# Spring Boot介绍

什么是Spring boot？

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特性理解

为基于Spring的开发提供更快的入门体验

开箱即用，没有代码生成，也无需XML配置。同时也可以修改默认值来满足特定的需求。

提供了一些大型项目中常见的非功能特性，如嵌入式服务器、安全、指标，健康检测、外部配置等。

Spring Boot并不是对Spring功能上的增强，而是提供了一种快速使用Spring的方式。

# 创建工程步骤

官网下载带springboot的Eclipse地址：https://spring.io/tools3/sts/all 选择windows版本 下载zip（它就是一个eclipse版本）

## Springboot控制层开发

第一步 创建一个普通的maven工程

就是和之前的操作不变

第二步 pom.xml的设置

<!-- spring boot 父节点依赖,引入这个之后相关的引入就不需要添加version配置，spring boot会自动选择最合适的版本进行添加。-->

<parent>

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

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

<version>1.4.1.RELEASE</version>

</parent>

<!-- 指定一下jdk的版本 ，这里我们使用jdk 1.8 ,默认是1.6 -->

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

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

</properties>

<dependencies>

<!-- spring-boot-starter-web: 等于加入了webMVC,AOP等spring基本依赖包....-->

<dependency>

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

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

<!-- <version></version> 由于我们在上面指定了 parent(spring boot) 不用配版本号-->

</dependency>

</dependencies>

<build>

<plugins>

<!-- 这是spring boot devtool plugin -->

<plugin>

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

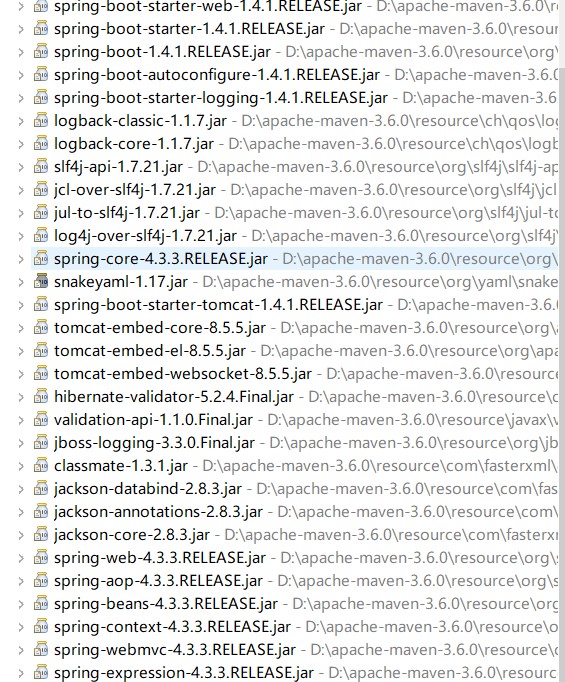
<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

加载的java如下



第三步 在同包目录下创建一个App.jar和UserController.java

UserController.java内容：主要是springmvc的Controller类

@RestController //等同于 Controller和ResponseBody的结合体，spring版本4.3以后支持

public class UserController {

@RequestMapping("nihao")

public String test() {

return "成功"; //这里作用就是响应客户端text文本信息了

}

}

页面显示：成功 //不知道为啥它不是乱码。Springmvc没有配置ContentType就是乱码，应该底层变成了utf-8了

App.jar此名字都可以可以随便取的，这里的内容重要，在这里它就是一个启动口

1、@SpringBootApplication 指定springboot是一个应用程序

2、SpringApplication.run(App.class, args);：这样就可以使用普通的java启动程序直接运行工程

/\*\*

\* 我们使用@SpringBootApplication指定这是一个spring boot的应用程序

\* @author WWL

\*

\*/

@SpringBootApplication

public class App {

//在main方法中进行启动我们的应用程序

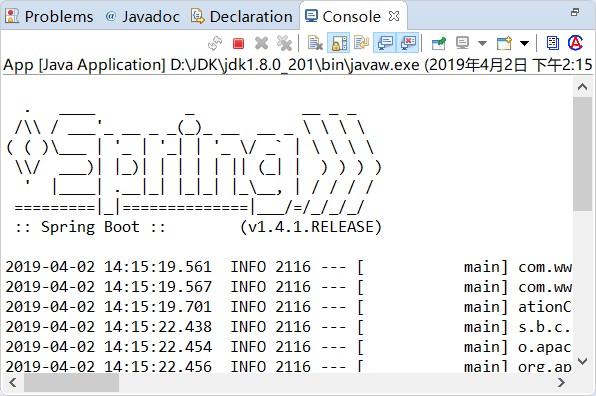
public static void main(String[] args) {

SpringApplication.run(App.class, args);

}

}

第四步 启动工程：在App.java页面直接右键 运行普通工程即可



## 热部署

在编写代码的时候，你会发现我们只是简单把打印信息改变了，就需要重新部署，如果是这样的编码方式，那么我们估计一天下来就真的是打几个Hello World就下班了。那么如何解决热部署的问题呢？那就是springloaded，(了解)

上面介绍的感觉没啥用，这里就不介绍了，放在上面让你还有印象：看下面内容spring-boot-devtools

spring-boot-devtools 是一个为开发者服务的一个模块，其中最重要的功能就是自动应用代码更改到最新的App上面去。原理是在发现代码有更改之后，重新启动应用，但是速度比手动停止后再启动还要更快，更快指的不是节省出来的手工操作的时间。

其深层原理是使用了两个ClassLoader，一个Classloader加载那些不会改变的类（第三方Jar包），另一个ClassLoader加载会更改的类，称为 restart ClassLoader

,这样在有代码更改的时候，原来的restart ClassLoader 被丢弃，重新创建一个restart ClassLoader，由于需要加载的类相比较少，所以实现了较快的重启时间（5秒以内）

使用方式步骤

第一步 添加依赖包和

spring-boot-devtool依赖包：作用：热部署

<dependency>

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

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

<optional>true</optional>

<scope>true</scope>

</dependency>

还有一个工具布局部署，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，上天添加依赖包devtools不会起作用，即应用不会restart -->

<fork>true</fork>

</configuration>

</plugin>

</plugins>

</build>

测试

说明：

1. devtools会监听classpath下的文件变动，并且会立即重启应用（发生在保存时机），注意：因为其采用的虚拟机机制，该项重启是很快的。

2. devtools可以实现页面热部署（即页面修改后会立即生效，这个可以直接在application.properties文件中配置spring.thymeleaf.cache=false来实现(这里注意不同的模板配置不一样)

测试：修改类-->保存：应用会重启

修改配置文件-->保存：应用会重启

修改页面-->保存：应用会重启，页面会刷新（原理是将spring.thymeleaf.cache设为false）

不能使用分析

1对应的spring-boot版本是否正确，这里使用的是1.4.1版本；

2是否加入plugin以及属性<fork>true</fork>

3Eclipse 菜单栏：Project—》Build Automatically 是否开启了（我自己就在这里栽了坑，不知道为什么我的工具什么时候关闭了自动编译的功能）。

4如果设置SpringApplication.setRegisterShutdownHook(false)，则自动重启将不起作用。

## 持久层mybatis开发

第一步 导包 和 application.properties

导包操作

<!-- spring boot parent节点，引入这个之后，在下面和spring boot相关的就不需要引入版本了; -->

<parent>

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

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

<version>1.4.1.RELEASE</version>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- 指定一下jdk的版本 ，这里我们使用jdk 1.8 ,默认是1.6 -->

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

</properties>

<dependencies>

<!-- web支持: 1、web mvc; 2、restful; 3、jackjson支持; 4、aop ........ -->

<dependency>

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

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

</dependency>

<!-- mysql 数据库驱动. -->

<dependency>

<groupId>mysql</groupId>

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

</dependency>

<!-- spring-boot mybatis依赖：

请不要使用1.0.0版本，因为还不支持拦截器插件，

1.1.1 是博主写帖子时候的版本，大家使用最新版本即可-->

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

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

<version>1.1.1</version>

</dependency>

Application.properties 这里只导入数据源即可

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

###datasource – mysql：useSSL=false遇到警告描述，原因是当前版本的MySQL需要指明是否进行SSL连接。

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

spring.datasource.url = jdbc:mysql://localhost:3306/test?characterEncoding=utf-8&useSSL=false

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

#mybatis设置

mybatis.configuration.map-underscore-to-camel-case: true #开启驼峰命名springboot默认是false如：数据库user\_name 🡪pojo字段userName就可以自动匹配了

mybatis.config-location=classpath:mybatis/mybatis-config.xml #配置mybatis的核心配置

第二步 启动类还需要添加扫描mapper接口类

Spring Boot启动类.

@SpringBootApplication

@MapperScan("com.kfit.\*")//扫描：该包下相应的class,主要是MyBatis的持久化类.

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class, args);

}

}

第三步 service层

@Transactional//添加事务.

public void save(Demo demo){

demoMappper.save(demo);

}

第四步 mapper层

使用注解方式

public interface DemoMappper {

//#{name}:参数占位符

@Select("select \*from Demo where name=#{name}")

public List<Demo> likeName(String name);

@Select("select \*from Demo where id = #{id}")

public Demo getById(long id);

@Select("select name from Demo where id = #{id}")

public String getNameById(long id);

//保存数据.

@Insert("insert into Demo(name) values(#{name})")

@Options(useGeneratedKeys=true,keyProperty="id",keyColumn="id") //返回主键id操作

public void save(Demo demo);

}

不使用注解方式:直接包xml写在同包下

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE mapper PUBLIC

"-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.wwl.mapper.UsersMapper">

<insert id="insert" parameterType="users">

insert into users(user\_name,user\_age) values(#{userName},#{userAge})

</insert>

</mapper>

<!--了解 需要包mapper.xml配置在不同的目录下：使用相对于类路径的资源：此配置没有硬性要求 -->

<mappers>

<package name="com.wwl.mapper"/>

<mapper resource="mybatis/mapper/UsersMapper.xml"/>

</mappers>

### PageHelper的使用

导包

<!-- MyBatis提供了拦截器接口，我们可以实现自己的拦截器，

将其作为一个plugin装入到SqlSessionFactory中。Github上有位开发者写了一个分页插件，我觉得使用起来还可以，挺方便的。

Github项目地址： https://github.com/pagehelper/Mybatis-PageHelper -->

<dependency>

<groupId>com.github.pagehelper</groupId>

<artifactId>pagehelper</artifactId>

<version>4.1.0</version>

</dependency>

第一步 复制以下类

类名称自定义

@Configuration //需要此注解

public class MyBatisConfiguration {

@Bean //需要此注解

public PageHelper pageHelper() {

System.out.println("MyBatisConfiguration.pageHelper()");

PageHelper pageHelper = new PageHelper();

Properties p = new Properties();

p.setProperty("offsetAsPageNum", "true");

p.setProperty("rowBoundsWithCount", "true");

p.setProperty("reasonable", "true");

pageHelper.setProperties(p);

return pageHelper;

}

}

第二步 然后就可以使用了

@RequestMapping("/likeName")

public List<Demo> likeName(String name){

/\*

\* 第一个参数：第几页;

\* 第二个参数：每页获取的条数.

\*/

PageHelper.startPage(1, 2); //在要查询之前写入这些代码即可

return demoService.likeName(name);

}

## 持久层Hibernate开发

### 介绍jpa

JPA(Java Persistence API)是Sun官方提出的Java持久化规范。它为Java开发人员提供了一种对象/关系映射工具来管理Java应用中的关系数据。

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。

我们先看一个描述：

Spring Data JPA能干什么

可以极大的简化JPA的写法，可以在几乎不用写实现的情况下，实现对数据的访问和操作。除了CRUD外，还包括如分页、排序等一些常用的功能。

首先我们需要清楚的是Spring Data是一个开源框架，在这个框架中Spring Data JPA只是这个框架中的一个模块，所以名称才叫Spring Data JPA。如果单独使用JPA开发，你会发现这个代码量和使用JDBC开发一样有点烦人，所以Spring Data JPA的出现就是为了简化JPA的写法，让你只需要编写一个接口继承一个类就能实现CRUD操作了。

### JPA/Hibernate 关系？

我们先看下别人的描述：

JPA是一种规范，而Hibernate是它的一种实现。除了Hibernate，还有EclipseLink(曾经的toplink)，OpenJPA等可供选择，所以使用Jpa的一个好处是，可以更换实现而不必改动太多代码。

### 简单步骤

这里的简单步骤其实是Spring Boot JPA再次封装了Hibernate简化使用操作

1. 在pom.xml添加mysql,spring-data-jpa依赖
2. 在application.properties文件中配置mysql连接配置文件；
3. 在application.properties文件中配置JPA配置信息；编写测试例子；

第0步 pom.xml添加jar包 添加了boot核心的pom配置默认就有hibernate核心的包

<!-- 添加MySQL数据库驱动依赖包. -->

<dependency>

<groupId>mysql</groupId>

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

</dependency>

<!-- 添加Spring-data-jpa依赖. -->

<dependency>

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

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

</dependency>

第一步 application.properties的配置

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

###datasource –数据库的配置方式如下.

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

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 – hibernate配置如下

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

# 连接那种数据库

spring.jpa.database = MYSQL #

# 显示sql语句

spring.jpa.show-sql = true

# (三个值create, create-drop, update) 常用update：即没有表进行创建，有表更新不删除数据；create：有表了也创建，并删除数据，

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

第二步 实体类操作(因为没了xml配置，都是注解方式进行操作的)

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 Cat {

/\*\*

\* 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 int id;//主键.

private String catName;//表中生成字段cat\_name

private int catAge;// 表中生成字段. cat\_age;

//getset省略

第三步 service层的操作

比较简单了：调用dao层继承的类中的方法

@Service

public class CatService {

@Resource

private CatRepository catRepository;

/\*\*

\* save,update ,delete 方法需要绑定事务.

\* 使用@Transactional进行事务的绑定.

\*/

//保存数据.

@Transactional

public void save(Cat cat){

catRepository.save(cat); //dao层继承类的方法

}

//删除数据

@Transactional

public void delete(int id){

catRepository.delete(id);

}

//查询数据.

public Iterable<Cat> getAll(){

return catRepository.findAll();

}

}

第四步 dao层的操作

1、创建接口类 继承CrudRepository带泛型类

//泛型参数：1、实体类名称，2、主键id的类型

public interface CatRepository extends CrudRep ository<Cat, Integer>{

}

这样空白即可：老师带我们看了CrudRepository源码就有很多增删改查的基本方法:还有别的接口类看[Dao层自定义方法](#_Dao层自定义方法)

控制层没什么可要注意的了

### Dao层自定义方法

Spring Boot Spring Data JPA介绍

[持久层Hibernate开发步骤](#_持久层Hibernate开发)最后dao层继承的CrudRepository就是其中的常用的一个类

Repository 接口是 Spring Data 的一个核心接口(所有都要继承它)：它不提供任何方法，开发者需要在自己定义的接口中声明需要的方法 ：

使用**Repository接口实现查询**需要注意以下几点

1. Repository是一个空接口，即是一个标记接口；

2. 若我们定义的接口继承了Repository，则该接口会被IOC容器识别为一个Repository Bean纳入到IOC容器中，进而可以在该接口中定义满足一定规范的方法。

3. 实际上也可以通过@RepositoryDefinition,注解来替代继承Repository接口。

4. 查询方法以find | read | get开头；

5. 涉及查询条件时，条件的属性用条件关键字连接，要注意的是条件属性以首字母大写。(条件关键字指定是实体类中的属性名称)

6.使用@Query注解可以自定义JPQL语句实现更灵活的查询。

两个例子：

public interface Cat2Repository extends Repository<Cat, Integer>{

/\*\*

\* 1/ 查询方法 以 get | find | read 开头.

\* 2/ 涉及查询条件时，条件的属性用条件关键字连接，要注意的是条件属性以首字母大写。(条件关键字指定是实体类中的属性名称)

\*/

//根据catName进行查询 : 根据catName进行查询.,这样定义即可，springboot就会给我么调出查询语句

public Cat findByCatName(String catName);

/\*\*

\* 如何编写JPQL语句， Hibernate -- HQL语句.

\* JPQL 语句 和HQL语句是类似的.

\*/

@Query("from Cat where catName=:cn")

public Cat findMyCatName(@Param("cn")String catName);

}

CrudRepository是继承Repository可以和它的查询方法 又扩展了几个方法

--T save(T entity);//保存单个实体

--Iterable<T> save(Iterable<? extends T> entities);//保存集合

--T findOne(ID id);//根据id查找实体

--boolean exists(ID id);//根据id判断实体是否存在

--Iterable<T> findAll();//查询所有实体,不用或慎用!

--long count();//查询实体数量

--void delete(ID id);//根据Id删除实体

--void delete(T entity);//删除一个实体

--void delete(Iterable<? extends T> entities);//删除一个实体的集合

--void deleteAll();//删除所有实体,不用或慎用!

PagingAndSortingRepository接口继承CrudRepository 它在CrudRepository又扩展了几个方法：分页需要的操作

该接口提供了分页与排序功能

--Iterable<T> findAll(Sort sort); //排序

--Page<T> findAll(Pageable pageable); //分页查询（含排序功能）

其它接口

JpaRepository：查找所有实体，排序、查找所有实体，执行缓存与数据库同步

JpaSpecificationExecutor：不属于Repository体系，实现一组 JPA Criteria 查询相关的方法，封装 JPA Criteria 查询条件。通常使用匿名内部类的方式来创建该接口的对象。

自定义 Repository：可以自己定义一个MyRepository接口继承它

# Application的配置

## Server配置

更改tomcat端口号、根URL访问路径，最大访问线程数，编码等

注意有些和老师的不一样的，应该是版本问题

#server.port=8080 #端口号

#server.address= # 绑定NIC地址

#server.session-timeout= # session会话超时时间

#the context path, defaults to '/'

#server.servlet.context-path=/spring-boot #老师的是server.context-path应该是版本问题

#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 # tomcat最大线程连接数

#server.tomcat.uri-encoding = UTF-8 # 编码

# Spring配置

视图解析器配置

spring.mvc.view.prefix=/WEB-INF/jsp/

spring.mvc.view.suffix=.jsp

# 模板的使用

## JSP使用

第一步 导包

<!-- 添加servlet接口依赖和普通的maven工程的servlet不同 -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<scope>provided</scope>

</dependency>

<!-- jstl标签库 -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

<!-- 添加tomcat的支持 两个配套 -->

<dependency>

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

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

<scope>provided</scope>

</dependency>

<dependency>

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

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

<scope>provided</scope>

</dependency>

<dependency>

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

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

<optional>true</optional>

<scope>true</scope>

第二步 创建目录

在Main下创建这两个目录/webapp/WEB-INF和文件web.xml

第三步 application.properties配置mvc视图解析器

注意：经测试不用配置静态文件的扫描了（js/html/css）

spring.mvc.view.prefix=/WEB-INF/jsp/

spring.mvc.view.suffix=.jsp

最后其它都一样了，该怎么配置就怎么配置

## Thymeleaf

简单说， Thymeleaf 是一个跟 Velocity、FreeMarker 类似的模板引擎，它可以完全替代 JSP 。

1.Thymeleaf 在有网络和无网络的环境下皆可运行，即它可以让美工在浏览器查看页面的静态效果，也可以让程序员在服务器查看带数据的动态页面效果。这是由于它支持 html 原型，然后在 html 标签里增加额外的属性来达到模板+数据的展示方式。浏览器解释 html 时会忽略未定义的标签属性，所以 thymeleaf 的模板可以静态地运行；当有数据返回到页面时，Thymeleaf 标签会动态地替换掉静态内容，使页面动态显示。

　　　2.Thymeleaf 开箱即用的特性。它提供标准和spring标准两种方言，可以直接套用模板实现JSTL、 OGNL表达式效果，避免每天套模板、该jstl、改标签的困扰。同时开发人员也可以扩展和创建自定义的方言。

　　　3. Thymeleaf 提供spring标准方言和一个与 SpringMVC 完美集成的可选模块，可以快速的实现表单绑定、属性编辑器、国际化等功能。

# 全局异常

非常简单：只要使用两个注解即可

1新建一个Class

2在class上添加注解，@ControllerAdvice

3在class中添加一个方法

4在方法上添加@ExceptionHandler拦截相应的异常信息

5 如果方法返回值得是View，方法返回值是ModelAndView

6如果返回的是String或者是json，方法上添加@ResponseBody注解

7 @author wenwenliang

@ControllerAdvice

public class GlobalDefaulExceptionHandler {

//这里使用String响应页面

@ResponseBody

@ExceptionHandler

public String defaultExceptionHandler(HttpServletRequest rea,Exception e) {

//返回的是String

return "对不起，服务器繁忙，请稍后再试";

}

}

# Json数据

其实看创建工程步骤完后：可以看到依赖进来很多jar包其中就有jackson给依赖进来了，

Pom.xml中会引入此jackson的配置

<!-- spring-boot-starter-web: webMVC,AOP等spring基本依赖包.... -->

<dependency>

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

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

</dependency>

Springboot底层已配置好加载jackson类，直接用即可

@RestController //等同于 Controller和ResponseBody的结合体，spring版本4.3以后支持

public class UserController {

@RequestMapping("ni")

public Demo testJson() {

Demo demo= new Demo(1, "@RestController就已实现ResponseBody，spring底层配置jackson类");

return demo; //

}

我们需要使用第三方的json解析框架：FastJson解析JSON数据

1. 引入FastJson包

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.2.15</version>

</dependency>

这里要说下很重要的话，官方文档说的1.2.10以后，会有两个方法支持HttpMessageconvert，一个是FastJsonHttpMessageConverter，支持4.2以下的版本，一个是FastJsonHttpMessageConverter4支持4.2以上的版本，具体有什么区别暂时没有深入研究。这里也就是说：低版本的就不支持了，所以这里最低要求就是1.2.10+。

有两种方式

第一种方法就是：

（1）启动类继承extends WebMvcConfigurerAdapter

（2）覆盖方法configureMessageConverters

@SpringBootApplication

public class App extends WebMvcConfigurerAdapter{

// 1、需要先定义一个 convert 转换消息的对象;

FastJsonHttpMessageConverter fastConverter = new FastJsonHttpMessageConverter();

//2、添加fastJson 的配置信息，比如：是否要格式化返回的json数据;

FastJsonConfig fastJsonConfig = new FastJsonConfig();

fastJsonConfig.setSerializerFeatures(

SerializerFeature.PrettyFormat

);

//3、在convert中添加配置信息.

fastConverter.setFastJsonConfig(fastJsonConfig);

//4、将convert添加到converters当中.

converters.add(fastConverter);

}

第二种方法 还是在启动类中进行操作

（1）在App.java启动类中，

注入Bean : HttpMessageConverters

/\*\*

\* 在这里我们使用 @Bean注入 fastJsonHttpMessageConvert

\* @return

\*/

@Bean

public HttpMessageConverters fastJsonHttpMessageConverters() {

// 1、需要先定义一个 convert 转换消息的对象;

FastJsonHttpMessageConverter fastConverter = new FastJsonHttpMessageConverter();

//2、添加fastJson 的配置信息，比如：是否要格式化返回的json数据;

FastJsonConfig fastJsonConfig = new FastJsonConfig();

fastJsonConfig.setSerializerFeatures(SerializerFeature.PrettyFormat);

//3、在convert中添加配置信息.

fastConverter.setFastJsonConfig(fastJsonConfig);

HttpMessageConverter<?> converter = fastConverter;

return new HttpMessageConverters(converter);

}