**Spring Boot开发**

20191230

目录

[第1章 SpringBoot简介 5](#_Toc2853)

[1.1. 微服务 5](#_Toc19377)

[1.2. SpringBoot简介 5](#_Toc11000)

[1.2.1. Spring、SpringMVC、SpringBoot的关系 5](#_Toc15119)

[1.2.2. SpringBoot的优点 5](#_Toc3834)

[1.3. 学习本书的建议 5](#_Toc7385)

[第2章 开发环境搭建 5](#_Toc21451)

[2.1. 搭建Java开发环境 5](#_Toc21753)

[2.2. Maven安装与配置 5](#_Toc18510)

[2.3. IDEA安装与配置 5](#_Toc3809)

[2.4. 第一个SpringBoot应用程序 5](#_Toc2532)

[第3章 SpringBoot基础 5](#_Toc23382)

[3.1. SpringBoot项目结构 5](#_Toc21373)

[3.2. SpringBoot的Starter 5](#_Toc18886)

[3.3. SpringBoot常用注解 5](#_Toc7712)

[3.4. 了解自动配置 6](#_Toc16672)

[3.5. HelloWorld程序解析 6](#_Toc26340)

[3.5.1. 依赖引入 6](#_Toc15488)

[3.5.2. 程序入口 6](#_Toc19814)

[第4章 配置文件 6](#_Toc29234)

[4.1. YAML简介 6](#_Toc5957)

[4.1.1. 基本语法 6](#_Toc24536)

[4.1.2. 数据类型 6](#_Toc18476)

[4.2. 配置文件加载顺序 6](#_Toc19639)

[4.3. Spring Boot基本配置介绍 6](#_Toc15212)

[4.4. 自动配置原理 7](#_Toc7034)

[第5章 日志 7](#_Toc21309)

[5.1. 日志简述 7](#_Toc20195)

[5.1.1. 日志的抽象层 7](#_Toc13075)

[5.1.2. 日志的实现层 7](#_Toc331)

[5.2. SpringBoot使用的日志-SLF4j 7](#_Toc26041)

[5.2.1. SpringBoot日志依赖关系 7](#_Toc19681)

[5.2.2. 使用SLF4j 7](#_Toc5380)

[5.2.3. 更改日志配置 7](#_Toc21244)

[5.3. 更改日志框架 7](#_Toc26717)

[第6章 SpringBoot Web开发 7](#_Toc17164)

[6.1. SpringMVC 7](#_Toc2173)

[6.1.1. 控制反转IOC 7](#_Toc1504)

[6.1.2. 面向切面AOP 8](#_Toc31236)

[6.1.3. 转发与重定向 8](#_Toc6241)

[6.1.4. 拦截器 8](#_Toc19928)

[6.1.5. 类型转换器 8](#_Toc14586)

[6.1.6. 异常处理 8](#_Toc23918)

[6.1.7. SpringMVC的自动配置 8](#_Toc25908)

[6.2. Thymeleaf模板引擎 8](#_Toc14955)

[6.2.1. Thymeleaf简介 8](#_Toc20827)

[6.2.2. 引入Thymeleaf 8](#_Toc29761)

[6.2.3. Thymeleaf语法规则 8](#_Toc4643)

[6.2.4. Thymeleaf应用实例 8](#_Toc11182)

[6.3. SpringBoot对JSP的支持 8](#_Toc12459)

[第7章 SpringBoot数据访问 8](#_Toc27469)

[7.1. Docker 8](#_Toc25417)

[7.1.1. 简介 9](#_Toc13723)

[7.1.2. Docker安装 9](#_Toc14853)

[7.1.3. Docker架构 9](#_Toc31764)

[7.1.4. Docker使用 9](#_Toc17332)

[7.1.5. Docker常用命令 9](#_Toc4187)

[7.1.6. 示例Docker安装MySQL 9](#_Toc6408)

[7.2. MyBatis 10](#_Toc26506)

[7.2.1. 简述 10](#_Toc25880)

[7.2.2. ORM框架原理 10](#_Toc2334)

[7.2.3. 引入mybatis 10](#_Toc32554)

[7.2.4. MyBatis注解方式开发 10](#_Toc3609)

[7.2.5. SpringBoot使用MyBatis 11](#_Toc1988)

[7.3. MongDB 11](#_Toc31307)

[7.3.1. 简介 11](#_Toc19280)

[7.3.2. 配置MongDB 11](#_Toc18664)

[7.3.3. 基本用法 11](#_Toc32470)

[7.3.4. SpringBoot使用MongDB 11](#_Toc3801)

[7.4. Redis 12](#_Toc20380)

[7.4.1. Redis简介 12](#_Toc17579)

[7.4.2. Redis安装与配置 12](#_Toc21756)

[7.4.3. Redis数据类型 12](#_Toc10630)

[7.4.4. Redis命令 12](#_Toc31713)

[7.4.5. SpringBoot使用Redis 12](#_Toc13210)

[第8章 SpringBoot Actuator 12](#_Toc25605)

[8.1. SpringBoot Actuator 简介 12](#_Toc24623)

[8.2. 原生端点使用 12](#_Toc18147)

[8.3. 定制 Actuator 12](#_Toc14050)

[第9章 Spring Security 12](#_Toc6531)

[9.1. Spring Security简介 12](#_Toc15601)

[9.2. 配置Spring Security 12](#_Toc1823)

[9.3. Spring Security入门 12](#_Toc2016)

[9.3.1. 框架原理 12](#_Toc12719)

[9.3.2. 核心组件 13](#_Toc4667)

[9.4. 实例 13](#_Toc28704)

[第10章 SpringBoot部署与测试 13](#_Toc19334)

[10.1. 热部署 13](#_Toc16135)

[10.2. 以jar文件部署 13](#_Toc25756)

[10.3. 以war文件部署 13](#_Toc3)

[10.4. @Profile注解 13](#_Toc9224)

[10.5. 单元测试 13](#_Toc17720)

[10.5.1. JUnit简介 13](#_Toc10781)

[10.5.2. Dao层单元测试 13](#_Toc4479)

[10.5.3. Service层单元测试 13](#_Toc2776)

[第11章 项目实战 13](#_Toc31553)

# SpringBoot简介

## SpringBoot简介

来看Spring官网对Spring Boot的介绍：Spring Boot使您可以轻松地创建独立的、生产级的、基于Spring的应用程序，您可以“只是运行”。我们对Spring平台和第三方库有一个独到的见解，这样您就可以从最少的麻烦开始了。大多数Spring引导应用程序需要最少的Spring配置。

Spring Boot对一些常用的功能、一些第三方库做了自动配置，使得你在使用Spring Boot的时候无需先进行重复而又烦人的工作，你只需要对项目做一些简单的、少量的配置就可以像运行一个最简单的jar程序一样运行一个Spring Boot应用。

### Spring、SpringMVC、SpringBoot的关系

Spring 是一个开源框架，是为了解决企业应用程序开发复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许您选择使用哪一个组件，同时为 J2EE 应用程序开发提供集成的框架。Spring这个轻量级的框架已经成为web开发事实上的标准。

Spring为javaEE开发主要提供了以下功能：

IOC：资源集中管理，实现资源的可配置和易管理。降低了使用资源双方的依赖程度，也就是我们说的耦合度。

AOP：的将一些通用任务，如安全、事务、日志等集中进行管理，提高了复用性和管理的便捷性

ORM和DAO提供了与第三方持久层框架的良好整合，简化了底层数据访问。

Web MVC：使得复杂项目更加容易维护，减少项目之间的耦合。

Spring Boot集成了Spring的所有组件，你可以在程序中使用这些组件。

Spring Boot在Spring的基础上扩充了许多功能，许多特性。而spring mvc 是只是spring 处理web层请求的一个模块。

因此他们的关系可以描述成这样：

spring mvc < spring < springboot。

### SpringBoot的优点

## 微服务

## 学习本书的建议

# 开发环境搭建

## 搭建Java开发环境

## Maven安装与配置

## IDEA安装与配置

## 第一个SpringBoot应用程序

# SpringBoot基础

## SpringBoot项目结构

Spring Boot框架对项目的结构并没有做强制约束，但是分工良好、结构清晰的项目结构可以规范开发的模块分工，增强代码的可读性，降低维护的难度与成本。维护一个结构清晰的项目结构是十分有必要的。

下面介绍一个典型的Spring Boot开发Web应用的项目结构。

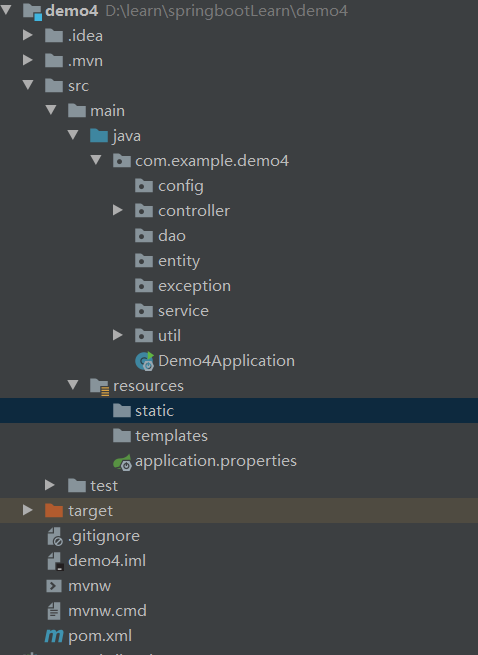


图3-1

### pom.xml

POM是项目对象模型(Project Object Model)的简称,它是Maven项目中的文件.主要用来管理项目的依赖关系，再这个文件中可以选择引入项目所需要的依赖。还可以用于管理：源代码、配置文件、开发者的信息和角色、问题追踪系统、组织信息、项目授权、项目的url等等。

### src

1. **main**

main层由java目录和resources组成，java目录存放java代码，resources目录存放资源。

**代码层java:**

1. 启动类Demo4Application,项目的入口类。
2. Config:存放项目的配置类
3. Controller: 存放作为controller的类
4. Dao:数据接口访问层
5. entity:实体层
6. Exception:存放自定义的异常类
7. Serivce:数据服务层
8. Util:放一些常用的工具类

**资源目录resources**

1. 项目配置文件application.yml/application.properties
2. 静态资源目录static: 用于存放html、css、js、图片等资源

3.放视图模板，springBoot的动态页面

1. **test**

test层的作用是在项目的开发阶段进行测试工作。

### Target

target用来存放项目构建后的文件和目录、jar包、war包、编译的class文件。

## SpringBoot的Starter

### Starter是什么

starter可以称为场景启动器。一个starter包含了这个应用这个场景所需要的jar包，和这个场景一些初始化配置。

### 怎么使用Starter

要使用starter,只需要将这个starter再pom文件中引入即可。例如我们想要使用web相关的功能。只需要：

1.在pom.xml文件中加入

<dependency>

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

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

</dependency>

这样就在项目总引入了web相关的jar包以及初始配置。

2.想要指定自己想要的配置，只要在配置文件中写入即可。例如想要指定程序的启动端口号，可以在application.properties写入server.port=”端口号”或在application.yml中写入:

server:

Port=”端口号”

1. 编写自己的业务代码

### Starter原理介绍

下面通过实现一个自定义的Starter来介绍Starter的原理。

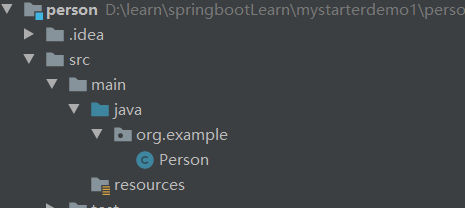
通过以下步骤来实现，

1. 编写要封装的组件
2. 编写自动配置组件
3. 编写starter
4. 导入starter并测试

下面完成上述四个步骤来实现一个自定义的starter

1. 编写要封装的组件

创建一个maven项目



**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">

<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>

<artifactId>person</artifactId>

<version>1.0-SNAPSHOT</version>

</project>

**Person.class**

public class Person {

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

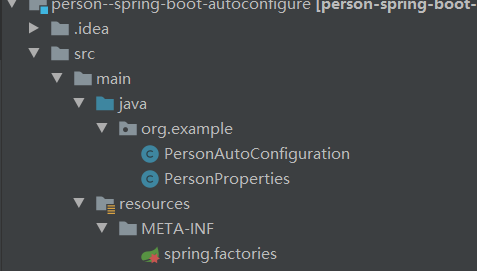
}

}

然后执行命令mvn clear install将组件打包到maven仓库。

2.编写自动配置组件

创建一个maven项目



**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">

<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>

<artifactId>person-spring-boot-autoconfigure</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

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

<artifactId>spring-boot-autoconfigure</artifactId>

<version>2.2.5.RELEASE</version>

</dependency>

<dependency>

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

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

<version>2.2.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.example</groupId>

<artifactId>person</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

</dependencies>

<build>

<resources>

<resource>

<directory>src/main/resources</directory>

<includes>

<include>META-INF/\*</include>

</includes>

</resource>

</resources>

</build>

</project>

引入spring-boot-autoconfigure、spring-boot-configuration-processor还有person组件依赖。

**PersonAutoConfiguration.class**

@Configuration

@EnableConfigurationProperties(PersonProperties.class)

public class PersonAutoConfiguration {

@Bean

public Person person(PersonProperties properties){

Person person = new Person();

person.setName(properties.getName());

person.setAge(properties.getAge());

return person;

}

}

这是一个自动配置类，可以为容器添加一个person组件（bean），这里用@EnableConfigurationProperties(PersonProperties.class)注解使得PersonProperties.class注入容器，person对象的属性值从PersonProperties.class中获得。

**PersonProperties.class**

@ConfigurationProperties(prefix = "person")

public class PersonProperties {

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

使用@ConfigurationProperties注解将配置文件（yml/properties）中指定前缀的配置（这里是 "person"）转为PersonProperties对象中属性的值。

**spring.factories**

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

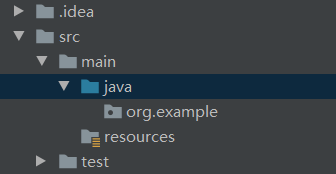
org.example.PersonAutoConfiguration

类路径下 META-INF/spring.factories 里面配置的所有EnableAutoConfiguration的值被加入到了容器中，用他们来做各种功能的自动配置。

然后执行命令mvn clear install将这个自动配置组件打包到maven仓库。

3.编写starter

新建一个maven项目



**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">

<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>

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

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

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

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

<version>2.2.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.example</groupId>

<artifactId>person-spring-boot-autoconfigure</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

</dependencies>

</project>

可以看到这个starter项目是没有java代码的，只有一个pom.xml文件，在pom.xml文件中引入了org.example.person-spring-boot-autoconfigure依赖。

1. 导入starter并测试

新建一个springBoot工程。在pom.xml文件中引入依赖：

<dependency>

<groupId>org.example</groupId>

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

<version>1.0-SNAPSHOT</version>

</dependency>

在配置文件application.properties中写入：

person.name=zhangsan

person.age=12

运行测试方法personTest()：

@SpringBootTest

class DemoApplicationTests {

@Autowired

Person person;

@Test

void personTest(){

System.out.println(person);

System.out.println(person.getName()+person.getAge());

}

}

运行结果如下：

starter4

这个运行结果说明了此person组件（bean）成功注入了容器，间接说明了自动配置类生效，自定义starter成功实现。

总结：starter本身没有java,它的作用是导入一些依赖包，依赖包中有自动配置类，向应用提供各种功能组件。

## SpringBoot常用注解

### @Component

带此注解的类看为组件，当使用基于注解的配置和类路径扫描的时候，这些类就会被实例化。

@Component

Class Person{

String name;

}

上例中这个Person就会被实例化，由Spring IOC容器管理。使用时可以用@Autowired注解来注入。

@Autowired

Person person;

这个注解也可以当作元注解使用，来注解其他注解。

如@Controller注解就是一个被@Component注解的注解。当@Controller被@Component注解了后，@Controller也就有了@Component注解的特性。所有被@Controller注解的类也会被IOC容器管理。

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Controller {

### @Bean

Spring的@Bean注解用于告诉方法，产生一个Bean对象，然后这个Bean对象交给Spring管理。产生这个Bean对象的方法Spring只会调用一次，随后这个Spring将会将这个Bean对象放在自己的IOC容器中。

bean都需要在@Configuration注解下进行创建，在一个方法上使用@Bean注解就表明这个方法需要交给Spring进行管理。

例如：

public class E2 {

public E2(){

System.out.println("E2 initialize");

}

}

@Configuration

public class BeanConfig {

@Bean

public E2 myBean(){

return new E2();

}

}

@Test  
public void BeanTest(){  
 ApplicationContext context = new AnnotationConfigApplicationContext(BeanConfig.class);

//bean名称为指定时，默认的时方法的名称,可以用@Bean(name = "xx")来指定bean的名称  
 Object myBean = context.getBean("myBean");  
}

### @Import

有时没有把某个类注入到IOC容器中，但在使用的时候需要获取该类对应的bean，这时就可以用到@Import注解。

@Import的用法有以下4种：

1.直接用类名导入bean

2.通过@Configuration注解的配置类导入bean

3.通过ImportSelector的实现类导入bean

4.通过ImportBeanDefinitionRegistrar的实现类导入bean

下面给出示例，四种方式的用法分别如下：

1.导入一个bean  
public class Bean1 {

}

@Configuration

@Import({Bean1.class})

public class MyConfig {

}

2.导入@Configuration注解的配置类

public class Bean2 {

}

@Configuration

public class BeanConfig {

@Bean

public Bean2 getBean2(){

return new Bean2();

}

}

@Configuration

@Import({BeanConfig.class})

public class MyConfig {

}

3.导入ImportSelector的实现类

public class Bean3 {

}

public class MyImportSelector implements ImportSelector {

@Override

public String[] selectImports(AnnotationMetadata annotationMetadata) {

return new String[]{"com.example.demo4.ImportDemo.Bean3"};

}

}

@Configuration

@Import({MyImportSelector.class})

public class MyConfig {

}

4.导入ImportBeanDefinitionRegistrar的实现类

public class Bean4 {

}

public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {

@Override

public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

RootBeanDefinition rootBeanDefinition = new RootBeanDefinition(Bean4.class);

registry.registerBeanDefinition("Bean4", rootBeanDefinition);

}

}

@Configuration

@Import({MyImportBeanDefinitionRegistrar.class})

public class MyConfig {

}

在此我们写一个统一的配置类MyConfigAll,和一个测试类同一测试以上4种方式。

@Configuration

@Import({Bean1.class,BeanConfig.class,MyImportSelector.class,MyImportBeanDefinitionRegistrar.class})

public class MyConfigAll {

}

再spring Boot的测试类里新建一个方法

@Test

public void ImportTest(){

AnnotationConfigApplicationContext annotationConfigApplicationContext = new AnnotationConfigApplicationContext(MyConfig.class);

String[] beanDefinitionNames = annotationConfigApplicationContext.getBeanDefinitionNames();

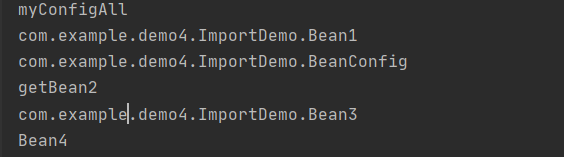
for (String beanDefinitionName: beanDefinitionNames) {

System.out.println(beanDefinitionName);

}

}

测试这个方法，结果如下：



可以看出Bean1,Bean2,Bean3,Bean4通过不同的4种导入方法被导入SpringIOC容器中。

### @Configuration

按住Ctrl点入@Configuration，可以进入此注解的定义

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Configuration {...}

可见它被@Component注解了，因此@Configuration注解有@Component的所有特性，意味也将会注册为bean, 也可以依赖注入。

被@Configuration注解的类内部包含有一个或者多个被@Bean注解的方法，这些方法将会被ApplicationContext上下文类进行扫描，并构建对应的bean,加入到Spring容器之中进行管理。

@Configuration

public class TestConfiguration {

@Bean

public Bean getBean() {

return new Bean();

}

}

### @ComponentScan

@ComponentScan主要就是定义扫描的路径从中找出标识了需要装配的类自动装配到spring的bean容器中

@ComponentScan的主要用法是：

1. 扫描路径下边带有@Component注解的类加入spring容器（@Controller，@Service，@Repository等被@Component注解的注解也是）

例：@ComponentScan(value = "com.demo.controller")

会扫描com.demo.controller包下带有@Component注解的类加入spring容器

如果只写@ComponentScan后面不指定特定的包，则会扫描此@ComponentScan注解所在包下的带有@Component注解的类加入spring容器。

1. 通过includeFilters来按照规则只包含某些包的扫描。

@ComponentScan(value = "com.demo",

includeFilters = {@Filter(type = FilterType.ANNOTATION, classes = {Controller. class})},useDefaultFilters = false)

public class MyBeanConfig {...}

以上，指定被 @Controller 注解的类才会被注册到容器中。includeFilters的参数是一个 Filter[] 数组，然后指定 FilterType 的类型为 ANNOTATION，也就是通过注解来过滤，最后的 value 则是Controller 注解类。配置之后，在 spring 扫描的时候，就会将 com.demo 包下，所有被 @Controller 注解标注的类加入容器。这里的useDefaultFilters的属性要设置为false。因为useDefaultFilters 的默认值是true,它的作用是会自动发现被 @Component、@Repository、@Service 和 @Controller 标注的类，所有这里把useDefaultFilters 的值设为false来关闭这个功能。

3.通过excludeFilters过滤出不用加入spring容器的类

用excludeFilters可以排除来按照规则排除某些包的扫描，它的用法和includeFilters 类似。

@ComponentScan(value = "com.demo",

excludeFilters = {@Filter(type = FilterType.ANNOTATION,

value = {Controller.class})})

public class MyBeanConfig {

在 spring 扫描的时候，就会跳过 com.demo 包下，所有被 @Controller 注解标注的类。

### @Conditional

@Conditional的作用是根据某个条件来决定是否创建Bean，通过实现Condition接口，并重写matches接口来构造判断条件，我们可以利用这个特性进行一些自动的配置。

下面将以不同的操作系统为条件，通过实现Condition接口，并重写其matches方法来构造判断条件。若在Windows系统下运行程序，则注入cat bean；若在Linux系统下运行程序，则注入dog bean。

1.创建Dog和Cat类

public class Cat {

}

public class Dog {

}

2.创建MyBeanConfig 配置类，来根据条件注入不同的bean

@Configuration

public class MyBeanConfig {

@Conditional(WindosCondition.class)

@Bean("cat")

public Cat getCat(){

return new Cat();

}

@Conditional(LinuxCondition.class)

@Bean("dog")

public Dog getDog(){

return new Dog();

}

}

3.创建LinuxCondition 和WindosCondition

public class LinuxCondition implements Condition {

@Override

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata){

return context.getEnvironment().getProperty("os.name").contains("Linux");

}

}

public class WindosCondition implements Condition {

@Override

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata){

return context.getEnvironment().getProperty("os.name").contains("Windows");

}

}

4.在测试类种创建一个测试方法，来看有注入了哪些bean

@Test

public void beanConditionTest(){

AnnotationConfigApplicationContext annotationConfigApplicationContext = new Annotati onConfigApplicationContext(MyBeanConfig.class);

String[] beanDefinitionNames = annotationConfigApplicationContext.getBeanDefinitionNa mes();

for (String beanDefinitionName: beanDefinitionNames) {

System.out.println(beanDefinitionName);

}

String osName = annotationConfigApplicationContext.getEnvironment().getProperty("os.n ame");

System.out.println("当前系统为：" + osName);

}

可以看到注入了cat这个bean:

beancondition

## 了解自动配置

在springboot启动的时候,加载主配置类过程中,开启了自动配置的功能.@EnableAutoConfiguration。

@SpringBootApplication是一个组合注解：

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),

@Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })

public @interface SpringBootApplication {...}

主要起作用的就是@EnableAutoConfiguration注解

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@AutoConfigurationPackage

@Import(AutoConfigurationImportSelector.class)

public @interface EnableAutoConfiguration {...}

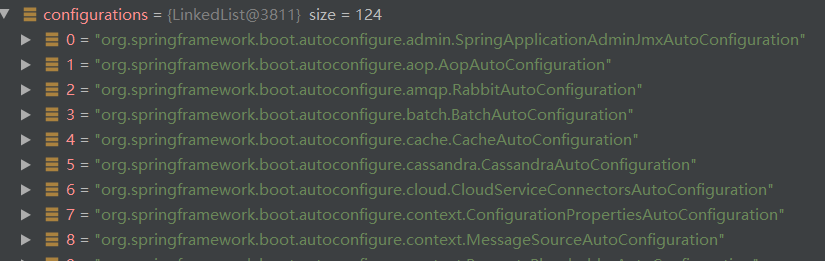
这里利用AutoConfigurationImportSelector给容器中导入一些,自动配置的组件。

看到AutoConfigurationImportSelector中的getCandidateConfigurations(annotationMetadata, attributes)方法，

这个方法中的SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(),getBeanClassLoader())方法会扫描所有jar包类路径下 META-INF/spring.factories

把扫描到的这些文件的内容包装成properties对象，再从properties中获取到EnableAutoConfiguration.class类（类名）对应的值，然后把他们添加在容器中。

在SpringFactoriesLoader.loadFactoryNames(...)处打一个断点，再以debug模式启动程序可以看到：



类路径下 META-INF/spring.factories 里面配置的所有EnableAutoConfiguration的值被加入到了容器中，这些用他们来做各种功能的自动配置。

这样各种XxxAutoConfiguratio被加载到了容器中，它们会向容器中加入各种功能的组件。

下面以HttpEncodingAutoConfiguration来阐述XxxAutoConfiguratio是怎么进行自动配置的。

@Configuration(proxyBeanMethods = false)

//将配置文件中对应的值和HttpEncodingProperties绑定起来；并把HttpEncodingProperties加入到ioc容器中

@EnableConfigurationProperties(HttpProperties.class)

//判断当前应用是否是web应用，如果是，当前配置类生效

@ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.SERVLET)

//判断应用中是否有CharacterEncodingFilter类,如果有 ,当前配置类生效

@ConditionalOnClass(CharacterEncodingFilter.class)

//判断配置文件中是否存在某个配置 spring.http.encoding.enabled；如果不存在，判断也是成立的（matchIfMissing = true的作用）

@ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled", matchIfMissing = true)

public class HttpEncodingAutoConfiguration {

private final HttpProperties.Encoding properties;

public HttpEncodingAutoConfiguration(HttpProperties properties) {

this.properties = properties.getEncoding();

}

@Bean//给容器中添加一个组件，此组件的一些属性的值可以从properties中获取

@ConditionalOnMissingBean///若容器中无组件CharacterEncodingFilter，这个bean才加入容器

public CharacterEncodingFilter characterEncodingFilter() {

CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();

filter.setEncoding(this.properties.getCharset().name());

filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));

filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));

return filter;

}

...

}

上面这个组件中属性的值可以从properties中获取，这个properties是开头@EnableConfigurationProperties(HttpProperties.class)中引入的HttpProperties的HttpProperties.Encoding属性注入的，而HttpProperties中的属性可以和配置文件中的值相绑定。这样配置文件中的值就注入到了这些组件的属性中。

Spring Boot启动时会加载很多自动配置类，这些自动配置类向容器中加入了很多功能组件。当我们需要使用这些组件时，可以查看相应的自动配置类，看看这些组件有哪些功能。当你要指定组件中属性的值时，可以查看此配置类绑定的XxxProperties类，然后再配置文件中按照 XxxProperties类的属性来进行配置。

## HelloWorld程序解析

### 依赖引入

#### 父项目

#### 启动器

### 程序入口

# 配置文件

## YAML简介

### 基本语法

### 数据类型

## 配置文件加载顺序

## Spring Boot基本配置介绍

## 自动配置原理

# 日志

## 日志简述

### 日志的抽象层

### 日志的实现层

## SpringBoot使用的日志-SLF4j

### SpringBoot日志依赖关系

### 使用SLF4j

### 更改日志配置

## 更改日志框架

# SpringBoot Web开发

## Spring核心技术

### 控制反转IOC

#### Bean属性

#### 基于注解的依赖注入

### 面向切面AOP

#### 注解式AOP开发

## SpringMVC

### SpringMVC架构

### 转发与重定向

### 拦截器

### 类型转换器

### 异常处理

### SpringMVC的自动配置

## Thymeleaf模板引擎

### Thymeleaf简介

### 引入Thymeleaf

### Thymeleaf语法规则

### Thymeleaf应用实例

## SpringBoot对JSP的支持

# SpringBoot数据访问

## Docker

### 简介

### Docker安装

#### Ubuntu安装

#### Windos安装

### Docker架构

### Docker使用

### Docker常用命令

### 示例Docker安装MySQL

## MyBatis

### 简述

### ORM框架原理

### 引入mybatis

### MyBatis注解方式开发

#### @Select注解

#### @Insert注解

#### @Update注解

#### @Delete注解

#### 动态查询

### SpringBoot使用MyBatis

## MongDB

### 简介

### 配置MongDB

### 基本用法

#### 连接

#### 创建与删除数据库

#### 创建与删除集合

#### 增删改查

#### 排序

### SpringBoot使用MongDB

## Redis

### Redis简介

### Redis安装与配置

### Redis数据类型

### Redis命令

### SpringBoot使用Redis

# SpringBoot Actuator

## SpringBoot Actuator 简介

## 原生端点使用

## 定制 Actuator

# Spring Security

## Spring Security简介

## 配置Spring Security

## Spring Security入门

### 框架原理

### 核心组件

## 实例

# SpringBoot部署与测试

## 热部署

## 以jar文件部署

## 以war文件部署

## @Profile注解

## 单元测试

### JUnit简介

### Dao层单元测试

### Service层单元测试

# 项目实战