

学习目标

1. 能够理解SpringBoot设计理念
2. 能够使用idea工具构建SpringBoot项目
3. 能够熟练的应用SpringBoot配置文件
4. 能够熟练的整合mybatis、redis
5. 能够运用代码测试工具
6. 理解版本控制的原理

1 Springboot的介绍

1.1 目标

能够理解springboot的设计理念，知道springboot用来解决什么问题

1.2 学习路径

- 目前项目开发的一些问题
- springboot的解决的问题
- springboot的特点和介绍

1.3 讲解

1.3.1 项目中开发的一些问题？

目前我们开发的过程当中，一般采用一个单体应用的开发采用SSM等框架进行开发，并在 开发的过程当中使用了大量的xml等配置文件，以及在开发过程中使用MAVEN的构建工具来进行构建项目，但是往往有时也会出现依赖的一些冲突，而且开发的时候测试还需要下载和使用tomcat等等这些servlet容器，所以开发的效率不高。

1.3.2 springboot解决的问题

那么在以上的问题中，我们就可以使用springboot来解决这些问题，当然他不仅仅是解决这些问题，来提高我们的开发人员的开发效率。使用springboot：可以不需要配置，可以不需要自己单独去获取tomcat,基本解决了包依赖冲突的问题，一键发布等等特性。

1.3.3 springboot的介绍

官方的说明：

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

特性:

创建独立的Spring应用程序

直接嵌入Tomcat, Jetty或Undertow (无需部署WAR文件)

提供“入门”依赖项(起步依赖), 以简化构建配置

尽可能自动配置Spring和第三方库

提供可用于生产的功能, 例如指标, 运行状况检查和外部化配置

完全没有代码生成, 也不需要XML配置

1.4 小结

springboot 就是一个基于spring的一个框架。提供了一些自动配置的依赖包, 自动嵌入servlet的容器, 简化了我们开发的配置, 提升开发人员的开发效率, 并解决了包依赖的问题。

2 Springboot使用入门

2.1 目标

能使用idea实现springboot的入门程序

2.2 学习路径

- 入门的需求
- 使用环境的准备
- 实现配置和开发
- 测试和小结

2.3 讲解

2.3.1 入门需求

使用springboot在页面中展示一个hello world

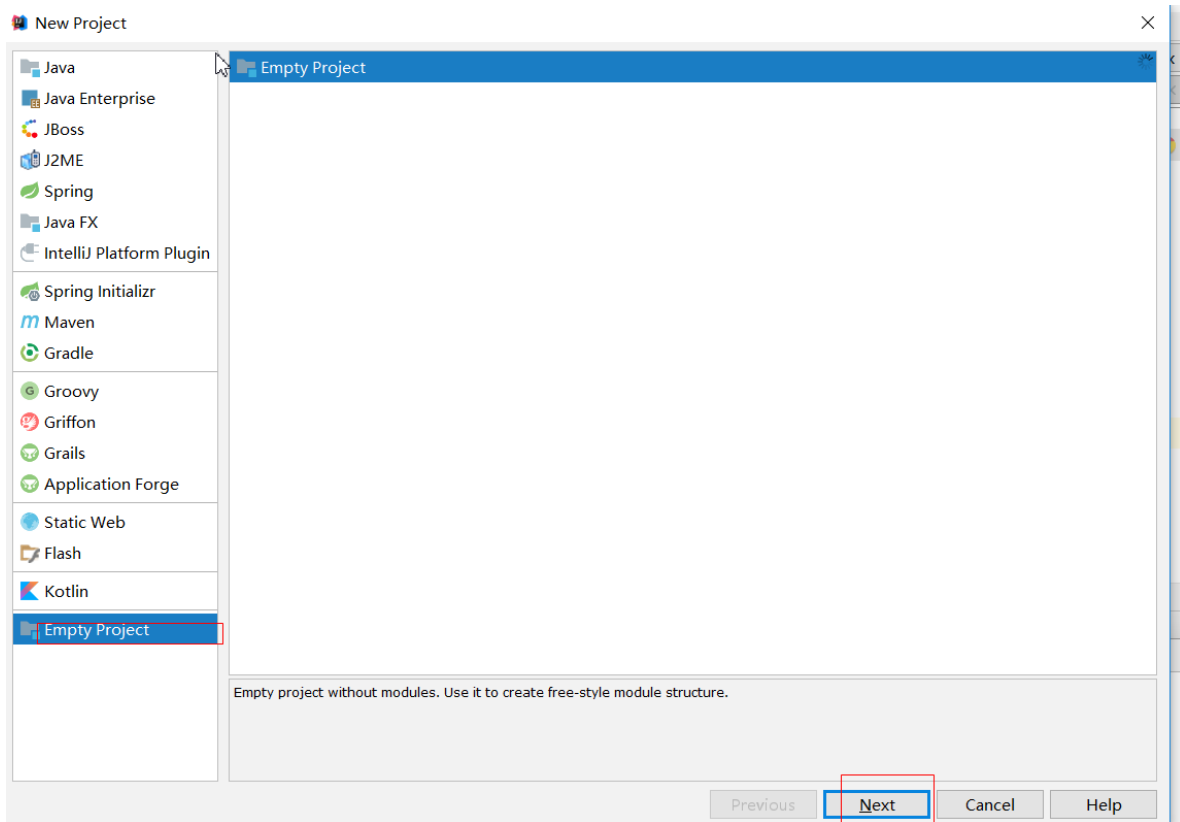
2.3.2 环境准备

推荐：使用springboot版本 springboot 2.1.4.RELEASE

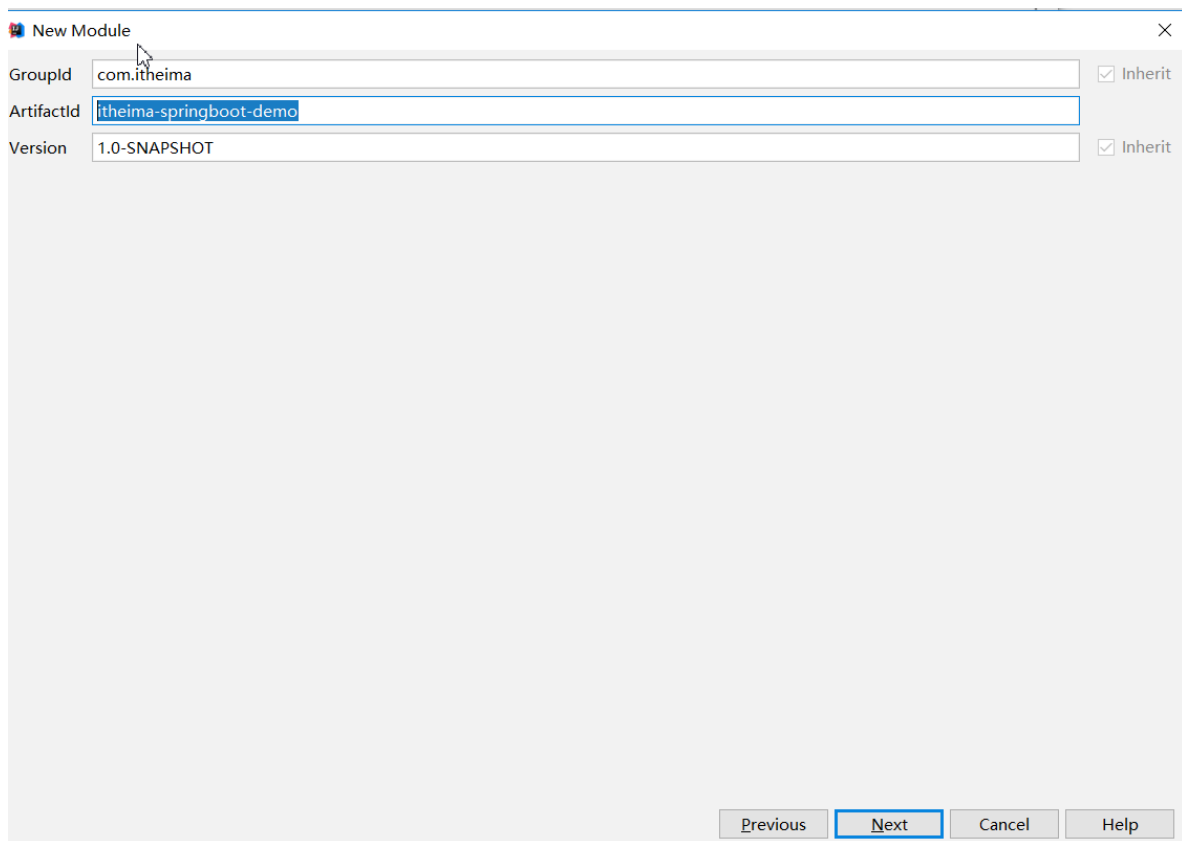
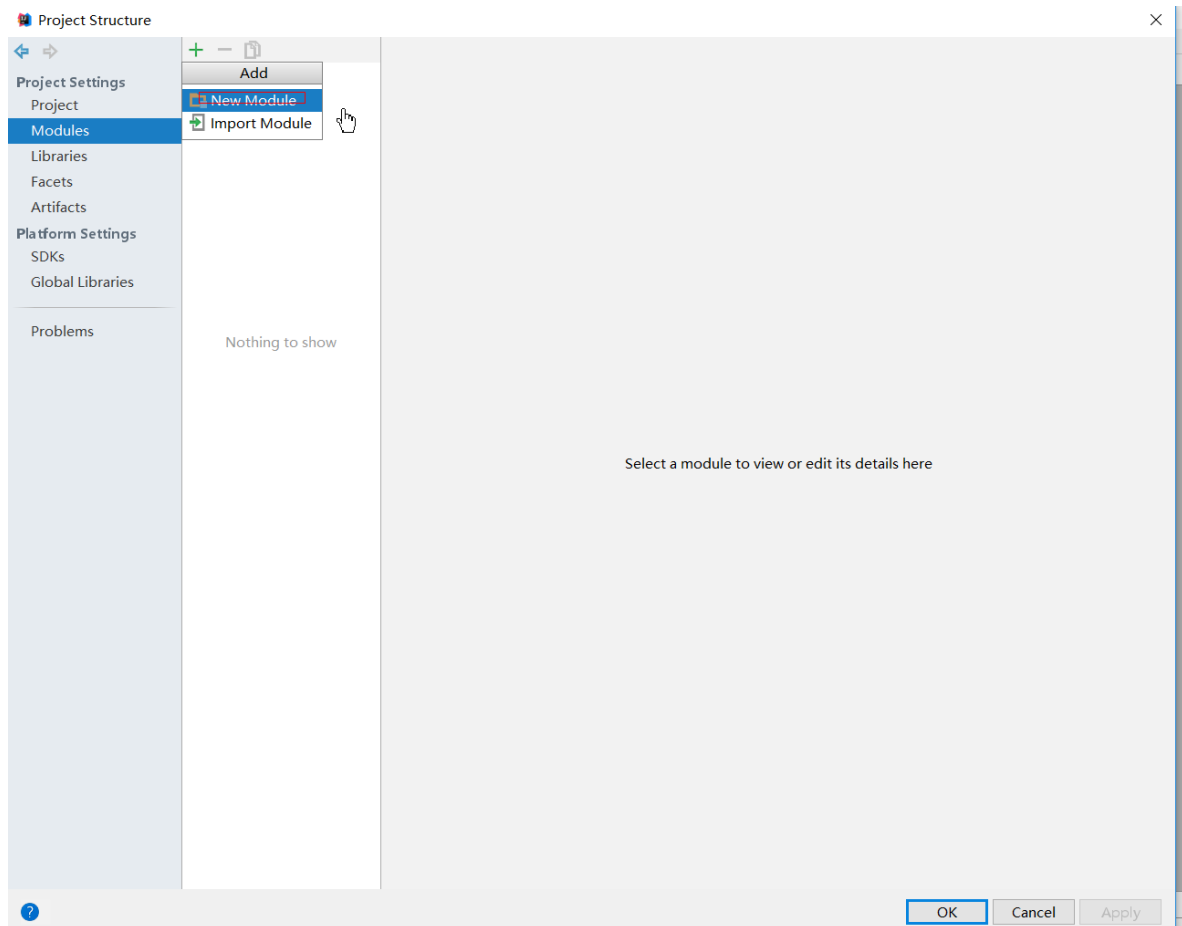
推荐：jdk1.8

2.3.3 开始开发和配置

(1) 创建空工程



(2) 创建maven工程



(3) 配置pom.xml

- 添加parent

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.4.RELEASE</version>
</parent>
```

- 添加起步依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

- pom.xml的整体代码如下

```
<?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>com.itheima</groupId>
  <artifactId>itheima-springboot-demo</artifactId>
  <version>1.0-SNAPSHOT</version>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>
</project>
```

(4) 创建启动类（或者叫引导类）

```
@SpringBootApplication
public class SpringbootApplication {

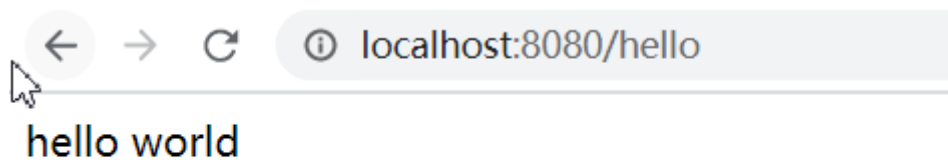
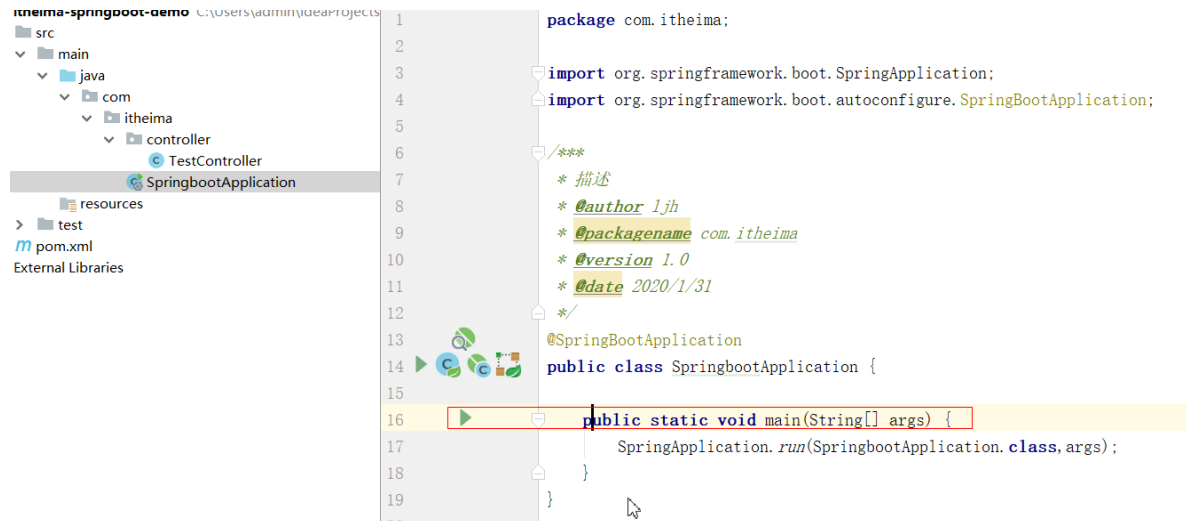
    public static void main(String[] args) {
        SpringApplication.run(SpringbootApplication.class,args);
    }
}
```

(5) 创建controller 实现展示hello world

```
@RestController
public class TestController {
    @RequestMapping("/hello")
    public String showHello(){
        return "hello world";
    }
}
```

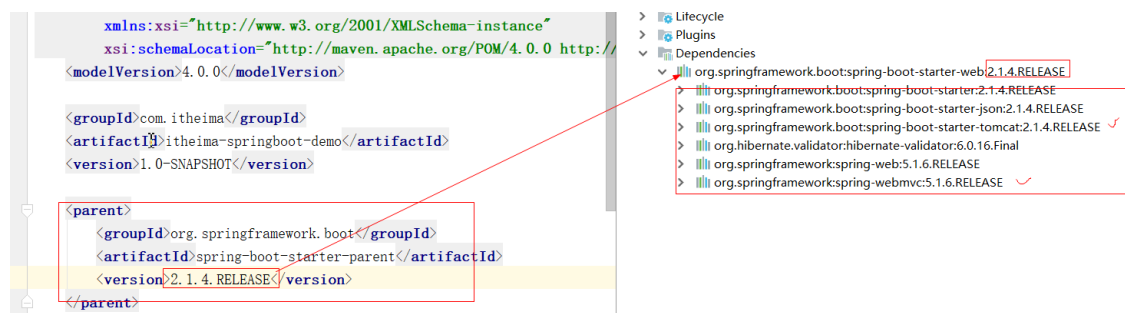
(6) 测试

启动启动类的main方法，在浏览器中输入localhost:8080/hello,则在页面中显示hello world



2.4 小结

上边的入门程序 很快开发完成，没有任何的配置，只需要添加依赖,设置springboot的parent，创建引导类即可。springboot的设置parent用于管理springboot的依赖的版本，起步依赖快速的依赖了在web开发中的所需要的依赖项。



2.5 Spring Initializr的方式创建springboot工程

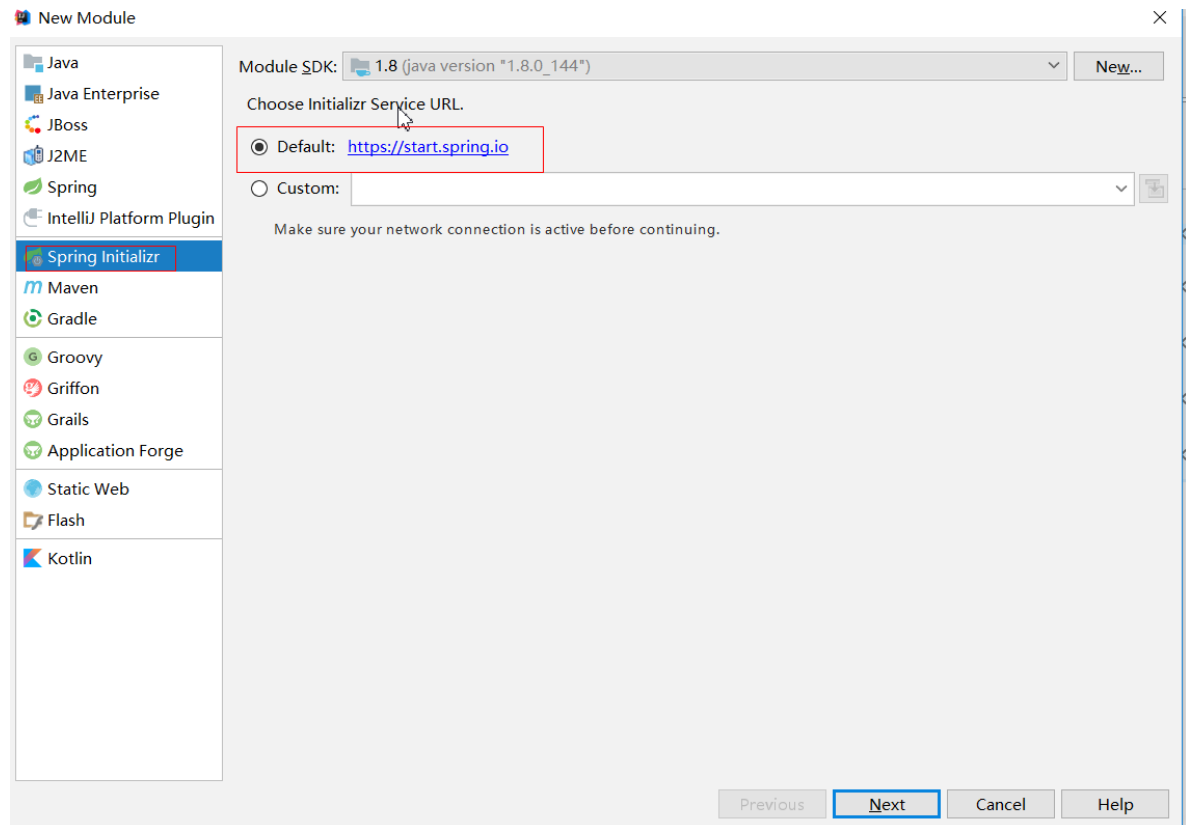
2.5.1 目标

了解相关的操作步骤实现创建springboot项目。

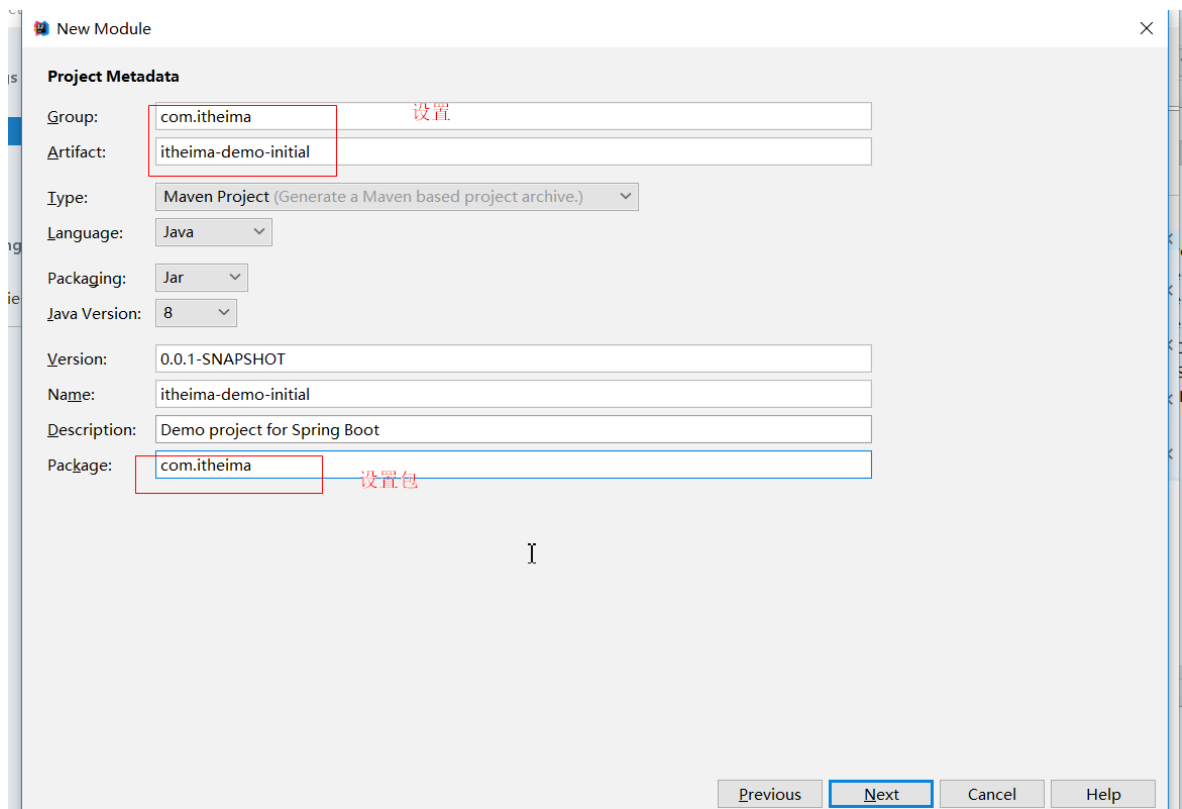
2.5.2 讲解

可以使用idea提供的方式来快速的创建springboot的项目，更加的方便，但是要求需要 联网而且网络需要比较稳定才行。如下步骤：

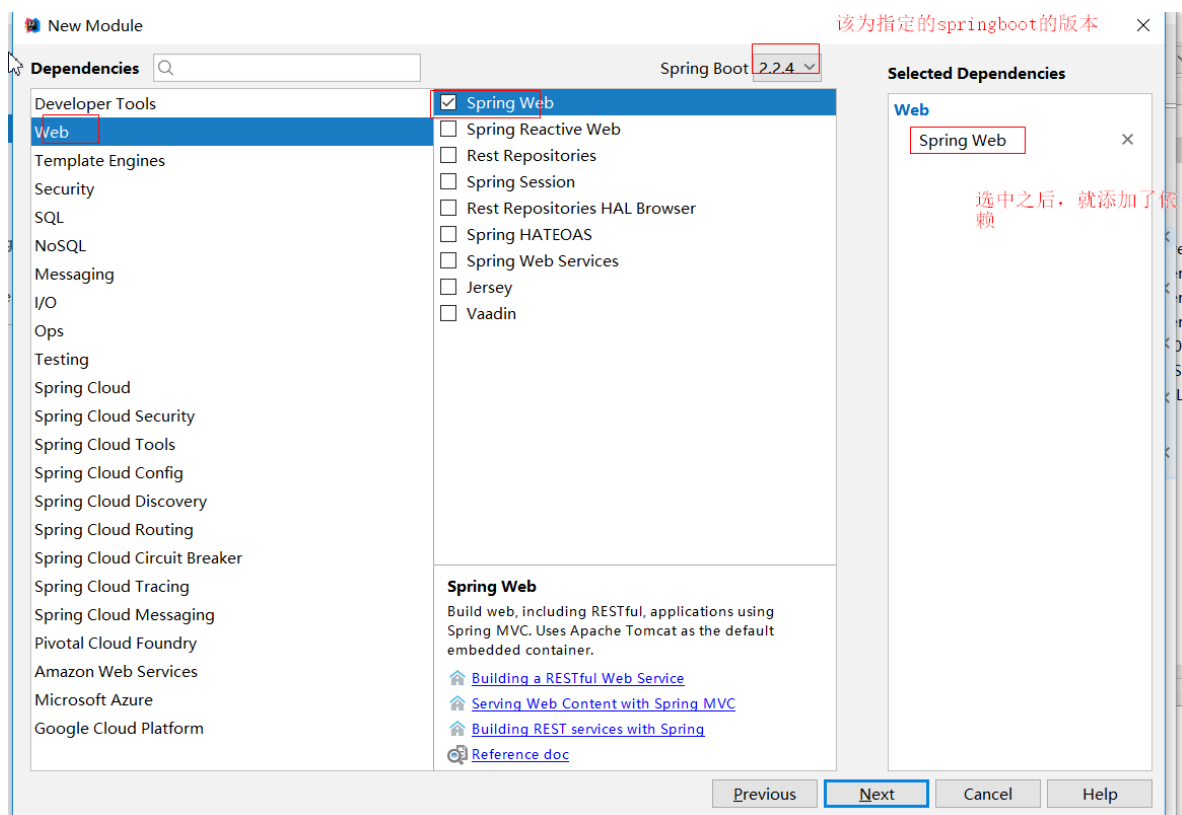
(1) 选中spring initializr



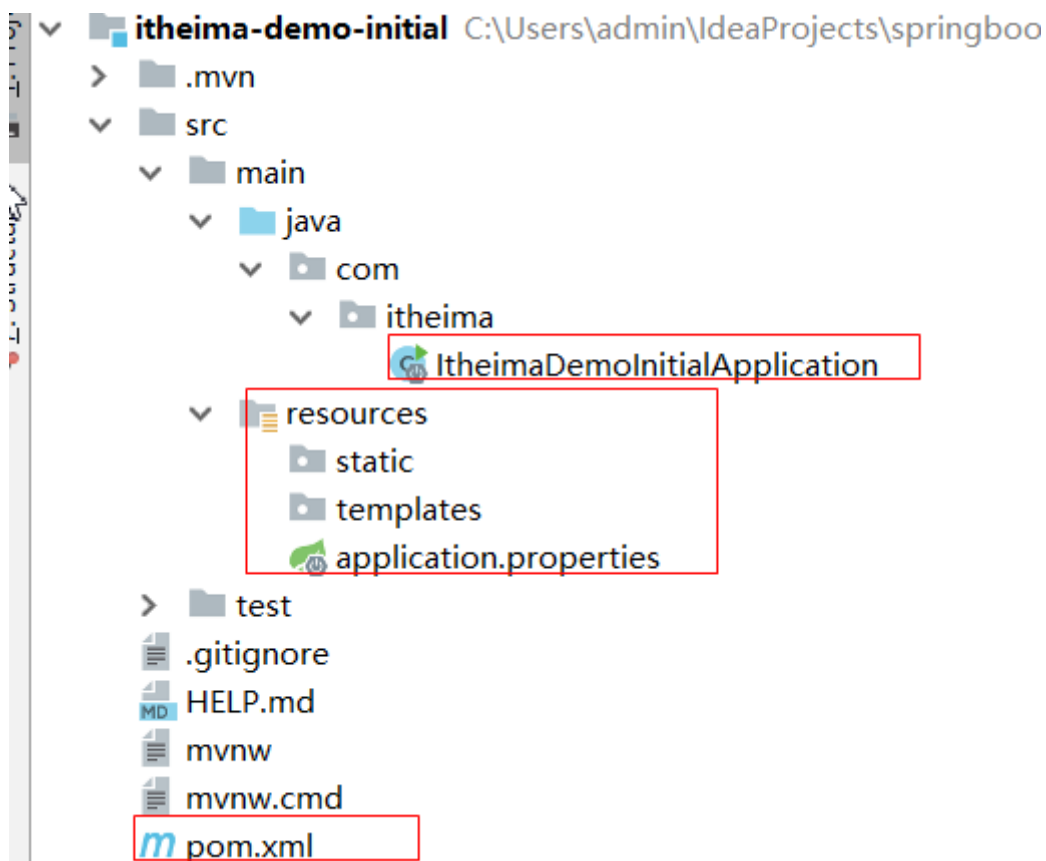
(2) 设置包名和坐标名



(3) 设置添加依赖



(4) 生成成功，自动生成相关引导类，添加依赖了。不需要手动的进行添加。如下图：



解释：

```
--ItheimaDemoInitialApplication类为引导类（启动类）
--static 目录用于存储静态资源
--templates 目录用于存储模板文件
--application.properties 用于配置相关的使用到的属性，所有的配置基本上都需要在这里配置，需要注意的是名字不能修改。
```

3 Springboot的配置文件

SpringBoot是**约定大于配置**的，所以很多配置都有默认值。如果想修改默认配置，可以用 `application.properties`或`application.yml(application.yaml)`自定义配置。SpringBoot默认从Resource目录加载自定义配置文件。

3.1 目标

- 掌握常用的properties的属性配置
- 掌握常用的yaml的属性配置
- 掌握获取配置文件中的属性值的常用的方式
- 掌握多配置文件切换

3.2 学习路径

- 常见的默认的配置
- 创建自定义配置格式
- 获取属性值

- 多文件配置和切换

3.3 讲解

3.3.1 properties文件

properties文件的配置多以 `key.key:key:va1ue` 的形式组成，那么springboot本身有默认的一些配置，如果要修改这些默认的配置，可以在`application.properties`中进行配置修改。

比如：修改端口配置

```
server.port=8081
```

3.3.2 yaml或者yml文件

yaml文件等价于properties文件，在使用过程中都是一样的效果。但是yaml文件书写的方式和properties文件不一样。更加简洁，那么我们可以根据需求选择性的使用properties和yaml文件。如果同时存在两个文件，那么优先级properties要高于yaml。

语法特点如下：

- 大小写敏感
- 数据值前必须有空格，作为分隔符
- 缩进的空格数目不重要，只需要对齐即可
- `#` 表示注释

书写格式如下要求如下：key和key之间需要换行以及空格两次。简单key value之间需要冒号加空格。

```
key1:
  key2:
    key3: value
key4: value4
```

比如：

```
server:
  port: 8081
```

注意：yaml语法中，相同缩进代表同一个级别

```
# 基本格式 key: value
name: zhangsan
# 数组 - 用于区分
city:
  - beijing
  - tianjin
  - shanghai
  - chongqing
```

```

#集合中的元素是对象形式
students:
  - name: zhangsan
    age: 18
    score: 100
  - name: lisi
    age: 28
    score: 88
  - name: wangwu
    age: 38
    score: 90
#map集合形式
maps: {"name": "zhangsan", "age": "15"}
#参数引用
person:
  name: ${name} # 该值可以获取到上边的name定义的值

```

3.3.3 获取配置文件中值

获取配置文件中的值我们一般有几种方式：

- @value注解的方式 只能获取简单值
- Environment的方式
- @ConfigurationProperties

演示如下：

yaml中配置：

```

# 基本格式 key: value
name: zhangsan
# 数组 - 用于区分
city:
  - beijing
  - tianjin
  - shanghai
  - chongqing
#集合中的元素是对象形式
students:
  - name: zhangsan
    age: 18
    score: 100
  - name: lisi
    age: 28
    score: 88
  - name: wangwu
    age: 38
    score: 90
#map集合形式
maps: {"name": "zhangsan", "age": "15"}
#参数引用
person:
  name: ${name} # 该值可以获取到上边的name定义的值
  age: 12

```

java代码:

controller

```
@RestController
public class Test2Controller {
    @Value("${name}")
    private String name;

    @Value("${city[0]}")
    private String city0;

    @Value("${students[0].name}")
    private String studentname;

    @Value("${person.name}")
    private String personName;

    @Value("${maps.name}")//value注解只能获简单的值对象
    private String name1;

    @Autowired
    private Student student;

    @RequestMapping("/show")
    public String showHello() {
        System.out.println(name);
        System.out.println(city0);
        System.out.println(studentname);
        System.out.println(personName);

        System.out.println(">>>>"+student.getAge());

        return "hello world";
    }
}
```

pojo:

```
@Component
@ConfigurationProperties(prefix = "person")
public class Student {
    private String name;
    private Integer age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

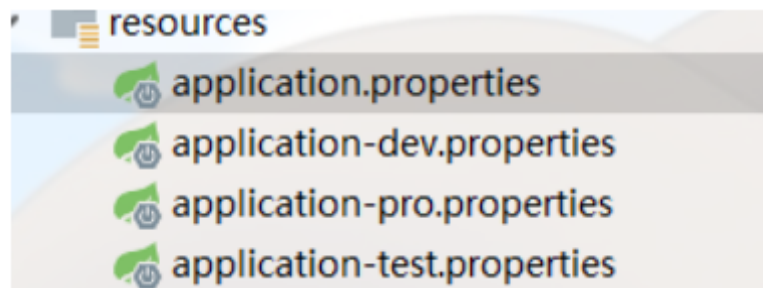
    public Integer getAge() {
        return age;
    }
}
```

```
public void setAge(Integer age) {  
    this.age = age;  
}  
  
}
```

3.3.4 profile

在开发的过程中，需要配置不同的环境，所以即使我们在application.yml中配置了相关的配置项，当时在测试是，需要修改数据源等端口路径的配置，测试完成之后，又上生产环境，这时配置又需要修改，修改起来很麻烦。

- properties配置方式



application.properties:

```
#通过active指定选用配置环境  
spring.profiles.active=test
```

application-dev.properties:

```
#开发环境  
server.port=8081
```

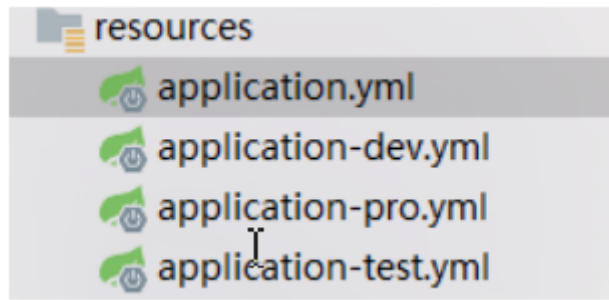
application-test.properties

```
server.port=8082
```

application-pro.properties

```
server.port=8083
```

- yml配置方式



application.yml:

```
#通过active指定选用配置环境
spring:
  profiles:
    active: pro
```

application-dev.yml:

```
#开发环境
server:
  port: 8081
```

application-test.yml:

```
#测试环境
server:
  port: 8082
```

applicatioin-pro.yml

```
#生产环境
server:
  port: 8083
```

还有一种是分隔符的方式（了解）

```
spring:
  profiles:
    active: dev

---
#开发环境
server:
  port: 8081
spring:
  profiles: dev

---
```

```
#测试环境
server:
  port: 8082
spring:
  profiles: test
```

```
#生产环境
server:
  port: 8083
spring:
  profiles: pro
```

- 激活profile的方式（了解）
 - 配置文件的方式（上边已经说过）
 - 运行是指定参数 `java -jar xxx.jar --spring.profiles.active=test`
 - jvm虚拟机参数配置 `-Dspring.profiles.active=dev`

4 Springboot集成第三方框架

在springboot使用过程中，基本是和第三方的框架整合使用的比较多，就是利用了springboot的简化配置，起步依赖的有效性。我们整合一些常见的第三方框架进行整合使用，后面会需要用到。

4.1 目标

- 掌握springboot整合mybatis
- 掌握springboot整合redis
- 掌握springboot整合junit

4.2 学习路径

- 学习springboot整合junit
- 学习springboot整合mybatis
- 学习springboot整合redis

4.3 讲解

4.3.1 springboot整合junit

(1)添加依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

(2)创建在/main/java/下一个类UserService用于测试:

```
package com.itheima.service;

/**
 * 描述
 * @author ljh
 * @package com.itheima.service
 * @version 1.0
 * @date 2020/2/26
 */
@Service
public class UserService {

    public String getUser() {
        System.out.println("获取用户的信息");
        return "zhangsan";
    }
}
```

(3)在test/java/下创建测试类，类的包名和启动类的报名一致即可

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBootApplicationTests {

    @Autowired
    private UserService userService;

    @Test
    public void getUser() {
        String userinfo = userService.getUser();
        System.out.println(userinfo);
    }
}
```

解释:

@RunWith(SpringRunner.class) 使用springrunner运行器
@SpringBootTest 启用springboot测试

使用的方式和之前的spring的使用方式差不多。

4.3.2 springboot整合mybatis

(1) 实现步骤说明

- 1.准备数据库创建表
- 2.添加起步依赖
- 3.创建POJO
- 4.创建mapper接口
- 5.创建映射文件
- 6.配置yml 指定映射文件位置
- 7.创建启动类，加入注解扫描
- 8.创建service controller 进行测试

(2) 创建数据库表

```
-- -----  
-- Table structure for `user`  
-- -----  
  
DROP TABLE IF EXISTS `user`;  
CREATE TABLE `user` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `username` varchar(50) DEFAULT NULL,  
  `password` varchar(50) DEFAULT NULL,  
  `name` varchar(50) DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;  
-- -----  
-- Records of user  
-- -----  
  
INSERT INTO `user` VALUES ('1', 'zhangsan', '123', '张三');  
INSERT INTO `user` VALUES ('2', 'lisi', '123', '李四');
```

(3)创建工程并添加依赖pom.xml如下参考

```
<?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>com.itheima</groupId>  
  <artifactId>itheima-springboot-mybatis-demo04</artifactId>  
  <version>1.0-SNAPSHOT</version>  
  
  <parent>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-parent</artifactId>  
    <version>2.1.4.RELEASE</version>  
  </parent>  
  
  <dependencies>  
    <!--驱动-->  
    <dependency>  
      <groupId>mysql</groupId>  
      <artifactId>mysql-connector-java</artifactId>  
      <scope>runtime</scope>
```

```

        </dependency>
        <!--mybatis的 起步依赖-->
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>2.0.1</version>
        </dependency>
        <!--spring web起步依赖-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>

```

(4)创建pojo

```

public class User implements Serializable{
    private Integer id;
    private String username;//用户名
    private String password;//密码
    private String name;//姓名
    //getter setter...
    //toString
}

```

(5)创建mapper接口

```

package com.itheima.dao;

import com.itheima.pojo.User;

import java.util.List;

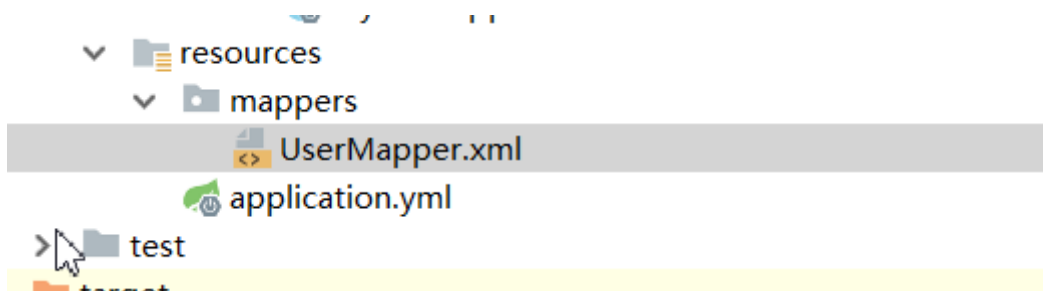
/**
 * mapper接口
 * @author ljh
 * @packagename com.itheima.dao
 * @version 1.0
 * @date 2020/2/23
 */

```

```
public interface UserMapper {
    public List<User> findAllUser();
}
```

(6)创建UserMapper映射文件，如下图所示

```
<?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.dao.UserMapper">
    <select id="findAllUser" resultType="com.itheima.pojo.User">
        SELECT * from user
    </select>
</mapper>
```



(7)创建配置application.yml文件:

```
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost/springboot_user?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
    username: root
    password: 123456
#配置mapper的映射文件的位置
mybatis:
  mapper-locations: classpath:mappers/*Mapper.xml
```

(8)创建启动类，加入mapper接口注解扫描

```
@SpringBootApplication
@MapperScan(basePackages = "com.itheima.dao")
//MapperScan 用于扫描指定包下的所有的接口，将接口产生代理对象交给spring容器
public class MybatisApplication {
    public static void main(String[] args) {
        SpringApplication.run(MybatisApplication.class,args);
    }
}
```

(9)创建service 实现类和接口

```
package com.itheima.service.impl;

import com.itheima.dao.UserMapper;
import com.itheima.pojo.User;
```

```

import com.itheima.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

/**
 * 描述
 * @author ljh
 * @package com.itheima.service.impl
 * @version 1.0
 * @date 2020/2/23
 */
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserMapper userMapper;

    @Override
    public List<User> findAllUser() {
        return userMapper.findAllUser();
    }
}

```

接口如下:

```

package com.itheima.service;

import com.itheima.pojo.User;

import java.util.List;

/**
 * 描述
 * @author ljh
 * @package com.itheima.service
 * @version 1.0
 * @date 2020/2/23
 */
public interface UserService {
    public List<User> findAllUser();
}

```

(10)创建controller

```

package com.itheima.controller;

import com.itheima.pojo.User;
import com.itheima.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

```

```

import java.util.List;

/**
 * 描述
 * @author ljh
 * @package com.itheima.controller
 * @version 1.0
 * @date 2020/2/23
 */
@RestController
@RequestMapping("/user")
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping("/findAll")
    public List<User> findAll(){
        return userService.findAllUser();
    }
}

```

(11)测试

浏览器中发送请求: `http://localhost:8080/user/findAll`

4.3.3 springboot整合redis

为了方便，我们就在之前的mybatis中的基础上进行测试。

4.3.3.1 整合redis实现数据添加

(1)springboot整合redis的步骤

1. 添加起步依赖
2. 准备好redis服务器 并启动
3. 在SerService中的方法中使用
 - 3.1 注入redisTemplate
 - 3.2 在方法中进行调用
4. 配置ym1 配置redis的服务器的地址

(2) 添加起步依赖

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

```

(3)配置redis链接信息

```

spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost/springboot_user?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
    username: root
    password: 123456
  redis:
    host: localhost
    port: 6379
#配置mapper的映射文件的位置
mybatis:
  mapper-locations: classpath:mappers/*Mapper.xml

```

(4)service中进行调用

```

@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserMapper userMapper;

    @Autowired
    private RedisTemplate redisTemplate;

    @Override
    public List<User> findAllUser() {

        //1. 获取redis中的数据
        List<User> list = (List<User>)
redisTemplate.boundValueOps("key_all").get();
        //2. 判断 是否有，如果有则返回，如果没有则从mysql中获取设置到redis中再返回
        if (list != null && list.size() > 0) {
            return list;
        }
        List<User> allUser = userMapper.findAllUser();




        //3 从mysql中获取设置到redis中再返回
        redisTemplate.boundValueOps("key_all").set(allUser);

        return allUser;
    }
}

```

(5)启动redis

如图双击exe启动

springboot > 资料 > redis2.8win32 > redis2.8win32		
名称	修改日期	类型
 redis.windows.conf	2019/9/21 14:10	CONF 文件
 redis-cli.exe	2015/2/9 16:00	应用程序
 redis-server.exe	2015/2/9 16:00	应用程序

(6) 测试，如下图所示

localhost:8080/user/findAll 2步骤

```
[{"id":1,"username":"zhangsan","password":"123","address":"北京"}, {"id":2,"username":"lisi","password":"123","address":"上海"}]
```

```
G:\courses\springboot\资料\redis2.8win32\redis2.8win32\redis-cli.exe
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> keys *
1) "\xac\xed\x00\x05t\x00\akey_all"
127.0.0.1:6379>
```

再次访问，则从redis中获取数据了。

4.3.3.2 redis的序列化机制

如上图所示，出现了乱码，这个是由于redis的默认的序列化机制导致的。这里需要注意下：并不是错误，由于序列化机制，导致我们数据无法正常显示。如果有代码的方式获取则是可以获取到数据的。

1. 默认的情况下redisTemplate操作key vlaue的时候 必须要求 key一定实现序列化 value 也需要实现序列化
2. 默认的情况下redisTemplate使用JDK自带的序列化机制：JdkSerializationRedisSerializer
3. JDK自带的序列化机制中要求需要key 和value 都需要实现Serializable接口
4. RedisTemplate支持默认以下几种序列化机制：机制都实现了RedisSerializer接口
 - + OxmSerializer
 - + GenericJackson2JsonRedisSerializer
 - + GenericToStringSerializer
 - + StringRedisSerializer
 - + JdkSerializationRedisSerializer
 - + Jackson2JsonRedisSerializer



```
ma-springboot-jpa-demo05 x MybatisApplication.java x UserServiceImpl.java x RedisTemplate.java x User.java x UserService.java x
*/
@Override
public void afterPropertiesSet() {

    super.afterPropertiesSet();

    boolean defaultUsed = false;
    if (defaultSerializer == null) {
        defaultSerializer = new JdkSerializationRedisSerializer(
            classLoader != null ? classLoader : this.getClass().getClassLoader());
    }

    if (enableDefaultSerializer) {
```

我们可以进行自定义序列化机制：例如：我们定义key 为字符串序列化机制， value：为JDK自带的方式，应当处理如下：

```
@Bean
public RedisTemplate<Object, Object> redisTemplate(
    RedisConnectionFactory redisConnectionFactory) throws
UnknownHostException {
    RedisTemplate<Object, Object> template = new RedisTemplate<>();
    template.setConnectionFactory(redisConnectionFactory);
    //设置key的值为字符串序列化方式 那么在使用过程中key 一定只能是字符串
    template.setKeySerializer(new StringRedisSerializer());
    //设置value的序列化机制为JDK自带的方式
    template.setValueSerializer(new JdkSerializationRedisSerializer());
    return template;
}
```

整体配置如下：

```
@SpringBootApplication
@ComponentScan(basePackages = "com.itheima.dao")
//MapperScan 用于扫描指定包下的所有的接口，将接口产生代理对象交给spring容器
public class MybatisApplication {
    public static void main(String[] args) {
        SpringApplication.run(MybatisApplication.class, args);
    }

    @Bean
    public RedisTemplate<Object, Object> redisTemplate(
        RedisConnectionFactory redisConnectionFactory) throws
UnknownHostException {
        RedisTemplate<Object, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory);
        //设置key的值为字符串序列化方式 那么在使用过程中key 一定只能是字符串
        template.setKeySerializer(new StringRedisSerializer());
        //设置value的序列化机制为JDK自带的方式
        template.setValueSerializer(new JdkSerializationRedisSerializer());
        return template;
    }
}
```


这里定义好了版本

如图所示：我们自己的springboot项目继承于spring-boot-starter-parent,而他又继承与spring-boot-dependencies，dependencies中定义了各个版本。通过maven的依赖传递特性从而实现了版本的统一管理。

Springboot第二天

学习目标

- 理解springboot自动配置原理
- 理解自动配置注解的原理解析
- 掌握自定义springboot的starter
- 了解springboot监听机制
- 了解springboot的启动流程
- 了解springboot监控
- 掌握springboot的部署

1 springboot的自动配置原理

在我们使用springboot的时候，能带来的方便性和便利性，不需要配置便可以实现相关的使用，开发效率极大的提升，那么实际上，springboot本身的基础依赖中封装了许许多多的配置帮我们自动完成了配置了。那么它是如何实现的呢？

1.1 Condition接口及相关注解

讲Springboot自动配置，逃不开ConditionalOnxxx等等注解，也逃不开condition接口所定义的功能。

1.1.1 condition接口

condition接口是spring4之后提供给了的接口，增加条件判断功能，用于选择性的创建Bean对象到spring容器中。

思考一个问题？

我们之前用过springboot整合redis 实现的步骤：就是添加redis起步依赖之后，直接就可以使用从spring容器中获取注入RedisTemplate对象了，而不需要创建该对象放到spring容器中了.意味着Spring boot redis的起步依赖已经能自动的创建该redisTemplate对象加入到spring容器中了。这里应用的一个重要点就是condition的应用。

我们来演示下，是否加入依赖就可以获取redisTemplate,不加依赖就不会获取到redisTemplate

1.1.1.1 效果演示

演示步骤：

1. 创建maven工程
2. 加入依赖
3. 创建启动类
4. 获取restTemplate的bean对象

(1) 创建工程添加依赖

```

<?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>com.itheima</groupId>
    <artifactId>itheima-springboot-demo01-condition</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.4.RELEASE</version>
    </parent>

    <dependencies>
        <!--加入springboot的starter起步依赖-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>

```

(2)com.itheima下创建启动类

```

package com.itheima;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class MySpringBootApplication {
    public static void main(String[] args) {
        ConfigurableApplicationContext context =
        SpringApplication.run(MySpringBootApplication.class, args);
    }
}

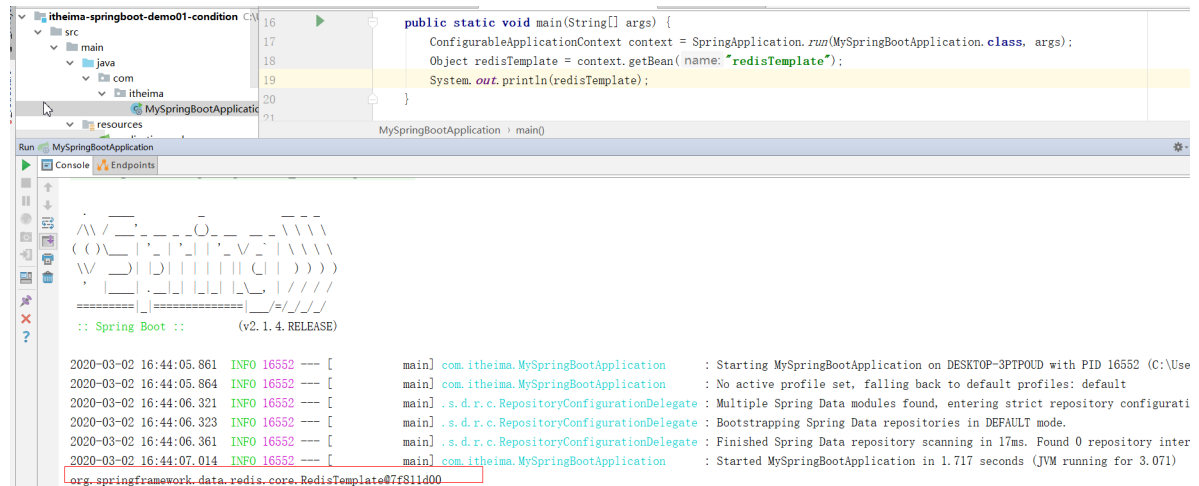
```

```

    Object redisTemplate = context.getBean("redisTemplate");
    System.out.println(redisTemplate);
}
}

```

(3)启动main方法，查看效果



(4)注释依赖则报错:



1.1.1.2 自定义实现类

刚才看到的效果，那么它到底是如何实现的呢？我们现在给一个需求：

(1)需求

在spring容器中有一个user的bean对象，如果导入了redisclient的坐标则加载该bean，如果没有导入则不加载该bean.

(2)实现步骤：

1. 定义一个接口condition的实现类
2. 实现方法 判断是否有字节码对象，有则返回true 没有则返回false
3. 定义一个User的pojo
4. 定义一个配置类用于创建user对象交给spring容器管理
5. 修改加入注解@conditional(value=condition)
6. 测试打印

(3)创建POJO

```
public class User {
    private String username;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

(4)创建condition的接口实现类

```
public class OnClassCondition implements Condition {
    /**
     * 返回true 则满足条件 返回false 则不满足条件
     *
     * @param context 上下文信息对象 可以获取环境的信息 和容器工程 和类加载器对象
     * @param metadata 注解的元数据 获取注解的属性信息
     * @return
     */
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

        //1. 获取当前的redis的类字节码对象
        try {
            //2. 加载成功则说明存在 redis的依赖 返回true,
            Class.forName("redis.clients.jedis.Jedis");
            return true;
        } catch (ClassNotFoundException e) {
            // 如果加载不成功则redis依赖不存在 返回false
            e.printStackTrace();
            return false;
        }
    }
}
```

(5) 定义配置类 在com.itheima.config下

```
package com.itheima.config;

import com.itheima.condition.OnClassCondition;
import com.itheima.pojo.User;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Conditional;
import org.springframework.context.annotation.Configuration;

/**
 * 描述
 * @author ljh
 * @package com.itheima.config
 * @version 1.0
 */
```

```

* @date 2020/3/2
*/
@Configuration
public class UserConfig {

    @Bean
    //conditional 用于指定当某一个条件满足并返回true时则执行该方法创建bean交给spring容器
    @Conditional(value = OnClassCondition.class)
    public User user() {
        return new User();
    }
}

```

解释:

`@Conditional(value = OnClassCondition.class)` 当符合指定类的条件返回true的时候则执行被修饰的方法，放入spring容器中。

(6)测试:

- 加入jedis的依赖时:

The screenshot shows an IDE with the following components:

- Project Structure:** Includes packages like `com.itheima.condition`, `com.itheima.config`, and `com.itheima.pojo`.
- Code:** `MySpringBootApplication` has a `main` method that prints `redisTemplate` and `user`. The `User` class is annotated with `@Conditional(OnClassCondition.class)`.
- Maven Dependencies:** The `pom.xml` file shows the inclusion of `jedis` version 3.2.0.
- Console Output:** Shows the application starting successfully on a Windows machine with PID 16268.

- 不加入jedis的依赖时:

The screenshot shows the same IDE setup but without the `jedis` dependency in `pom.xml`. The console output shows a runtime exception:

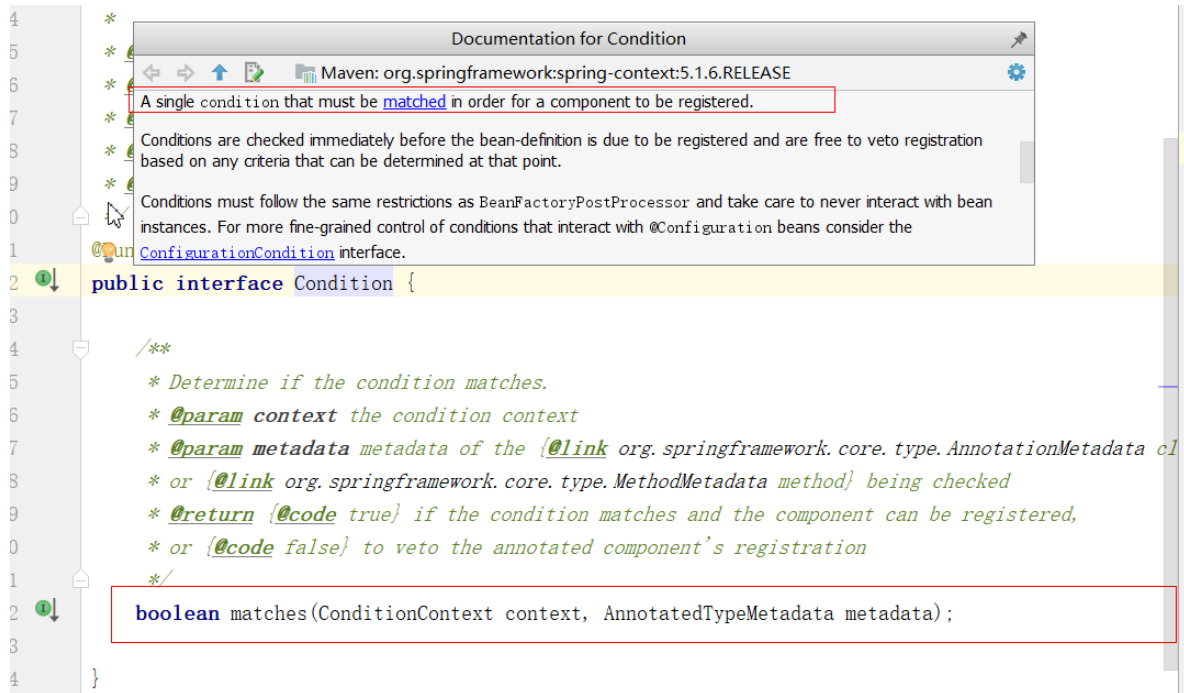
```

Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'user' available
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanDefinition(DefaultListableBeanFactory.java:775)
    at org.springframework.beans.factory.support.AbstractBeanFactory.getMergedLocalBeanDefinition(AbstractBeanFactory.java:1221)
    at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:294)
    at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:199)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1105)
    at com.itheima.MySpringBootApplication.main(MySpringBootApplication.java:20)

```

1.1.1.3 小结

我们由上边的看出。由于有了条件接口，那么我们可以选择性的在某种条件小才进行bean的注册和初始化等操作。他的接口的说明也描述了这有点；



1.1.2 需求优化

我们希望这个类注解可以进行动态的加载某一个类的全路径，不能写死为redis.将来可以进行重用。

(1) 需求：

1. 可以自定义一个注解 用于指定具体的类全路径 表示有该类在类路径下时才执行注册
2. 在配置类中使用该自定义注解 动态的指定类路径
3. 在条件的实现类中进行动态的获取并加载类即可

(2)实现步骤

1. 自定义注解
2. 配置类使用注解
3. 条件实现类中修改方法实现

(3) com.itheima.annotation下自定义注解：

```
package com.itheima.annotation;  
  
import com.itheima.condition.OnClassCondition;  
import org.springframework.context.annotation.Conditional;  
  
import java.lang.annotation.*;  
  
/**  
 * 描述  
 * @author ljh  
 * @package com.itheima.annotation  
 * @version 1.0  
 * @date 2020/3/2  
 */
```

```

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(OnClassCondition.class)
public @interface ConditionalOnClass {
    /**
     * 指定所有的类全路径的字符数组
     * @return
     */
    String[] name() default {};
}

```

(4)修改配置类

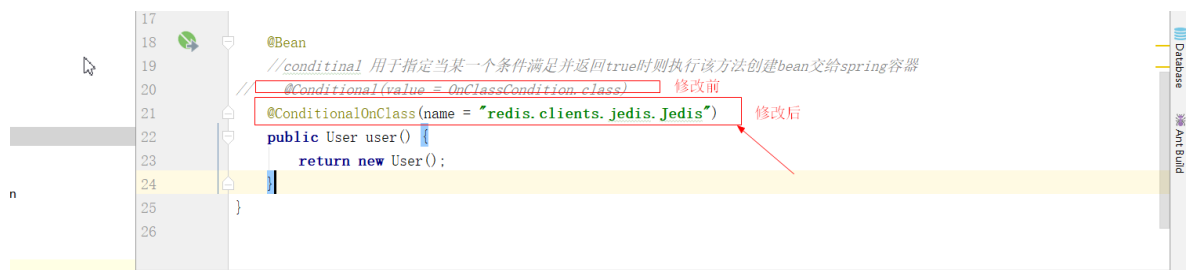
```

@Configuration
public class UserConfig {

    @Bean
    //conditional 用于指定当某一个条件满足并返回true时则执行该方法创建bean交给spring容器
    // @Conditional(value = OnClassCondition.class)
    @ConditionalOnClass(name = "redis.clients.jedis.Jedis")
    public User user() {
        return new User();
    }
}

```

如下图:



(5) 修改实现类

```

public class OnClassCondition implements Condition {
    /**
     * 返回true 则满足条件 返回false 则不满足条件
     *
     * @param context 上下文信息对象 可以获取环境的信息 和容器工程 和类加载器对象
     * @param metadata 注解的元数据 获取注解的属性信息
     * @return
     */
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

        //1. 获取当前的redis的类字节码对象
        /*try {
            //2. 加载成功则说明存在 redis的依赖 返回true,
            Class.forName("redis.clients.jedis.Jedis");

```



```

        return true;
    } catch (ClassNotFoundException e) {
        // 如果加载不成功则redis依赖不存在 返回false
        e.printStackTrace();
        return false;
    }*/
    //获取注解的信息
    Map<String, Object> annotationAttributes =
metadata.getAnnotationAttributes(ConditionalOnClass.class.getName());
    //获取注解中的name的方法的数据值
    String[] values = (String[]) annotationAttributes.get("name");
    for (String value : values) {
        try {
            Class.forName(value);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            return false;
        }
    }
    return true;
}
}
}

```

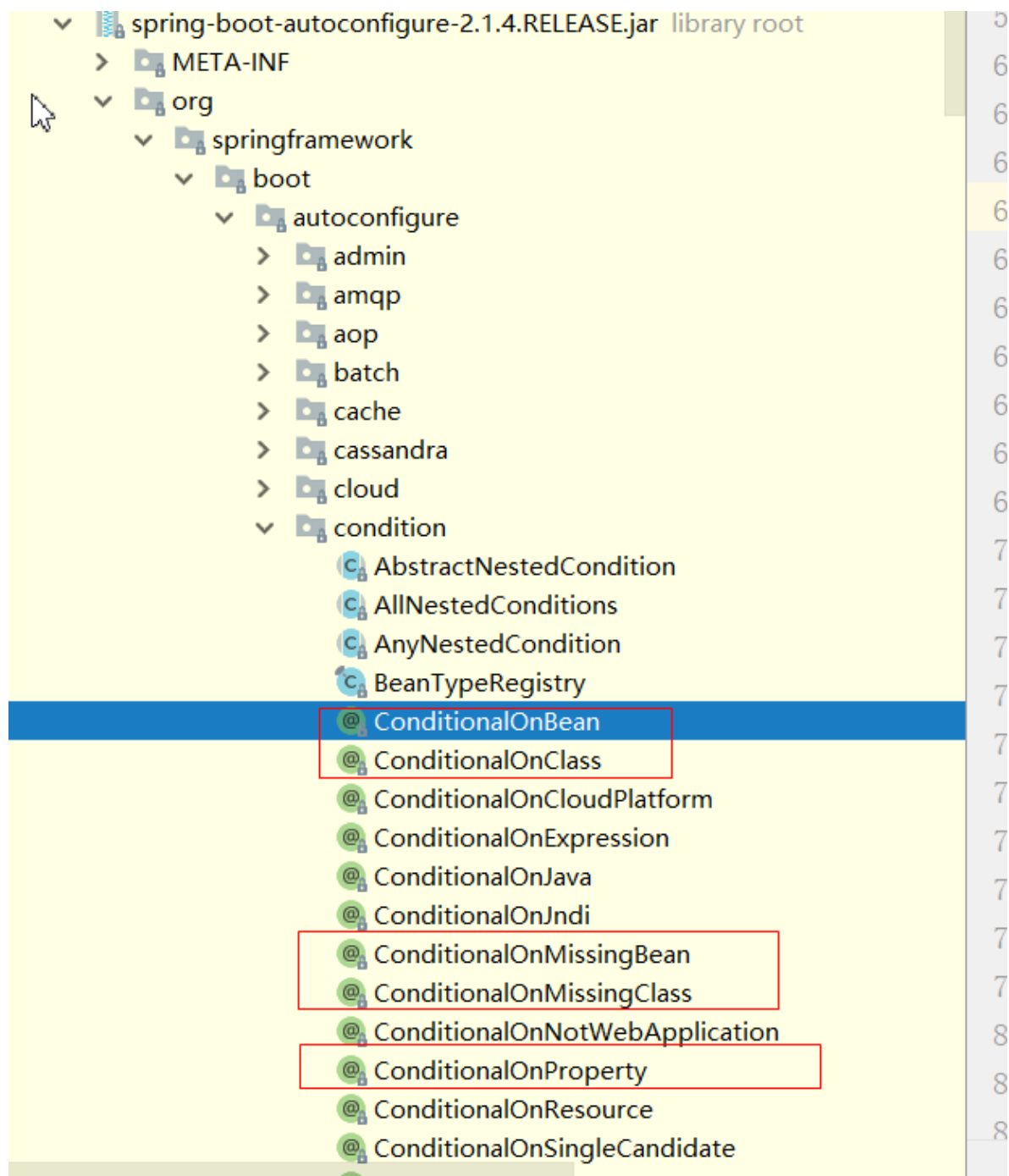
1.1.3 相关的条件的注解说明

常用的注解如下：

```

ConditionalOnBean    当spring容器中有某一个bean时使用
ConditionalOnClass   当判断当前类路径下有某一个类时使用
ConditionalOnMissingBean 当spring容器中没有某一个bean时才使用
ConditionalOnMissingClass 当当前类路径下没有某一个类的时候才使用
ConditionalOnProperty 当配置文件中有一个key value的时候才使用
....

```



1.1.4 小结

condition 用于自定义某一写条件类，用于当达到某一个条件时使用。关联的注解为@conditional结合起来使用。当然我们springboot本身已经提供了一系列的注解供我们使用。如上图所示。

1.2 切换内置的web容器

我们知道在springboot启动的时候如果我们使用web起步依赖，那么我们默认就加载了tomcat的类嵌入了tomcat了，不需要额外再找tomcat。

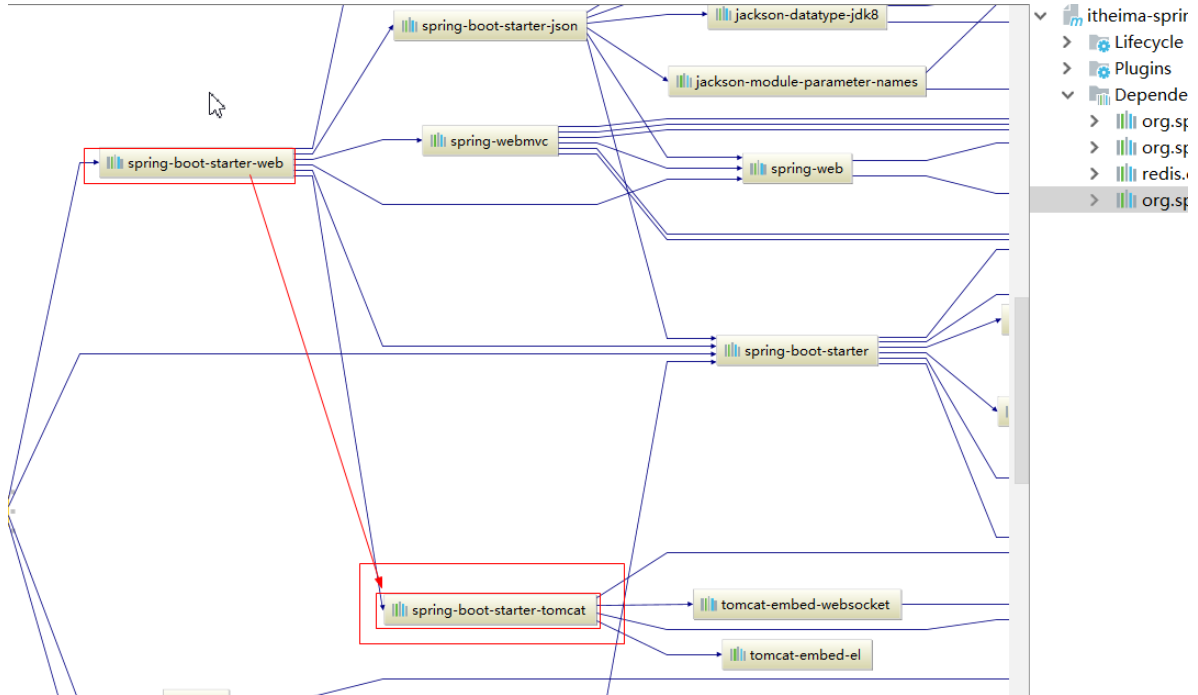
加载配置tomcat的原理：

- (1) 加入pom.xml中起步依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

(2)查看依赖图

web起步依赖依赖于spring-boot-starter-tomcat，这个为嵌入式的tomcat的包。



(3) 自动配置类说明:

Maven: org.springframework.boot:spring-boot-autoconfigure:2.1.4.RELEASE
spring-boot-autoconfigure-2.1.4.RELEASE.jar library root

- mail
- mongo
- mustache
- orm
- quartz
- reactor
- security
- sendgrid
- session
- solr
- task
- template
- thymeleaf
- transaction
- validation
- web
 - client
 - embedded
 - EmbeddedWebServerFactoryCustomizerAutoConfiguration
 - JettyWebServerFactoryCustomizer
 - NettyWebServerFactoryCustomizer
 - TomcatWebServerFactoryCustomizer
 - UndertowWebServerFactoryCustomizer
- format
- reactive

```

45  @ConditionalOnWebApplication  当前为web环境时加载配置
46  @EnableConfigurationProperties(ServerProperties.class)
47  public class EmbeddedWebServerFactoryCustomizerAutoConfiguration {
48
49      /**
50       * Nested configuration if Tomcat is being used.
51       */
52      @Configuration  当前类路径下有tomcat类时进行初始化创建配置使用
53      @ConditionalOnClass({ Tomcat.class, UpgradeProtocol.class })
54      public static class TomcatWebServerFactoryCustomizerConfiguration {
55
56          @Bean
57          public TomcatWebServerFactoryCustomizer tomcatWebServerFactoryCus
58              Environment environment, ServerProperties serverProperties
59              return new TomcatWebServerFactoryCustomizer(environment, serv
60      }
61
62  }
63

```

以上如图所示:

web容器有4种类型:

- + tomcat容器
- + jetty
- + netty
- + undertow

默认spring-boot-starter-web加入的是tomcat，所以根据上图配置，会配置tomcat作为web容器

启动时如下：

```
2 20:32:27.972 INFO 1784 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data repositories in DEFAULT mode.
2 20:32:27.999 INFO 1784 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 15ms. Found 0 repository int
2 20:32:28.519 INFO 1784 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2 20:32:28.553 INFO 1784 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2 20:32:28.554 INFO 1784 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.17]
2 20:32:28.686 INFO 1784 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2 20:32:28.687 INFO 1784 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1331 ms
2 20:32:28.953 INFO 1784 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2 20:32:29.447 INFO 1784 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path '/'
2 20:32:29.449 INFO 1784 --- [main] com.itheima.MySpringBootApplication : Started MySpringBootApplication in 2.606 seconds (JVM running for 3.516)
ma.pojo.User@771d1ffb
```

(4)可以尝试修改web容器：

如上，我们可以通过修改web容器，根据业务需求使用性能更优越的等等其他的web容器。这里我们演示使用jetty作为web容器。

在pom.xml中排出tomcat依赖，添加jetty依赖即可：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <artifactId>spring-boot-starter-tomcat</artifactId>
      <groupId>org.springframework.boot</groupId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

再次启动如下图所示：

```
org.eclipse.jetty.server.session : node0 Scavenging every 600000ms
o.e.jetty.server.handler.ContextHandler : Started o.s.b.w.e.j.JettyEmbeddedWebAppContext@181d7f28{application/, [file
org.eclipse.jetty.server.Server : Started @2695ms
o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
o.e.j.s.h.ContextHandler.application : Initializing Spring DispatcherServlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Completed initialization in 7 ms
o.e.jetty.server.AbstractConnector : Started ServerConnector@3d8b319e{HTTP/1.1,[http/1.1]}{0.0.0.0:8080}
o.s.b.web.embedded.jetty.JettyWebServer : Jetty started on port(s) 8080 (http/1.1) with context path '/'
com.itheima.MySpringBootApplication : Started MySpringBootApplication in 2.458 seconds (JVM running for 3.412)
```

1.3 @Enable*类型的注解说明

在我们使用的代码当中，其中启动类中有一个注解：

@SpringBootApplication注解里面有@EnableAutoConfiguration，那么这种@Enable*开头就是springboot中定义的一些动态启用某些功能的注解，他的底层实现原理实际用的就是@import注解导入一些配置，自动进行配置，加载Bean。

那么我们自己定义写Eanble相关的注解来学习下这种注解所实现的功能，以下 来实现相关功能。

1.3.1 @SpringbootConfiguration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 {
```

如上图所示，就是该注解实际上是在启动类上的注解中的一个注解，我们再点击进去：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {

}
```

我们发现其实该注解就是一个@configuration注解，那么意味着我们的启动类被注解修饰后，意味着它本身也是一个配置类，该配置类就可以当做spring中的applicationContext.xml的文件，用于加载配置使用。

1.3.2 @ComponentScan注解

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM,
        classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {

    /**
```

如上图，在启动类的注解@springbootapplication注解里面又修饰了@compnetScan注解，该注解的作用用于组件扫描包类似于xml中的context-componet-scan，如果不指定扫描路径，那么就扫描该注解修饰的启动类所在的包以及子包。这就是为什么我们在第一天的时候写了controller 并没有扫描也能使用的原因。

1.3.3 实现加载第三方的Bean

需求：

1. 定义两个工程demo2 demo3 demo3中有bean
2. demo2依赖了demo3
3. 我们希望demo2直接获取加载demo3中的bean

(1) 定义工程: demo2

pom.xml:只加入springbootstarter

```
<?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>com.itheima</groupId>
    <artifactId>itheima-springboot-demo02-enable</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.4.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

(2)在包com.itheima下定义demo2启动类，并加载第三方的依赖中的bean

```
package com.itheima;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

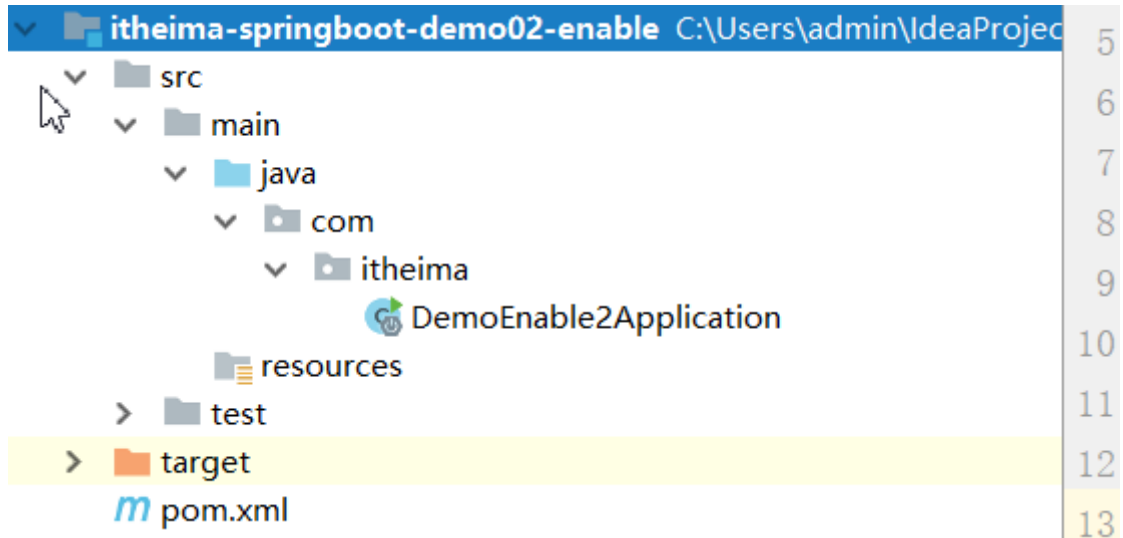
@SpringBootApplication
public class DemoEnable2Application {
```

```

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
SpringApplication.run(DemoEnable2Application.class, args);
        //获取加载第三方的依赖中的bean
        Object user = context.getBean("user");
        System.out.println(user);
    }
}

```

工程结构如下:



(3)定义工程demo3

pom.xml:

```

<?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>com.itheima</groupId>
    <artifactId>itheima-springboot-demo03-enable</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.4.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
    </dependencies>

</project>

```

(4) 在demo3工程中定义配置类和POJO

pojo:在com.itheima.pojo下创建

```
public class User {  
  
}
```

配置类: **注意**, 在com.config下创建配置类:

```
@Configuration  
public class UserConfig {  
  
    @Bean  
    public User user(){  
        return new User();  
    }  
}
```

(5)修改demo2工程的pom.xml:加入demo3的依赖如下图所示:

```
<dependencies>  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter</artifactId>  
    </dependency>  
    <!-- 依赖于 -->  
    <dependency>  
        <groupId>com.itheima</groupId>  
        <artifactId>itheima-springboot-demo03-enable</artifactId>  
        <version>1.0-SNAPSHOT</version>  
    </dependency>  
</dependencies>  
  
<build>
```

加入该依赖

(6) 启动测试: 发现报错:

```
Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'user' available  
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanDefinition(DefaultListableBeanFactory.java:775)  
    at org.springframework.beans.factory.support.AbstractBeanFactory.getMergedLocalBeanDefinition(AbstractBeanFactory.java:1221)  
    at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:294)  
    at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:199)  
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1105)  
    at com.itheima.DemoEnable2Application.main(DemoEnable2Application.java:19)
```

(7) 解决该错误的方式:

- 1.第一种使用组件扫描 扫描包路径放大
- 2.第二种使用import注解进行导入配置类的方式即可

```
@SpringBootApplication
@ComponentScan("com")
@Import(UserConfig.class)
public class DemoEnable2Application {
    public static void main(String[] args) {
        ConfigurableApplicationContext context = SpringApplication.run(DemoEnable2Application.class, args);
        //获取加载第三方的依赖中的bean
        Object user = context.getBean(name: "user");
        System.out.println(user);
    }
}
```

```
C:\Program Files\Java\jdk1.8.0_144\bin\java" ...  
  
 _   _      _       _           _          _         _  
/\ / __' -__-_-(_)-__--_\ \\\ \  
( )\_ | '_|'_||'_V_ |\ \\\  
\W _)|_| || || || |( | ))))  
' |_|. |_|_|_|_|_| \ , // //  
=====|_|=====|_/=//_/_/  
  
:: Spring Boot ::                (v2.1.4.RELEASE)  
  
2020-03-02 21:37:09.406 INFO 15928 --- [main] com.itheima.DemoEnable2Application  
2020-03-02 21:37:09.410 INFO 15928 --- [main] com.itheima.DemoEnable2Application  
2020-03-02 21:37:10.822 INFO 15928 --- [main] com.itheima.DemoEnable2Application  
com.itheima.pojo.User@79c3f01f
```

Process finished with exit code 0

1.3.4 实现优化加载第三bean

上一节中我们使用了import和componenet扫描的方式，都可以解决问题，但是这两种方式相对要麻烦一些，不那么优雅，而且类多那么容易编写麻烦，能否有一种一个注解就能搞定，一看就明白的方式呢？答案是肯定的。

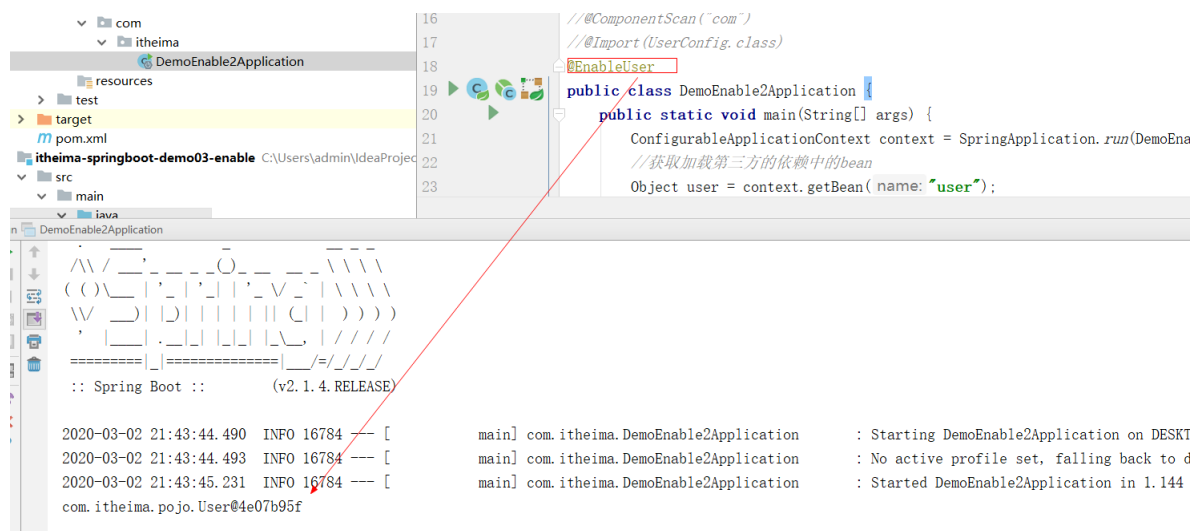
- (1) 在demo03中com.config下创建一个自定义注解@EnableUser:

```
package com.config;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(UserConfig.class)
public @interface EnableUser {

}
```

- (2) 在demo2中使用该注解即可



如上一目了然，当然这里面用的功能点不在于自定义的注解，而在于import的注解。

1.3.5 @import注解

import注解用于导入其他的配置，让spring容器进行加载和初始化。import的注解有以下几种方式使用：

- 直接导入Bean
- 导入配置类
- 导入ImportSelector的实现类，通常用于加载配置文件中的Bean
- 导入ImportBeanDefinitionRegistrar实现类

如上直接导入Bean和导入配置类这个比较简单，我们不再说明，其实上一节我们已经用过了。

1.3.5.1 使用ImportSector实现类方式

在demo3工程中定义类

```

package com.config;

public class MyImportSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        // 返回要注册到spring容器中的Bean的全路径
        return new String[]{"com.itheima.pojo.Role", "com.itheima.pojo.User"};
    }
}

```

定义POJO:

```

package com.itheima.pojo;

public class Role {

}

```

在demo2中修改导入：



打印:

```
2020-03-02 21:58:08.582 INFO 7716 --- [           main] com.itheima.DemoE
com.itheima.pojo.User@2b9ed6da
com.itheima.pojo.Role@6c61a903
```

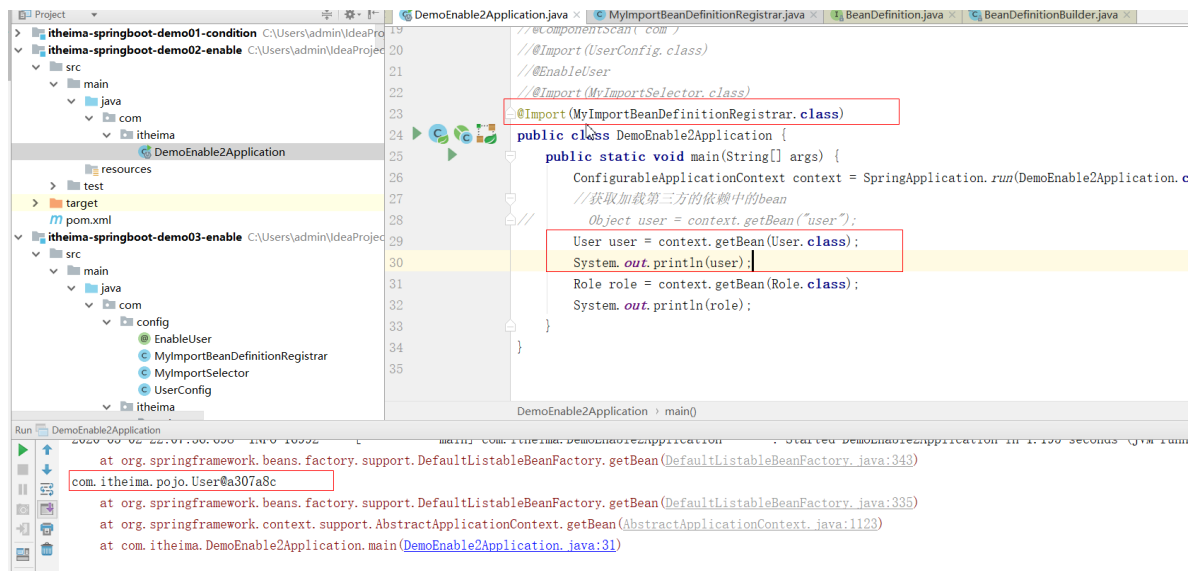
1.3.5.2 使用ImportBeanDefinitionRegistrar实现类方式

(1) 在demo3下定义实现类

```
package com.config;
import com.itheima.pojo.User;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.context.annotation.ImportBeanDefinitionRegistrar;
import org.springframework.core.type.AnnotationMetadata;

public class MyImportBeanDefinitionRegistrar implements
ImportBeanDefinitionRegistrar {
    @Override
    public void registerBeanDefinitions(AnnotationMetadata
importingClassMetadata, BeanDefinitionRegistry registry) {
        //创建BeanDefinition
        AbstractBeanDefinition beanDefinition =
BeanDefinitionBuilder.rootBeanDefinition(User.class).getBeanDefinition();
        //注册bean 目前只注册User
        registry.registerBeanDefinition("user", beanDefinition);
    }
}
```

(2) 修改demo2的启动类配置:



```

@SpringBootApplication
// @ComponentScan("com")
// @Import(UserConfig.class)
// @EnableUser
// @Import(MyImportSelector.class)
// @Import(MyImportBeanDefinitionRegistrar.class)
public class DemoEnable2Application {
    public static void main(String[] args) {
        ConfigurableApplicationContext context =
        SpringApplication.run(DemoEnable2Application.class, args);
        // 获取加载第三方的依赖中的bean
        Object user = context.getBean("user");
        User user = context.getBean(User.class);
        System.out.println(user);
        Role role = context.getBean(Role.class);
        System.out.println(role);
    }
}

```

如上图所示也能成功，针对User成功，针对Role失败，这里我们只加载了User。

1.4 @EnableAutoConfiguration

自动配置流程：

1. @import注解 导入配置
2. selectImports导入类中的方法中加载配置返回Bean定义的字符数组
3. 加载META-INF/spring.factories 中获取Bean定义的全路径名返回
4. 最终返回回去即可

流程截图：

```

SpringBootApplication.java x EnableAutoConfiguration.java x AutoConfigurationImport
76 @Retention(RetentionPolicy. RUNTIME)
77 @Documented
78 @Inherited
79 @AutoConfigurationPackage
80 @Import(AutoConfigurationImportSelector.class)
81 public @interface EnableAutoConfiguration {
82
83     String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enablea
84
85     /**
86      * Exclude specific auto-configuration classes such that the
87      * @return the classes to exclude
88      */
89     Class<?>[] exclude() default {};

```

```

public class AutoConfigurationImportSelector
    implements DeferredImportSelector, BeanClassLoaderAware, ResourceLoaderAware,
        BeanFactoryAware, EnvironmentAware, Ordered {

    private static final AutoConfigurationEntry EMPTY_ENTRY = new AutoConfigurationEntry();

    private static final String[] NO_IMPORTS = {};

    private static final Log logger = LogFactory
        .getLog(AutoConfigurationImportSelector.class);

    private static final String PROPERTY_NAME_AUTOCONFIGURE_EXCLUDE = "spring.autoconfigure.exclude";

    private ConfigurableListableBeanFactory beanFactory;

    private Environment environment;

    private ClassLoader beanClassLoader;

    private ResourceLoader resourceLoader;

    @Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        if (!isEnabled(annotationMetadata)) {
            return NO_IMPORTS;

```

```

@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }

    AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
        .loadMetadata(this.beanClassLoader);
    AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(
        autoConfigurationMetadata, annotationMetadata);
    return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}

```

```

protected AutoConfigurationEntry getAutoConfigurationEntry(
    AutoConfigurationMetadata autoConfigurationMetadata,
    AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }

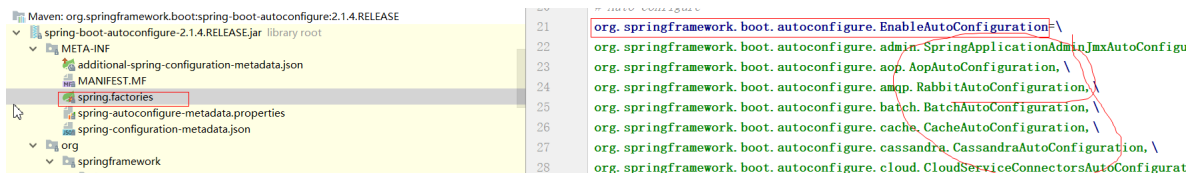
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    List<String> configurations = getCandidateConfigurations(annotationMetadata,
        attributes);
    configurations = removeDuplicates(configurations);
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    configurations.removeAll(exclusions);
    configurations = filter(configurations, autoConfigurationMetadata);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return new AutoConfigurationEntry(configurations, exclusions);
}

```

```

protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
    AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
        getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
    Assert.notEmpty(configurations,
        message: "No auto configuration classes found in META-INF/spring.factories. If you "
            + "are using a custom packaging, make sure that file is correct.");
    return configurations;
}

```



其中就有：redis的自动配置类，redis的自动配置我们在说condition的时候已经说过只要加入依赖则配置交到spring容器中自动配置了。

```

org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfiguration, \
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration, \
org.springframework.boot.autoconfigure.data.redis.RedisReplicatingAutoConfiguration, \

```

2.Springboot自动配置 自定义starter

以上我们学习了springboot的自动配置原理，那么我们通过一个案例来强化我们的学习。

2.1 需求说明

当加入redis客户端的坐标的时候，自动配置jedis的bean 加载到spring容器中。

2.2 实现步骤

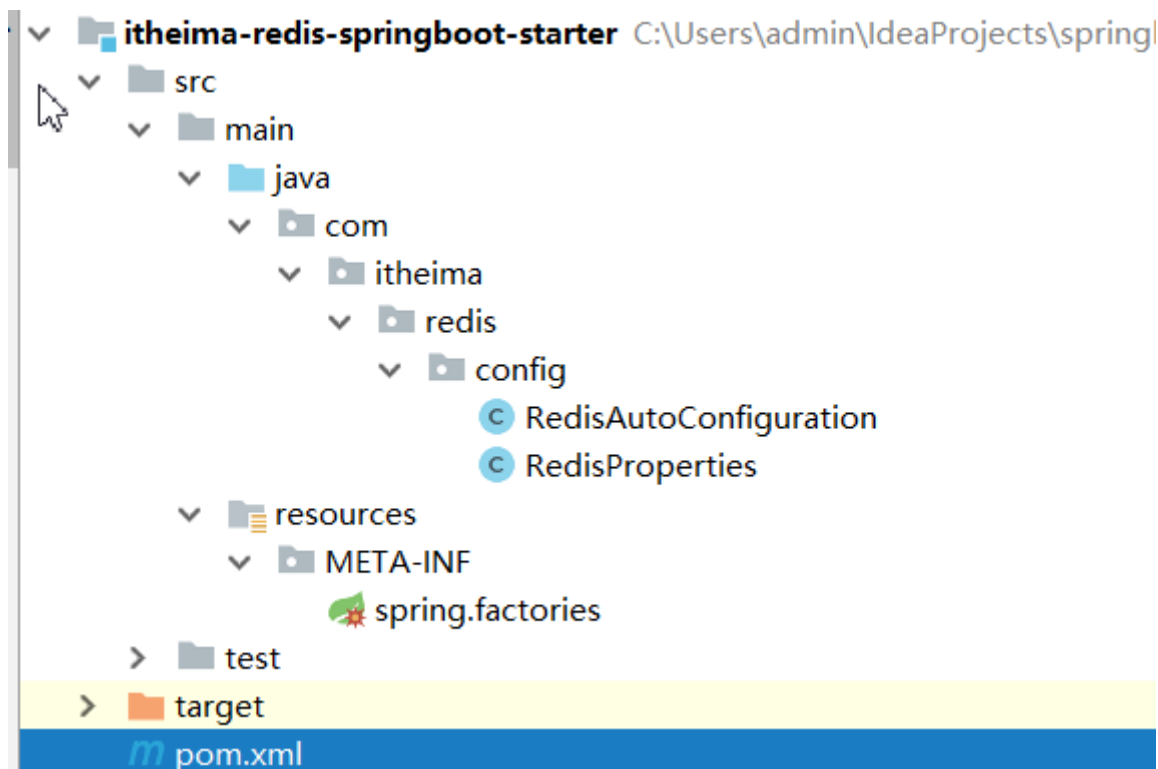
我们可以参考springboot整合mybatis的依赖进行配置实现。为了简单起见我们只定义一个工程即可。

- 1.创建工程 `itheima-redis-springboot-starter` 用作起步依赖
- 2.添加依赖
- 3.创建自动配置类和POJO
- 4.创建工程 `itheima-test-starter` 用于测试使用

(1) 创建工程itheima-redis-springboot-starter

该工程创建不需要启动类，不需要测试类，只需要spring-boot-starter以及jedis的依赖坐标。

项目结构如下：



```
<?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>com.itheima</groupId>
  <artifactId>itheima-redis-springboot-starter</artifactId>
  <version>1.0-SNAPSHOT</version>

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

```

        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.4.RELEASE</version>
    </parent>

    <dependencies>
        <!--springboot的starter-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <!--redis的依赖jedis-->
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
            <version>3.2.0</version>
        </dependency>
    </dependencies>

</project>

```

(2) 创建自动配置类

```

@Configuration
@EnableConfigurationProperties(RedisProperties.class)
@ConditionalOnClass(Jedis.class)
public class RedisAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean(name = "jedis")
    public Jedis jedis(RedisProperties redisProperties) {
        System.out.println("哈哈哈哈哈=====" + redisProperties.getHost() + ":" +
redisProperties.getPort());
        return new Jedis(redisProperties.getHost(), redisProperties.getPort());
    }
}

```

(3) 创建POJO

```

@ConfigurationProperties(prefix = "redis")
public class RedisProperties {
    private String host = "localhost"; //给与默认值
    private Integer port = 6379; //给与默认值

    public String getHost() {
        return host;
    }

    public void setHost(String host) {
        this.host = host;
    }

    public Integer getPort() {
        return port;
    }
}

```



```
public void setPort(Integer port) {  
    this.port = port;  
}  
}
```

(4)在resources下创建META-INF/spring.factories文件并定义内容如下:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
com.itheima.redis.config.RedisAutoConfiguration
```

(5)创建测试工程itheima-test-starter, 添加依赖

```
<?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>com.itheima</groupId>  
    <artifactId>itheima-test-starter</artifactId>  
    <version>1.0-SNAPSHOT</version>  
  
    <parent>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-parent</artifactId>  
        <version>2.1.4.RELEASE</version>  
    </parent>  
  
    <dependencies>  
        <!--springboot的起步依赖-->  
        <dependency>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-starter</artifactId>  
        </dependency>  
        <!--加入itheima的redis的起步依赖-->  
        <dependency>  
            <groupId>com.itheima</groupId>  
            <artifactId>itheima-redis-springboot-starter</artifactId>  
            <version>1.0-SNAPSHOT</version>  
        </dependency>  
    </dependencies>  
</project>
```



(2)定义启动类

```
package com.itheima;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.Bean;
import redis.clients.jedis.Jedis;

/**
 * 描述
 * @author ljh
 * @package com.itheima
 * @version 1.0
 * @date 2020/3/3
 */
@SpringBootApplication
public class ItheimaRedisTestApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext =
        SpringApplication.run(ItheimaRedisTestApplication.class, args);
        Object jedis = applicationContext.getBean("jedis");
        System.out.println(jedis);
    }

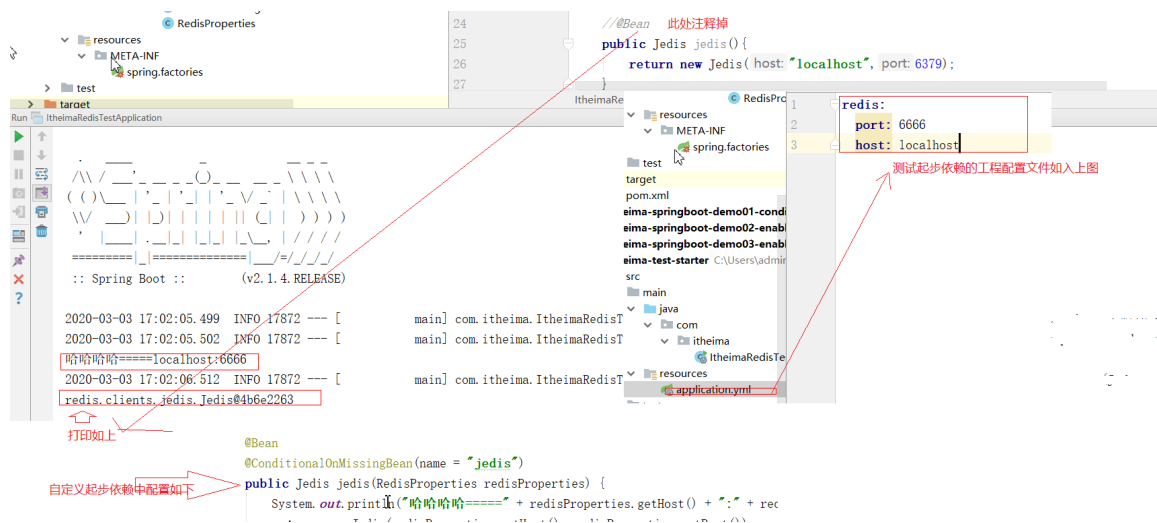
    //@Bean
    public Jedis jedis(){
        return new Jedis("localhost", 6379);
    }
}
```

(3)配置application.yml:

```
redis:
  port: 6666
  host: localhost
```

(4)测试:

(1) 注释掉则出现如下结果



(2) 不注释掉出现如下结果:

```
2020-03-03 17:06:02.850 INFO 13480 --- [main] co
2020-03-03 17:06:03.838 INFO 13480 --- [main] co
redis.clients.jedis.Jedis@7803bfd
```

3 SpringBoot的监控

时常我们在使用的项目的时候, 想知道相关项目的一些参数和调用状态, 而SpringBoot自带监控功能Actuator, 可以帮助实现对程序内部运行情况监控, 比如监控状况、Bean加载情况、配置属性、日志信息等。

3.1 Actuator

Actuator是springboot自带的组件可以用来进行监控, Bean加载情况、环境变量、日志信息、线程信息等等, 使用简单

3.1.1 使用actuator

(1)操作步骤:

1. 创建springboot工程
2. 添加Actuator的起步依赖
3. 配置开启端点和相关配置项
4. 通过端点路径查看信息

(2)创建工程, pom.xml,

注意主要需要添加依赖如下：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

总体如下：

```
<?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>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.itheima</groupId>
  <artifactId>itheima-actuator</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>itheima-actuator</name>
  <description>Demo project for Spring Boot</description>

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

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>

  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
```

```

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

</project>

```

(3)编写启动类:

```

@SpringBootApplication
public class ItheimaActuatorApplication {

    public static void main(String[] args) {
        SpringApplication.run(ItheimaActuatorApplication.class, args);
    }

    @RestController
    @RequestMapping("/test")
    class TestController {

        @GetMapping("/index")
        public String show() {
            return "hello world";
        }
    }
}

```

(4) 配置application.properties:

```

# 配置健康端点开启所有详情信息
management.endpoint.health.show-details=always
# 设置开放所有web相关的端点信息
management.endpoints.web.exposure.include=*
# 设置info前缀的信息设置
info.name=zhangsan
info.age=18

```

(5)在浏览器输入 地址: `http://localhost:8080/actuator`

```

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     "beans-actuator": {

```

显示如上的信息，就可以看到相关的路径，这些路径分表代表不同的信息的含义。

3.1.2 监控路径列表说明

以下展示部分列表

路径	描述
/beans	描述应用程序上下文里全部的Bean，以及它们的关系
/env	获取全部环境属性
/env/{name}	根据名称获取特定的环境属性值
/health	报告应用程序的健康指标，这些值由HealthIndicator的实现类提供
/info	获取应用程序的定制信息，这些信息由info打头的属性提供
/mappings	描述全部的URI路径，以及它们和控制器(包含Actuator端点)的映射关系
/metrics	报告各种应用程序度量信息，比如内存用量和HTTP请求计数
/metrics/{name}	报告指定名称的应用程序度量值
/trace	提供基本的HTTP请求跟踪信息(时间戳、HTTP头等)

3.2 SpringBoot admin

如果使用actuator使用起来比较费劲，没有数据直观感受。我们可以通过插件来展示。

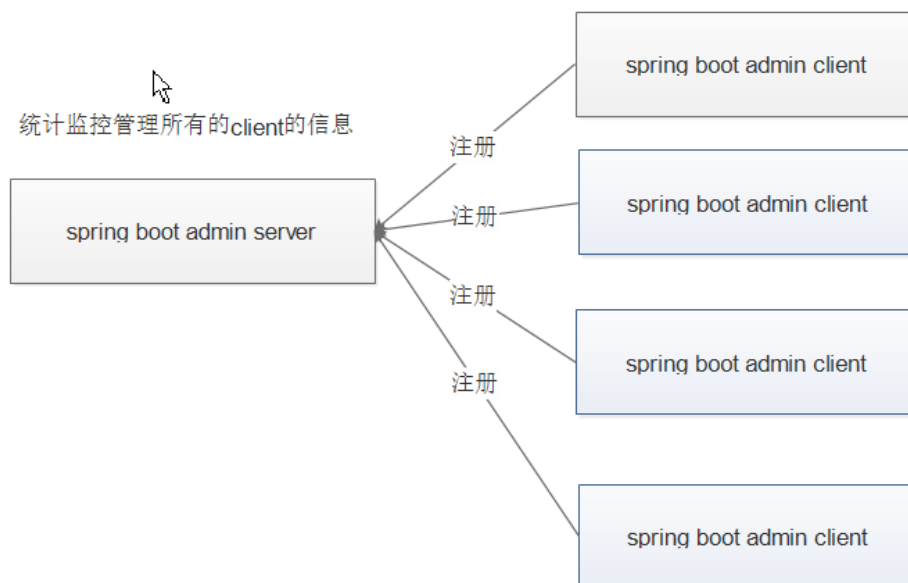
3.2.1 介绍

- Spring Boot Admin是一个开源社区项目，用于管理和监控SpringBoot应用程序。
- Spring Boot Admin 有两个角色，客户端(Client)和服务端(Server)。
- 应用程序作为Spring Boot Admin Client向为Spring Boot Admin Server注册
- Spring Boot Admin Server 通过图形化界面方式展示Spring Boot Admin Client的监控信息。

3.2.2 使用

spring boot admin的架构角色

- admin server 用于收集统计所有相关client的注册过来的信息进行汇总展示
- admin client 每一个springboot工程都是一个client 相关的功能展示需要汇总到注册汇总到server



(1) 创建admin server 工程 itheima-admin-server

可使用spring initianzer来创建：

```
<?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
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.13.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.itheima</groupId>
  <artifactId>ithima-admin-server</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>ithima-admin-server</name>
```

```

<description>Demo project for Spring Boot</description>

<properties>
  <java.version>1.8</java.version>
  <spring-boot-admin.version>2.1.6</spring-boot-admin.version>
</properties>

<dependencies>
  <dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>de.codecentric</groupId>
      <artifactId>spring-boot-admin-dependencies</artifactId>
      <version>${spring-boot-admin.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

</project>

```

(2)创建启动类


```

@SpringBootApplication
@EnableAdminServer
public class IthimaAdminServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(IthimaAdminServerApplication.class, args);
    }

}

```

注意:

@EnableAdminServer 该注解用于启用Server功能。

(3)修改application.properties文件

```
server.port=9000
```

(4)创建admin client 工程 itheima-admin-client

```

<?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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.13.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.itheima</groupId>
    <artifactId>ithima-admin-client</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>ithima-admin-client</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
        <spring-boot-admin.version>2.1.6</spring-boot-admin.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>de.codecentric</groupId>
            <artifactId>spring-boot-admin-starter-client</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>

```

```

        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>de.codecentric</groupId>
            <artifactId>spring-boot-admin-dependencies</artifactId>
            <version>${spring-boot-admin.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

</project>

```

(5) 创建启动类:

```

@SpringBootApplication
public class IthimaAdminClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(IthimaAdminClientApplication.class, args);
    }

    @RestController
    @RequestMapping("/user")
    class TestController {

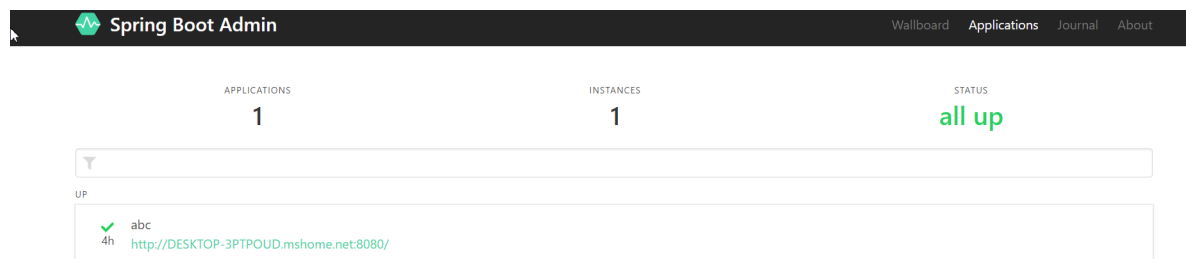
        @RequestMapping("/findAll")
        public String a() {
            return "aaaa";
        }
    }
}

```

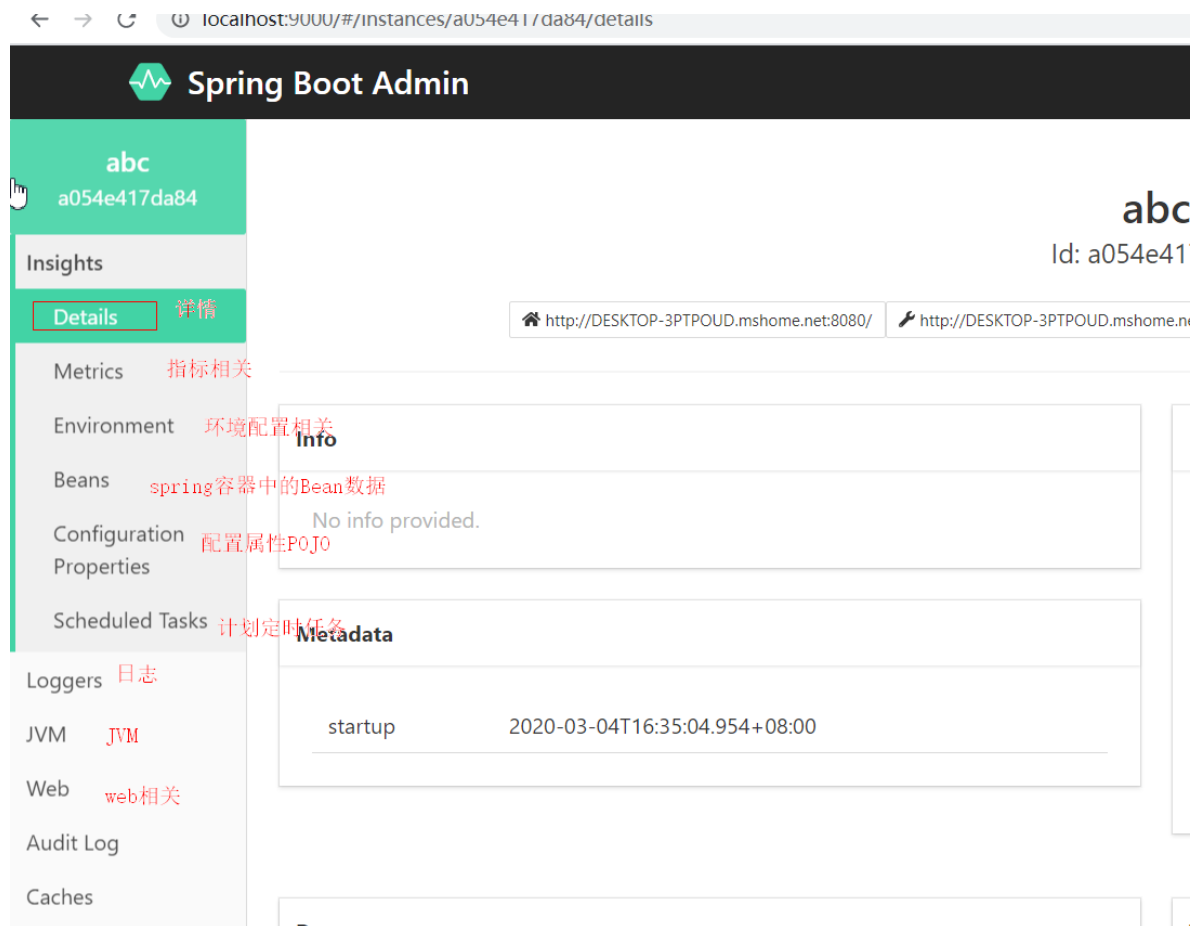
(6)配置application.properties:

```
# 配置注册到的admin server的地址
spring.boot.admin.client.url=http://localhost:9000
# 启用健康检查 默认就是true
management.endpoint.health.enabled=true
# 配置显示所有的监控详情
management.endpoint.health.show-details=always
# 开放所有端点
management.endpoints.web.exposure.include=*
# 设置系统的名称
spring.application.name=abc
```

(7)启动两个系统。访问路径 <http://localhost:9000/>



我们简单认识下：并点击相关界面链接就能看到相关的图形化展示了。



4.SpringBoot部署项目

在springboot项目中，我们部署项目有两种方式：

- jar包直接通过java命令运行执行
- war包存储在tomcat等servlet容器中执行

4.1 jar包部署

(1) 新建项目用于测试 pom.xml

```
<?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>com.itheima</groupId>
    <artifactId>itheima-demo</artifactId>
    <version>1.0-SNAPSHOT</version>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.4.RELEASE</version>
    </parent>

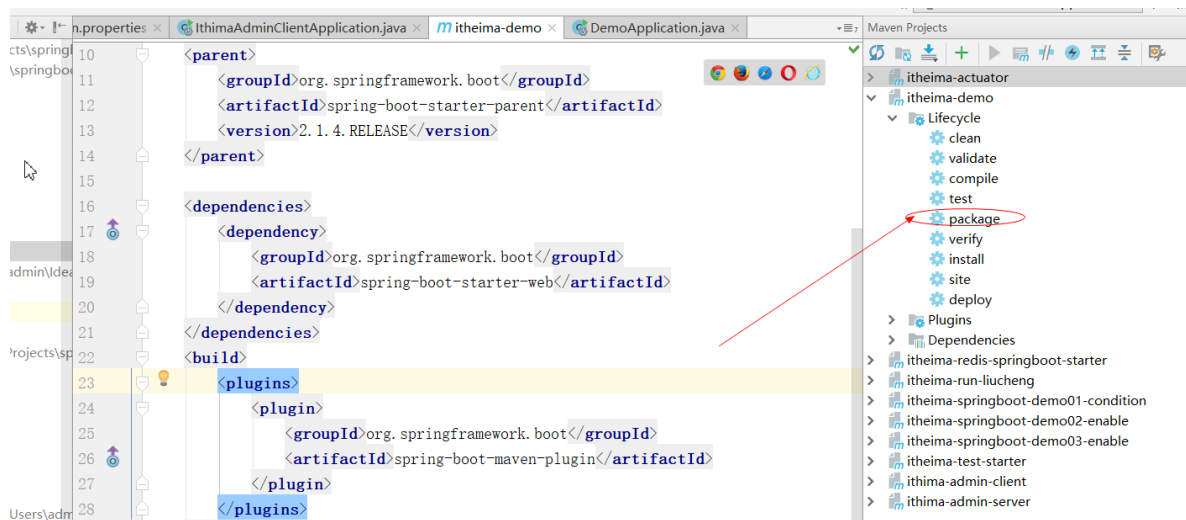
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

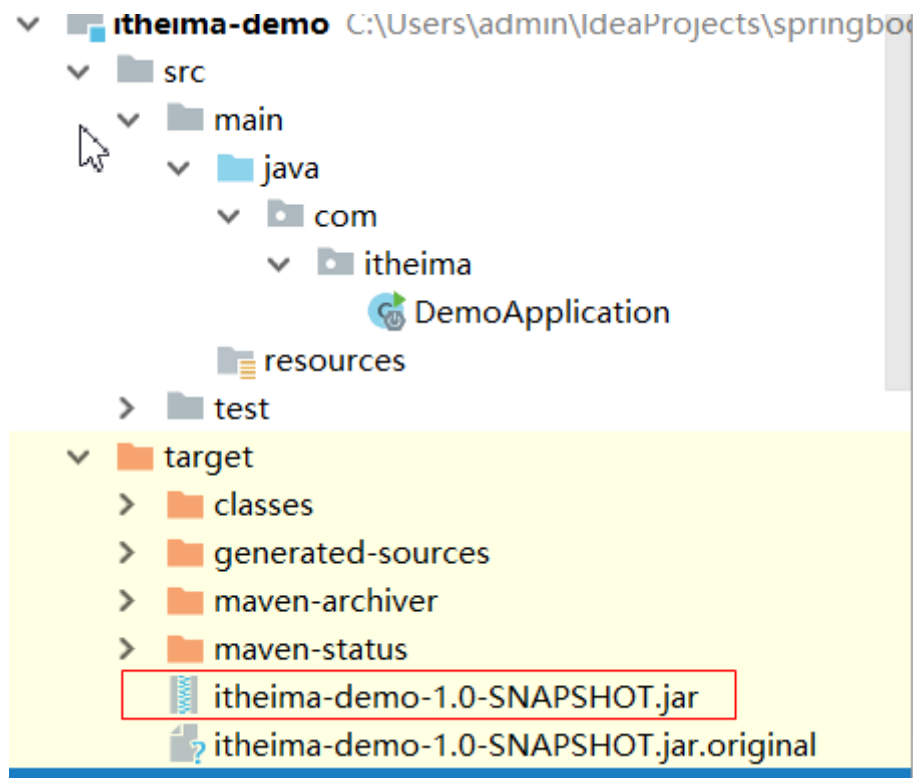
(2)定义启动类：

```
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

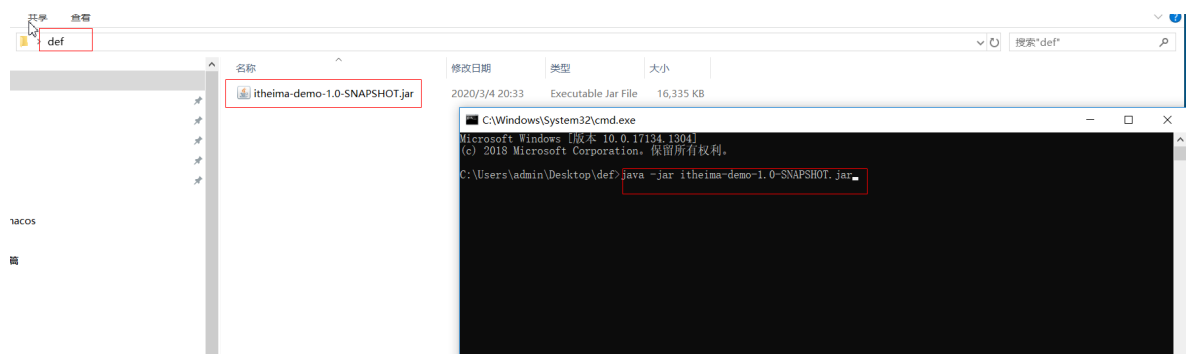
(3)执行如下图命令即可



(4) copy jar包到任意目录，可以直接执行java命令执行系统



执行命令：如下图



```
java -jar itheima-demo-1.0-SNAPSHOT.jar
```

执行效果如下：

```
2020-03-04 20:36:34.653 INFO 14076 --- [main] com.itheima.DemoApplication : Starting DemoApplication v1.0-SNAPSHOT on DESKTOP-PTPOUD with PID 14076 (C:\Users\admin\Desktop\def\itheima-demo-1.0-SNAPSHOT.jar started by admin in C:\Users\admin\Desktop\def)
2020-03-04 20:36:34.656 INFO 14076 --- [main] com.itheima.DemoApplication : No active profile set, falling back to default profiles: default
2020-03-04 20:36:36.358 INFO 14076 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2020-03-04 20:36:36.406 INFO 14076 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-03-04 20:36:36.406 INFO 14076 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.17]
2020-03-04 20:36:37.065 INFO 14076 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-03-04 20:36:37.066 INFO 14076 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 2353 ms
2020-03-04 20:36:37.316 INFO 14076 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-03-04 20:36:37.579 INFO 14076 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2020-03-04 20:36:37.582 INFO 14076 --- [main] com.itheima.DemoApplication : Started DemoApplication in 3.495 seconds (JVM running for 4.313)
```

4.2 war包部署

(1) 首先修改打包方式和修改相关配置依赖

```
<groupId>com.itheima</groupId>
<artifactId>itheima-demo</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>
<parent>
```

```
<?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>com.itheima</groupId>
  <artifactId>itheima-demo</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
  </parent>

  <dependencies>
    <dependency>
      <!-- To build an executable war use one of the profiles below -->
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <exclusions>
```

```

        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
</dependencies>
<build>
    <finalName>demo</finalName>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

</project>

```

(2) 修改启动类配置 需要继承SpringBootServletInitializer

```

@SpringBootApplication
public class DemoApplication extends SpringBootServletInitializer{
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @RestController
    class TestController{

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

```

(3)执行命令 打包 之后变成一个war包

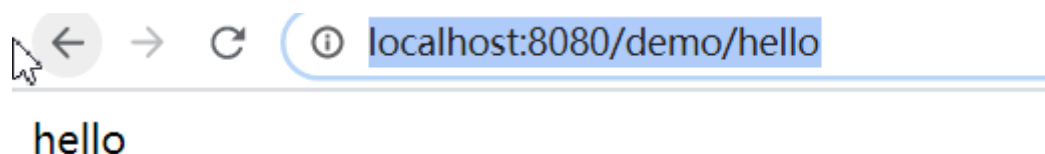


(4) copy该war包到tomcat中

这个tomcat大家自己找一个自己熟悉的，最好是tomcat8.5以上。

> apache-tomcat-8.5.23-windows-x64 > apache-tomcat-8.5.23 > webapps >				
名称	修改日期	类型	大小	
demo	2020/3/4 20:48	文件夹		
docs	2017/9/28 11:31	文件夹		
examples	2017/9/28 11:31	文件夹		
host-manager	2017/9/28 11:31	文件夹		
manager	2017/9/28 11:31	文件夹		
ROOT	2020/2/23 8:32	文件夹		
demo.war	2020/3/4 20:48	WAR 文件	12,716 KB	

(5) 浏览器输入地址测试即可：



小结：

推荐使用jar，特别在微服务领域中使用jar的方式简单许多。

5 Springboot监听机制（了解）

springboot事件监听机制，实际上是对java的事件的封装。

java中定义了以下几个角色：

- 事件 Event
- 事件源 Source
- 监听器 Listener 实现EventListener接口的对象

Springboot在启动的时候，会对几个监听器进行回调，完成初始化的一些操作，我们可以实现这个监听器来实现相关的业务，比如缓存的一些处理。springboot 提供了这些接口，我们只需要实现这些接口就可以在启动的时候进行回调了。

```
ApplicationContextInitializer  
SpringApplicationRunListener  
CommandLineRunner  
ApplicationRunner
```

解释：

`CommandLineRunner` 在容器准备好了之后可以回调 `@componet`修饰即可
`ApplicationRunner` 在容器准备好了之后可以回调 `@componet`修饰即可

`ApplicationContextInitializer`

在spring 在刷新之前 调用该方法 用于：做些初始化工作 通常用于web环境，用于激活配置，web上下文的属性注册。

注意：需要配置META/spring.factories 配置之后才能加载调用

`SpringApplicationRunListener` 也需要配置META/spring.factories 配置之后才能加载调用

他是SpringApplication run方法的监听器，当我们使用SpringApplication调用Run方法的时候触发该监听器回调方法。注意：他需要有一个公共的构造函数，并且每一次RUN的时候都需要重新创建实例

接下来通过跟踪源码SpringApplication.run启动流程来看他们的一些使用。

6.附加说明

springboot初始化的Run流程的链接地址：

<https://www.processon.com/view/link/59812124e4b0de2518b32b6e>