1. Spring分析
2. Spring优点分析

Spring是Java企业版（Java Enterprise Edition，J2EE）的轻量级代替品，无需开发重量级的Enterprise JavaBean(EJB)，Spring为企业级Java开发提供了一种相对简单的方法，通过依赖注入和面向切面编程，用简单的Java对象（POJO）实现了EJB的功能。

1. Spring的缺点分析

Spring的配置过程十分地繁琐，一开始，Spring使用XML进行配置，Spring2.5引入了基于注解的组件扫描后消除了大量XML配置；Spring3.0后引入了基于Java的配置，这是一种类型安全的可重构配置方式，可以替代XML。

同时，项目的依赖管理十分恼人，一旦导入的包的版本选择错误，那么包之间的不兼容问题会十分地恼人，且很难发现。

1. SpringBoot
2. SpringBoot的特点
3. 约定优于配置，没有代码生成，也没有XML配置，同时也可以修改默认值来满足特定的需求。
4. 提供了一些大型项目中常见的非功能性特性，如嵌入式服务器，安全，指标，健康监测，外部配置等。
5. SpringBoot不是对Spring功能上地增强，而是提供了一种快速使用Spring的方式。
6. SpringBoot的核心功能
7. 起步依赖

起步依赖的本质是一个Maven项目对象模型（Project Object Model，POM），定义了对其他库的传递依赖，这些东西加在一起支持某种功能，即打包一些坐标，提供默认的功能。

1. 自动配置

SpringBoot的自动配置是应用程序启动时考虑了多种因素，由SpringBoot自动决定Spring配置应该选择什么配置的过程。

1. SpringBoot快速入门
2. 创建Maven工程

在IDEA中创建一个Maven工程。

1. 添加SpringBoot的起步依赖

SpringBoot的核心功能是起步依赖，我们需要在项目中继承SpringBoot的起步依赖spring-boot-starter-parent，

<parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.0.1.RELEASE</version>  
</parent>

同时，由于SpringBoot要集成SpringMVC进行Controller的开发，因此项目要导入Web的起步依赖，在之前的Spring项目中，我们不仅要导入Spring的坐标，还要导入SpringMVC的坐标以及其他坐标，现在，我们直接导入一个web的起步依赖就可以了。

<dependencies>  
 <dependency>  
 <!--Web功能的起步依赖-->  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
</dependencies>

1. 导入SpringBoot引导类

配置完基本的信息后，我们需要添加一个SpringBoot的引导类，该类用于启动Spring，即运行SpringApplication，注意，SpringApplication的run函数中的参数是该引导类的字节码。同时要在该类上添加注解@SpringBootApplication标识这是一个SpringBoot的引导类。

package com.springboot;  
  
import org.springframework.boot.SpringApplication;  
  
@org.springframework.boot.autoconfigure.SpringBootApplication  
public class SpringBootApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringBootApplication.class);  
 }  
}

1. 编写Controller

在引导类的同级包或者子级包中创建一个Controller。Controller的写法以及注解与Spring中是相同的。

package com.springboot.controller;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.ResponseBody;  
  
@Controller  
public class QuickController {  
  
 @RequestMapping("/quick")  
 @ResponseBody  
 public String quick(){  
 return "hello springboot";  
 }  
}

1. SpringBoot工程热部署

在开发阶段，我们会反复地修改类，页面等资源，每次修改后都要重新启动修改才能够生效，这样非常麻烦，如何在不重启的情况下降修改生效呢？这就需要热部署来帮忙了。

1. 首先在pom.xml中添加如部署的依赖

<!-- 热部署配置-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-devtools</artifactId>  
</dependency>

1. 接下来设置IDEA的自动编译

选择File -> Settings -> Compiler 勾选 Build project automatically；

然后按住 Shift + Ctrl + Alt + /，选择Registry，勾选 compiler.automake.allow.when.app.running。

1. 设置IDEA中的Spring Boot 设置， Edit Configuration，将 On “Update” action 和 On frame deactivation 都改为 Update classes and resources。
2. 使用IDEA创建SpringBoot项目

在新建项目处，选择Spring Initializer，初始化一个Spring项目，重点是选择各种SpringBoot需要的启动依赖，根据自己的需要进行选择，SpringBoot会自动将依赖添加到pom.xml文件中。

1. SpringBoot原理分析
2. 起步依赖原理分析

（1） 分析spring-boot-starter-parent

我们在pom.xml中继承了spring-boot-starter-parent，点进去可以发现它又继承了spring-boot-starter-dependencies，再点进去发现，spring-boot-starter-dependencies中都是依赖的坐标，依赖管理，插件管理。也就是说在SpringBoot工程中继承了spring-boot-starter-parent后已经具备版本锁定等配置了，因此起步依赖的作用就是进行依赖的传递。

（2）分析spring-boot-starter-web

我们在pom.xml中导入了web的依赖spring-boot-starter-web，只需要这一个依赖，我们就可以使用springmvc来开发，点进去看看发现spring-boot-starter-web就是将web开发要使用的spring-web，spring-webmvc等坐标进行了打包，同样是依赖传递的作用。

1. 自动配置原理解析

SpringBoot的自动配置是通过引导类的@SpringBootApplication注解完成的，点进去可以看到其中包括了很多注解，主要的是@SpringBootConfiguration，@EnableAutoConfiguration和@ComponentScan注解。

（1）@SpringBootConfiguration

这个注解表示这是一个配置类，我们可以在引导类中进行Spring的配置。

（2）@EnableAutoConfiguration

这个注解表示开启SpringBoot的自动配置功能。

（3）@ComponentScan

这个注解表示将会扫描引导类所在包下的所有类

1. SpringBoot配置文件
2. SpringBoot配置文件作用

SpringBoot是基于约定的，所以很多配置都有默认值，但是如果想使用自己的配置替换默认的配置的话，就可以使用application.properties或者application.yml(application.yaml)来进行配置。

SpringBoot默认从Resources目录下加载application\*.properties或者application.yml(application.yaml)文件。.properties的加载顺序在.yam文件之后，因此，.properties文件中的内容会覆盖.yml中的内容。

1. application\*.yml配置文件
2. 概述

YML文件格式是YAML（YAML Aint Markup Language）编写的文件格式，YAML是一种直观地能够被电脑识别的数据序列化格式，容易阅读也容易与脚本交互，可以被不同语言导入。扩展名为.yaml或者.yaml。

1. yml配置文件语法
   1. 普通元数据

yml用缩进表示等级，如普通的元数据是在冒号后面加上一个空格，如下所示：

key: value

* 1. 对象数据

对象拥有多个数据，同一层级的用相同的缩进表示，如下所示，person对象有多个属性：

person:

name: haohao

age: 31

addr: beijing

或者可以写在同一行中，属性之间用逗号分隔开：

person: {name: haohao,age: 31,addr: beijing}

* 1. 数组集合数据

对于数组，集合数据，就要使用 “-” 加上 空格来表示这是集合中的一个数据，如下所示：

city:

* Beijing
* Tianjin
* shanghai

或者写在一行之中：

city: [Beijing,tianjin,shanhai]

若数组中有对象数据，那么直接将对象的属性一一写在数组的一个数据之中即可，如下所示：

student:

* name: zhangsan

age: 18

score: 100

* name: list
* age: 21
* score: 88

1. 配置文件与配置类的属性映射方式
   1. 使用注解@Value进行映射

我们在配置文件中配置了值以后，如何让Spring知道这些配置的作用呢？像Spring默认的配置键名Spring是认识的，但是对于那些我们自己创建的键名就无法识别了。因此，我们可以使用@Value注解将配置的键值映射到一个Spring管理的Bean的字段上。

如，配置文件application.yml中配置如下：

person:

name: zhangsan

age: 18

我们在类中将person对象的name属性和age属性一一映射：

@Controller

public class QuickController{

@Value(“${person.name}”)

private String name;

@Value(“${person.age}”)

private Integer age;

@RequestMapping(“/quick”)

@ResponseBody

public String quick(){

return name + age;

}

}

这种方式映射地非常准确，但是也非常地繁琐，需要一个个地进行映射。

* 1. 使用注解@ConfigurationProperties映射

注解@ConfigurrationProperties能够将配置文件中的配置自动与实体进行映射，如在注解的参数中设置配置文件中键值作为前缀，只要类中的参数与配置文件中相同，就可以直接映射，且该参数必须提供get和set方法。

application.yml配置如下：

person:

name: zhangsan

age: 18

类中的代码如下所示：

@Controller

@ConfigurationProperties(prefix = "person")

public class QuickStartController {

private String name;

private Integer age;

@RequestMapping("/quick")

@ResponseBody

public String quick(){

return "springboot 访问成功! name="+name+",age="+age;

}

public void setName(String name) {

this.name = name;

}

public void setAge(Integer age) {

this.age = age;

}

}

1. SpringBoot整合其他技术
2. SpringBoot整合Mybatis
   1. 添加Mybatis的起步依赖

<!--mybatis起步依赖-->

<dependency>

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

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

<version>1.1.1</version>

</dependency>

* 1. 添加数据库驱动坐标

<!-- MySQL连接驱动 -->

<dependency>

<groupId>mysql</groupId>

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

</dependency>

* 1. 添加数据库连接信息

在application.properties配置文件中添加数据库连接信息

#DB Configuration:

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

spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test?

useUnicode=true&characterEncoding=utf8

spring.datasource.username=root

spring.datasource.password=root

* 1. 创建实体Bean

public class User {

// 主键

private Long id;

// 用户名

private String username;

// 密码

private String password;

// 姓名

private String name;

//此处省略getter和setter方法 .. ..

}

* 1. 编写Mapper

@Mapper注解标记该类是一个Mybatis的mapper接口，可以被SpringBoot自动扫描到Spring上下文中。

@Mapper

public interface UserMapper {

public List<User> queryUserList();

}

* 1. 配置Mapper映射文件

在src\main\resources\mapper路径下添加UserMapper.xml配置文件：

<?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.itheima.mapper.UserMapper">

<select id="queryUserList" resultType="user">

select \* from user

</select>

</mapper>

* 1. 在application.properties中添加Mybatis的信息

我们需要修改SpringBoot的默认配置，让SpringBoot知道MyBatis扫描包以及映射文件位置：

#spring集成Mybatis环境

#pojo别名扫描包

mybatis.type-aliases-package=com.itheima.domain

#加载Mybatis映射文件

mybatis.mapper-locations=classpath:mapper/\*Mapper.xml

* 1. 编写测试Controller

@Controller

public class MapperController {

@Autowired

private UserMapper userMapper;

@RequestMapping("/queryUser")

@ResponseBody

public List<User> queryUser(){

List<User> users = userMapper.queryUserList();

return users;

}

}

1. SpringBoot整合Junit
   1. 添加Junit的起步依赖

一般SpringBoot中默认已经给我们设置了测试的起步依赖。

<!--测试的起步依赖-->

<dependency>

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

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

<scope>test</scope>

</dependency>

* 1. 编写测试类

@RunWith(SpringRunner.class)

@SpringBootTest(classes = MySpringBootApplication.class)

public class MapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void test() {

List<User> users = userMapper.queryUserList();

System.out.println(users);

}

}

之前的@RunWith注解中的参数是SpringJunit4ClassRunner，这里的SpringRunner继承自SpringJunit4ClassRunner。

@SpringBootTest注解中的参数指定的是SpringBoot引导类的字节码对象。意思是测试这个类的项目。

1. SpringBoot整合Redis
   1. 添加Redis的起步依赖

<!-- 配置使用redis启动器 -->

<dependency>

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

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

</dependency>

* 1. 配置Redis的链接信息

#Redis

spring.redis.host=127.0.0.1

spring.redis.port=6379

* 1. 注入RedisTemplate测试Redis

@RunWith(SpringRunner.class)

@SpringBootTest(classes = SpringbootJpaApplication.class)

public class RedisTest {

@Autowired

private UserRepository userRepository;

@Autowired

private RedisTemplate<String, String> redisTemplate;

@Test

public void test() throws JsonProcessingException {

//从redis缓存中获得指定的数据

1

2

3

4

5

6

1

2

3

4

5

1

2

3

1

2

3

4

5

6

7

8

9

10

11

12

13

String userListData = redisTemplate.boundValueOps("user.findAll").get();

//如果redis中没有数据的话

if(null==userListData){

//查询数据库获得数据

List<User> all = userRepository.findAll();

//转换成json格式字符串

ObjectMapper om = new ObjectMapper();

userListData = om.writeValueAsString(all);

//将数据存储到redis中，下次在查询直接从redis中获得数据，不用在查询数据库

redisTemplate.boundValueOps("user.findAll").set(userListData);

System.out.println("===============从数据库获得数据===============");

}else{

System.out.println("===============从redis缓存中获得数据===============");

}

System.out.println(userListData);

}

}