**Springboot**

# 什么是Springboot

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。

## Spring Boot特性

1. 创建独立的Spring应用程序

2. 嵌入的Tomcat，无需部署WAR文件

3. 简化Maven配置

4. 自动配置Spring

5. 提供生产就绪型功能，如指标，健康检查和外部配置

6.开箱即用，没有代码生成，也无需XML配置。

## Spring Boot完美使用FastJson解析JSON数据

### 引入fastjson依赖库

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.2.15</version>

</dependency>

### 配置fastjon(支持两种方法)

第一种方法就是：

（1）启动类继承extends WebMvcConfigurerAdapter

（2）覆盖方法configureMessageConverters

第一种方式代码

@SpringBootApplication

**public** **class** ApiCoreApp **extends** WebMvcConfigurerAdapter {

@Override

**public** **void** configureMessageConverters(List<HttpMessageConverter<?>> converters) {

**super**.configureMessageConverters(converters);

FastJsonHttpMessageConverter fastConverter = **new** FastJsonHttpMessageConverter();

FastJsonConfig fastJsonConfig = **new** FastJsonConfig();

fastJsonConfig.setSerializerFeatures(

SerializerFeature.***PrettyFormat***

);

fastConverter.setFastJsonConfig(fastJsonConfig);

converters.add(fastConverter);

}

}

第二种方法

在App.java启动类中，

注入Bean : HttpMessageConverters

第二种方式代码

@Bean

public HttpMessageConverters fastJsonHttpMessageConverters() {

FastJsonHttpMessageConverter fastConverter = new FastJsonHttpMessageConverter();

FastJsonConfig fastJsonConfig = new FastJsonConfig();

fastJsonConfig.setSerializerFeatures(SerializerFeature.PrettyFormat);

fastConverter.setFastJsonConfig(fastJsonConfig);

HttpMessageConverter<?> converter = fastConverter;

return new HttpMessageConverters(converter);

}

### Springboot的FastJson的使用

HelloController.java

@RestController//等价于@Controller和@ResponseBody

public class HelloController {

@RequestMapping("/hello")//建立请求映射

public String hello(){

return "hello";

}

@RequestMapping("/getUser")

public User getUser(){

User user=new User();

user.setName("张三");

user.setPass("123");

user.setCreateTime(new Date());

user.setRemarks("this is remark");

return user;

}

}

User.java

public class User {

String name;

String pass;

/\*

如果不想返回remarks

deserialize是否需要序列化属性

\*/

@JSONField(serialize = false)

String remarks;

public String getRemarks() {

return remarks;

}

public void setRemarks(String remarks) {

this.remarks = remarks;

}

public Date getCreateTime() {

return createTime;

}

public void setCreateTime(Date createTime) {

this.createTime = createTime;

}

@JSONField(format = "yyyy-MM-dd HH:mm:ss")

Date createTime;//创建时间

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getPass() {

return pass;

}

public void setPass(String pass) {

this.pass = pass;

}

}

在User.java中，可以在属性上加注解 @JSONField(format = “yyyy-MM-dd HH:mm:ss”)格式化日期

@JSONField(serialize = false)属性不序列化，以及其他spring-fastjson注解对解析json

### Spring Boot热部署

添加spring-boot-devtools依赖包：

<dependency>

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

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

<optional>true</optional>

<scope>true</scope>

</dependency>

添加spring-boot-maven-plugin

<build>

<plugins>

<plugin>

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

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

<configuration>

<!--fork : 如果没有该项配置，肯呢个devtools不会起作用，即应用不会restart -->

<fork>true</fork>

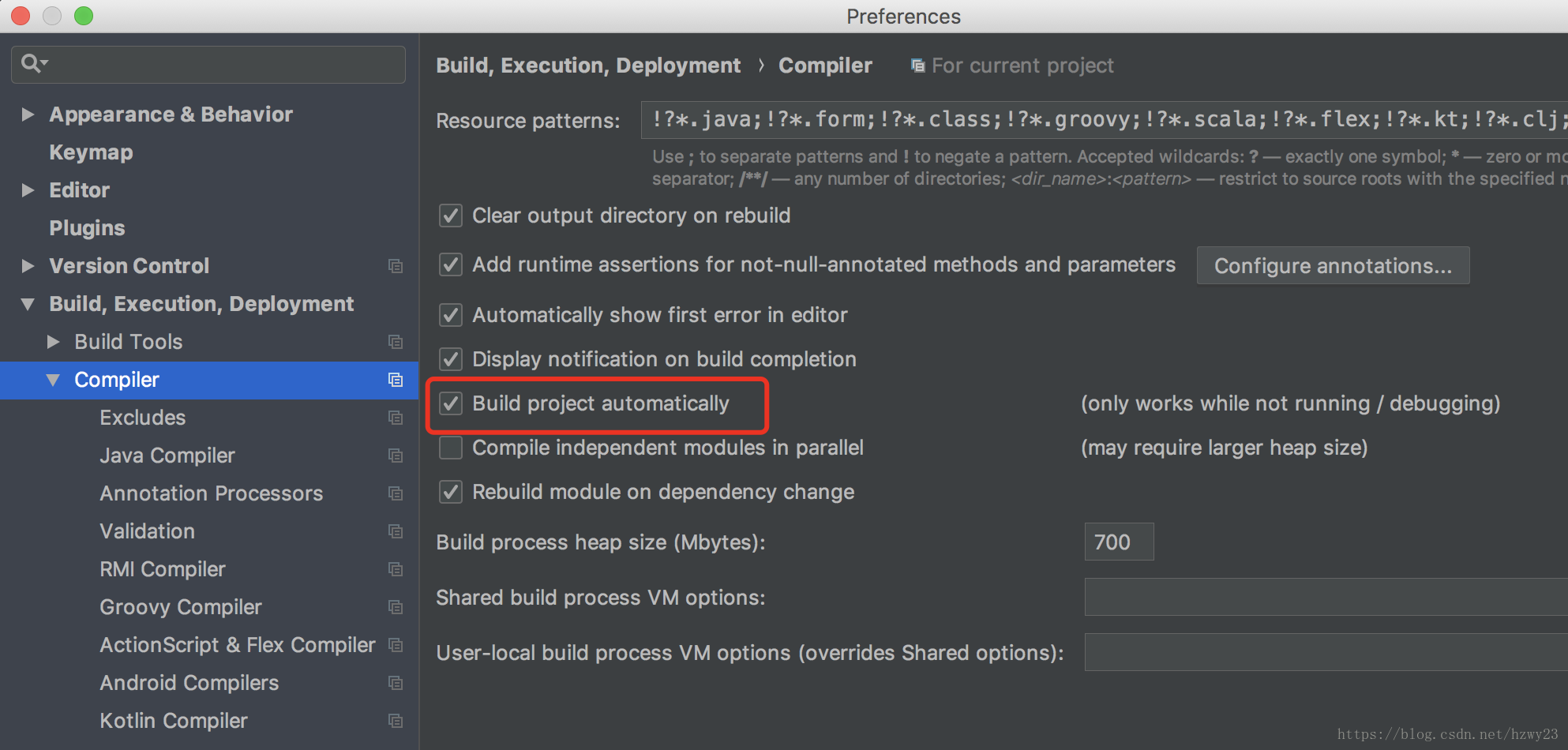
</configuration>

</plugin>

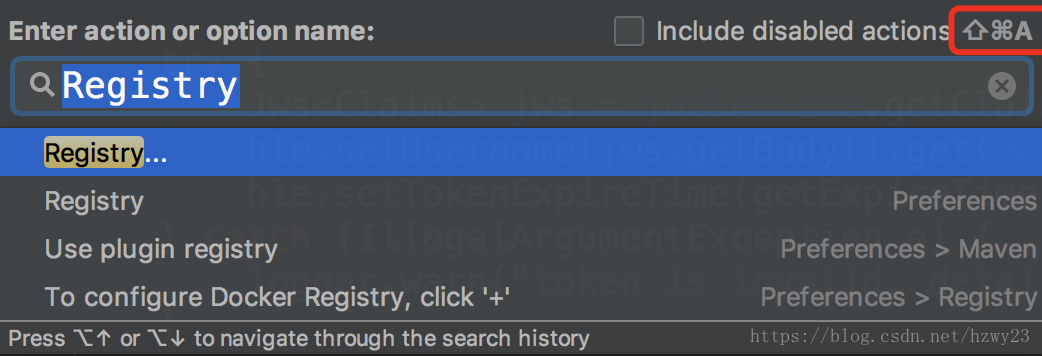
</plugins>

</build>

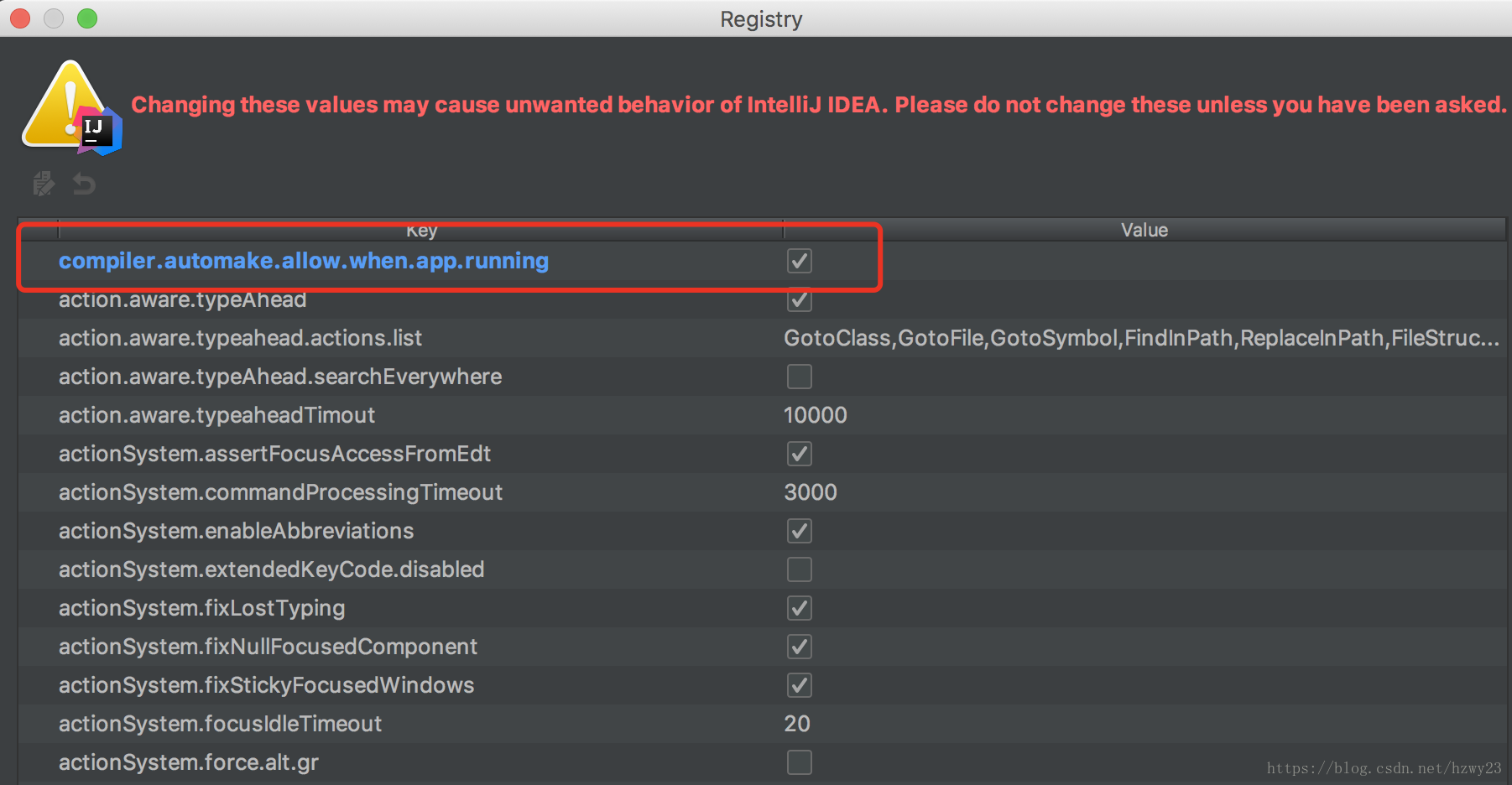
修改mac上的Intellij IDEA配置文件



修改Registry ，mac上快捷键：Command + Shift + A



勾选自动编译选项



# Spring Boot Spring Data JPA

## 什么是JPA?

JPA全称Java Persistence API.JPA通过JDK 5.0注解或XML描述对象－关系表的映射关系，并将运行期的实体对象持久化到数据库中

持久化（Persistence），即把数据（如内存中的对象）保存到可永久保存的存储设备中（如磁盘）。持久化的主要应用是将内存中的对象存储在的数据库中，或者存储在磁盘文件中、XML数据文件中等等。

持久化是将程序数据在持久状态和瞬时状态间转换的机制。

JDBC就是一种持久化机制。文件IO也是一种持久化机制。

## 什么是Hibernate

Hibernate是一个开放源代码的对象关系映射框架，它对JDBC进行了非常轻量级的对象封装，它将POJO与数据库表建立映射关系，是一个全自动的orm框架，hibernate可以自动生成SQL语句，自动执行，使得Java程序员可以随心所欲的使用对象编程思维来操纵数据库。 Hibernate可以应用在任何使用JDBC的场合，既可以在Java的客户端程序使用，也可以在Servlet/JSP的Web应用中使用，最具革命意义的是，Hibernate可以在应用EJB的[J2EE](http://baike.baidu.com/view/1507.htm)架构中取代CMP，完成[数据持久化](http://baike.baidu.com/view/4549557.htm)的重任。

## 什么是ORM

ORM是对象关系映射的意思，英语：Object Relational Mapping简称ORM，是一种程序技术，用于实现面向对象编程语言里不同系统类型的系统之间的数据转换。

## 什么是Spring Data

Spring Data是一个用于简化数据库访问，并支持云服务的开源框架。其主要目标是使得数据库的访问变得方便快捷，并支持map-reduce框架和云计算数据服务。此外，它还支持基于关系型数据库的数据服务，如Oracle RAC等。对于拥有海量数据的项目，可以用Spring Data来简化项目的开发，就如Spring Framework对JDBC、ORM的支持一样，Spring Data会让数据的访问变得更加方便

## Spring Boot JPA使用

pom.xml添加mysql,spring-data-jpa依赖

<dependency>

<groupId>mysql</groupId>

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

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

配置application.properties配置文件

########################################################

###datasource – 数据库配置\u6307\u5b9amysql\u6570\u636e\u5e93\u8fde\u63a5\u4fe1\u606f.

########################################################

spring.datasource.url = jdbc:mysql://localhost:3306/test

spring.datasource.username = root

spring.datasource.password = root

spring.datasource.driverClassName = com.mysql.jdbc.Driver

spring.datasource.max-active=20

spring.datasource.max-idle=8

spring.datasource.min-idle=8

spring.datasource.initial-size=10

########################################################

### Java Persistence Api -- Spring JPA的配置 JPAjpa\u7684\u914d\u7f6e\u4fe1\u606f.

########################################################

# Specify the DBMS

spring.jpa.database = MYSQL

# Show or not log for each sql query

spring.jpa.show-sql = true

# Hibernate ddl auto (create, create-drop, update)

spring.jpa.hibernate.ddl-auto = update

# Naming strategy

#[org.hibernate.cfg.ImprovedNamingStrategy #org.hibernate.cfg.DefaultNamingStrategy]

spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy

# stripped before adding them to the entity manager)

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

创建pojo层的实体类

/\*\*  
 \* 创建了一个实体类。  
 \* 如何持久化呢？  
 \* 1、使用@Entity进行实体类的持久化操作，当JPA检测到我们的实体类当中有  
 \* **@Entity** 注解的时候，会在数据库中生成对应的表结构信息。  
 \* 如何指定主键以及主键的生成策略？  
 \* 2、使用@Id指定主键.  
 \* \*/  
@Entity  
**public class** Dog {  
 /\*\*  
 \* 使用@Id指定主键.  
 \* 使用代码@GeneratedValue(strategy=GenerationType.AUTO)  
 \* 指定主键的生成策略，mysql默认的是自增长。  
 \*/  
 @Id  
 @GeneratedValue(strategy = GenerationType.**AUTO**)  
  
 **private** Integer **dogId**;  
 **private** String **dogName**;  
 **private** Integer **dogAge**;  
  
 **public** Integer getDogId() {  
 **return dogId**;  
 }  
  
 **public void** setDogId(Integer dogId) {  
 **this**.**dogId** = dogId;  
 }  
  
 **public** String getDogName() {  
 **return dogName**;  
 }  
  
 **public void** setDogName(String dogName) {  
 **this**.**dogName** = dogName;  
 }  
 **public** Integer getDogAge() {  
 **return dogAge**;  
 }  
  
 **public void** setDogAge(Integer dogAge) {  
 **this**.**dogAge** = dogAge;  
 }  
}

需要创建一个持久层的接口继承CrudRepository;

**public interface** CatDao **extends** CrudRepository<Dog,Integer>{  
   
}

idea的@Autowid 不能映射持久层的则需要进行一下操作进行设置

Editor –-> Inspection 🡪 spring🡪spring code🡪Autowiring for bean class 🡪 修改为Warning

**import** com.shi.springboot.pojo.Dog;  
  
**public interface** DogService {  
 **public void** save(Dog dog);  
 **public void** delete(**int** id);  
 **public** Iterable<Dog> findAll();  
}

@Service  
**public class** DogServiceImpl **implements** DogService {  
 @Autowired  
 **private** DogDao **dogDao**;  
 @Transactional  
 @Override  
 **public void** save(Dog dog) {  
 **dogDao**.save(dog);  
 }  
 @Transactional  
 @Override  
 **public void** delete(**int** id) {  
 **dogDao**.delete(id);  
 }  
 @Transactional  
 @Override  
 **public** Iterable<Dog> findAll() {  
 **return dogDao**.findAll();  
 }  
}

@RestController  
@RequestMapping(**"/dog"**)  
**public class** DogController {  
 @Autowired  
 **private** DogService **dogService**;  
 @RequestMapping(**"/save"**)  
 **public** String save(){  
 Dog dog = **new** Dog();  
 dog.setDogName(**"bigDog"**);  
 dog.setDogAge(15);  
 **dogService**.save(dog);  
 **return "save OK"**;  
 }  
 @RequestMapping(**"/findAll"**)  
 **public** Iterable<Dog> findAll(){  
 **return dogService**.findAll();  
 }  
 @RequestMapping(**"/delete"**)  
 **public** String delete(){  
 **dogService**.delete(2);  
 **return "delete OK"**;  
 }  
}

## jdbcTemplate

@Repository  
**public class** DogDao2 {  
 @Autowired  
 **private** JdbcTemplate **jdbcTemplate**;  
 **public** Dog selectByDogName(String dogName){  
 String sql = **"select** *\** **from dog where dog\_name = ?"**;  
 RowMapper<Dog> rowMapper = **new** BeanPropertyRowMapper<>(Dog.**class**);  
 Dog dog = **jdbcTemplate**.queryForObject(sql,**new** Object[]{dogName},rowMapper);  
 **return** dog;  
 }

**public interface** DogService {  
 **public void** save(Dog dog);  
 **public void** delete(**int** id);  
 **public** Iterable<Dog> findAll();  
 public Dog selectByDaoName(String dogName);  
}

@Service  
**public class** DogServiceImpl **implements** DogService {  
 @Autowired  
 **private** DogDao **dogDao**;  
 @Autowired  
 **private** DogDao2 **dogDao2**;  
 @Transactional  
 @Override  
 **public void** save(Dog dog) {  
 **dogDao**.save(dog);  
 }  
 @Transactional  
 @Override  
 **public void** delete(**int** id) {  
 **dogDao**.delete(id);  
 }  
 @Transactional  
 @Override  
 **public** Iterable<Dog> findAll() {  
 **return dogDao**.findAll();  
 }  
 @Override  
 public Dog selectByDaoName(String dogName) {  
 return dogDao2.selectByDogName(dogName);  
 }  
}

@RestController  
@RequestMapping(**"/dog"**)  
**public class** DogController {  
 @Autowired  
 **private** DogService **dogService**;  
 @RequestMapping(**"/save"**)  
 **public** String save(){  
 Dog dog = **new** Dog();  
 dog.setDogName(**"hello"**);  
 dog.setDogAge(15);  
 **dogService**.save(dog);  
 **return "save OK"**;  
 }  
 @RequestMapping(**"/findAll"**)  
 **public** Iterable<Dog> findAll(){  
 **return dogService**.findAll();  
 }  
 @RequestMapping(**"/delete"**)  
 **public** String delete(){  
 **dogService**.delete(2);  
 **return "delete OK"**;  
 }  
 @RequestMapping("/selectByDogName")  
 public Dog selectDogByDogName (String dogName){  
 return dogService.selectByDaoName(dogName);  
 }  
  
}

# 全局异常捕捉

新建一个类GlobalDefaultExceptionHandler，

在class注解上@ControllerAdvice,

在方法上注解上@ExceptionHandler(value = Exception.class)

@ControllerAdvice  
**public class** GlobleDefaultExceptionHandle {  
 @ExceptionHandler(Exception.**class**)  
 **public** String exceptionHandle(HttpServletRequest request , Exception e ,ModelAndView mv){  
 **return "对不起，网络错误"**;  
 }  
}

# 访问404

（1）404 -- 确定地址是否输入正确，，此路径非彼路径  
（2）404 -- 是否用对注解，此注解非彼注解  
（3）404 -- 包路径是否正确，此包非彼包

（4）404 -- 确认类包是否正确，此类包非彼类包

# springboot 配置properties文件

## 修改端口号

Spring boot 默认端口是8080，如果想要进行更改的话，只需要修改applicatoin.properties文件，在配置文件中加入：

server.port=8081

## 配置路径

Spring boot 默认路径为：/

server.context-path=/spring-boot

## 其他

#server.port=8080

#server.address= # bind to a specific NIC

#server.session-timeout= # session timeout in seconds

#the context path, defaults to '/'

#server.context-path=/spring-boot

#server.servlet-path= # the servlet path, defaults to '/'

#server.tomcat.access-log-pattern= # log pattern of the access log

#server.tomcat.access-log-enabled=false # is access logging enabled

#server.tomcat.protocol-header=x-forwarded-proto # ssl forward headers

#server.tomcat.remote-ip-header=x-forwarded-for

#server.tomcat.basedir=/tmp # base dir (usually not needed, defaults to tmp)

#server.tomcat.background-processor-delay=30; # in seconds

#server.tomcat.max-threads = 0 # number of threads in protocol handler

#server.tomcat.uri-encoding = UTF-8 # character encoding to use for URL decoding

# pringboot 中使用模版引擎

## spring boot使用thymeleaf

操作步骤

（1）在pom.xml中引入thymeleaf;

（2）如何关闭thymeleaf缓存

（3）编写模板文件.html

（4）编写访问模板文件controller

在pom.xml加入thymeleaf的依赖：

<dependency>

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

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

</dependency>

如何关闭thymeleaf缓存

###THYMELEAF (ThymeleafAutoConfiguration)

########################################################

#spring.thymeleaf.prefix=classpath:/templates/

#spring.thymeleaf.suffix=.html

#spring.thymeleaf.mode=HTML5

#spring.thymeleaf.encoding=UTF-8

# ;charset=<encoding> is added

#spring.thymeleaf.content-type=text/html

# set to false for hot refresh

spring.thymeleaf.cache=false

在templates文件中写入模版

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"

xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">

<head>

<title>Hello World!</title>

</head>

<body>

<h1>Hello.v.2</h1>

<p>${hello}</p>

</body>

</html>

编写contoller

@Controller  
@RequestMapping(**"/templates"**)  
**public class** TemplatesContrllor {  
 @RequestMapping(**"/hello"**)  
 **public** String hello(){**return "hello"**;  
 }  
 @RequestMapping(**"/hello2"**)  
 **public** String hello2(Map<String,Object> map){  
 map.put(**"name"**,**"大黄"**);  
 **return "hello"**;  
 }  
}

## Spring Boot 使用freemarker模版

步骤

（1）在pom.xml中引入freemarker;

（2）如何关闭freemarker缓存

（3）编写模板文件.ftl

（4）编写

在pom.xml中引入freemarker;

<dependency>

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

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

</dependency>

如何关闭freemarker缓存

###FREEMARKER (FreeMarkerAutoConfiguration)

########################################################

spring.freemarker.allow-request-override=false

spring.freemarker.cache=true

spring.freemarker.check-template-location=true

spring.freemarker.charset=UTF-8

spring.freemarker.content-type=text/html

spring.freemarker.expose-request-attributes=false

spring.freemarker.expose-session-attributes=false

spring.freemarker.expose-spring-macro-helpers=false

#spring.freemarker.prefix=

#spring.freemarker.request-context-attribute=

#spring.freemarker.settings.\*=

#spring.freemarker.suffix=.ftl

#spring.freemarker.template-loader-path=classpath:/templates/ #comma-separated list

#spring.freemarker.view-names= # whitelist of view names that can be resolved

@RequestMapping(**"/helloFtl"**)  
**public** String helloFtl(){  
 **return "helloFtl"**;  
}

# SpringBoot Web应用返回JSP页面配置

1.要返回jsp页面需要以下配置：

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

</dependency>

2.在application.yml里配置如下信息：

#路径名称按自己需要命名,在WEB-INF下新建文件夹page

spring:

mvc:

view:

prefix: /WEB-INF/page/

suffix: .jsp

3.在Controller里没什么变化

import org.springframework.ui.ModelMap;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/index")

public class IndexController {

@RequestMapping("/page")

public String indexPage(ModelMap model) {

model.addAttribute("hello", "13212");

return "hello";

}

4.jsp内容：

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>hello</title>

</head>

<body>

<h3>Hello</h3>

<div style="width:200px;height:100px;border:1px solid gray;">${hello }</div>

</body>

</html>