

SpringBoot入门

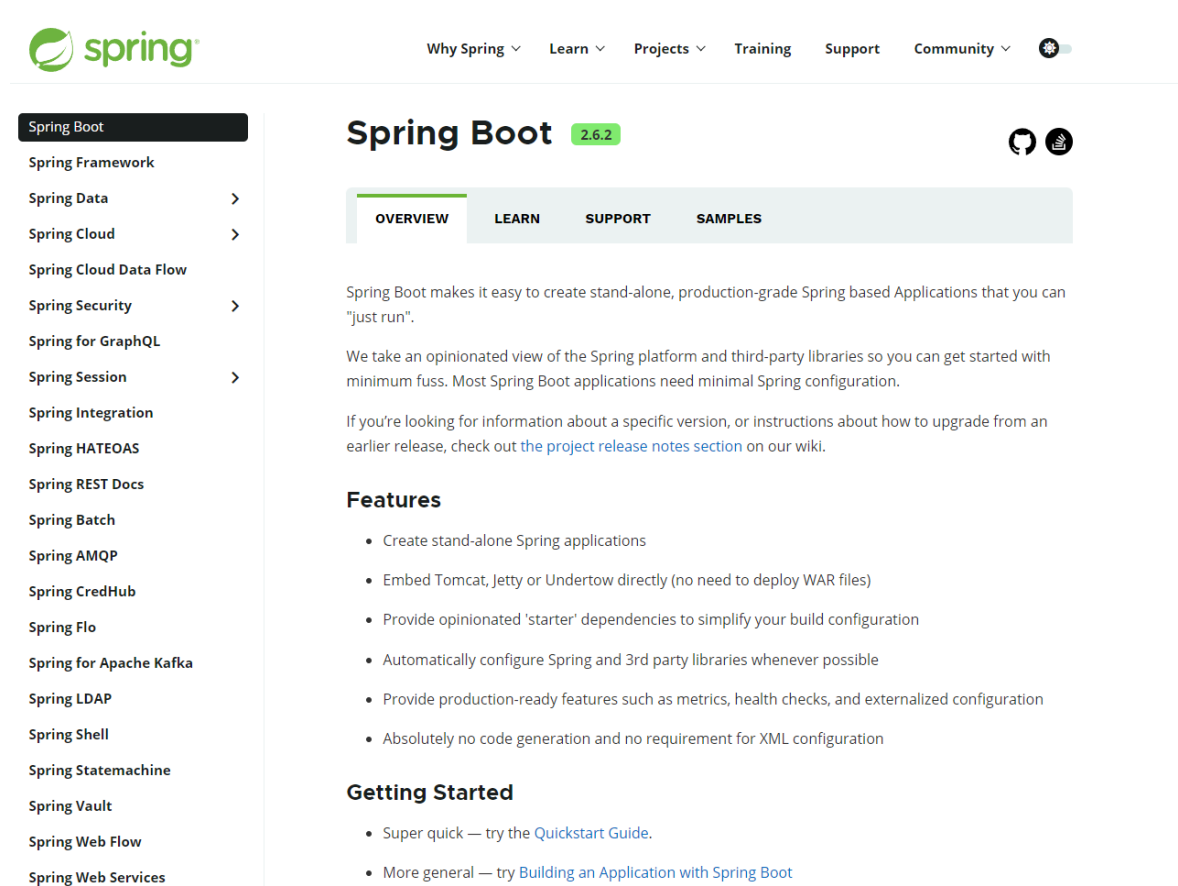
第一章 概述

1、今日内容

- SpringBoot概述、快速入门
- SpringBoot配置
- SpringBoot整合-重点

2、SpringBoot概述-面试

(1) 官网: <https://spring.io/>



(2) spring缺点:

a、配置繁琐

搭建ssm项目, 需要配置大量xml。 application.xml spring-mybatis.xml spring-mvc.xml,大量的bean。

b、依赖繁琐

pom.xml要写大量依赖。 pom.xml spring-core spring-bean spring-mvc spring-mybatis java-connector

版本冲突 spring-core 4.0 spring-mvc 5.0

3) SpringBoot概念

SpringBoot提供了一种快速使用Spring的方式，基于**约定优于配置**的思想，可以让开发人员不必在配置与逻辑业务之间进行思维的切换，全身心的投入到逻辑业务的代码编写中，从而大大提高了开发的效率。

(4) SpringBoot功能

a、自动配置

Spring Boot的自动配置是一个运行时（更准确地说，是应用程序启动时）的过程，考虑了众多因素，才决定Spring配置应该用哪个，不该用哪个。该过程是SpringBoot自动完成的。

b、起步依赖

起步依赖本质上是一个Maven项目对象模型（Project Object Model，POM），定义了对其他库的**传递依赖**，这些东西加在一起即支持某项功能。**依赖太多 版本冲突**。

简单的说，起步依赖就是将具备某种功能的坐标打包到一起，并提供一些默认的功能。

c、辅助功能

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（tomcat）、安全、指标、健康检测、外部配置等。

注意：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。

3、SpringBoot快速入门

官网：<https://docs.spring.io/spring-boot/docs/current/reference/html>

(1) **需求**：搭建SpringBoot工程，定义HelloController.hello()方法，返回”Hello SpringBoot!”。

(2) **实现步骤**：

①创建Maven项目 springboot-helloworld

②导入SpringBoot起步依赖

```
1  <!--springboot工程需要继承的父工程-->
2      <parent>
3          <groupId>org.springframework.boot</groupId>
4          <artifactId>spring-boot-starter-parent</artifactId>
5          <version>2.1.8.RELEASE</version>
6      </parent>
7
8      <dependencies>
9          <!--web开发的起步依赖-->
10         <dependency>
11             <groupId>org.springframework.boot</groupId>
12             <artifactId>spring-boot-starter-web</artifactId>
13         </dependency>
14     </dependencies>
```

③定义Controller编写引导类

```
1  @SpringBootApplication//表示这个类 是springboot主启动类。
2  public class HelloApplication {
3      public static void main(String[] args) {
4          SpringApplication.run(HelloApplication.class,args);
5      }
6  }
```

④定义Controller

```
1  @RestController
2  @RequestMapping("/hello")
3  public class HelloController {
4
5      @GetMapping("/test")
6      public String test1(){
7          return "Hello SpringBoot!";
8      }
9  }
```

⑤启动测试 访问: <http://localhost:8080/hello/test>

(3) 总结

1启动springboot一个web工程

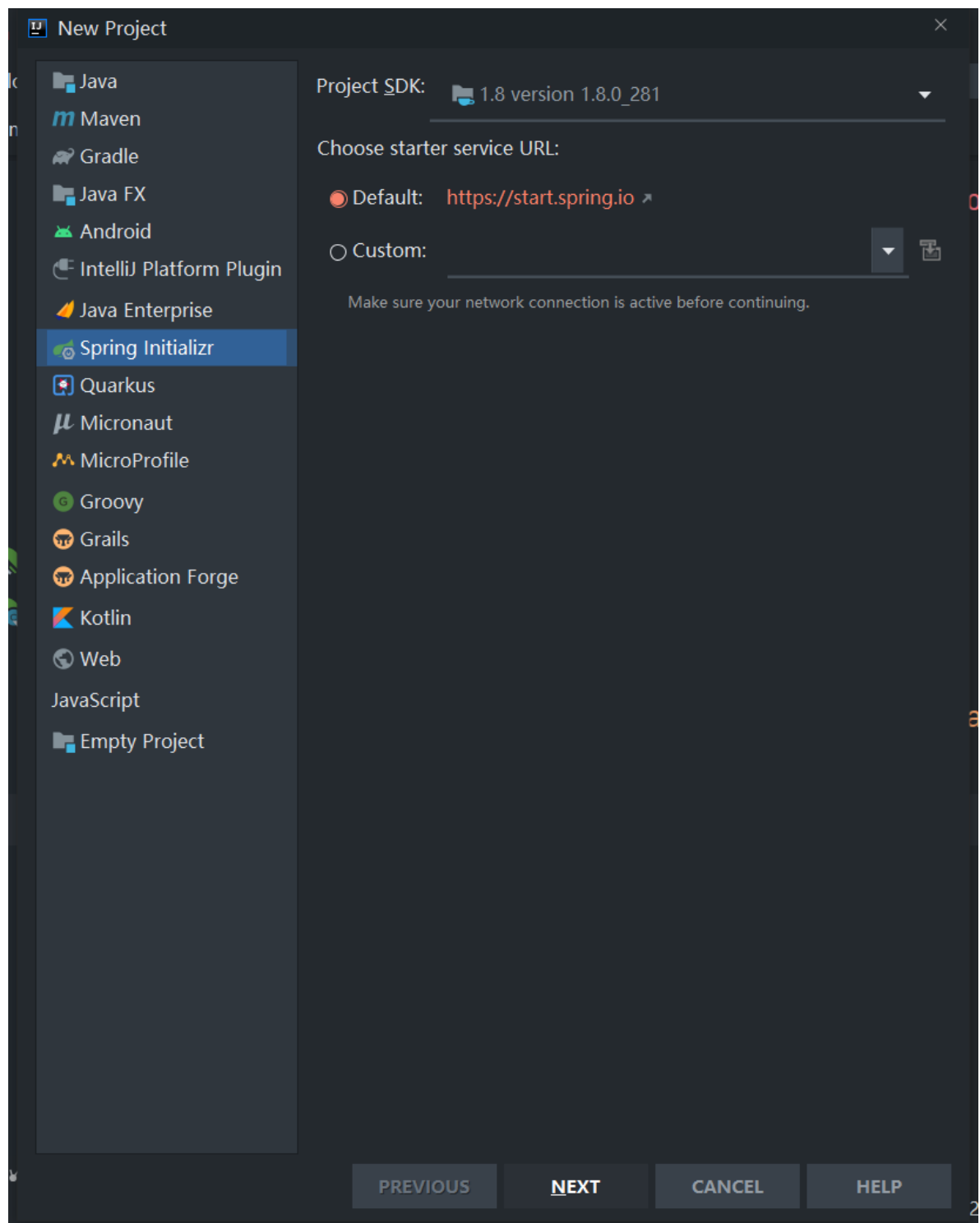
1pom 规定父工程, 导入web的起步依赖

2主启动类 @SpringBootApplication、main

3业务逻辑 controller, service,dao

- SpringBoot在创建项目时, 使用jar的打包方式。 java -jar xxx.jar
- SpringBoot的引导类, 是项目入口, 运行main方法就可以启动项目。
- 使用SpringBoot和Spring构建的项目, 业务代码编写方式完全一样。

4、快速构建SpringBoot工程



New Project

Spring Initializr Project Settings

Group:

com.ydl

Artifact:

springboot-init

Type:

☒ Maven

☐ Gradle

Language:

☒ Java

☐ Kotlin

☐ Groovy

Packaging:

☒ Jar

☐ War

Java version:

8

Version:

0.0.1-SNAPSHOT

Name:

springboot-init

Description:

Demo project for Spring Boot

Package:

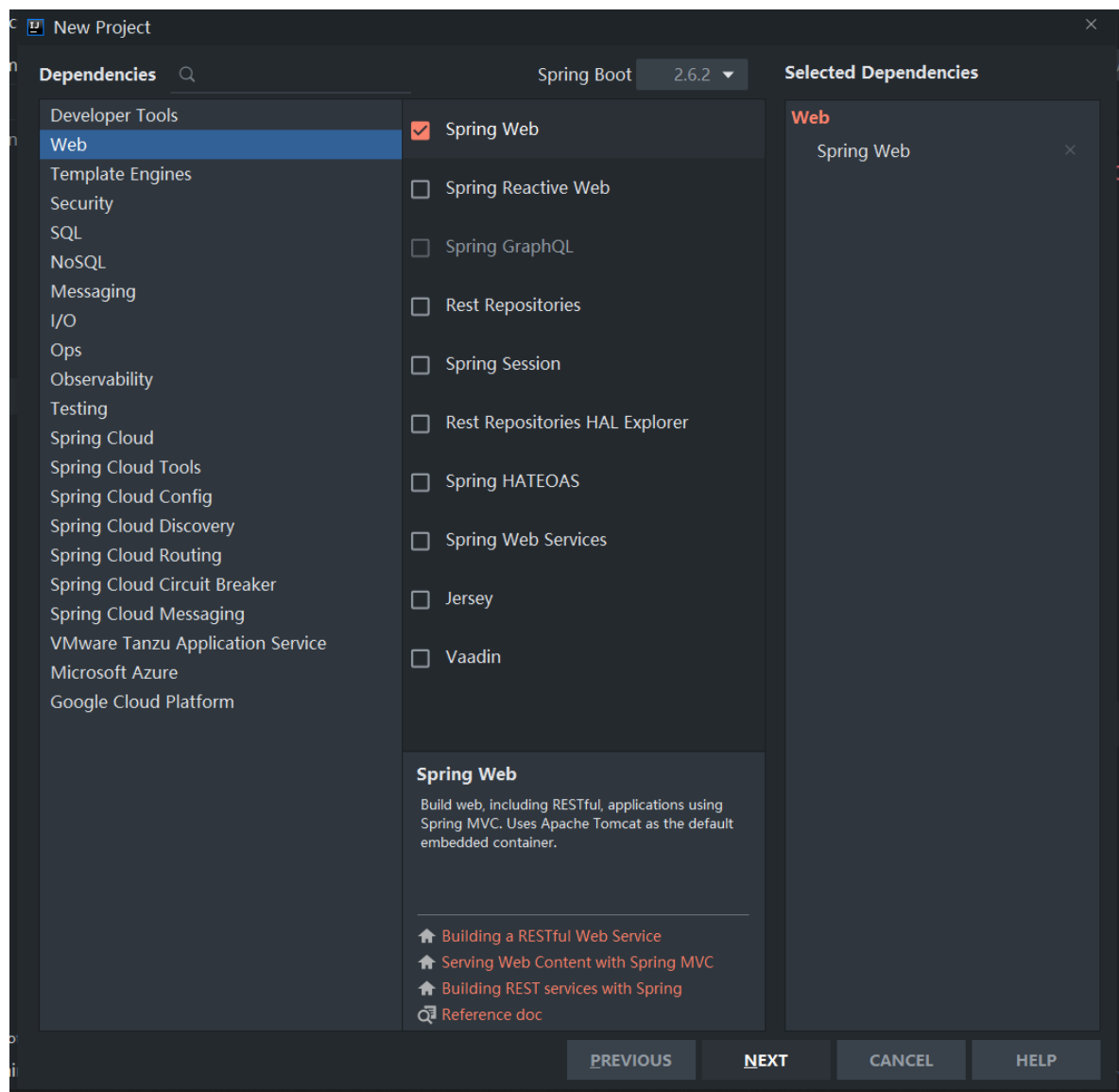
com.ydl.springbootinit

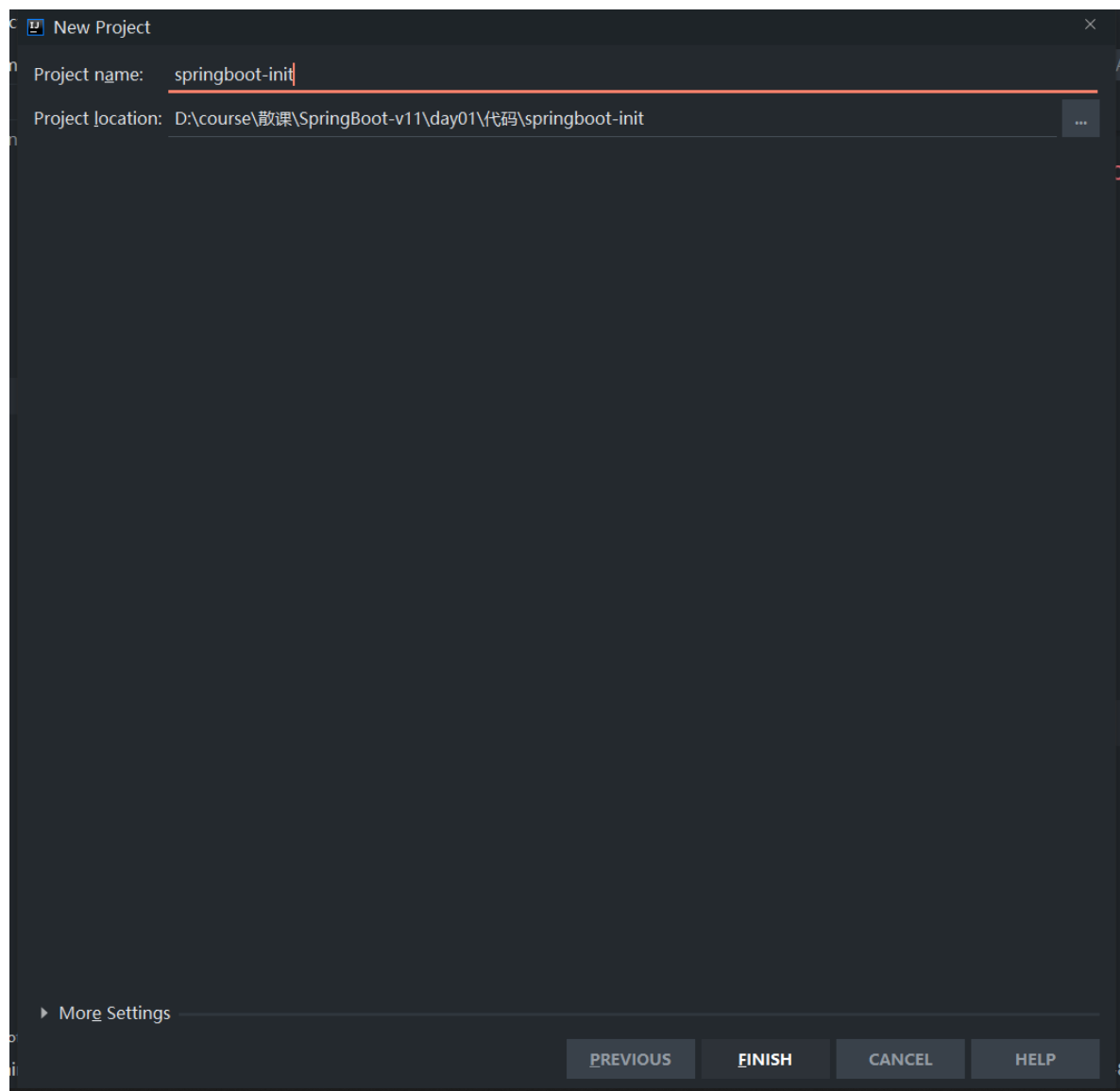
PREVIOUS

NEXT

CANCEL

HELP





编写controller

```
1  @RestController
2  public class HelloController {
3      @RequestMapping("/hello")
4      public String hello(){
5          return " hello Spring Boot !";
6      }
7  }
```

启动测试 访问: <http://localhost:8080/hello/test>

5、SpringBoot起步依赖原理分析-理解

- 在spring-boot-starter-parent中定义了各种技术的版本信息，组合了一套最优搭配的技术版本。
- 在各种starter中，定义了完成该功能需要的坐标合集，其中大部分版本信息来自于父工程。
- 我们的工程继承parent，引入starter后，通过**依赖传递**，就可以简单方便获得需要的jar包，并且不会存在版本冲突等问题。

第二章 配置文件

1、SpringBoot配置-配置文件分类

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

1. 默认配置文件名称：application
2. 在同一级目录下优先级为：properties>yml > yaml

例如：配置内置Tomcat的端口

properties：

```
1  server.port=8080
```

yml:

```
1  server:
2    port: 8080
```

init工程：

修改application.properties

```
1  server.port=8081
```

新建application.yml

```
1  server:
2    port: 8082
```

新建application.yml

```
1  server:
2    port: 8083
```

)2、SpringBoot配置-yaml基本语法

(1) 概念：

YAML是一种直观的能够被电脑识别的数据序列化格式，并且容易被人类阅读，容易和脚本语言交互的，可以被支持YAML库的不同的编程语言程序导入。

(2) 语法特点：

- 大小写敏感
- 数据值前边必须有空格，作为分隔符
- 使用缩进表示层级关系
- 缩进时不允许使用Tab键，只允许使用空格（各个系统 Tab对应的空格数目可能不同，导致层次混乱）。
- 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可
- "#" 表示注释，从这个字符一直到行尾，都会被解析器忽略。

```
1  server:
2    port: 8080
3    address: 127.0.0.1
4  name: abc
```

3、SpringBoot配置-yaml数据格式

对象(map)：键值对的集合。


```

1  person:
2      name: itlils
3  # 行内写法
4  person: {name: itlils}

```

数组：一组按次序排列的值

```

1  address:
2      - beijing
3      - shanghai
4  # 行内写法
5  address: [beijing, shanghai]

```

纯量：单个的、不可再分的值

```

1  msg1: 'hello \n world' # 单引忽略转义字符
2  msg2: "hello \n world" # 双引识别转义字符

```

参数引用

```

1  name: itlils
2  person:
3      name: ${itlils} # 引用上边定义的名称值

```

4、SpringBoot配置-获取数据

1@Value

```

1      #获取普通配置
2      @Value("${name}")
3      private String name;
4      #获取对象属性
5      @Value("${person.name}")
6      private String name2;
7      #获取数组
8      @Value("${address[0]}")
9      private String address1;
10     #获取纯量
11     @Value("${msg1}")
12     private String msg1;

```

2Environment

```

1  @Autowired
2  private Environment env;
3
4  System.out.println(env.getProperty("person.name"));
5
6  System.out.println(env.getProperty("address[0]"));

```

3 @ConfigurationProperties

注意：prefix一定要写

```

1  @Component
2  @ConfigurationProperties(prefix = "person")
3  public class Person {
4

```

```

5     private String name;
6     private int age;
7     private String[] address;
8
9     public String getName() {
10        return name;
11    }
12
13    public void setName(String name) {
14        this.name = name;
15    }
16
17    public int getAge() {
18        return age;
19    }
20
21    public void setAge(int age) {
22        this.age = age;
23    }
24
25    public String[] getAddress() {
26        return address;
27    }
28
29    public void setAddress(String[] address) {
30        this.address = address;
31    }
32
33    @Override
34    public String toString() {
35        return "Person{" +
36            "name='" + name + '\'' +
37            ", age=" + age +
38            '}';
39    }
40 }

```

修改controller

```

1     @Autowired
2     private Person person;
3
4     System.out.println(person);
5     String[] address = person.getAddress();
6     for (String s : address) {
7         System.out.println(s);
8     }

```

去掉报警提示:

```

1     <dependency>
2         <groupId>org.springframework.boot</groupId>
3         <artifactId>spring-boot-configuration-processor</artifactId>
4         <optional>true</optional>
5     </dependency>

```

5、SpringBoot配置-profile-运维

1. 背景：profile是用来完成不同环境下，配置动态切换功能的。

2. profile配置方式

多profile文件方式：提供多个配置文件，每个代表一种环境。主配置文件application.properties配置：

```
1 spring.profiles.active=dev
```

application-dev.properties/yml 开发环境

application-test.properties/yml 测试环境

application-pro.properties/yml 生产环境

yml多文档方式：

在yml中使用 --- 分隔不同配置

```
1  ---
2  server:
3      port: 8081
4  spring:
5      profiles: dev
6  ---
7  server:
8      port: 8082
9  spring:
10     profiles: pro
11  ---
12  server:
13     port: 8083
14  spring:
15     profiles: test
16  ---
17  spring:
18     profiles:
19         active: dev
```

1. profile激活方式

- 配置文件：再配置文件中配置：spring.profiles.active=dev
- 虚拟机参数：在VM options 指定：-Dspring.profiles.active=pro
- 命令行参数：--spring.profiles.active=dev

相当于上线时，运行jar包：java -jar xxx.jar --spring.profiles.active=dev

测试：使用maven 打包此项目，在target包中出现springboot-profiles-0.0.1.jar

cmd 输入

```
1 java -jar springboot-profiles-0.0.1.jar --spring.profiles.active=test
```

6、SpringBoot配置-项目内部配置文件加载顺序

加载顺序为下文的排列顺序，高优先级配置的属性会生效

- file:./config/：当前项目下的/config目录下
- file:./：当前项目的根目录
- classpath:/config/：classpath的/config目录
- classpath/：classpath的根目录

测试:

新建springboot-config目录, 分别在以上目录创建配置文件。

注意: 1项目根目录为springboottest。

2高级配置文件只覆盖低级配置文件的重复项。低级配置文件的独有项任然有效。最低级配置文件中增加:

```
1 server.servlet.context-path = /test
```

访问: <http://localhost:8084/test/hello>

7、SpringBoot配置-项目外部配置加载顺序

外部配置文件的使用是为了不修改配置文件做的

1.命令行

```
1 java -jar app.jar --name="Spring" --server.port=9000
```

2.指定配置文件位置

```
1 java -jar myproject.jar --spring.config.location=d://application.properties
```

<https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.external-config>

作用: 生产环境, 随时改变环境变量时, 可以通过改变配置文件来做。不需重新打包项目。

第三章 整合框架-重要

1、SpringBoot整合Junit

1. 搭建SpringBoot工程 springboot-test。不用任何起步依赖。
2. 引入starter-test起步依赖

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter</artifactId>
5     </dependency>
6
7     <dependency>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-test</artifactId>
10        <scope>test</scope>
11    </dependency>
12 </dependencies>
```

1. 编写service com.ydl.springboottest

```
1
```

```
@Service public class UserService { public void add() { System.out.println("add....."); } }
```

```
1 4. 编写测试类 com.ydl.springboottest
2
```

```

3  ``java
4  @SpringBootTest
5  @RunWith(SpringRunner.class)
6  public class UserServiceTest {
7      @Autowired
8      UserService userService;
9
10     @Test
11     public void testAdd() {
12         userService.add();
13     }
14 }

```

1. 测试

2、SpringBoot整合mybatis-最重点

①搭建SpringBoot工程 springboot-mybatis

②引入mybatis起步依赖，添加mysql驱动

```

1      <dependencies>
2          <dependency>
3              <groupId>org.mybatis.spring.boot</groupId>
4              <artifactId>mybatis-spring-boot-starter</artifactId>
5              <version>2.1.0</version>
6          </dependency>
7
8          <dependency>
9              <groupId>mysql</groupId>
10             <artifactId>mysql-connector-java</artifactId>
11             <!--<scope>runtime</scope>-->
12         </dependency>
13         <dependency>
14             <groupId>org.springframework.boot</groupId>
15             <artifactId>spring-boot-starter-test</artifactId>
16             <scope>test</scope>
17         </dependency>
18     </dependencies>

```

③定义表和实体类 com.ydl.springbootmybatis.domain

```

1  public class User {
2      private int id;
3      private String username;
4      private String password;
5
6
7      public int getId() {
8          return id;
9      }
10
11     public void setId(int id) {
12         this.id = id;
13     }
14
15     public String getUsername() {
16         return username;

```

```

17     }
18
19     public void setUsername(String username) {
20         this.username = username;
21     }
22
23     public String getPassword() {
24         return password;
25     }
26
27     public void setPassword(String password) {
28         this.password = password;
29     }
30
31     @Override
32     public String toString() {
33         return "User{" +
34             "id=" + id +
35             ", username='" + username + '\'' +
36             ", password='" + password + '\'' +
37             '}';
38     }
39 }

```

④编写DataSource和MyBatis相关配置

application.yml

```

1  # datasource
2  spring:
3      datasource:
4          url: jdbc:mysql://127.0.0.1:3306/springboot?serverTimezone=UTC
5          username: root
6          password: root
7          driver-class-name: com.mysql.jdbc.Driver

```

⑤纯注解开发 新建接口com.ydl.springbootmybatis.mapper

```

1  @Mapper
2  public interface UserMapper {
3
4      @Select("select * from t_user")
5      public List<User> findAll();
6  }

```

测试

```

1  @SpringBootTest
2  class SpringbootMybatisApplicationTests {
3      @Autowired
4      private UserMapper userMapper;
5
6      @Test
7      void testFindAll() {
8          List<User> all = userMapper.findAll();
9          System.out.println(all);
10     }
11 }

```

⑥xml开发 新建接口 com.ydl.springbootmybatis.mapper

```
1  @Mapper
2  public interface UserXmlMapper {
3      public List<User> findAll();
4  }
```

resources下建立xml文件 UserMapper.xml

```
1  <?xml version="1.0" encoding="UTF8" ?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3  <mapper namespace="com.ydl.mapper.UserXmlMapper">
4      <select id="findAll" resultType="com.ydl.domain.User">
5          select * from t_user
6      </select>
7  </mapper>
```

修改application.yml 新增如下配置

```
1  mybatis:
2      mapper-locations: classpath:mapper/*
3      type-aliases-package: com.ydl.springbootmybatis.domain
```

测试

```
1  @Autowired
2  private UserXmlMapper userXmlMapper;
3
4      @Test
5      void testFindAllByXml() {
6          List<User> all = userXmlMapper.findAll();
7          System.out.println(all);
8      }
```

3、SpringBoot整合redis

①搭建SpringBoot工程 springboot-redis

②引入redis起步依赖

```
1  <dependencies>
2      <dependency>
3          <groupId>org.springframework.boot</groupId>
4          <artifactId>spring-boot-starter-data-redis</artifactId>
5      </dependency>
6
7      <dependency>
8          <groupId>org.springframework.boot</groupId>
9          <artifactId>spring-boot-starter-test</artifactId>
10         <scope>test</scope>
11     </dependency>
12 </dependencies>
```

③编写测试类

```
1  @SpringBootTest
2  class SpringbootRedisApplicationTests {
```

```

3      @Autowired
4      private RedisTemplate redisTemplate;
5
6      @Test
7      void testSet() {
8          redisTemplate.boundValueOps("name").set("zhangsan");
9      }
10
11     @Test
12     void testGet() {
13         Object name = redisTemplate.boundValueOps("name").get();
14         System.out.println(name);
15     }
16 }

```

④启动redis

⑤测试

⑥application.yml 配置redis相关属性

```

1  spring:
2      redis:
3          host: 127.0.0.1 # redis的主机ip
4          port: 6379

```

作业：ssm项目改造成boot

总结：上手

1boot不是spring的增强，快速使用 Spring 的方式。

1.1pom parent starter

1.2主启动类 SpringBootApplication

1.3application.yml

快速构建

2配置文件

properties yml yaml

语法

自定义属性值，获取 @value Environment 对象

3重要--其他技术整合

junit: starter

mybatis: starter 注解 xml

redis:starter redisTemplate 两套api

SpringBoot高级

今日内容

- SpringBoot自动配置原理
- SpringBoot自定义starter

- SpringBoot事件监听
- SpringBoot流程分析
- SpringBoot监控
- SpringBoot部署

第一章 condition

1、SpringBoot自动配置-Condition-1

Condition是Spring4.0后引入的条件化配置接口，通过实现Condition接口可以完成有条件的加载相应的Bean

@Conditional要配和Condition的实现类（ClassCondition）进行使用

- 创建模块 springboot-condition
- 一、观察spring自动创建bean过程

改造启动类

```

1  @SpringBootApplication
2  public class SpringbootConditionApplication {
3
4      public static void main(String[] args) {
5          //返回spring容器
6          ConfigurableApplicationContext context=
SpringApplication.run(SpringbootConditionApplication.class, args);
7
8          // 获取redisTemplate 这个bean对象
9          Object redisTemplate = context.getBean("redisTemplate");
10         System.out.println(redisTemplate);
11     }
12 }
```

启动：获取不到对象

导入 redis依赖，再启动则可以获取到bean对象。

```

1      <dependency>
2          <groupId>org.springframework.boot</groupId>
3          <artifactId>spring-boot-starter-data-redis</artifactId>
4      </dependency>
```

- 二、自定义bean对象创建

新建user实体类 com.ydlclass.springbootcondition.domain

```

1  public class User {
2  }
```

新建配置类 com.ydlclass.springbootcondition.config

```

1  @Configuration
2  public class UserConfig {
3
4      @Bean
5      public User user() {
6          return new User();
7      }
8
9  }

```

启动类获取。测试可以获取到

```

1  Object user = context.getBean("user");
2  System.out.println(user);

```

- 三、自定义bean 根据条件创建

创建condition类 com.ydlclass.springbootcondition.condition

```

1  public class ClassCondition implements Condition {
2      @Override
3      public boolean matches(ConditionContext conditionContext,
4          AnnotatedTypeMetadata annotatedTypeMetadata) {
5          return false;
6      }
7  }

```

改造UserConfig

```

1      @Bean
2      @Conditional(ClassCondition.class)
3      public User user() {
4          return new User();
5      }

```

测试不能自动创建user这个bean

- 四、改造ClassCondition。根据是否导入redis来决定是否创建userBean

```

1  package com.ydlclass.springbootcondition.condition;
2
3
4  import org.springframework.context.annotation.Condition;
5  import org.springframework.context.annotation.ConditionContext;
6  import org.springframework.core.type.AnnotatedTypeMetadata;
7
8  /**
9   * @Created by IT李老师
10  * 个人微 itlils
11  */
12  public class ClassCondition implements Condition {
13
14      //通过boolean返回值，就能确定是否生成bean对象
15      @Override
16      public boolean matches(ConditionContext context, AnnotatedTypeMetadata
17          metadata) {
18          //业务逻辑，返回true或false来决定某个bean对象是否生成
19
20          //需求1: 必须引入jedis，你的项目才生成user对象。

```

```

20         try {
21             Class.forName("redis.clients.jedis.Jedis");
22             return true;
23         } catch (ClassNotFoundException e) {
24             return false;
25         }
26     }
27 }
28 }

```

测试。获取不到userBean

```

1  package com.ydlclass.springbootcondition;
2
3  import com.ydlclass.springbootcondition.domain.User;
4  import io.lettuce.core.output.StatusOutput;
5  import org.springframework.boot.SpringApplication;
6  import org.springframework.boot.autoconfigure.SpringBootApplication;
7  import org.springframework.context.ConfigurableApplicationContext;
8
9  @SpringBootApplication
10 public class SpringbootConditionApplication {
11
12     public static void main(String[] args) {
13         ConfigurableApplicationContext context =
14             SpringApplication.run(SpringbootConditionApplication.class, args);
15
16         //只要引入redis起步依赖，有redisTemplate对象。没引入，容器中没这个对象。
17         //Object redisTemplate = context.getBean("redisTemplate");
18         //System.out.println(redisTemplate);
19
20         //通过名字拿bean对象
21         //User user = (User) context.getBean("user");
22         //System.out.println(user);
23
24         //通过类型拿bean对象
25         User user = context.getBean(User.class);
26         System.out.println(user);
27     }
28 }

```

导入依赖,再测试，可以获取到userBean

```

1  <dependency>
2      <groupId>redis.clients</groupId>
3      <artifactId>jedis</artifactId>
4  </dependency>

```

2、SpringBoot自动配置-Condition-2

需求：将类的判断定义为动态的。判断哪个字节码文件存在可以动态指定。

自定义条件注解类

```

1  import org.springframework.context.annotation.Conditional;
2
3  import java.lang.annotation.*;
4
5
6  @Target({ElementType.TYPE, ElementType.METHOD})
7  @Retention(RetentionPolicy.RUNTIME)
8  @Documented
9  @Conditional(ClassCondition.class)
10 public @interface ConditionOnClass {
11     String[] value();
12 }

```

ClassCondition

```

1  package com.ydlclass.springbootcondition.condition;
2
3
4  import org.springframework.context.annotation.Condition;
5  import org.springframework.context.annotation.ConditionContext;
6  import org.springframework.core.type.AnnotatedTypeMetadata;
7
8  import java.util.Map;
9
10 /**
11  * @Created by IT李老师
12  * 公主号 “IT李哥交朋友”
13  * 个人微 itlils
14  */
15 public class ClassCondition implements Condition {
16
17     //通过boolean返回值，就能确定是否生成bean对象
18     @Override
19     public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
20         //业务逻辑，返回true或false来决定某个bean对象是否生成
21
22         //需求1: 必须引入jedis, 你的项目才生成user对象。
23         //try {
24         //    Class.forName("redis.clients.jedis.Jedis");
25         //    return true;
26         //} catch (ClassNotFoundException e) {
27         //    return false;
28         //}
29
30
31         try {
32             //需求2: 必须引入 动态传来的包名 , 你的项目才生成user对象。
33             Map<String, Object> annotationAttributes =
metadata.getAnnotationAttributes("com.ydlclass.springbootcondition.condition.Con
ditionalOnClass");
34             System.out.println(annotationAttributes);
35             String[] values = (String[]) annotationAttributes.get("value");
36             for (String value : values) {
37                 Class.forName(value);
38             }
39             return true;
40         } catch (Exception e) {

```

```

41         return false;
42     }
43
44
45     }
46 }

```

注意：此处@ConditionOnClass为自定义注解

```

1  package com.ydlclass.springbootcondition.config;
2
3  import com.ydlclass.springbootcondition.condition.ClassCondition;
4  import com.ydlclass.springbootcondition.condition.ConditionalOnClass;
5  import com.ydlclass.springbootcondition.domain.User;
6  import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
7  import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
8  import org.springframework.context.annotation.Bean;
9  import org.springframework.context.annotation.Conditional;
10 import org.springframework.context.annotation.Configuration;
11
12 /**
13  * @Created by IT李老师
14  * 个人微 itlils
15  */
16 //配置类
17 @Configuration
18 public class UserConfig {
19
20     @Bean
21     //@Conditional(ClassCondition.class) //条件满足，new这个对象。条件不满足，不new这个
22     bean这个对象
23     @ConditionalOnClass({"redis.clients.jedis.Jedis"}) //这个注解，不用你写。
24     springboot 已经写好了
25     public User user(){
26         return new User();
27     }
28
29     @Bean
30     @ConditionalOnProperty(name = "ydlclass",havingValue = "itlils")
31     //@ConditionalOnClass(name="redis.clients.jedis.Jedis")
32     public User user2(){
33         return new User();
34     }
35 }

```

测试User对象的创建

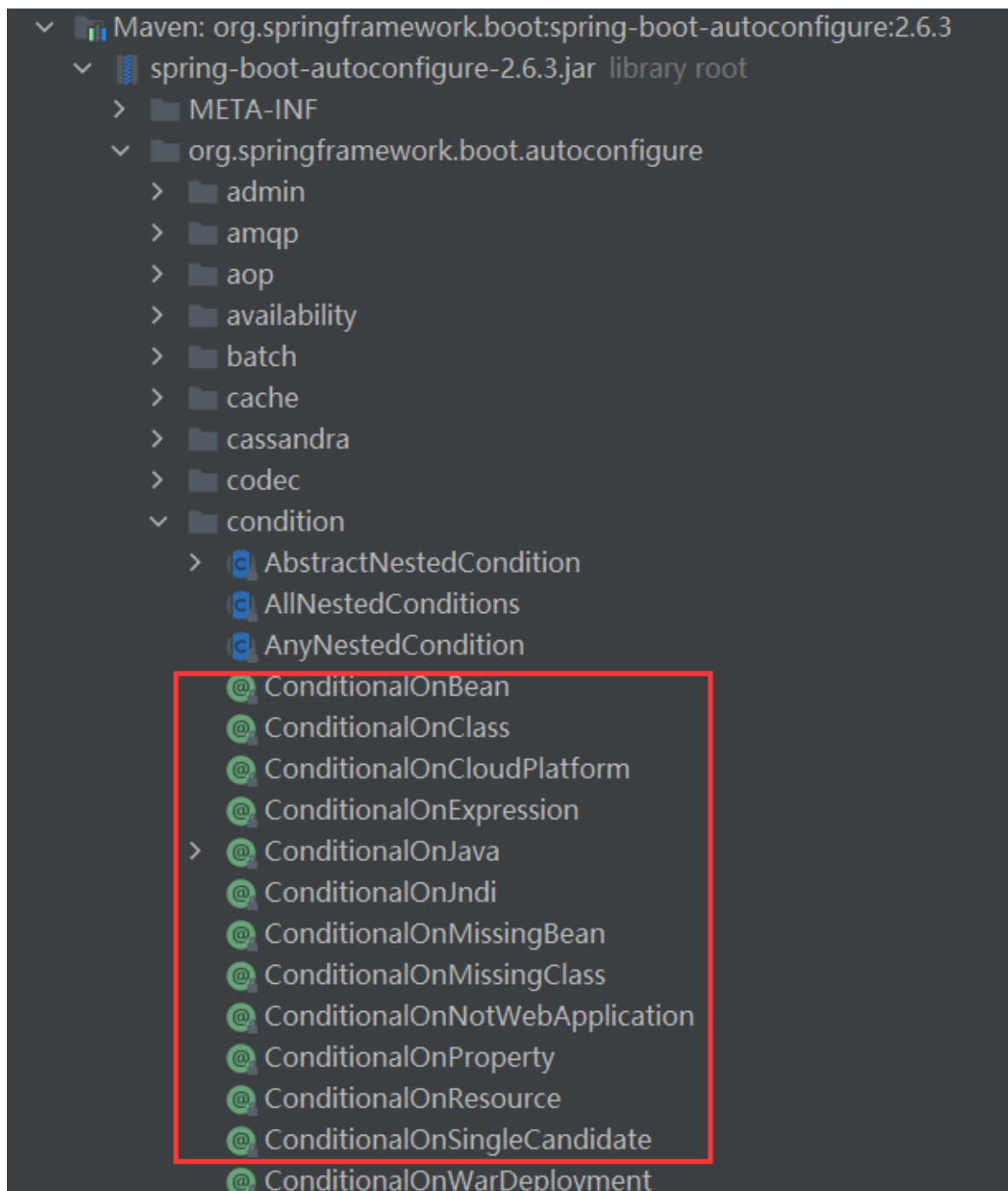
```

1  package com.ydlclass.springbootcondition;
2
3  import com.ydlclass.springbootcondition.domain.User;
4  import io.lettuce.core.output.StatusOutput;
5  import org.springframework.boot.SpringApplication;
6  import org.springframework.boot.autoconfigure.SpringBootApplication;
7  import org.springframework.context.ConfigurableApplicationContext;
8
9  @SpringBootApplication
10 public class SpringbootConditionApplication {

```

```
11
12     public static void main(String[] args) {
13         ConfigurableApplicationContext context =
SpringApplication.run(SpringbootConditionApplication.class, args);
14
15         //只要引入redis起步依赖，有redisTemplate对象。没引入，容器中没这个对象。
16         //Object redisTemplate = context.getBean("redisTemplate");
17         //System.out.println(redisTemplate);
18
19         //通过名字拿bean对象
20         //User user = (User) context.getBean("user");
21         //System.out.println(user);
22
23         //通过类型拿bean对象
24         //User user = context.getBean(User.class);
25         //System.out.println(user);
26
27         Object user2 = context.getBean("user2");
28         System.out.println(user2);
29
30     }
31
32 }
```

[查看条件注解源码](#)



SpringBoot 提供的常用条件注解：

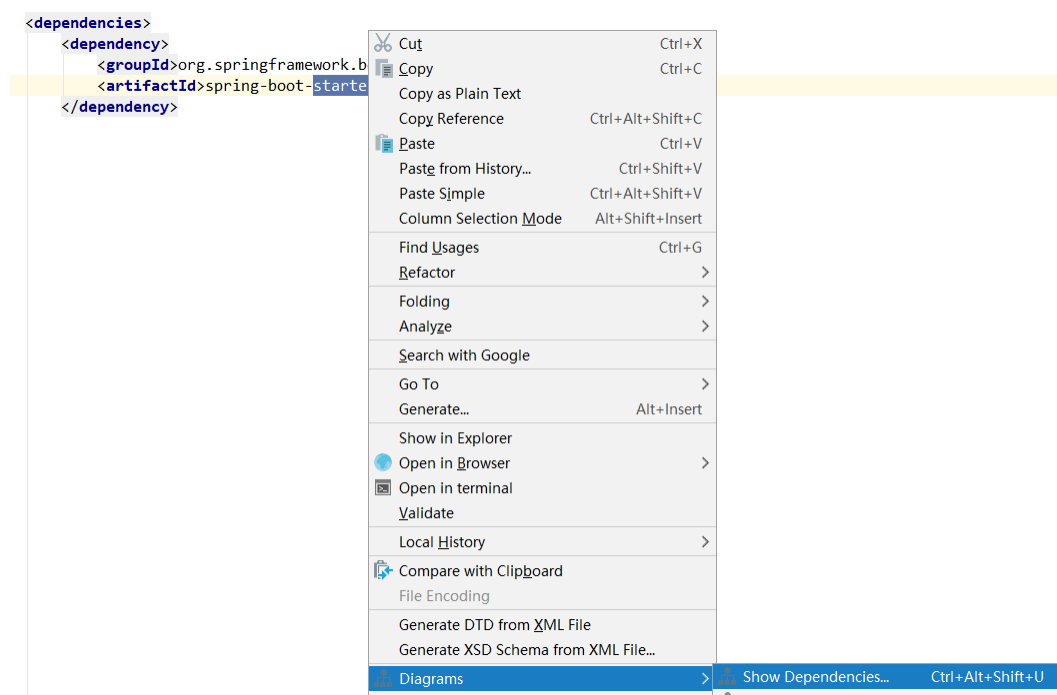
ConditionalOnProperty：判断配置文件中是否有对应属性和值才初始化Bean

ConditionalOnClass：判断环境中是否有对应字节码文件才初始化Bean

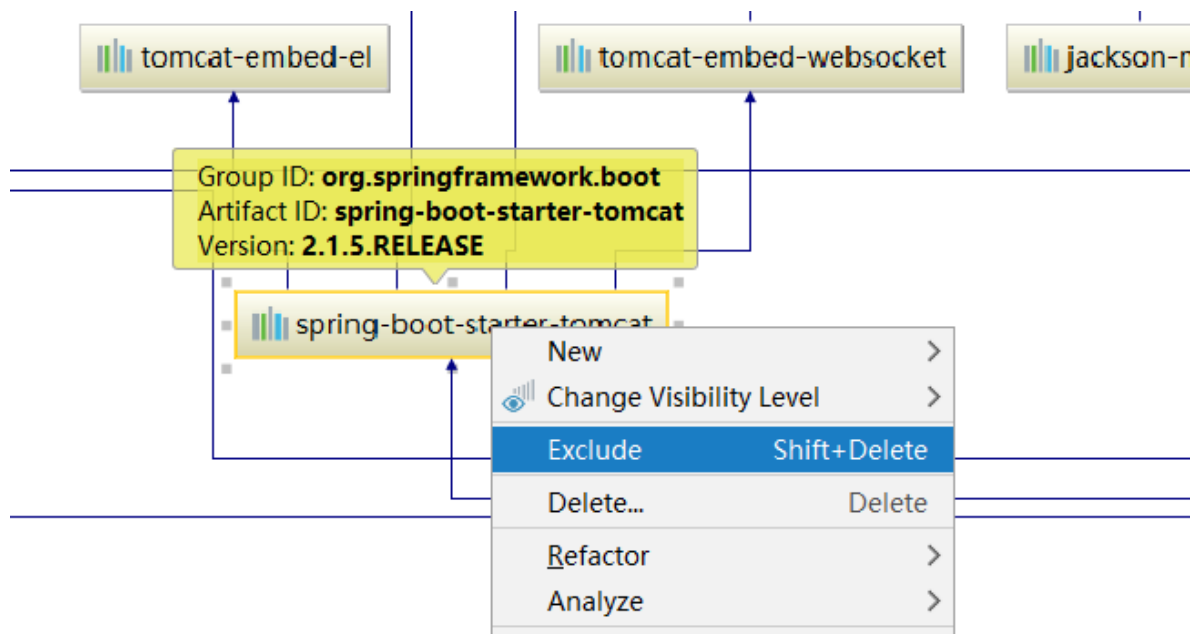
ConditionalOnMissingBean：判断环境中没有对应Bean才初始化Bean

3、SpringBoot自动配置-切换内置web服务器

[查看继承关系图](#)



排除Tomcat



pom文件中的排除依赖效果

```

1      <dependency>
2          <groupId>org.springframework.boot</groupId>
3          <artifactId>spring-boot-starter-web</artifactId>
4          <!--排除tomcat依赖-->
5          <exclusions>
6              <exclusion>
7                  <artifactId>spring-boot-starter-tomcat</artifactId>
8                  <groupId>org.springframework.boot</groupId>
9              </exclusion>
10         </exclusions>
11     </dependency>
12
13     <!--引入jetty的依赖-->
14     <dependency>
15         <artifactId>spring-boot-starter-jetty</artifactId>
16         <groupId>org.springframework.boot</groupId>

```


问你：为什么引入了starter-data-redis，我们就能项目中，直接拿redistemplate？

springboot中的autoconfig工程里把常用的对象的配置类都有了，只要工程中，引入了相关起步依赖，这些对象在我们本项目的容器中就有了。



4、SpringBoot自动配置-Enable注解原理-重点

- SpringBoot不能直接获取在其他工程中定义的Bean

演示代码：

springboot-enable工程

```

1  /**
2   * @ComponentScan 扫描范围：当前引导类所在包及其子包
3   *
4   * com.ydlclass.springbootenable
5   * com.ydlclass.config
6   * //1.使用@ComponentScan扫描com.ydlclass.config包
7   * //2.可以使用@Import注解，加载类。这些类都会被Spring创建，并放入IOC容器
8   * //3.可以对Import注解进行封装。
9   */
10
11  //@ComponentScan("com.qiniu.config")
12  //@Import(UserConfig.class)
13  @EnableUser
14  @SpringBootApplication
15  public class SpringbootEnableApplication {
16
17      public static void main(String[] args) {
18          ConfigurableApplicationContext context =
19              SpringApplication.run(SpringbootEnableApplication.class, args);
20
21          //获取Bean
22          Object user = context.getBean("user");
23          System.out.println(user);
24      }
25  }
26  }
```

pom中引入springboot-enable-other

```

1      <dependency>
2          <groupId>com.ydlclass</groupId>
3          <artifactId>springboot-enable-other</artifactId>
4          <version>0.0.1-SNAPSHOT</version>
5      </dependency>

```

springboot-enable-other工程

UserConfig

```

1  @Configuration
2  public class UserConfig {
3
4      @Bean
5      public User user() {
6          return new User();
7      }
8
9  }

```

确实，本工程中没有这个第三方jar包中的bean对象



EnableUser注解类

```

1  import org.springframework.context.annotation.Import;
2
3  import java.lang.annotation.*;
4
5  @Target(ElementType.TYPE)
6  @Retention(RetentionPolicy.RUNTIME)
7  @Documented
8  @Import(UserConfig.class)
9  public @interface EnableUser {
10 }

```

原因： @ComponentScan 扫描范围：当前引导类所在包及其子包

三种解决方案：

- 1.使用@ComponentScan扫描com.ydlclass.config包
- 2.可以使用@Import注解，加载类。这些类都会被Spring创建，并放入IOC容器

3.可以对Import注解进行封装。

重点：Enable注解底层原理是使用@Import注解实现Bean的动态加载

重要：springbootapplication 由三个注解组成

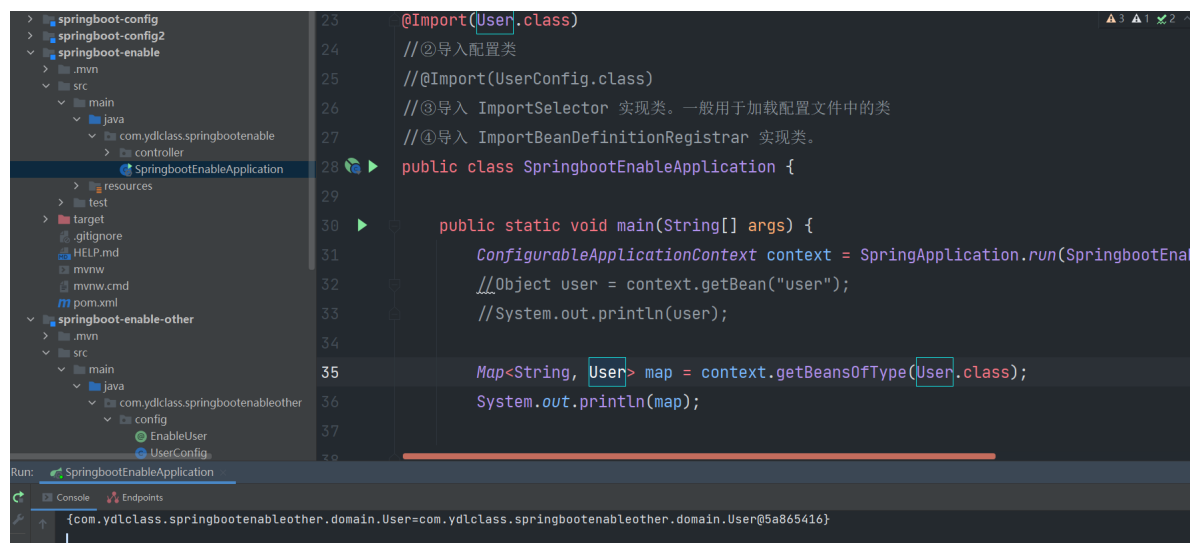
```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
                                   @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

- 1 @SpringBootConfiguration 自动配置相关
- 2 @EnableAutoConfiguration
- 3 @ComponentScan 扫本包及子包

5、SpringBoot自动配置-@Import详解

@Enable底层依赖于@Import注解导入一些类，使用@Import导入的类会被Spring加载到IOC容器中。而@Import提供4中用法：

①导入Bean。注意bean名字是全限定名。



②导入配置类

③导入 ImportSelector 实现类。一般用于加载配置文件中的类

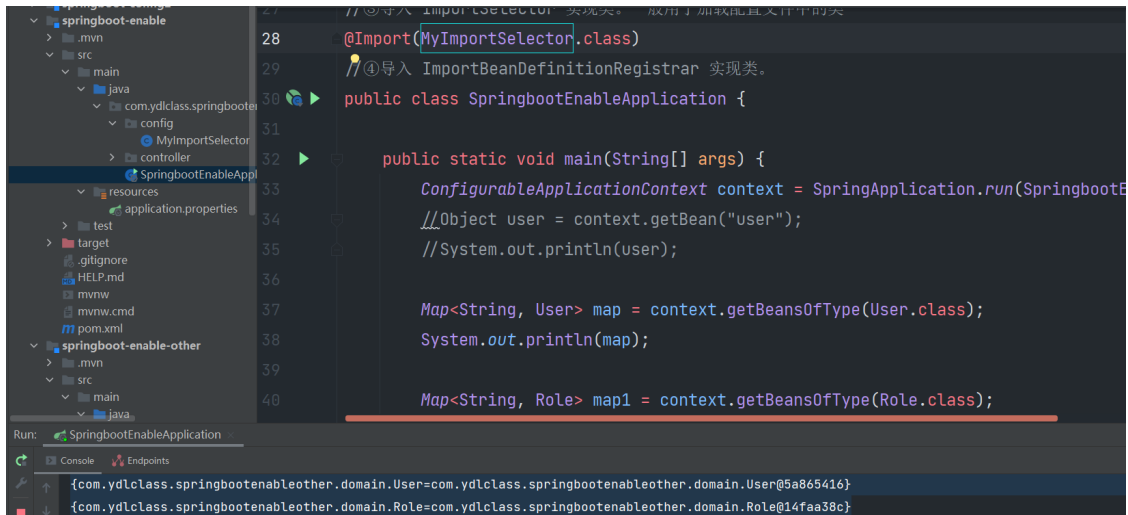
④导入 ImportBeanDefinitionRegistrar 实现类。

- 导入Bean @Import(User.class)
 - 导入配置类 @Import(UserConfig.class)
 - 导入 ImportSelector 实现类 @Import(MyImportSelector.class)
- MyImportSelector

```

1 public class MyImportSelector implements ImportSelector {
2     @Override
3     public String[] selectImports(AnnotationMetadata importingClassMetadata)
4     {
5         return new String[]{"com.ydlclass.domain.User",
6             "com.ydlclass.domain.Role"};
7     }
8 }

```

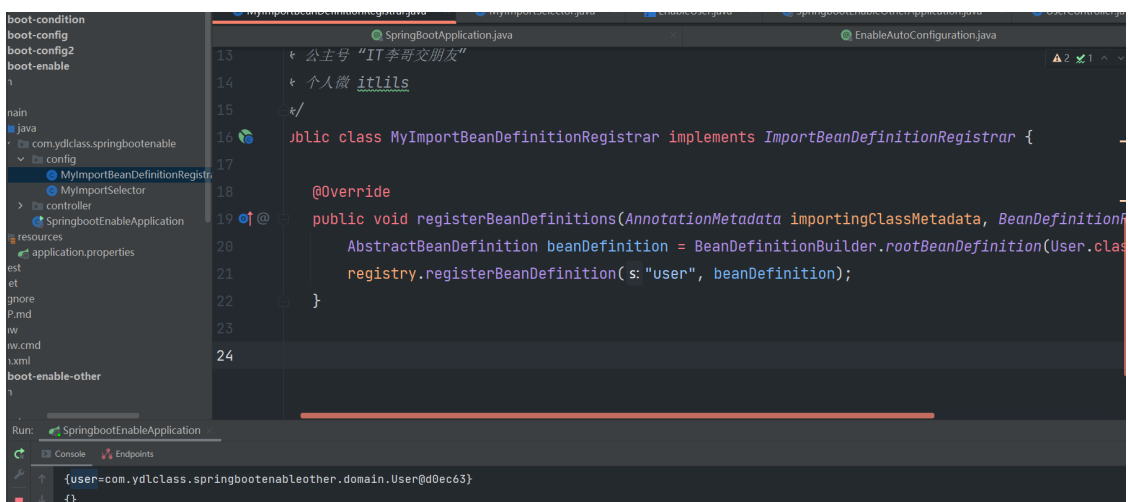


- 导入 ImportBeanDefinitionRegistrar 实现类。@Import({MyImportBeanDefinitionRegistrar.class})

```

1 public class MyImportBeanDefinitionRegistrar implements
2     ImportBeanDefinitionRegistrar {
3     @Override
4     public void registerBeanDefinitions(AnnotationMetadata
5         importingClassMetadata, BeanDefinitionRegistry registry) {
6         AbstractBeanDefinition beanDefinition =
7             BeanDefinitionBuilder.rootBeanDefinition(User.class).getBeanDefinition();
8         registry.registerBeanDefinition("user", beanDefinition);
9     }
10 }

```



SpringbootEnableApplication测试代码

- 1 Import4中用法:
- 2
- 3 * 1. 导入Bean

```

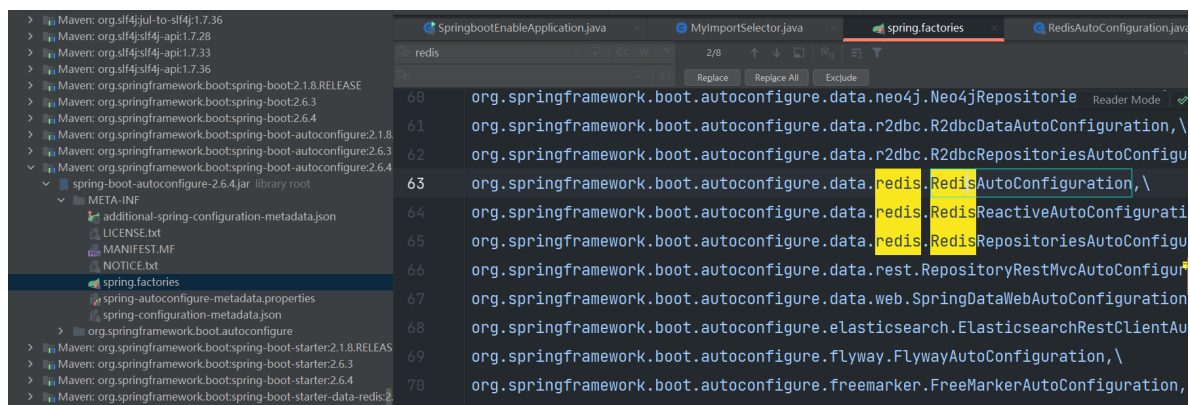
4  * 2. 导入配置类
5  * 3. 导入ImportSelector的实现类。
6  * 4. 导入ImportBeanDefinitionRegistrar实现类
7      */
8
9      //@Import(User.class)
10     //@Import(UserConfig.class)
11     //@Import(MyImportSelector.class)
12     //@Import({MyImportBeanDefinitionRegistrar.class})
13
14     @SpringBootApplication
15     public class SpringbootEnableApplication {
16     public static void main(String[] args) {
17         ConfigurableApplicationContext context =
18         SpringApplication.run(SpringbootEnableApplication.class, args);
19
20         /*//获取Bean
21         Object user = context.getBean("user");
22         System.out.println(user);*/
23
24         /*User user = context.getBean(User.class);
25         System.out.println(user);
26
27         Role role = context.getBean(Role.class);
28         System.out.println(role);*/
29
30         /* Object user = context.getBean("user");
31         System.out.println(user);*/
32         Map<String, User> map = context.getBeansOfType(User.class);
33         System.out.println(map);
34     }
35 }

```

@EnableAutoConfiguration中使用的是第三种方式：@Import(AutoConfigurationImportSelector.class)

6、SpringBoot自动配置-@EnableAutoConfiguration详解

面试题：springboot 自动配置原理？



- @EnableAutoConfiguration 注解内部使用 @Import(AutoConfigurationImportSelector.class)来加载配置类。
- 配置文件位置：META-INF/spring.factories，该配置文件中定义了大量的配置类，当 SpringBoot 应用启动时，会自动加载这些配置类，初始化Bean
- 并不是所有的Bean都会被初始化，在配置类中使用Condition来加载满足条件的Bean

第二章 自定义starter - 了解

1、SpringBoot自动配置-自定义starter步骤分析

需求：自定义redis-starter。要求当导入redis-starter坐标时，SpringBoot自动创建Jedis的Bean。

步骤：

- ①创建 redis-spring-boot-autoconfigure 模块
- ②创建 redis-spring-boot-starter 模块,依赖 redis-spring-boot-autoconfigure的模块
- ③在 redis-spring-boot-autoconfigure 模块中初始化 Jedis 的 Bean。并定义META-INF/spring.factories 文件
- ④在测试模块中引入自定义的 redis-starter 依赖，测试获取 Jedis 的Bean，操作 redis。

2、SpringBoot自动配置-自定义starter实现-1

1. 创建redis-spring-boot-starter工程

pom文件中引入redis-spring-boot-autoconfigure

```
1      <!--引入configure-->
2      <dependency>
3          <groupId>com.ydlclass</groupId>
4          <artifactId>redis-spring-boot-autoconfigure</artifactId>
5          <version>0.0.1-SNAPSHOT</version>
6      </dependency>
```

1. 创建redis-spring-boot-autoconfigure配置工程

创建RedisProperties配置文件参数绑定类

```
1  @ConfigurationProperties(prefix = "redis")
2  public class RedisProperties {
3
4      private String host = "localhost";
5      private int port = 6379;
6
7
8      public String getHost() {
9          return host;
10     }
11
12     public void setHost(String host) {
13         this.host = host;
14     }
15
16     public int getPort() {
17         return port;
18     }
19
20     public void setPort(int port) {
21         this.port = port;
22     }
23 }
```

创建RedisAutoConfiguration自动配置类

```

1  @Configuration
2  @EnableConfigurationProperties(RedisProperties.class)
3  public class RedisAutoConfiguration {
4
5      /**
6       * 提供Jedis的bean
7       */
8      @Bean
9      public Jedis jedis(RedisProperties redisProperties) {
10         return new Jedis(redisProperties.getHost(), redisProperties.getPort());
11     }
12 }

```

在resource目录下创建META-INF文件夹并创建spring.factories

注意：“\”是换行使用的

```

1  org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
2  com.ydlclass.redis.config.RedisAutoConfiguration

```

1. 在springboot-enable工程中引入自定义的redis的starter

```

1  <!-- 自定义的redis的starter-->
2      <dependency>
3          <groupId>com.ydlclass</groupId>
4          <artifactId>redis-spring-boot-starter</artifactId>
5          <version>0.0.1-SNAPSHOT</version>
6      </dependency>

```

在SpringbootEnableApplication启动类中测试

```

1  Jedis jedis = context.getBean(Jedis.class);
2  System.out.println(jedis);

```

3、SpringBoot自动配置-自定义starter实现-2

测试springboot-enable工程中的application.properties中的配置参数

```

1  redis.port=6380

```

使用注解完成有条件加载配置类

```

1  @Configuration
2  @EnableConfigurationProperties(RedisProperties.class)
3  @ConditionalOnClass(Jedis.class)
4  public class RedisAutoConfiguration {
5
6
7      /**
8       * 提供Jedis的bean
9       */
10     @Bean
11     @ConditionalOnMissingBean(name = "jedis")
12     public Jedis jedis(RedisProperties redisProperties) {
13         System.out.println("RedisAutoConfiguration...");
14         return new Jedis(redisProperties.getHost(), redisProperties.getPort());
15     }
16 }

```

第三章 事件监听

1、SpringBoot事件监听

Java中的事件监听机制定义了以下几个角色：

- ①事件：Event，继承 java.util.EventObject 类的对象
- ②事件源：Source，任意对象Object
- ③监听器：Listener，实现 java.util.EventListener 接口的对象

SpringBoot 在项目启动时，会对几个监听器进行回调，我们可以实现这些监听器接口，在项目启动时完成一些操作。

- ApplicationContextInitializer
- SpringApplicationRunListener
- CommandLineRunner
- ApplicationRunner

自定义监听器的启动时机：MyApplicationRunner和MyCommandLineRunner都是当项目启动后执行，使用@Component放入容器即可使用

MyApplicationRunner

```
1  /**
2   * 当项目启动后执行run方法。
3   */
4  @Component
5  public class MyApplicationRunner implements ApplicationRunner {
6      @Override
7      public void run(ApplicationArguments args) throws Exception {
8          System.out.println("ApplicationRunner...run");
9          System.out.println(Arrays.asList(args.getSourceArgs()));
10     }
11 }
```

MyCommandLineRunner

```
1  @Component
2  public class MyCommandLineRunner implements CommandLineRunner {
3      @Override
4      public void run(String... args) throws Exception {
5          System.out.println("CommandLineRunner...run");
6          System.out.println(Arrays.asList(args));
7      }
8  }
```

MyApplicationContextInitializer的使用要在resource文件夹下添加META-INF/spring.factories

```
1  org.springframework.context.ApplicationContextInitializer=com.ydlclass.springbootl
   istener.listener.MyApplicationContextInitializer
```



```

1  @Component
2  public class MyApplicationContextInitializer implements
    ApplicationContextInitializer {
3      @Override
4      public void initialize(ConfigurableApplicationContext applicationContext) {
5          System.out.println("ApplicationContextInitializer...initialize");
6      }
7  }

```

MySpringApplicationRunListener的使用要添加**构造器**

spring.factories加

```

1  org.springframework.boot.SpringApplicationRunListener=com.ydlclass.springbootlistener.listener.MySpringApplicationRunListener

```

```

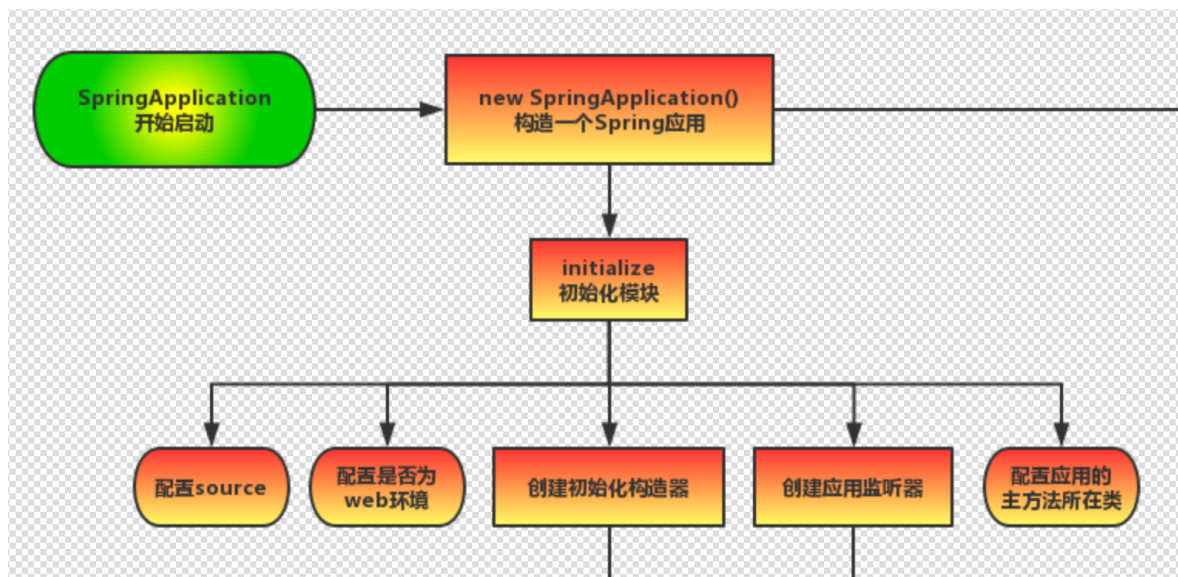
1  public class MySpringApplicationRunListener implements
    SpringApplicationRunListener {
2
3      public MySpringApplicationRunListener(SpringApplication application,
        String[] args) {
4      }
5
6      @Override
7      public void starting() {
8          System.out.println("starting...项目启动中");
9      }
10
11     @Override
12     public void environmentPrepared(ConfigurableEnvironment environment) {
13         System.out.println("environmentPrepared...环境对象开始准备");
14     }
15
16     @Override
17     public void contextPrepared(ConfigurableApplicationContext context) {
18         System.out.println("contextPrepared...上下文对象开始准备");
19     }
20
21     @Override
22     public void contextLoaded(ConfigurableApplicationContext context) {
23         System.out.println("contextLoaded...上下文对象开始加载");
24     }
25
26     @Override
27     public void started(ConfigurableApplicationContext context) {
28         System.out.println("started...上下文对象加载完成");
29     }
30
31     @Override
32     public void running(ConfigurableApplicationContext context) {
33         System.out.println("running...项目启动完成，开始运行");
34     }
35
36     @Override
37     public void failed(ConfigurableApplicationContext context, Throwable
        exception) {
38         System.out.println("failed...项目启动失败");
39     }

```

第四章 SpringBoot启动流程

1、SpringBoot流程分析-初始化

1. 配置启动引导类（判断是否有启动主类）
2. 判断是否是Web环境
3. 获取初始化类、监听器类

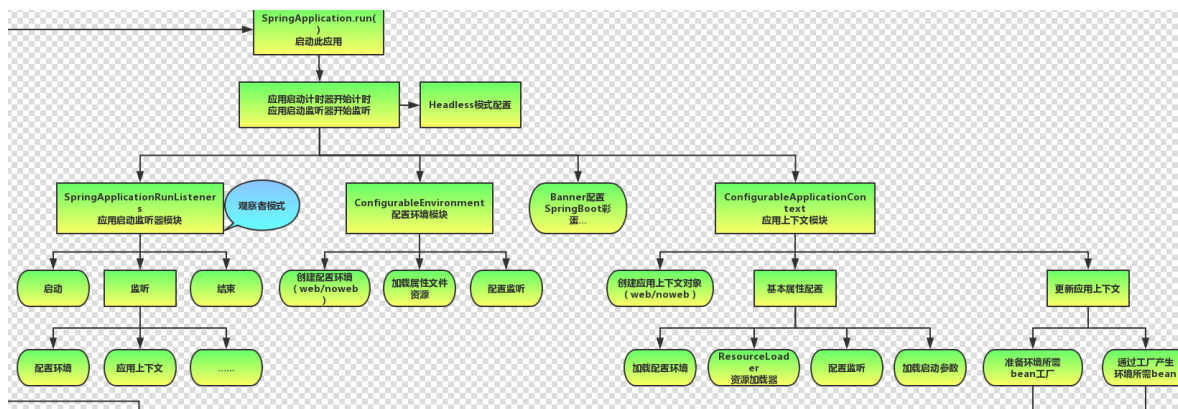


2、SpringBoot流程分析-run

1. 启动计时器
2. 执行监听器
3. 准备环境
4. 打印banner：可以在resource下粘贴自定义的banner
5. 创建context

```
1 refreshContext(context);
```

执行refreshContext方法后才真正创建Bean



第五章 监控-运维

1、SpringBoot监控-actuator基本使用

①导入依赖坐标

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-actuator</artifactId>
4 </dependency>
```

②访问 <http://localhost:8080/actuator>

```
1 {
2   "_links":{
3     "self":{
4       "href":"http://localhost:8080/actuator",
5       "templated":false
6     },
7     "health":{
8       "href":"http://localhost:8080/actuator/health",
9       "templated":false
10    },
11    "health-component-instance":{
12
13      "href":"http://localhost:8080/actuator/health/{component}/{instance}",
14      "templated":true
15    },
16    "health-component":{
17      "href":"http://localhost:8080/actuator/health/{component}",
18      "templated":true
19    },
20    "info":{
21      "href":"http://localhost:8080/actuator/info",
22      "templated":false
23    }
24  }
```

<http://localhost:8080/actuator/info>

在application.properties中配置

```
1 info.name=lucy
2 info.age=99
```

<http://localhost:8080/actuator/health>

开启健康检查详细信息

```
1 management.endpoint.health.show-details=always
```

```
1 {
2   "status":"UP",
3   "details":{
4     "diskSpace":{
5       "status":"UP",
```

```

6         "details":{
7             "total":159579508736,
8             "free":13558104064,
9             "threshold":10485760
10        }
11    },
12    "redis":{
13        "status":"UP",
14        "details":{
15            "version":"2.4.5"
16        }
17    }
18 }
19 }

```

2、SpringBoot监控-actuator开启所有endpoint

开启所有endpoint

在application.properties中配置：

```
1 management.endpoints.web.exposure.include=*
```

开启所有endpoint的返回结果：

```

1  {
2      "_links":{
3          "self":{
4              "href":"http://localhost:8080/actuator",
5              "templated":false
6          },
7          "auditevents":{
8              "href":"http://localhost:8080/actuator/auditevents",
9              "templated":false
10         },
11         "beans":{
12             "href":"http://localhost:8080/actuator/beans",
13             "templated":false
14         },
15         "caches-cache":{
16             "href":"http://localhost:8080/actuator/caches/{cache}",
17             "templated":true
18         },
19         "caches":{
20             "href":"http://localhost:8080/actuator/caches",
21             "templated":false
22         },
23         "health-component-instance":{
24             "href":"http://localhost:8080/actuator/health/{component}/{instance}",
25             "templated":true
26         },
27         "health":{
28             "href":"http://localhost:8080/actuator/health",
29             "templated":false
30         },
31         "health-component":{
32             "href":"http://localhost:8080/actuator/health/{component}",

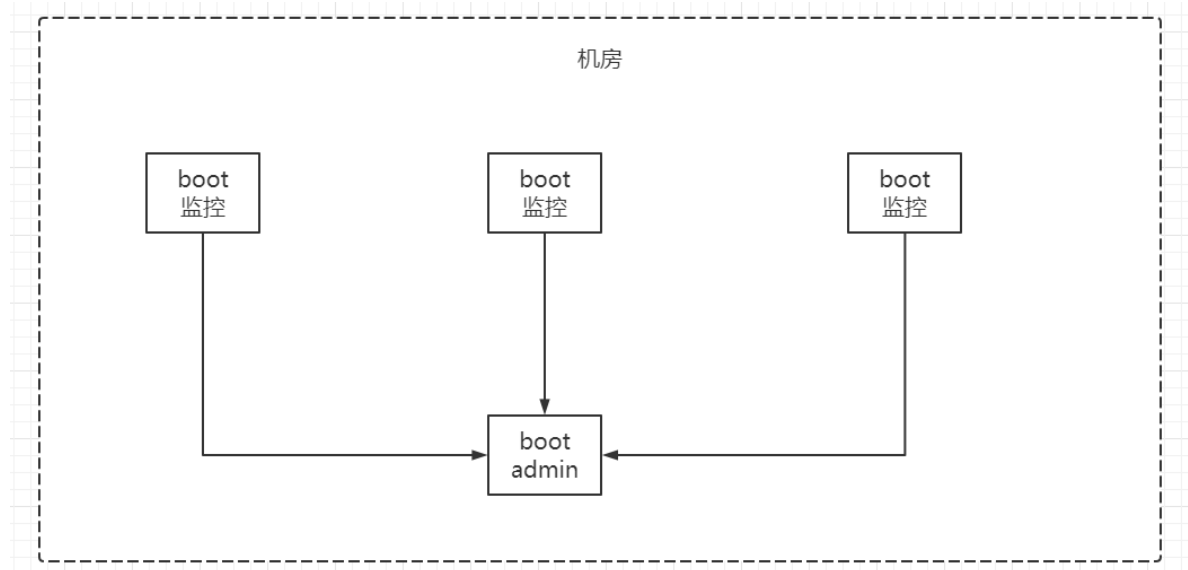
```

```
33         "templated":true
34     },
35     "conditions":{
36         "href":"http://localhost:8080/actuator/conditions",
37         "templated":false
38     },
39     "configprops":{
40         "href":"http://localhost:8080/actuator/configprops",
41         "templated":false
42     },
43     "env":{
44         "href":"http://localhost:8080/actuator/env",
45         "templated":false
46     },
47     "env-toMatch":{
48         "href":"http://localhost:8080/actuator/env/{toMatch}",
49         "templated":true
50     },
51     "info":{
52         "href":"http://localhost:8080/actuator/info",
53         "templated":false
54     },
55     "loggers":{
56         "href":"http://localhost:8080/actuator/loggers",
57         "templated":false
58     },
59     "loggers-name":{
60         "href":"http://localhost:8080/actuator/loggers/{name}",
61         "templated":true
62     },
63     "heapdump":{
64         "href":"http://localhost:8080/actuator/heapdump",
65         "templated":false
66     },
67     "threaddump":{
68         "href":"http://localhost:8080/actuator/threaddump",
69         "templated":false
70     },
71     "metrics-requiredMetricName":{
72         "href":"http://localhost:8080/actuator/metrics/{requiredMetricName}",
73         "templated":true
74     },
75     "metrics":{
76         "href":"http://localhost:8080/actuator/metrics",
77         "templated":false
78     },
79     "scheduledtasks":{
80         "href":"http://localhost:8080/actuator/scheduledtasks",
81         "templated":false
82     },
83     "httptrace":{
84         "href":"http://localhost:8080/actuator/httptrace",
85         "templated":false
86     },
87     "mappings":{
88         "href":"http://localhost:8080/actuator/mappings",
89         "templated":false
```

```
90     }
91 }
92 }
```

3、SpringBoot监控-springboot admin图形化界面使用

SpringBoot Admin 有两个角色，客户端(Client)和服务端(Server)。

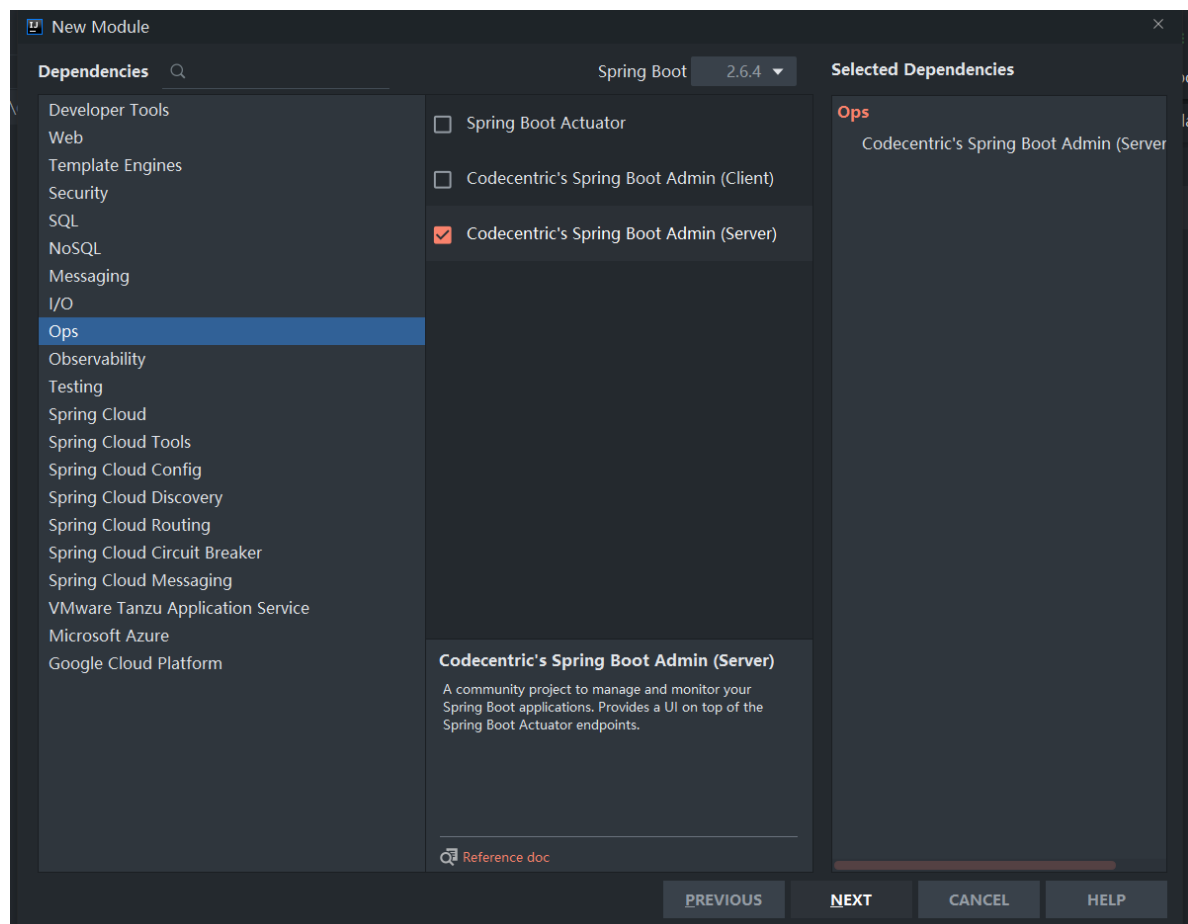


以下为创建服务端和客户端工程步骤：

admin-server:

①创建 admin-server 模块

②导入依赖坐标 admin-starter-server



```

1      <dependency>
2          <groupId>de.codecentric</groupId>
3          <artifactId>spring-boot-admin-starter-server</artifactId>
4      </dependency>

```

③在引导类上启用监控功能@EnableAdminServer

```

1  @EnableAdminServer
2  @SpringBootApplication
3  public class SpringbootAdminServerApplication {
4
5      public static void main(String[] args) {
6          SpringApplication.run(SpringbootAdminServerApplication.class, args);
7      }
8
9  }

```

admin-client:

①创建 admin-client 模块

②导入依赖坐标 admin-starter-client

```

1      <dependency>
2          <groupId>de.codecentric</groupId>
3          <artifactId>spring-boot-admin-starter-client</artifactId>
4      </dependency>

```

③配置相关信息：server地址等

```

1  # 执行admin.server地址
2  spring.boot.admin.client.url=http://localhost:9000
3
4  management.endpoint.health.show-details=always
5  management.endpoints.web.exposure.include=*

```

④启动server和client服务，访问server

The screenshot displays the Spring Boot Admin web interface. The top navigation bar includes links for '应用端' (Client), '应用' (Application), '日志报表' (Log Reports), '关于我们' (About Us), and '简体中文' (Simplified Chinese). The main content area shows the details for a 'spring-boot-application' instance with ID 'b6bbca1e7345'. The interface is divided into several sections:

- Insights:** A sidebar menu with options like '性能' (Performance), '环境' (Environment), '类' (Classes), '配置属性' (Configuration Properties), '计划任务' (Scheduled Tasks), '日志配置' (Log Configuration), 'JVM', '映射' (Mapping), and '缓存' (Cache).
- 信息 (Info):** A section indicating that no information is currently provided.
- 元数据 (Metadata):** A table showing the startup time as '2022-03-08T17:21:31.304+08:00'.
- 健康 (Health):** A section showing the instance status as 'UP' and disk space details: total 1.11 TB, free 551 GB, threshold 10.5 MB, and exists true.
- 进程 (Process):** A table showing process details: PID 19772, runtime 0d 0h 4m 18s, process CPU usage 0.00, system CPU usage 0.11, and CPU core count 8.
- 线程 (Threads):** A bar chart showing thread counts: 24 for active threads (活跃线程), 19 for guarded threads (守护线程), and 28 for thread pool threads (线程池).

第六章 SpringBoot部署

SpringBoot 项目开发完毕后，支持两种方式部署到服务器：

①jar包(官方推荐)

②war包

更改pom文件中的打包方式为war

```
1 <packaging>war</packaging>
```

修改启动类

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.boot.builder.SpringApplicationBuilder;
4 import
  org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
5
6 @SpringBootApplication
7 public class SpringbootDeployApplication extends SpringBootServletInitializer {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SpringbootDeployApplication.class, args);
11     }
12
13
14     @Override
15     protected SpringApplicationBuilder configure(SpringApplicationBuilder
  builder) {
16         return builder.sources(SpringbootDeployApplication.class);
17     }
18 }
```

指定打包的名称

```
1 <build>
2     <finalName>springboot</finalName>
3     <plugins>
4         <plugin>
5             <groupId>org.springframework.boot</groupId>
6             <artifactId>spring-boot-maven-plugin</artifactId>
7         </plugin>
8     </plugins>
9 </build>
```