  String[] imgs = image.split(",");  return imgs;  }  return null;  } } |

## 展示商品描述

请求的url：/item/desc/{itemId}.html

相应的内容就是商品描述的html片段。

参数：商品ID

### Service层

接受商品id，从Url中获取该Id，调用Service层获得商品描述数据，响应给浏览器。响应一个字符串。

参数：商品id

返回值是一个Html片段

|  |
| --- |
| @Override public String getgItemDescById(Long itemId) {  // 根据商品id调用taotao-rest的服务获取到数据  // http://localhost:8081/rest/item/desc/1022302785  String json = HttpClientUtil.*doGet*(REST\_BASE\_URL + REST\_ITEM\_DESC\_URL + itemId);  // 将json转换成java对象  TaotaoResult taotaoResult = TaotaoResult.*formatToPojo*(json, TbItemDesc.class);  // 获取商品描述信息  TbItemDesc itemDesc = (TbItemDesc)taotaoResult.getData();  String desc = itemDesc.getItemDesc();  return desc; } |

### Controller层

返回的是一个html页面，所以需要设置produce为MediaType.TEXT\_HTML\_VALUE，文字格式为utf-8

|  |
| --- |
| @RequestMapping(value = "/item/desc/{itemId}" ,produces = MediaType.*TEXT\_HTML\_VALUE*+";charset=utf-8") @ResponseBody public String getItemDesc(@PathVariable Long itemId){  String desc = itemService.getgItemDescById(itemId);  return desc; } |

## 展示规格参数

当点击tab标签时，触发一个点击事件。JS做ajax请求，请求规格参数数据。将响应的内容展示到页面。

请求的url：/item/param/{itemId}.html

相映红的内容：一个html片段。在java代码中生成。

参数：商品id。

### Service层

接受商品id，调用taotao-rest的服务，获得规格参数数据。转换成java对象，转换成html片段，返回给Controller。

### Controller层

接受商品id，调用Service获得规格参数的html片段，直接响应。

参数：商品id

返回值：String

# 网页的静态化

## 分析

每个商品要对应生成一个html文件，保存到磁盘。访问文件的时候，没必要使用tomcat，可以使用nginx访问。

使用freemaker工具生成。

## freemarker

freemarker是一个用Java语言别写的模板影青，它是基于模板来生成文本输出。FreeMarker与Web容器无关，即在Web运行时，它不会接触到Servlet和Http。它不仅可以用作表现层的实现技术，而且还可以用于生成XML，JSP或Java等。

目前企业中，主要用freemarker做静态页面或是页面展示。

## fremarker的使用方法

第一步：把fareemarker的jar包添加到工程中。

第二步：freemarker的运行不依赖web容器，可以在java工程中运行。创建一个测试方法进行测试。

第三步：创建一个Configuration对象。

第四步：设置Configuration对象模板文件存放的路径。

第五步：设置Configuration的默认字符集。一般是UTF-8。

第六步：从Configuration对象中获得模板对象。需要指定一个模板文件的名字。

第七步：创建模板需要的数据集。可以是一个map对象也可以是一个pojo。将模板需要的数据放入数据集。

第八步：创建一个Writer对象，指定生成文件所存放的路径及文件名。

第九步：丢哦啊用模板对象的process方法生成静态文件。需要数据集与writer对象。

第十步：关闭writer对象

## 使用代码实例

### Java代码

|  |
| --- |
| @Test public void testFreeMarker() throws Exception{  //第一步：把fareemarker的jar包添加到工程中。  //第二步：freemarker的运行不依赖web容器，可以在java工程中运行。创建一个测试方法进行测试。  //第三步：创建一个Configuration对象。  Configuration configuration = new Configuration(Configuration.*getVersion*());  //第四步：设置Configuration对象模板文件存放的路径。  configuration.setDirectoryForTemplateLoading(new File("E:\\taotao\\taotao-portal\\src\\main\\webapp\\WEB-INF\\jsp\\ftl"));  //第五步：设置Configuration的默认字符集。一般是UTF-8。  configuration.setDefaultEncoding("UTF-8");  //第六步：从Configuration对象中获得模板对象。需要指定一个模板文件的名字。  Template template = configuration.getTemplate("first.ftl");  //第七步：创建模板需要的数据集。可以是一个map对象也可以是一个pojo。将模板需要的数据放入数据集。  Map root = new HashMap();  root.put("hello","hello freemarker");  //第八步：创建一个Writer对象，指定生成文件所存放的路径及文件名。  Writer out = new FileWriter(new File("E:\\ftl\\hello.html"));  //第九步：丢哦啊用模板对象的process方法生成静态文件。需要数据集与writer对象。  template.process(root,out);  //第十步：关闭writer对象。  out.flush();  out.close(); } |

### first.ftl代码

**${**hello**}**

# freemarker模板的写法

## 取简单类型数据

使用EL表达式

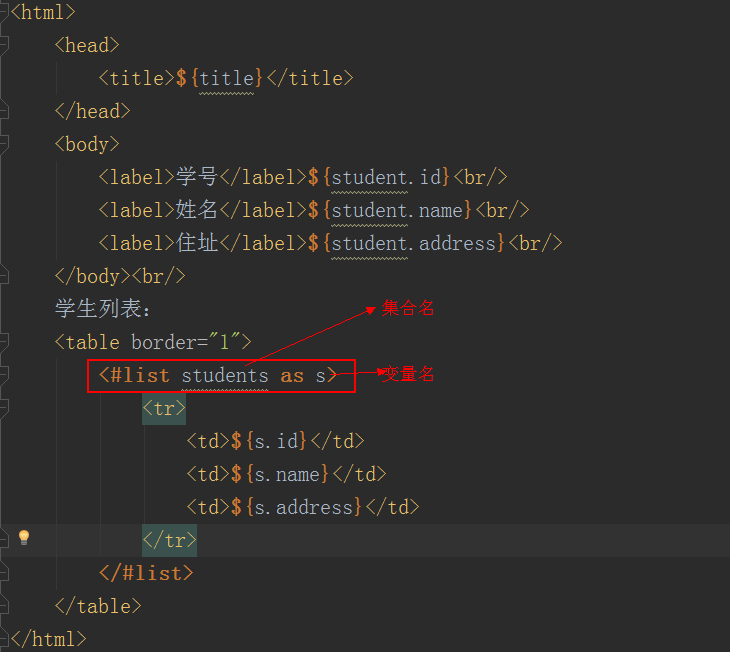
${hello}

## 取包装数据类型

### ftl模板

|  |
| --- |
| <html>  <head>  <title>**${**title**}**</title>  </head>  <body>  <label>学号</label>**${**student.id**}**<br/>  <label>姓名</label>**${**student.name**}**<br/>  <label>住址</label>**${**student.address**}**<br/>  </body> </html> |

## 遍历集合/数组



## 在模板中进行条件判断

|  |
| --- |
| **<#if** 判断条件**>  <#else> </#if>** 逻辑运算符(== != || &&) |

例：

|  |
| --- |
| **<#list** students **as** s**>  <#if** s\_index % 2 == 0**>** <tr style="color:red">  **<#else>** <tr>  **</#if>** <td>**${**s\_index**}**</td>  <td>**${**s.id**}**</td>  <td>**${**s.name**}**</td>  <td>**${**s.address**}**</td>  </tr> **</#list>** |

## 格式化日期

默认格式

1：date

${cur\_time?date}

2：datetime

${cur\_time?datetime}

3：time

${cur\_time?time}

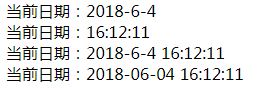
4：自定义格式

${cur\_time?string(“yyyy-MM-dd HH:mm:ss”)}

代码：

|  |
| --- |
| <br/> 当前日期：**${**curdate?date**}** <br/> 当前日期：**${**curdate?time**}** <br/> 当前日期：**${**curdate?datetime**}** <br/> 当前日期：**${**curdate?string("yyyy-MM-dd HH:mm:ss")**}** |

显示效果：



## 处理空值

### 普通数据

在Freemarker中，若是存在取空值会报错，只需要在取值后面加入！即可，之后可跟默认值。



### 日期类型

|  |
| --- |
| **<#if** curdate ??**>** 当前日期：**${**curdate?string("yyyy-MM-dd HH:mm:ss")**}  <#else>** curdate属性为null **</#if>** |

## include

将其他页面引入当前页面。



# freemarker在项目中的使用

## 使用freemarker整合spring

|  |
| --- |
| <!-- freemarker整合spring --> <bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">  <property name="templateLoaderPath" value="/WEB-INF/ftl/" />  <property name="defaultEncoding" value="UTF-8" /> </bean> |

使用FreeMarkerConfigurer代替Configuration交由Spring管理。

### 使用到的依赖

|  |
| --- |
| <dependency>  <groupId>org.freemarker</groupId>  <artifactId>freemarker</artifactId> </dependency> <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId> </dependency> |

### 联想

freemarker是根据模板生成对应的文件，数据是由调用方直接进行传入的，所以，可以进行多方面使用，只需要你将数据替换为自己想要的数据，将模板替换为想要的模板即可，实现原理应该是比较简单的，将模板数据读取到内存，模板数据中的标识替换为想要的数据，之后以文本的形式写入文件，并修改文件的后缀名为指定后缀名即可。

联想：

分布式服务器，使用zookeeper统一管理配置文件，但是使用freemarker虽然不能统一配置，但是可以将生成的文件均放到各个服务器上，进行文件覆盖。

## 静态文件生成的时机

1. 当用户第一次访问时生成静态文件。不推荐使用。
   1. 当高并发时，可能文件未生成完整即被下一个调用返回。
2. 提前生成，当后台添加、编辑商品时生成静态网页。
   1. 使用后台与portal联动，由portal发布服务，供后台系统调用，当后台添加或者修改商品时，调用此服务。
   2. 此时taotao-portal的功能就是生成静态页面。

# 发布生成页面的服务

1. 获得数据的方式：调用rest发布的服务获得数据。
2. 生成静态页面。
3. 响应生成成功。

## Service层

获得的参数：商品id

根据商品id调用rest发布的服务，获得数据(商品基本信息、商品描述、商品规格参数)，生成静态页面。返回成功。

参数：商品id

返回值：TaotaoResult。

|  |
| --- |
| public TaotaoResult getItemHtml(Long itemId) throws IOException, TemplateException {  // 根据商品id获得商品数据  // 商品基本信息  TbItem item = itemService.getItemById(itemId);  // 商品详细信息  String itemDesc = itemService.getItemDescById(itemId);  // 商品规格参数  String itemParam = itemService.getItemParamById(itemId);   // 生成静态文件  Configuration configuration = freeMarkerConfigurer.getConfiguration();   Template template = configuration.getTemplate("item.ftl");  // 创建一个数据集  Map root = new HashMap();  // 向数据集中添加数据  root.put("item",item);  root.put("itemDesc",itemDesc);  root.put("itemParam",itemParam);   // 创建Writer对象  Writer out = new FileWriter(new File(STATIC\_PAGE\_PATH+ itemId + ".html"));  // 生成静态文件  template.process(root,out);  out.flush();  out.close();  return TaotaoResult.*ok*(); } |

## 模板文件

详见item.ftl

## Controller层

接收商品id，调用service生成静态页面。返回json数据。

参数：商品id。

返回值：taotaoResult

|  |
| --- |
| @RequestMapping("/gen/item/{itemId}") @ResponseBody public TaotaoResult getItemPage(@PathVariable Long itemId){  try {  TaotaoResult result = staticPageService.getItemHtml(itemId);  return result;  } catch (Exception e) {  e.printStackTrace();  return TaotaoResult.*build*(500,ExceptionUtil.*getStackTrace*(e));  } } |

### 对于Nginx

静态文件访问服务器。

可以配置文件路径，由文件名访问，也可以配置其他服务器地址，由nginx转发请求。

所以说，以前使用的react实际上使用了npm的专用未知服务器。

# 解决406问题

问题原因：

数据访问的路径是以.html结尾，浏览器会自动进行SEO优化。但是在ConTroller中使用了@ResponseBody注解，返回.json数据。导致浏览器不知应该使用何种方式进行数据解析，所以报406异常。

解决方式：  
在web.xml中配置除.html之外的拦截方式即可。

# 淘淘商城第十天

## 课程回顾

展示商品详情页面

1. 第一种方式：使用jsp展示页面。
   1. 商品的基本信息 请求即加载
   2. 商品描述 使用ajax延迟加载
   3. 商品的规格参数 动态加载(访问时才加载)
   4. 调用服务获得的数据
2. 第二种方式：网页静态化使用freenarker
   1. 创建一个模板文件，类似于jsp
   2. 创建一个数据集将模板需要的数据准备好
   3. freemarker加载模板根据数据集生成html页面。
   4. 使用nginx访问html

## 课程计划

1. 单点登录系统(一处登录到处使用)
   1. 创建一个单点登录系统 独立工程
   2. 发布服务供其他系统调用
   3. 前台系统调用单点登录系统
2. session共享问题

# 单点登录系统

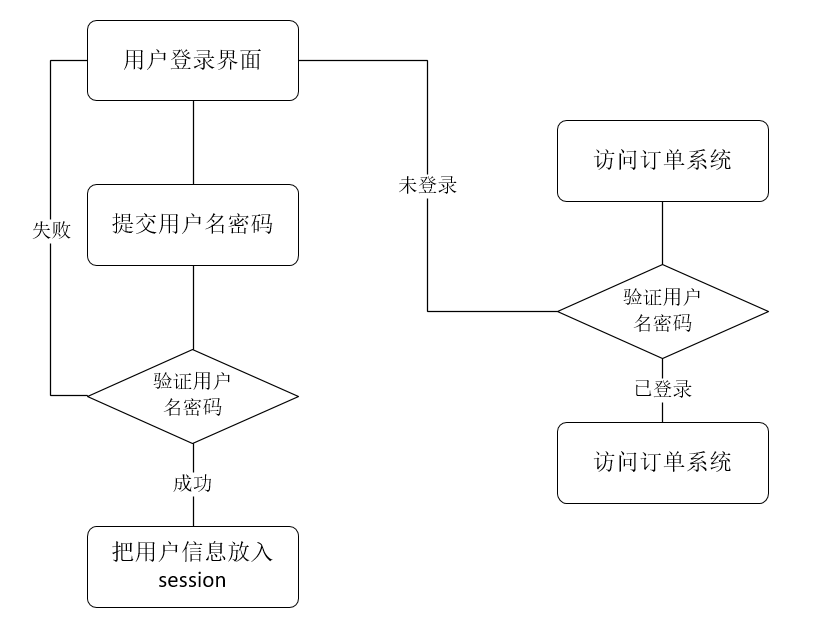
## SSO

SSO：单点登录系统。

Single Sign On 单点登录。SSO是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。它包括可以将这次主要的登录映射到其他应用中用户同一个用户的登录的机制。是目前比较流行的企业业务整合的解决方案之一。

## 为什么要有点点登录系统

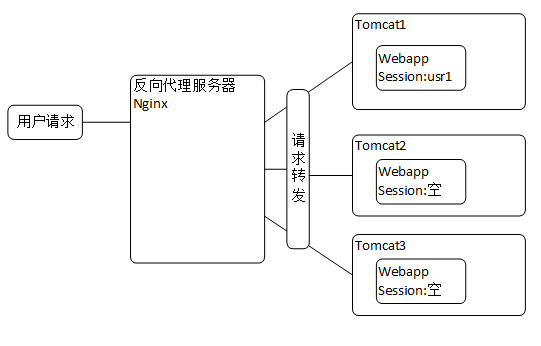
### 传统的登录



## 并发量高的情况

访问量2000-3000的并发

一个tomcat无法满足业务需求，需要做集群。



## Session共享的问题

### 方式一：tomcat集群

Tomcat做集群配置session复制。当其中一个tomcat的session被刷新后，将session数据向集群内所有Tomcayt广播该session对象，使其他tomcat刷新该sesion，增加删除或修改。

#### 问题

如果集群中节点很多，会形成网络风暴。因为当其他服务器接收到该session广播后，会继续向其他服务器发送自己的广播。做集群推荐节点数据量不要超过5个。

### 方式二：分布式架构

将系统拆分成多个子系统。

分布式架构，将验证的内容存储进入redis中，进行session共享的意义在于可以在其他信任的系统中获取到对应的用户共享信息，但是还有一个问题是用户的短链接过多问题。

将用户信息放入到单点登录系统中，在单点登录系统中使用redis模拟session。

单点登录系统对外发布服务进行单点登录系统管理。

# 单点登录系统处理流程



使用单点登录系统，当每一个子系统的子tomcat校验用户是否登录时，需要一个唯一标识码，所以这里使用token进行校验，token类似于sessionId，进行唯一标识。key即是token。将token写入cookie中，并返回成功。其他系统再从cookie中获取该token值作为单点登录验证基础。

## 系统最新架构



# 创建单点登录系统

使用的技术：

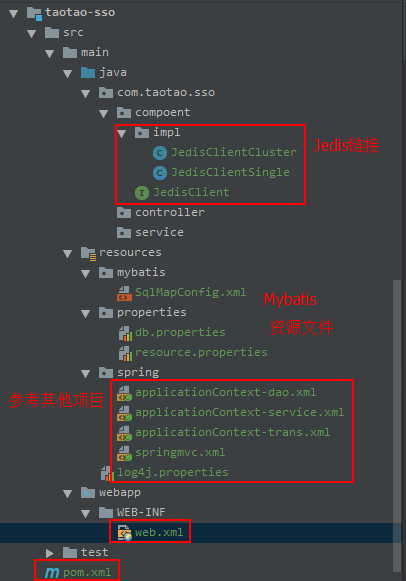
1. mybatis
2. spring
3. spring-mvc (对外发布服务、登录界面)
4. jedis

## pom文件

|  |
| --- |
| <?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">  <parent>  <artifactId>taotao-parent</artifactId>  <groupId>com.caxins.taotao</groupId>  <version>0.0.1-SNAPSHOT</version>  <relativePath>../taotao-parent/pom.xml</relativePath>  </parent>  <modelVersion>4.0.0</modelVersion>   <artifactId>taotao-sso</artifactId>  <packaging>war</packaging>    <dependencies>  <dependency>  <groupId>com.caxins.taotao</groupId>  <artifactId>taotao-dao</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <!--Jedis-->  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  </dependency>  <!-- 添加spring的依赖jar包 -->  <!-- Spring -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  </dependency>  <!-- Servlet依赖，编译时期使用 -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>servlet-api</artifactId>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jsp-api</artifactId>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency>  </dependencies> </project> |

## 框架整合 整合ssm

参考taotao-rest项目



# 单点登录系统的实现

注册功能、登录功能、查询功能。

## 注册接口

### 数据校验接口

#### 分析

要查询的表tb\_user

根据用户名、电话、email去数据库中进行查询，如果查询到结果应该返回false，如果查询结构为空则返回true。

使用taotaoResult包装一下即可。需要支持jsonp，根据callback判断

#### dao层

可以使用逆向工程

#### Service层

接收两个参数，一个是要校验的数据，一个是数据类型。根据与不同的数据类型生成不同的查询条件，到tb\_user中进行查询。

参数：String param；Int type

返回值：TaotaoResult

|  |
| --- |
| @Override public TaotaoResult checkData(String param, int type) {  TbUserExample example = new TbUserExample();  TbUserExample.Criteria criteria = example.createCriteria();  // 1、2、3分别代表username、phone、email 添加查询条件  if (1 == type){  criteria.andUsernameEqualTo(param);  }else if (2 == type){  criteria.andPhoneEqualTo(param);  }else if (3 == type){  criteria.andEmailEqualTo(param);  }  // 根据数据类型检查数据  List<TbUser> tbUsers = userMapper.selectByExample(example);  // 判断查询结果是否为空  if( tbUsers == null || tbUsers.isEmpty()){  return TaotaoResult.*ok*();  }  return TaotaoResult.*ok*(false); } |

#### Controller层

Controller只是发布服务。接收三个参数，一个是要校验的数据，一个是数据类型，一个callback。调用Service校验。返回json数据。还需要支持jsonp，需要判断callback

接收的参数：username,password,phone,email

返回值：TaotaoResult

|  |
| --- |
| @RequestMapping("/check/{param}/{type}") @ResponseBody public Object checkData(@PathVariable String param,@PathVariable Integer type,String callback){  try {  TaotaoResult result = registerService.checkData(param, type);  if (StringUtils.*isNotBlank*(callback)) {  // 请求为jsonp调用，需要支持jsonp  MappingJacksonValue mappingJacksonValue = new MappingJacksonValue(result);  mappingJacksonValue.setJsonpFunction(callback);  return mappingJacksonValue;  }  return result;  }catch (Exception e){  e.printStackTrace();  return TaotaoResult.*build*(500,ExceptionUtil.*getStackTrace*(e));  } } |

### 用户注册接口

#### 分析

使用POST请求，从request中获取到用户名、密码、电话与email插入到数据库

#### dao层

单表操作调用逆向工程代码即可。

#### service层

参数：TbUser

返回值：TaotaoResult

接收参数，对数据进行校验，校验成功，插入数据，返回结果。

|  |
| --- |
| @Override public TaotaoResult register(TbUser user) {  // 校验用户名与密码是否为空  if(StringUtils.*isBlank*(user.getUsername()) || StringUtils.*isBlank*(user.getPassword())){  return TaotaoResult.*build*(400,"用户名与密码不能为空");  }  // 校验数据是否重复  // 校验用户名  TaotaoResult result = checkData(user.getUsername(), 1);  if(!(boolean) result.getData()){  return TaotaoResult.*build*(400,"用户名重复");  }  // 校验手机号  if(user.getPhone() != null){  result = checkData(user.getPhone(), 2);  if(!(boolean) result.getData()){  return TaotaoResult.*build*(400,"手机号已注册");  }  }  // 校验邮箱  if(user.getEmail() != null){  result = checkData(user.getEmail(), 3);  if(!(boolean) result.getData()){  return TaotaoResult.*build*(400,"邮箱已注册");  }  }  // 补全数据  user.setCreated(new Date());  user.setUpdated(new Date());  // 对密码进行MD5加密,Spring中自带Md5加密  user.setPassword(DigestUtils.*md5DigestAsHex*(user.getPassword().getBytes()));  // 插入数据  userMapper.insert(user);   return TaotaoResult.*ok*(); } |

#### Controller层

|  |
| --- |
| @RequestMapping(value = "/register", method = RequestMethod.*POST*) @ResponseBody public TaotaoResult register(TbUser user){  try {  TaotaoResult result = registerService.register(user);  return result;  }catch (Exception e){  e.printStackTrace();  return TaotaoResult.*build*(500,ExceptionUtil.*getStackTrace*(e));  } } |

## 用户登录接口

### 分析

1. 接收用户名密码
2. 校验用户名密码
3. 生成token，可以使UUID
4. 将用户信息以token值为key写入到redis
5. 将token写入cookie
6. 返回登录成功，将token返回给客户端

将token写入cookie中,cookie在系统之间进行共享。

### Dao层

单表操作，使用逆向工程即可

### Service层

接收参数：用户名、密码，HttpServletRequest，HttpServletResponse

返回值：TaotaoResult

数据校验，校验成功后成圣token，向redis中写入用户信息，把token写入cookie，返回TaotaoResult，包含token。

|  |
| --- |
| @Override public TaotaoResult login(String username, String password, HttpServletRequest request, HttpServletResponse response) {  // 判断用户密码是否正确  TbUserExample example = new TbUserExample();  TbUserExample.Criteria criteria = example.createCriteria();  criteria.andUsernameEqualTo(username);  List<TbUser> tbUsers = userMapper.selectByExample(example);  // 获取对应的用户信息  if(tbUsers == null || tbUsers.isEmpty()){  return TaotaoResult.*build*(400,"用户名不存在");  }  TbUser user = tbUsers.get(0);  // 校验密码  if(!user.getPassword().equals(DigestUtils.*md5DigestAsHex*(password.getBytes()))){  return TaotaoResult.*build*(400,"密码错误");  }  // 登录成功  // 生成token  String token = UUID.*randomUUID*().toString();  // 将用户信息写入redis  // key: REDIS\_SESSION:{TOKEN}  // value: user(JSON格式)  // 将密码清空  user.setPassword(null);  jedisClient.set(REDIS\_SSION\_KEY+":"+token,JsonUtils.*objectToJson*(user));  // 设置redis模拟的session的过期时间  jedisClient.expire(REDIS\_SSION\_KEY+":"+token,SESSION\_EXPIRE);  return TaotaoResult.*ok*(token); } |

Cookie中的数据时不支持跨服务器共享的，不过，在CookieUtils中，将域名进行截取，只保留不会更改的部分，

使得Cookie数据的存取不受分布式项目的约束。

### Controller层

请求的url：/user/login

接收参数：username、password

调用service，返回taotaoResult对象。

响应json数据。

|  |
| --- |
| @RequestMapping(value = "/user/login",method = RequestMethod.*POST*) @ResponseBody public TaotaoResult login(String username, String password, HttpServletRequest request, HttpServletResponse response){   try {  TaotaoResult result = loginController.login(username, password, request, response);  return result;  } catch (Exception e){  e.printStackTrace();  return TaotaoResult.*build*(500,ExceptionUtil.*getStackTrace*(e));  } } |

## 通过Token查询用户信息

根据token到redis查询用户信息，如果用户信息不存在，说明session已经过期。

若是已经过去，返回400并提示用户redis中数据已经过期。如果查询到用户，返回用户信息，并且更新用户的过期时间。

请求的url：/user/token/{token}

需要支持jsonp

返回taotaoResult。

### Dao层

使用JedisClient实现

### Service层

参数：String token

根据token查询redis，查询到结果返回用户对象，更新过期时间。如果查询不到结果，返回session已经过期，状态码400

返回值：TaotaoResult。

session的作用是用来进行数据存储而不是进行服务器对象标识的，这是一个理解性的问题

### Controller层

请求的url：/user/token/{token}

从url获取token的内容，调用service去用户信息，返回TaotaoResult。

|  |
| --- |
| @RequestMapping(value = "/user/token/{token}") @ResponseBody public Object getUserByToken(@PathVariable String token,String callback){  try {  TaotaoResult result = loginService.getUserByToken(token);  // 支持jsonp  if(StringUtils.*isNotBlank*(callback)){  MappingJacksonValue jacksonValue = new MappingJacksonValue(result);  jacksonValue.setJsonpFunction(callback);  return jacksonValue;  }  return result;  }catch (Exception e){  e.printStackTrace();  return TaotaoResult.*build*(500,ExceptionUtil.*getStackTrace*(e));  } } |

Java如何支持jsonp

方式一：

返回字符串

指定produces = MediaType.APPLICATION\_JSON\_VALUE+";charset=utf-8"

taotao-rest\src\main\java\com\taotao\rest\controller\ItemCatController.java

方式二：

返回MappingJackValue。

使用该对象包含对应值

## 安全退出

接收token，在redis中将与之对应的用户信息即可。返回TaotaoResult.ok()

分析：接收对应的token值，在redis中将数据删除即可

### Dao层

调用JedisClient即可

### Service层

接收的参数：String token

返回的结果：TaotaoResult.ok()

|  |
| --- |
| @Override public TaotaoResult loginOut(String token) {  // 根据token值设置对应用户数据过期  jedisClient.expire(REDIS\_SSION\_KEY+":"+token,-2);  return TaotaoResult.*ok*(); } |

### Controller层

|  |
| --- |
| @RequestMapping("/user/logout/{token}") @ResponseBody public Object logout(@PathVariable String token,String callback){  try {  TaotaoResult result = loginOutService.loginOut(token);  if(StringUtils.*isNotBlank*(callback)){  MappingJacksonValue jacksonValue = new MappingJacksonValue(result);  jacksonValue.setJsonpFunction(callback);  return jacksonValue;  }  return result;  }catch (Exception e){  e.printStackTrace();  return TaotaoResult.*build*(500,ExceptionUtil.*getStackTrace*(e));  } } |

# 门户系统整合SSO

当用户在首页点击登录或者注册的时候需要跳转到sso系统进行相对应的操作。当登录成功应该跳转到首页。首页应该显示当前登录的用户名。

# 登录拦截器

## 分析

当用户提交订单时此时必须要求用户灯具，可以使用拦截器来实现。

拦截器的处理流程：

1. 拦截请求的url
2. 从Cookie中获取token值
3. 如果未获得token跳转到登录页面
4. 取到token，调用sso系统的服务查询用户信息。
5. 若是用户Redis中模拟的session已经过期，跳转到登录界面
6. 若是没有过期，放行，并将过期时间刷新为最大

## Service层

根据token获取用户信息，如果渠道返回TbUser对象，如果取不到返回null。

参数：HttpServletRequest、HttpServletRespone

返回值：TaotaoResult

## 拦截器

在springmvc中需要实现HandleInterceptor接口。

|  |
| --- |
| public class LoginInterceptor implements HandlerInterceptor {   @Autowired  private UserService userService;  @Value("${SSO\_LOGIN\_URL}")  private String SSO\_LOGIN\_URL;   // 执行之前,false  @Override  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  //1.拦截请求的url  //2.从Cookie中获取token值  //3.如果未获得token跳转到登录页面  //4.取到token，调用sso系统的服务查询用户信息。  TbUser user = userService.getUserByToken(request, response);  //5.若是用户Redis中模拟的session已经过期，跳转到登录界面  if(user == null){  // URL跳转  response.sendRedirect(SSO\_LOGIN\_URL);  return false;  }  //6.若是没有过期，放行，并将过期时间刷新为最大  return true;  } |

### 在springmvc中配置拦截器

|  |
| --- |
| <!-- 拦截器配置 --> <mvc:interceptors>  <mvc:interceptor>  <!--<mvc:mapping path="/order/\*\*"></mvc:mapping>-->  <mvc:mapping path="/item/\*\*"></mvc:mapping>  <bean class="com.taotao.portal.LoginInterceptor" />  </mvc:interceptor> </mvc:interceptors> |

# 处理登录回调

登录的url中包含回调的参数，参数就是回调的url，sso登录Controller接收参数，当登录成功后跳转到回调的url。

|  |
| --- |
| */\*\*  \* 展示登录页面  \*/* @RequestMapping("/page/login") public String showLogin(String redirectURL,Model model){  model.addAttribute("redirect",redirectURL);  // 将参数传递给jsp  return "login"; }  */\*\*  \* 展示注册页面  \*/* @RequestMapping("/page/register") public String showRegister(String redirectURL,Model model){  model.addAttribute("redirect",redirectURL);  return "register"; } |

## 拦截器的处理



# 淘淘商城第十一天

## 内容回顾

1. 单点登录系统
   1. 分布式环境下session数据共享问题
   2. session中放到redis中
   3. 使用UUID生成token值，将token放入cookie中
   4. 根据token获取用户信息
2. SSO系统集成portal
   1. 用户登录注册功能都跳转到SSO(单点登录系统)
   2. 登录完成后跳转到首页
   3. 首页需要在加载完成后，需要从cookie中获取token，使用jsonp调用sso系统的服务，根据token查询用户信息，将用户名显示到首页。
   4. 强制用户时，使用拦截器实现用户强制登录
   5. 回调页面实现

## 课程计划

1. 购物车的实现
2. 订单系统的实现

# 购物车

## 分析

添加购物车不需要用户登录。购物车数据应该放入cookie中。

当向购物车添加同一款商品时，购物车中商品的数量增加。

购物车中还应该可以删除商品。

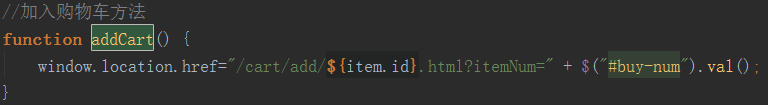
购物车可以修改商品数量。商品的总价格需要重新计算。

点击去结算按钮要求用户登录。

## 添加购物车

### 分析

请求的url



1. 接收商品id，根据商品id获取商品的图片、title、单价。并且将数据保存到cookie中，还应该包含商品的数量。
2. 将商品数据转换成json数据，保存到cookie中。对购物车数据进行转码。
3. 最终展示添加购物车成功页面。

### Service

参数：

1. 商品id
2. 商品数量

业务逻辑：

1. 接收商品id
2. 从购物车获取商品列表
3. 从商品列表中查询列表中是否存在此商品
4. 如果存在商品，则在原有数量上加上当期增加的数量
5. 如果不存在，调用rest，根据商品id获得商品数据
6. 将商品数据添加到列表中
7. 将购物车列表写入cookie
8. 返回taotaoResult

|  |
| --- |
| @Override public TaotaoResult addCart(Long itemId, Integer num, HttpServletRequest request, HttpServletResponse response) {  //1. 接收商品id  //2. 从购物车获取商品列表  List<CartItem> cartItemList = getCartItemList(request);  //3. 从商品列表中查询列表中是否存在此商品  boolean haveFlg = false;  for(CartItem cartItem : cartItemList){  //4. 如果存在商品，则在原有数量上加上当期增加的数量  if(cartItem.getId() == itemId){  cartItem.setNum(cartItem.getNum()+num);  haveFlg = true;  break;  }  }  //5. 如果不存在，调用rest，根据商品id获得商品数据  if(!haveFlg){  CartItem cartItem = new CartItem();  PortalItem item = itemService.getItemById(itemId);  // 转换后才能CartItem  cartItem.setId(itemId);  cartItem.setNum(num);  cartItem.setPrice(item.getPrice());  cartItem.setTitle(item.getTitle());  if(StringUtils.*isNotBlank*(item.getImage())) {  cartItem.setImage(item.getImage());  }  //6. 将商品数据添加到列表中  cartItemList.add(cartItem);  }  //7. 将购物车列表写入cookie  CookieUtils.*setCookie*(request,response,"TT\_CART",JsonUtils.*objectToJson*(cartItemList),COOKIE\_EXPIRE,true);  //8. 返回taotaoResult  return TaotaoResult.*ok*(); }  */\*\*  \* 获取购物车商品列表  \** ***@param*** *request  \** ***@return*** *\*/* private List<CartItem> getCartItemList(HttpServletRequest request){  try {  // 从cookie中获取商品列表  String json = CookieUtils.*getCookieValue*(request, "TT\_CART", true);  // 将json转换成java对象  List<CartItem> list = JsonUtils.*jsonToList*(json, CartItem.class);  return list == null ? new ArrayList<CartItem>() : list;  }catch (Exception e){  e.printStackTrace();  return new ArrayList<CartItem>();  } } |

### Controller层

请求的url：/cart/add/${item.id}.html?num=" + $("#buy-num").val()\

接收两个参数：

1. 商品id
2. 商品数量

调用Service将商品添加到购物车

返回jsp页面，添加成功页面。

|  |
| --- |
| @RequestMapping("/cart/add/{itemId}") public String addCart(@PathVariable Long itemId, Integer num, HttpServletRequest request, HttpServletResponse response){  try {  TaotaoResult result = cartService.addCart(itemId, num, request, response);  return "cartSuccess";  }catch (Exception e){  return "index";  } } |

## 展示购物车商品列表

### 分析

请求的<url:/cart/cart>

HttpServletRequest

从cookie中获取购物车商品列表，在页面展示。

Jsp页面需要一个购物车商品列表即可。

### Service层

无参数，直接从Cookie中获取购物车商品列表返回即可

返回值是：List<CartItem>

|  |
| --- |
| */\*\*  \* 获取购物车商品列表  \** ***@param*** *request  \** ***@return*** *\*/* private List<CartItem> getCartItemList(HttpServletRequest request){  try {  // 从cookie中获取商品列表  String json = CookieUtils.*getCookieValue*(request, "TT\_CART", true);  // 将json转换成java对象  List<CartItem> list = JsonUtils.*jsonToList*(json, CartItem.class);  return list == null ? new ArrayList<CartItem>() : list;  }catch (Exception e){  e.printStackTrace();  return new ArrayList<CartItem>();  } }  @Override public List<CartItem> getCartItems(HttpServletRequest request) {  List<CartItem> list = getCartItemList(request);  return list; } |

### Controller层

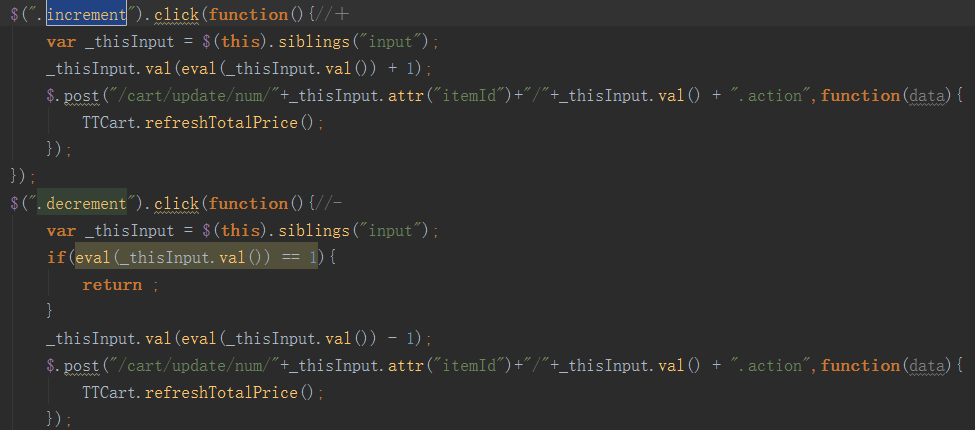
接收请求

请求的URL：/cart/cart

调用Service去商品列表，传递给jsp。

## 修改购物车商品数量

修改商品数量的js请求：



请求的url:

/cart/update/num/{itemId}/{num}.action

1. 把购物车商品列表从cookie中取出来
2. 根据itemId找到对应的商品
3. 使用参数中的数量更新cookie中的数量信息
4. 将购物车商品列表再次写入cookie
5. 返回json数据。返回taotaoResult即可

### Service层

接收两个参数：

1. 商品id
2. 商品数量

业务逻辑：

1. 把购物车商品列表从cookie中取出来
2. 根据itemId找到对应的商品
3. 使用参数中的数量更新胡亮
4. 将购物车商品列表写会cookie
5. 返回TaotaoResult

|  |
| --- |
| @Override public TaotaoResult updateCartItem(Long itemId, Integer num, HttpServletRequest request, HttpServletResponse response) {  // 从Cookie中获取购物车商品列表  List<CartItem> itemList = getCartItemList(request);  // 根据商品id查询商品  for(CartItem cartItem : itemList){  if(cartItem.getId().equals(itemId)){  // 更新数量  cartItem.setNum(num);  break;  }  }  // 将数据写回Cookie  CookieUtils.*setCookie*(request,response,"TT\_CART",JsonUtils.*objectToJson*(itemList),COOKIE\_EXPIRE,true);   return TaotaoResult.*ok*(); } |

### Controller层

请求的url：/cart/update/num/{itemId}/{num}.action

从url中获取两个参数，调用Service更新商品数量。

返回TaotaoResult.ok() 返回json格式的数据。

|  |
| --- |
| @RequestMapping("/cart/update/num/{itemId}/{num}") @ResponseBody public TaotaoResult updateCartItemNum(@PathVariable Long itemId,@PathVariable Integer num,HttpServletRequest request,HttpServletResponse response){  TaotaoResult result = cartService.updateCartItem(itemId, num, request, response);  return result; } |

## 删除购物车商品

请求的url：/cart/delete/{itemId}.html

### Service层

接收的参数：itemId

业务逻辑：

1. 接收itemId
2. 从cookie中获取itemList
3. 删除list中对应的信息
4. 将list写入的cookie中
5. 返回TaotaoResult

|  |
| --- |
| @Override public TaotaoResult deleteCartItem(Long itemId, HttpServletRequest request, HttpServletResponse response) {  // 从Cookie中获取对应的商品列表信息  List<CartItem> itemList = getCartItemList(request);  // 遍历itemList并删除对应信息  for(CartItem item : itemList){  if(item.getId().equals(itemId)){  itemList.remove(item);  break;  }  }  // 将数据写回到Cookie  CookieUtils.*setCookie*(request,response,"TT\_CART",JsonUtils.*objectToJson*(itemList),COOKIE\_EXPIRE,true);  return TaotaoResult.*ok*(); } |

### Controller层

请求的url：/cart/delete/${cart.id}.html

|  |
| --- |
| @RequestMapping("/cart/delete/{itemId}") public String deleteCartItem(@PathVariable Long itemId,HttpServletRequest request,HttpServletResponse response){  TaotaoResult result = cartService.deleteCartItem(itemId, request, response);  return "redirect:/cart/cart.html"; } |

## 购物车畅想

1. 未登录的情况下可以把购物车写入cookie。
2. 已登录的情况下，需要把购物车写入redis。
3. 登录时如果cookie中存在购物车商品，应将cookie中的数据转移到redis中。key为用户id。value即购物车商品列表。
4. 如果redis中转义商品时，redis的购物车中已经存在了商品，此时需要把商品合并。如果是同一款商品数量叠加，如果是新商品就合并商品。

# 订单系统的实现

## 分析

1. 订单确认页面，确认购买的商品、使用优惠券或积分、确认配送地址。
2. 确认页面上存在订单提交按钮，点击按钮生成订单。
3. 生成订单需要调用服务。需要发布一个生成订单的服务。

需要创建一个订单系统。

## 订单系统创建

需要直接使用关系型数据库。

使用的技术：

mybatis

spring

spring-mvc

Redis

可以参考taotao-rest创建服务。

POM文件

|  |
| --- |
| <?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">  <parent>  <artifactId>taotao-parent</artifactId>  <groupId>com.caxins.taotao</groupId>  <version>0.0.1-SNAPSHOT</version>  <relativePath>../taotao-parent/pom.xml</relativePath>  </parent>  <modelVersion>4.0.0</modelVersion>   <artifactId>taotao-order</artifactId>  <packaging>war</packaging>    <dependencies>  <dependency>  <groupId>com.caxins.taotao</groupId>  <artifactId>taotao-dao</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <!--Jedis-->  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  </dependency>  <!-- 添加spring的依赖jar包 -->  <!-- Spring -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  </dependency>  <!-- Servlet依赖，编译时期使用 -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>servlet-api</artifactId>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jsp-api</artifactId>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency>  </dependencies> </project> |

# 创建订单服务

## 分析

接收json数据，需要使用@RequestBody注解，对应提交的json数据格式，创建一个POJO接收

需要用到的表：

tb\_order 订单表

tb\_order\_item 订单商品明细表

tb\_order\_shipping 物流表

可以使用订单相关的三个表对应的pojo，重新封装一个pojo来接收表单中提交的json数据。

定义Pojo

|  |
| --- |
| public class OrderInfo extends TbOrder {   private List<TbOrderItem> orderItems;   private TbOrderShipping orderShipping;   public List<TbOrderItem> getOrderItems() {  return orderItems;  }   public void setOrderItems(List<TbOrderItem> orderItems) {  this.orderItems = orderItems;  }   public TbOrderShipping getOrderShipping() {  return orderShipping;  }   public void setOrderShipping(TbOrderShipping orderShipping) {  this.orderShipping = orderShipping;  } } |

接收json数据后，插入到对应表中。

tb\_order 订单表

tb\_order\_item 订单明细表

tb\_order\_shipping物流表

订单号的生成：要求可读性好，不能太长，应该是纯数字。

1. UUID
2. 取毫秒值
3. 使用redis的incr命令(redis是单线程，不会出现重复，设置一个基数) 采取该方式

安装Redishttps://blog.csdn.net/xu470438000/article/details/42971091

不签订单的相关字段。插入到相对应的表中。

需要返回taotaoResult，其中包含订单号。

## Dao层

向tb\_order tb\_order\_item tb\_order\_shipping三个表中插入数据。

可以使用逆向工程

## Service层

一：插入订单表

1 接收数据，OrderInfo

2 生成订单号

3 补全字段

4 插入订单表

二：插入订单明细表

1 生成订单明细ID，使用redis的incr命令生成。

2 补全字段

3 插入数据

三：插入物流表

1 补全字段

2 插入数据

四：返回TaotaoResult包装订单号

参数：OrderInfo

返回值TaotaoResult

|  |
| --- |
| @Override public TaotaoResult createOrder(OrderInfo orderInfo) {  //一：插入订单表  //1 接收数据，OrderInfo  //2 生成订单号  // 去订单号  String id = jedisClient.get(REDIS\_ORDER\_GEN\_KEY);  if(StringUtils.*isBlank*(id)){  // 如果订单生成key不存在，设置初始值  jedisClient.set(REDIS\_ORDER\_GEN\_KEY,ORDER\_ID\_BEGIN);  }  Long orderId = jedisClient.incr(REDIS\_ORDER\_GEN\_KEY);  //3 补全字段  orderInfo.setOrderId(orderId.toString());  // 设置状态：状态：1、未付款，2、已付款，3、未发货，4、已发货，5、交易成功，6、交易关闭  orderInfo.setStatus(1);  Date date = new Date();  orderInfo.setCreateTime(date);  orderInfo.setUpdateTime(date);  //4 插入订单表  orderMapper.insert(orderInfo);  //二：插入订单明细表   List<TbOrderItem> orderItems = orderInfo.getOrderItems();  for(TbOrderItem orderItem : orderItems){  //1 生成订单明细ID，使用redis的incr命令生成。  Long detailId = jedisClient.incr(REDIS\_ORDER\_DETAIL\_GEN\_KEY);  //2 补全字段  // 订单明细号  orderItem.setItemId(detailId.toString());  // 订单号  orderItem.setOrderId(orderId.toString());  // 插入数据  orderItemMapper.insert(orderItem);  }  //三：插入物流表  TbOrderShipping orderShipping = orderInfo.getOrderShipping();  //1 补全字段  orderShipping.setOrderId(orderId.toString());  orderShipping.setCreated(date);  orderShipping.setUpdated(date);  //2 插入数据  orderShippingMapper.insert(orderShipping);  //四：返回TaotaoResult包装订单号  return TaotaoResult.*ok*(orderId); } |

## Controller层

发布服务

接收客户端发送的数据，接收json格式的数据，使用@RequestBody 使用OrderIndo接收数据，调用service插入数据生成订单，返回TaotaoResult。

|  |
| --- |
| @RequestMapping(value = "/order/create",method = RequestMethod.*POST*) @ResponseBody public TaotaoResult createOrder(@RequestBody OrderInfo orderInfo){  try {  TaotaoResult result = orderService.createOrder(orderInfo);  return result;  } catch (Exception e) {  e.printStackTrace();  return TaotaoResult.*build*(500,ExceptionUtil.*getStackTrace*(e));  } } |

# 创建订单Portal的处理

## 分析

* + 1. 显示订单确认页面
    2. 提交订单处理

## 显示订单确认页面

请求的url：/order/order-cart

1. 数据从购物车中获得
2. 访问此页面要求用户登录
3. 需要根据用户id查询常用地址列表(待实现)

## Service层

可以使用购物车Service

## Controller层

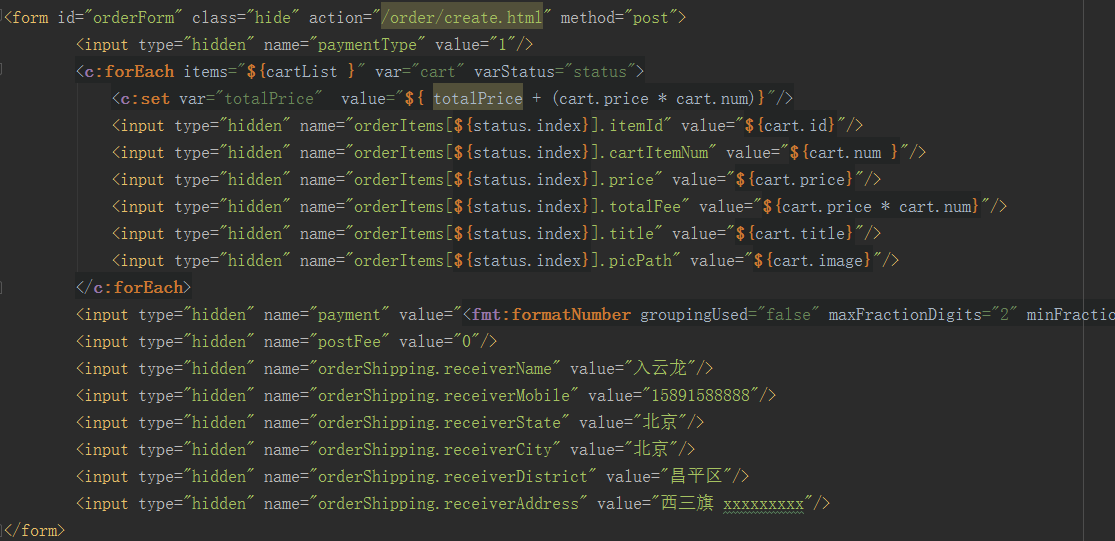
请求的url：/order/oder-cart

取购物车商品列表，传递给jsp展示。

|  |
| --- |
| @RequestMapping("/order-cart") public String showOrderCart(HttpServletRequest request,Model model){  List<CartItem> cartItems = cartService.getCartItems(request);  model.addAttribute("cartList",cartItems);  return "order-cart"; } |

# 提交订单

在订单确认页面，点击提交订单按钮，进行提交订单的处理。把表单的内容提交给taotao-portal的后台，后台调用taotao-order发布的服务生成订单。

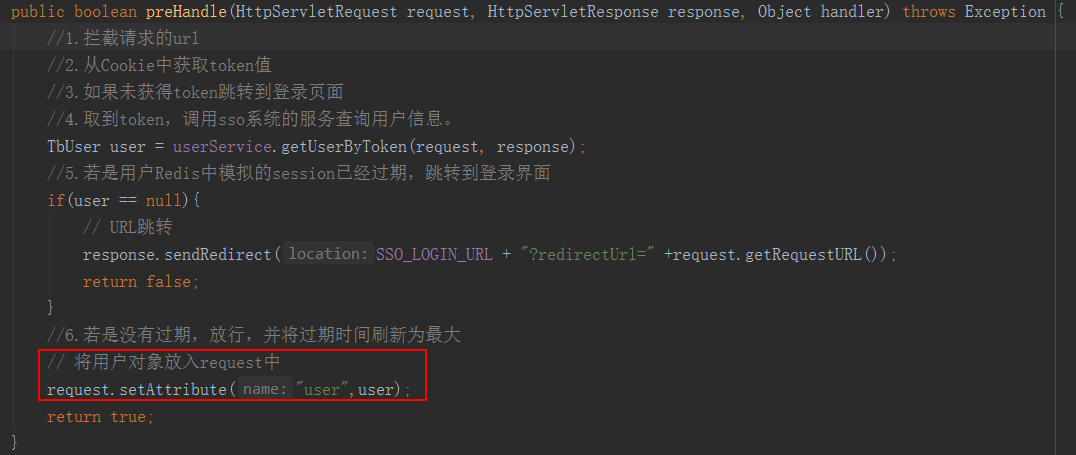


接收表单中的内容：

使用OrderInfo来接收

需要通过token去userId，使用userId与昵称

配置拦截器，在拦截器中将用户对象放入request



## Service层

接收表单中的内容OrderInfo，调用taotao-order的服务，生成订单。提交的数据格式是：json格式。

返回TaotaoResult，其中包含订单号，取出订单号，并返回。

参数：OrderInfor

返回值：String 订单号

|  |
| --- |
| @Override public String createOrder(OrderInfo orderInfo) {  // 调用服务  // 将orderInfo转换成json  String json = JsonUtils.*objectToJson*(orderInfo);  // 提交订单数据  String result = HttpClientUtil.*doPostJson*(ORDER\_BASE\_URL+ORDER\_CREATE\_URL, json);  // 转换成java对象  TaotaoResult taotaoResult = TaotaoResult.*format*(result);  String orderId = taotaoResult.getData().toString();  return orderId; } |

## Controller层

接收提交的订单信息，使用OrderInfo接收。调用Service创建订单，跳转到成功页面，并且把订单号传递给页面。

请求的<url:/order/create>

|  |
| --- |
| @RequestMapping(value = "/create", method = RequestMethod.*POST*) public String createOrder(OrderInfo orderInfo,Model model,HttpServletRequest request){  // 获取用户信息  TbUser user = (TbUser)request.getAttribute("user");  // 补全OrderInfo属性  orderInfo.setUserId(user.getId());  orderInfo.setBuyerNick(user.getUsername());  // 调用服务  String orderId = orderService.createOrder(orderInfo);  // 将订单号传递给页面  model.addAttribute("orderId",orderId);  model.addAttribute("payment",orderInfo.getPayment());  model.addAttribute("date",new DateTime().plusDays(3).toString("yyyy-MM-dd"));  // 返回逻辑视图  return "success"; } |

# 淘淘商城第十一天

## 内容回顾

1. 购物车
   1. Cookie，把购物车放入cookie
   2. 实现简单，不占用服务端的资源
   3. 更换设备后购物车信息不同步
      1. 解决方案：当用户登录后，需要将Cookie的购物车信息，同步到Redis。
2. 订单系统
   1. 发布订单相关的服务。
   2. 创建订单。
   3. pc端和移动端可以共同使用同一个服务。
   4. 生成订单的过程全部使用关系型数据库，需要保证数据的完整性。

## 课程计划

1. 项目的部署
   1. Linux系统安装mysql
   2. 系统的架构
   3. 网络拓扑图
   4. 域名的规划
   5. 服务器的规划
   6. 实际部署，使用maven在tomcat下热部署
2. 项目总结
   1. 十二天的内容复习
   2. 面试中遇到的问题
   3. 答疑

# 系统架构

mycat分布式数据库解决方案

## 网络拓扑图

# 项目的部署

## 服务器规划

* + 1. 图片服务器 1台
    2. Nginx 1台
    3. 数据库服务器 1台
    4. taotao-manager 1台
    5. taotao-portal 1台 表现层
    6. redis 1台
    7. solr 1台
    8. rest、search、order、sso 1台 服务层

需要8台虚拟机

## 域名规划



使用一个公网ip，要访问很多系统，使用nginx的反向代理。

需要host文件来使用域名。

域名都需要指向nginx所在服务器的IP地址。

## 系统的部署方法

服务器层的ip:192.168.31.145

服务层同一台服务器上部署多个系统：

Taotao-rest 8080

taotao-search 8081

taotao-sso 8082

taotao-order 8083

## 进行tomcat热部署

修改tomcat-user.xml配置文件，配置用户、密码和权限

# LinkedList源码解析

LinkedList是一个双向列表，使用的是内部的Node内置类进行数据存储，主要存储的是，上一个节点，当前值，下一个节点，通过上下节点对象实现链接。

当前LinkedList，其实只存储了首节点与末节点两个数据，遍历时，是根据这两个数据节点，从前或从后遍历，获取对应值。

对于遍历方面，LinkedList是进行了中间切分(1.8版本)，若是index/2小于当前节点索引，从后往前遍历，若是大于当前节点，从前往后遍历，当遍历到当前索引时，结束循环，返回当前节点，从而获取其中item,即当前值。

# Quartz

quartz是OpenSymphony开源组织在Job SCHEDULING领域的开源项目，它可以与J2EE与J2SE莹莹程序相结合也可以单独使用。Qyarrz可以用来创建简单或为运行是个，摆个，甚至是好几万个Jobs这样复杂的日程序表。Jobs可以做成标准的Java组件或是EJBs。

Quartz是一个任务日程关系系统，一个在预先确定(被纳入日程)的时间到达时，负责执行(或者通知)其他软件组件的系统。

Quartz用一个小Java库发布文件(.jar文件)，这股库文件包含了所有Quartz核心功能。这些功能的主要接口(API)是Scheduler接口。它提供了简单的操作，例如：将任务纳入日程或者从日程中取消，开始/停止/暂停日程进度。

## 应用场景

下单后不付款，规定事件之后取消订单的动作。可以使用定时框架Quartz任务来完成。还有Timer

## Maven依赖

|  |
| --- |
| <dependency>  <groupId>org.quartz-scheduler</groupId>  <artifactId>quartz</artifactId>  <version>${quartz.version}</version> </dependency> <dependency>  <groupId>org.springframework</groupId>//和spring整合需要的jar包  <artifactId>spring-context-support</artifactId>  <version>${spring.version}</version> </dependency> |

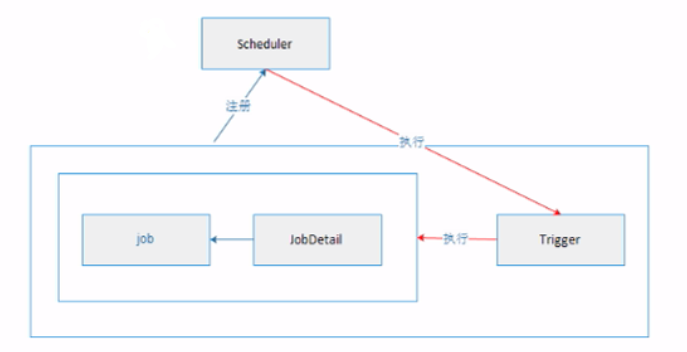
## Quartz框架架构

Scheduler - 核心调度器

Job - 任务 相当于接口

JobDetail - 任务描述 相当于实现类

Trigger - 触发器



### 流程描述

1. 定义job和jobdetail
2. 定义trigger和上面定义的job一起注册到scheduler中
3. scheduler通过trigger执行job

## 使用方法

### 第一步：创建Job

开发一个job类。创建一个java类，创建一个普通方法，作为任务处理方法。

|  |
| --- |
| public class Scheduler1 {  public void execute(){  System.*out*.println("任务已执行"+new Date());  } } |

### 第二步：配置Job到Spring容器

将类放入spring容器中，可以使用配置文件，也可以使用注解。

|  |
| --- |
| <!-- 配置Job类 --> <bean id="scheduler1" class="com.taotao.order.scheduler.Scheduler1" /> |

### 第三步：将Job类配置JobDetail

配置jobDetail，指定job对象。

|  |
| --- |
| <!-- 配置JobDetail --> <bean id="springQtzJobMethod" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">  <!-- 配置Job类 -->  <property name="targetObject" ref="scheduler1"></property>  <!-- 配置要执行的Job类中的方法 -->  <property name="targetMethod" value="execute"/> </bean> |

### 第四步：配置调度触发器

配置Trigger，需要指定一个cron表达式用来配置任务的执行时机。

|  |
| --- |
| <!-- 配置触发器trigger --> <bean id="cronTriggerFactoryBean" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">  <property name="jobDetail" ref="springQtzJobMethod" />  <property name="cronExpression" value="0/5 \* \* \* \* ?" /> </bean> |

### 第五步：配置调度工厂

配置一个TriggerFactory。

|  |
| --- |
| <!-- 配置调度工厂 --> <bean id="schedulerFactoryBean" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">  <property name="triggers">  <list>  <ref bean="cronTriggerFactoryBean"></ref>  </list>  </property> </bean> |

## Cron表达式

cron表达式被用来配置DCronTrigger实例。Cron的表达式是字符串，实际上是7个子表达式，描述个别细节的时间表。

这些子表达式使用空白进行分割

1. Seconds
2. Minutes
3. Hours
4. Day-of-Month
5. Month
6. Day-of-week
7. Year(可选字段)

例如：”0 0 12 ? \* WED” 在每星期三下午12点执行

个别子表达式可以包含范围，例如在前面的例子里WED可以替换成”MON-FRI”,”MON,WED,FRI”甚至是”MON-WED，SAT”

“\*”代表整个事件端，每一个字段都有以压力可以指定有效值。

# HashMap看源码笔记

HashMap是基于hashCode的散列值存储，使用散列值标识数据，并在散列值的基础上进行数据存储。

所以猜测，HashMap是基于散列值的数组存储，在这个基础上，为了避免HashCode碰撞，进行链式存储。

纠正：HashMap是基于散列值进行存储，但是HashCode的碰撞率很低很低很低，并不是为了避免HashCode碰撞才产生的链式存储，而是因为HashMap使用的是定长数组，所谓的索引是数组.length&hash值，这个值的碰撞率较高，所以才在此基础上进行的链式存储。

HashMap会在声明的时候就占据内存(合理猜测)。

HashMap存在最大值 1 >> 30 表明HashMap的最大存储空间为10亿左右

纠错：HashMap中数组的最大长度是 1>>30，但是每个数组值中理论上可以无限存储数据。

HashMap中存在一个内部静态类Node，Node会存储对应的hash值、key值、value值、与下一个Node对象 next。

猜测：看了Node对象，很天真的想HashMap是一个单项链表。但这样与猜测不符合，所以深入猜测Node对象可能会存储下一个节点对象，但是在hashCode值发生碰撞之后才会存储的，真正的存储方式还应继续。

可以在HashMap中发现getNode方法中

声明了一个变量Node<K,V>[] tab; 这个预期是相符合的。

数组中依据HashCode值进行数据链式存储。

在源码中发现了Node[]集合，这个应该是存储所有HashMap数据的真正存储空间标识。在源码395行。

在1388行，可以看到声明了一个tab，经过一系列繁杂的计算之后，得到的cap值。

根据猜测，Node<K,V>[]是用来存储HashMap数据的结果，全局变量中，唯有table是这种数据结果，推论是table是存储的对象介质。

在阅读源码中，唯有在readObject()中存在给table赋值，该方法是递归方法，但是没有找到初始调用方法。

纠错：在之后resize()中也发现了给table赋值语句。

在网络查询中，readObject并不是进行初始化赋值的，是根据HashMap对象，读取序列化数据所产生的方法，所以这种猜测错误。

重新对代码进行查询，发现，在HashMap中还有一个地方对table值进行了赋值，在resize()方法中，该方法是用来对HashMap进行扩容所存在的。

加断点，重新进行测试程序运行，发现此时的table为null，表明此时数据还未被初始化，所以可以肯定，resize()是不光进行扩容也进行初始化的方法。

找到了HashMap的初始化方法与存储结构。

但是HashMap的存储效率与读取效率够高，肯定有原因，并且还存在一种情况是HashCode值碰撞，会产生链表结构。

所以定位了HashMap的get方法。get方法定位之后，会调用getNode方法。

在getNode方法中，table值赋予了tab，在if语句中进行了赋值。

在569行可以看出，是通过(n-1)&hash值当做下标进行存储数据的。

n的值是table.length，是数组的长度-1，并和hash值进行与运算得到的值。

在getNode中对性能的主要体现是使用数组长度与hash值进行与运算作为数组的下标进行数据存储。

关于负载因子：

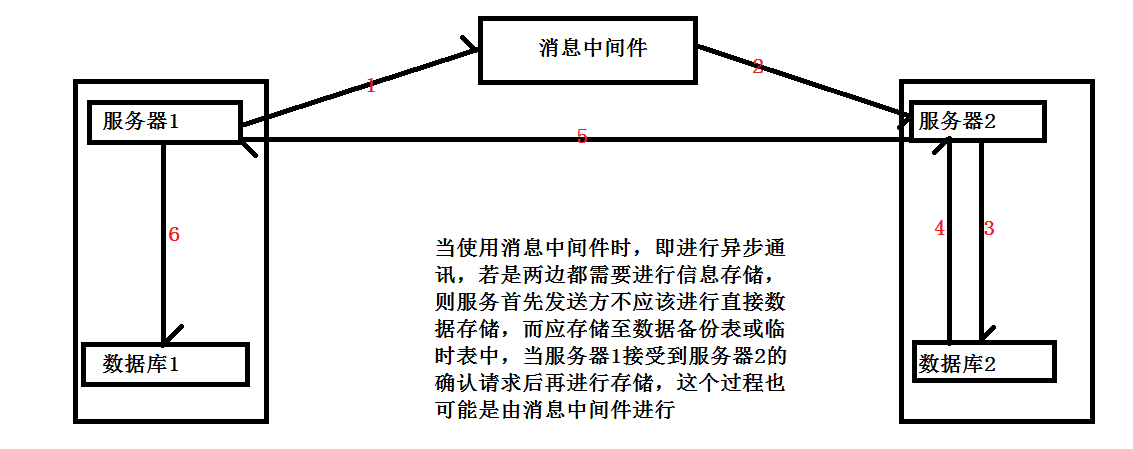
因为数组的长度固定，所以进行与hash的与运算时会出现重复的情况，这个时候就需要扩容当前数组与hash值与运算值相同所在节点，即进行链式存储，这就是一个基数问题，16个数分布在10个空间上与16个数分布在5个空间上的深度是不一样的，这个值其实可以无限大，比如说2,100等等，就会导致数据查询缓慢。

在hashMap中可以说是存在索引的，这个索引值是数组长度&键的hashCode值，当密度过大时，索引的效率依然存在，但是，链的深度过高，导致在链中的搜索时间过长，影响性能。

是很明显的以空间换时间理论。

谈及至此，数据库的索引，应该也类似于此，得出数组长度，计算key的hashCode值，从而确定每一个数据所在数组中的位置，在数组中直接获取数据。

分布式事务



不管是对于分布式系统还是对于单个项目，集群项目，其中所传递的东西全部都是数据，对象是无法进行传递的。

所以spring将所有的对象进行了统一的管理，但是却将需要传递的、会发生改变的数据信息全部交由外部程序去完成，可以说，spring已经将所有能以技术问题解决的业务问题解决，所剩下的全部都属于业务问题了。