SpringBoot高级之原理分析

一、SpringBoot自动配置--注解说明

1.1、Condition条件判断

1.1.1、创建Condition模块

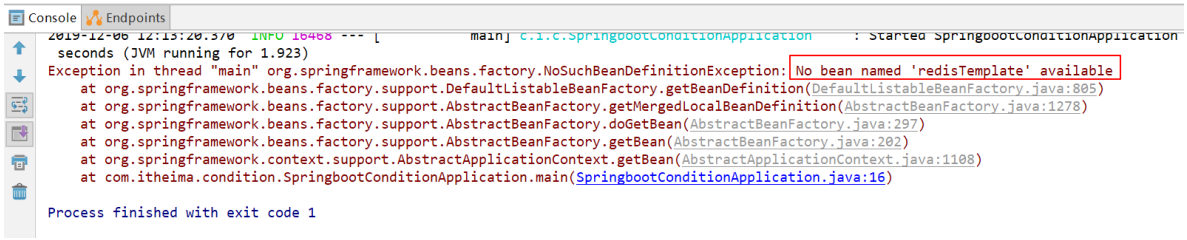
> Condition（条件）：Condition是在Spring4.0增加的条件判断功能，通过这个可以功能可以实现选择性的创建Bean操作。

思考：

> SpringBoot是如何知道要创建哪个Bean的？比如SpringBoot是如何知道要创建RedisTemplate的？

创建一个模块，springboot-condition：

```
@SpringBootApplication
public class SpringbootConditionApplication {

    public static void main(String[] args) {
        //启动springBoot的应用，返回spring的IOC容器
        ConfigurableApplicationContext context =
SpringApplication.run(SpringbootConditionApplication.cla
ss, args);

        //获取一个Bean，RedisTemplate
        Object redis = context.getBean("redisTemplate");
        System.out.println(redis);
    }
}
```

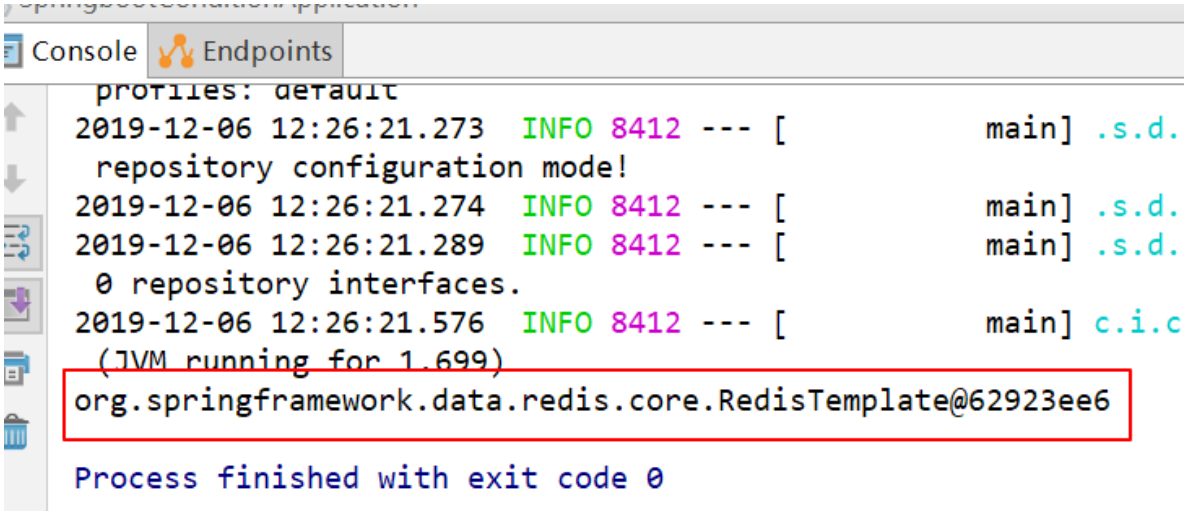


没有引入坐标，所以没有redisTemplate的Bean。

增加redis坐标

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

redisTemplate的Bean已经引入



1.1.2、Condition案例1

需求：

```
在 Spring 的 IOC 容器中有一个 User 的 Bean，现要求：
    1. 导入Jedis坐标后，加载该Bean，没导入，则不加载。
    2. 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。
```

第一步：创建User实体类

```
package com.itheima.condition.domain;

public class User {
}
```

第二步：创建User配置类

```
package com.itheima.condition.config;

import com.itheima.condition.domain.User;
import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration;

@Configuration
public class UserConfig {

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

第三步：修改启动类

SpringbootConditionApplication 中增加user的Bean获取

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

```
2019-12-06 12:29:48.315  INFO 14504 --- [
 seconds (JVM running for 1.454)
com.itheima.condition.domain.User@cdc3aae

Process finished with exit code 0
```

现在是任何情况下都能加载User这个类。

第四步：实现Condition

新建一个ClassCondition类，实现Condition接口里的matches方法来控制类的加载

新建一个类实现Condition接口

```
public class ClassCondition implements Condition {
    @Override
    public boolean matches(ConditionContext
conditionContext, AnnotatedTypeMetadata
annotatedTypeMetadata) {
        return false;
    }
}
```

修改condition.config的UserConfig类：

```
@Bean
@Conditional(ClassCondition.class)
public User user(){
    return new User();
}
```

将user对象加入Bean里的时候增加一个@Conditional注解

测试：当matches返回false时，不加载User类，返回true时，加载User类

第五步：导入Jedis坐标

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
```

第六步：修改matches方法

```
public boolean matches(ConditionContext
conditionContext, AnnotatedTypeMetadata
annotatedTypeMetadata) {
    boolean flag = true;
    try {
        Class<?> aClass =
Class.forName("redis.clients.jedis.Jedis");
    } catch (ClassNotFoundException e) {
        flag = false;
    }
    return flag;
}
```

当Jedis坐标导入后，可以加载到 redis.clients.jedis.Jedis 这个类，未导入时，加载不到redis.clients.jedis.Jedis这个类，所以通过异常捕获可以返回是否加载。

1.1.3、Condition案例2

现在的 Class.forName("redis.clients.jedis.Jedis") 这个是写死的，是否可以动态的加载呢?

第一步：新建注解类

新建ConditionOnClass注解类，增加@Conditional注解

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(ClassCondition.class)
public @interface ConditionOnClass {
    String[] value();
}
```

注解类中增加value变量。

第二步：修改UserConfig

```
@Bean
//@Conditional(ClassCondition.class)
@ConditionOnClass("redis.clients.jedis.Jedis")
public User user(){
    return new User();
}
```

现在新建的注解@ConditionOnClass和原注解@Conditional作用一致

第三步：修改matches方法

```
/**
 * @param conditionContext  上下文对象，用于获取类加载，Bean工厂等信息
 * @param annotatedTypeMetadata  注解的元对象，可以用于获取注解定义的属性值
 * @return
 */
@Override
public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata annotatedTypeMetadata) {
    Map<String, Object> map = annotatedTypeMetadata.getAnnotationAttributes(ConditionOnClass.class.getName());
    //System.out.println(map);
    String[] strings = (String[]) map.get("value");
    boolean flag = true;
    try {
        for (String className : strings) {
            Class<?> aClass = Class.forName(className);
        }
    } catch (ClassNotFoundException e) {
        flag = false;
    }
    return flag;
}
```

此时，通过UserConfig注解注入的类存在就加载User类，如果注入的类不存在就不加载User类

测试，引入fastjson坐标，加载User类

第四步：查看源代码jar包

org.springframework.boot.autoconfigure.condition.ConditionalOnClass

org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration

第五步：判断配置文件

```
@Bean
@ConditionalOnProperty(name = "itcast", havingValue = "itheima")
public User user2(){
    return new User();
}
```

修改启动类

```
Object user = context.getBean("user2");
System.out.println(user);
```

- 在UserConfig类中新增一个方法，同样加载User类
- 使用@ConditionalOnProperty注解，name是itcast，value是itheima
- 修改配置文件application.properties，增加itcast=itheima
- 测试加载User类

1.1.4、Condition小结

- User实体类，UserConfig配置类将User放入Bean工厂，ClassCondition类重写matches方法，ConditionOnClass注解类增加注解。
- **自定义条件**：
    - **自定义条件类**：自定义类实现Condition接口，重写 matches 方法，在 matches 方法中进行逻辑判断，返回boolean值 。matches 方法两个参数：

        - **context**：上下文对象，可以获取属性值，获取类加载器，获取FactoryBean等。
        - **metadata**：元数据对象，用于获取注解属性。
    - 判断条件：在初始化Bean时，使用@Conditional（条件类.class）注解。

- **SpringBoot常用条件注解**：
    - **ConditionalOnProperty**：判断配置文件中是否有对应的属性和值才初始化Bean。
    - **ConditionalOnClass**：判断环境中是否有对应的字节码文件才初始化Bean。
    - **ConditionalOnMissingBean**：判断环境中是否有对应的Bean才初始化Bean。

1.2、SpringBoot切换内置服务器

1.2.1、启用内置web服务器

引入starter-web坐标之后，服务器内置tomcat启动了

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

1.2.2、查看一下源jar包

org.springframework.boot.autoconfigure.web.embedded.TomcatWebServerFactoryCustomizer

修改坐标：

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

```
: Started o.s.b.w.e.j
'Temp/jetty-docbase.169771738408749757.8080/],AVAILABLE}
: Started @1689ms
: Initializing ExecutorService 'applicationTaskExecutor'
: Initializing Spring DispatcherServlet

: Initializing Servlet 'dispatcherServlet'
: Completed initialization in 3 ms
: Started ServerConnector@42257bdd{HTTP/1.1,[http/1.1]}{0

: Jetty started on port(s) 8080 (http/1.1) with context

: Started SpringbootConditionApplication in 1.278
```

Jetty服务器启动

1.3、@Enable*注解

SpringBoot中提供了很多Enable开头的注解，这些注解都是用于动态启用某些功能的。而其底层原理是使用@Import注解导入一些配置类，实现Bean的动态加载。

问题：

SpringBoot 工程是否可以直接获取jar包中定义的Bean？

1.3.1、创建两个模块

一个是springboot-enable，一个是springboot-enable-other

1.3.2、创建实体类与配置类

```
package com.itheima.domain;

public class User {
}
```

```
@Configuration
public class UserConfig {

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

1.3.3、引入enable-other坐标

```
<dependency>
    <groupId>com.itheima</groupId>
    <artifactId>springboot-enable-other</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

1.3.4、修改enable启动类

```
public static void main(String[] args) {
    ConfigurableApplicationContext context =
SpringApplication.run(SpringbootEnableApplication.class,
args);

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

Bean里没有User这个类

```
  profiles: default
Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'user' available
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanDefinition(DefaultListableBeanFactory.java:805)
    at org.springframework.beans.factory.support.AbstractBeanFactory.getMergedLocalBeanDefinition(AbstractBeanFactory.java:1278)
    at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:297)
    at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:202)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1108)
    at com.itheima.enable.SpringbootEnableApplication.main(SpringbootEnableApplication.java:13)
2019-12-06 13:58:52.044  INFO 14492 --- [           main] c.i.enable.SpringbootEnableApplication   : Started SpringbootEnableApplication in 0.707 seconds
 (JVM running for 1.159)
```

```
@ComponentScan扫描范围：当前引导类所在包以其子包
com.itheima.enable
com.itheima.config.UserConfig
两个包明显是平级的
```

第一种：增加扫描包的范围

```
@SpringBootApplication
@ComponentScan("com.itheima.config")
public class SpringbootEnableApplication {}
```

第二种：使用@Import注解

使用@Import注解，都会被Spring创建，放入IOC容器

```
@SpringBootApplication
@Import(UserConfig.class)
public class SpringbootEnableApplication {}
```

第三种：对@Import进行封装

创建EnableUser注解类，使用@Import注解

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

此时，查看@SpringBootApplication注解时发现，里面有@EnableAutoConfiguration注解，而这个注解中使用了@Import({AutoConfigurationImportSelector.class})注解，加入了一个类。

1.4、@Import注解

@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：

- 导入Bean
- 导入配置类

- 导入ImportSelector实现类，一般用于加载配置文件中的类
- 导入ImportBeanDefinitionRegistrar实现类

1.4.1、导入Bean

```
@SpringBootApplication
@Import(User.class)
public class SpringbootEnableApplication {}
```

```
 (JVM running for 0.823)
Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'user' available
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanDefinition(DefaultListableBeanFactory.java:8
    at org.springframework.beans.factory.support.AbstractBeanFactory.getMergedLocalBeanDefinition(AbstractBeanFactory.java:1278
    at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:297)
    at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:202)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1108)
    at com.itheima.enable.SpringbootEnableApplication.main(SpringbootEnableApplication.java:36)

Process finished with exit code 1
```

这样导入不了的原因在于，我们是通过Bean的名称"user"来获取对象，而导入的这个User.class不一定叫"user"这个名字，所以需要修改启动类，通过类的类型来获取。

```
public static void main(String[] args) {
    ConfigurableApplicationContext context =
SpringApplication.run(SpringbootEnableApplication.class,
args);

    /*Object user = context.getBean("user");
        System.out.println(user);*/
    User user = context.getBean(User.class);
    System.out.println(user);
}
```

如果想要获取Bean的名称，那么可以使用context.getBeansOfType

```
Map<String, User> map =
context.getBeansOfType(User.class);
System.out.println(map);
```

```
 (JVM running for 0.818)
com.itheima.domain.User@68f4865
{com.itheima.domain.User=com.itheima.domain.User@68f4865}
```

所以自动创建的Bean名称为com.itheima.domain.User

那么通过这个名字就可以获取这个类

```
Object user1 =
context.getBean("com.itheima.domain.User");
System.out.println(user1);
```

```
 (JVM running for 0.822)
com.itheima.domain.User@57576994
{com.itheima.domain.User=com.itheima.domain.User@57576994}
com.itheima.domain.User@57576994
```

1.4.2、导入配置类

配置类指的是前面创建好的UserConfig，那么现在再创建一个实体类

```
package com.itheima.domain;

public class Role {
}
```

修改UserConfig配置类，将Role这个类也加入到Bean里

```
@Bean
public Role role(){
    return new Role();
}
```

修改启动类，增加Role的类加载

```
Role role = context.getBean(Role.class);
System.out.println(role);
```

```
 profiles: default
2019-12-06 14:47:43.356  INFO 8764 --- [
 (JVM running for 0.851)
com.itheima.domain.User@934b6cb
com.itheima.domain.Role@55cf0d14
```

此时，两个类都可以被加载。

1.4.3、实现ImportSelector接口

创建一个MyImportSelector类，实现ImportSelector接口，重写里面的selectImports方法

```
public class MyImportSelector implements ImportSelector
{
    @Override
    public String[] selectImports(AnnotationMetadata
annotationMetadata) {
        return new String[]{"com.itheima.domain.Role",
"com.itheima.domain.User"};
    }
}
```

修改启动类

```
@SpringBootApplication
@Import(MyImportSelector.class)
public class SpringbootEnableApplication {}
```

```
2019-12-06 14:55:51.049  INFO 10764 ---
 (JVM running for 1.216)
com.itheima.domain.User@2474f125
com.itheima.domain.Role@7357a011
```

此时，两个类都可以被加载。

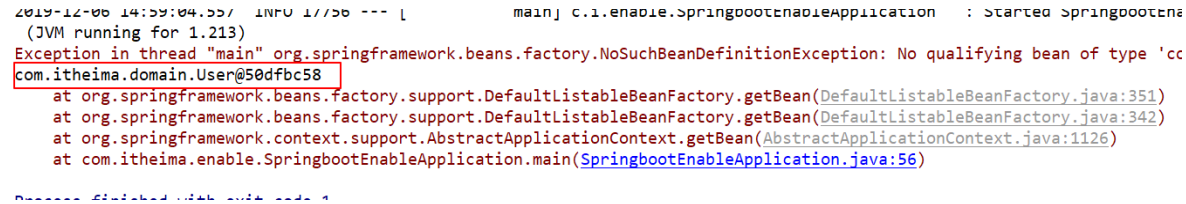
1.4.4、实现ImportBeanDefinitionRegistrar接口

创建一个MyImportBeanDefinitionRegistrar类，实现ImportBeanDefinitionRegistrar接口，重写registerBeanDefinitions方法

```
AbstractBeanDefinition beanDefinition =
BeanDefinitionBuilder.rootBeanDefinition(User.class).get
BeanDefinition();
registry.registerBeanDefinition("user", beanDefinition);
```

修改启动类

```
@SpringBootApplication
@Import(MyImportBeanDefinitionRegistrar.class)
public class SpringbootEnableApplication {}
```



此时，User类被加载，而Role没有被加载，想要加载Role类，再次修改registerBeanDefinitions方法

```
@Override
public void registerBeanDefinitions(AnnotationMetadata
importingClassMetadata, BeanDefinitionRegistry registry)
{
    //注册User类
    AbstractBeanDefinition beanDefinition =
BeanDefinitionBuilder.rootBeanDefinition(User.class).get
BeanDefinition();
    registry.registerBeanDefinition("user",
beanDefinition);

    //注册Role类
    beanDefinition =
BeanDefinitionBuilder.rootBeanDefinition(Role.class).get
BeanDefinition();
    registry.registerBeanDefinition("role",
beanDefinition);
}
```

而通过名称也可以将User类加入到Bean中，只需要修改启动类用名称获取Bean即可。

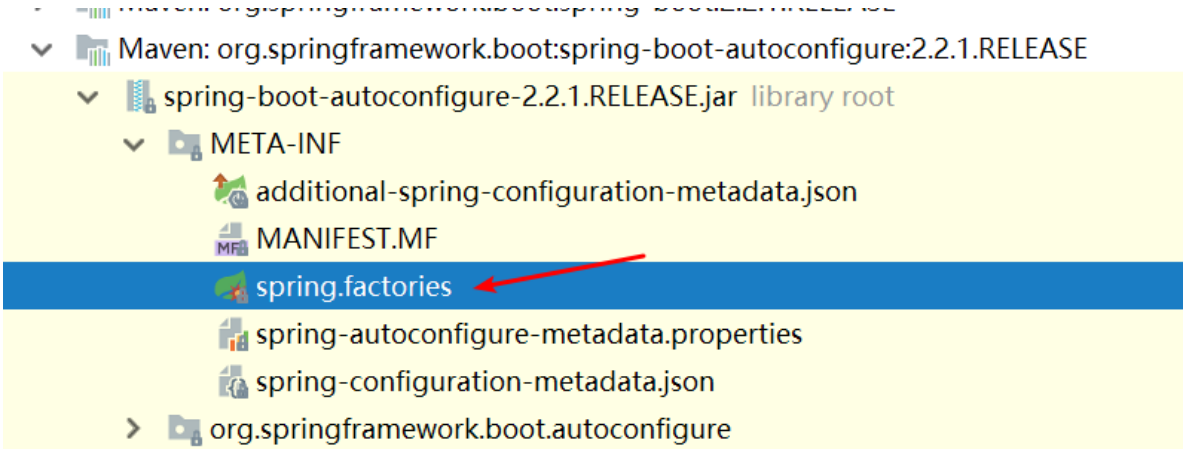
1.5、@EnableAutoConfiguration

@SpringBootApplication中的@EnableAutoConfiguration注解，也是通过@Import({AutoConfigurationImportSelector.class})来实现类的加载，那么说明Springboot中也是使用第三种方法导入类。

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!this.isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    } else {
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
        AutoConfigurationImportSelector.AutoConfigurationEntry autoConfigurationEntry = this.getAutoConfigurationEntry(autoConfigurationMetac
        return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
    }
}
```

```
protected AutoConfigurationImportSelector.AutoConfigurationEntry getAutoConfigurationEntry(AutoConfiguration
    if (!this.isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    } else {
        AnnotationAttributes attributes = this.getAttributes(annotationMetadata);
        List<String> configurations = this.getCandidateConfigurations(annotationMetadata, attributes);
        configurations = this.removeDuplicates(configurations);
        Set<String> exclusions = this.getExclusions(annotationMetadata, attributes);
        this.checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
        configurations = this.filter(configurations, autoConfigurationMetadata);
        this.fireAutoConfigurationImportEvents(configurations, exclusions);
        return new AutoConfigurationImportSelector.AutoConfigurationEntry(configurations, exclusions);
    }
}
```

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderFactoryClass(), this.getBear
    Assert.notEmpty(configurations, message: "No auto configuration classes found in META-INF/spring.factories. If you are using a
    return configurations;
}
```



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。

二、SpringBoot自动配置--自定义Starter

2.1、分析MyBatis加载过程

引入坐标

```
<!--
https://mvnrepository.com/artifact/org.mybatis.spring.boot/mybatis-spring-boot-starter -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
</dependency>
```

加载过程

2.2、自定义Starter需求

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

步骤：

- 创建 redis-spring-boot-autoconfigure 模块
- 创建 redis-spring-boot-starter 模块，依赖 redis-spring-boot-autoconfigure的模块
- 在 redis-spring-boot-autoconfigure 模块中初始化 Jedis 的 Bean。并定义META-INF/spring.factories 文件
- 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。

2.3、创建两个模块

redis-spring-boot-autoconfigure

redis-spring-boot-starter

在starter的坐标中加入autoconfigure

```
<dependency>
    <groupId>com.itheima</groupId>
    <artifactId>redis-spring-boot-
autoconfigure</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

在autoconfigure的坐标中加入Jedis

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
```

2.4、创建RedisAutoConfigure配置类

在autoconfigure模块中创建RedisAutoConfigure配置类

```
@Configuration
@EnableConfigurationProperties(RedisProperties.class)
public class RedisAutoConfigure {
    /**
     * 提供Jedis的Bean
     */
    @Bean
    public Jedis jedis(RedisProperties redisProperties){
        return new Jedis(redisProperties.getHost(),
redisProperties.getPort());
    }
}
```

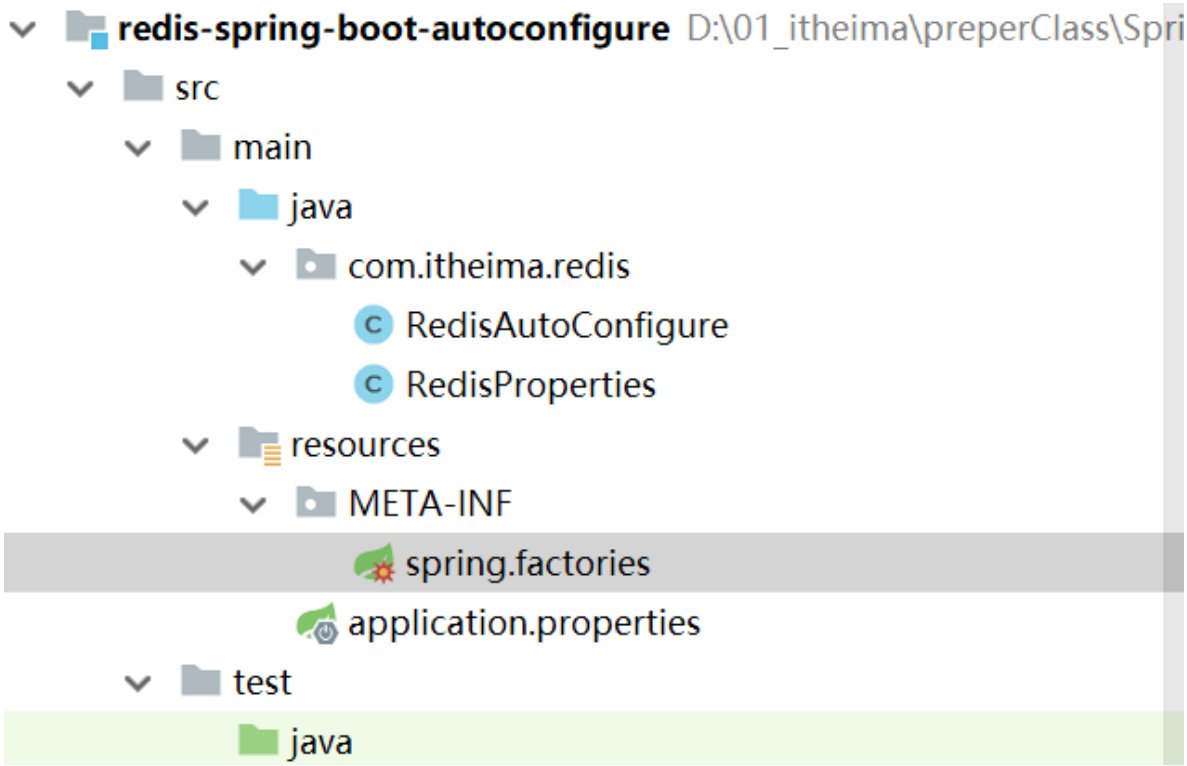
在autoconfigure模块中创建RedisProperties配置类

```
@ConfigurationProperties(prefix = "redis")
public class RedisProperties {
```

```
    private String host = "localhost";
    private int port = 6379;

    public String getHost() {
        return host;
    }

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

    public int getPort() {
        return port;
    }

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

2.5、创建META-INF/spring.factories

在resources下新建META-INF/spring.factories

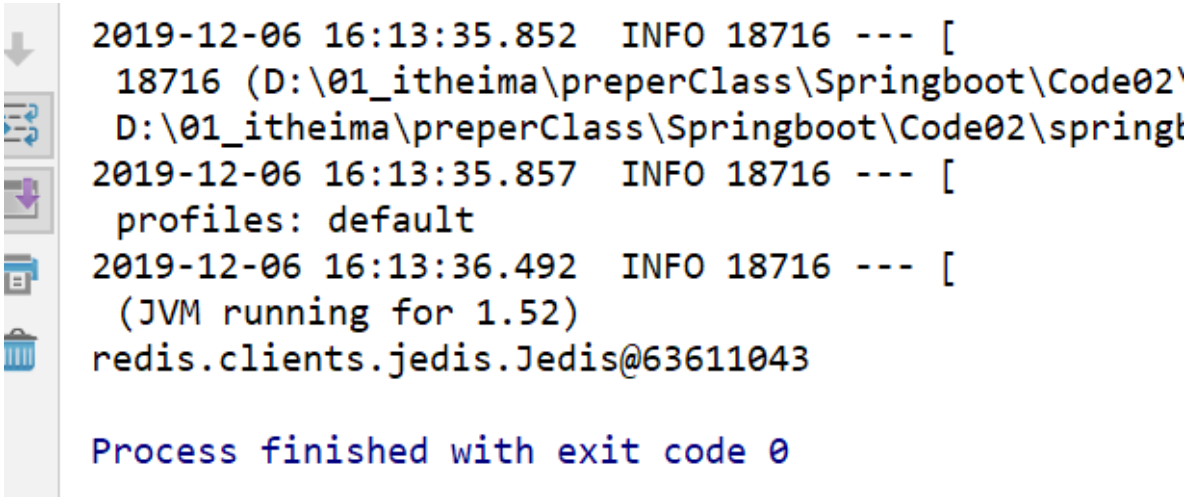


```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.itheima.redis.RedisAutoConfigure
```

2.6、修改启动类

修改springboot-enable启动类

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

        Jedis jedis = context.getBean(Jedis.class);
        System.out.println(jedis);
    }
}
```



```
2019-12-06 16:13:35.852  INFO 18716 --- [
 18716 (D:\01_itheima\preperClass\Springboot\Code02\
 D:\01_itheima\preperClass\Springboot\Code02\springb
2019-12-06 16:13:35.857  INFO 18716 --- [
 profiles: default
2019-12-06 16:13:36.492  INFO 18716 --- [
 (JVM running for 1.52)
redis.clients.jedis.Jedis@63611043

Process finished with exit code 0
```

此时，Jedis可以加载到。

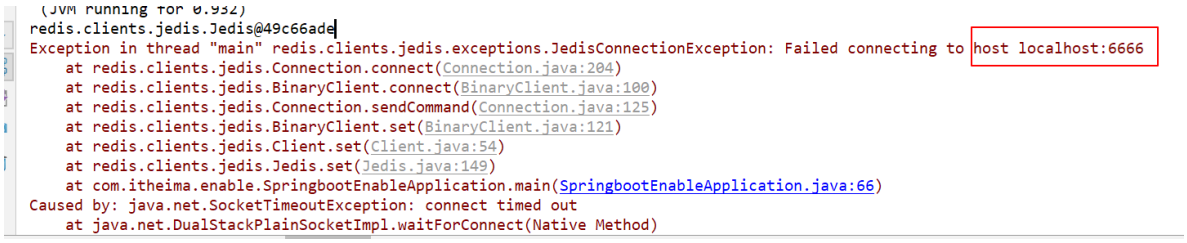
2.7、使用Jedis

```
jedis.set("name", "itcast");
String value = jedis.get("name");
System.out.println(value);
```

2.8、使用配置文件

修改enable里的application.properties

```
redis.port=6666
```



启动时报错，连接不到本地的localhost:6666，证明redis配置文件已经起作用了。

2.9、优化RedisAutoConfigure

在RedisAutoConfigure类上增加注解@ConditionalOnClass(Jedis.class)

加载的时候判断Jedis类存在的时候才加载Bean

```
@Configuration
@EnableConfigurationProperties(RedisProperties.class)
@ConditionalOnClass(Jedis.class)
public class RedisAutoConfigure {
    /**
     * 提供Jedis的Bean
     */
    @Bean
    @ConditionalOnMissingBean(name = "jedis")
    public Jedis jedis(RedisProperties redisProperties){
        System.out.println("RedisAutoConfigure....");
        return new Jedis(redisProperties.getHost(),
redisProperties.getPort());
    }
}
```

在jedis方法上增加@ConditionalOnMissingBean注解，当jedis没有被创建时加载这个类，并写入Bean

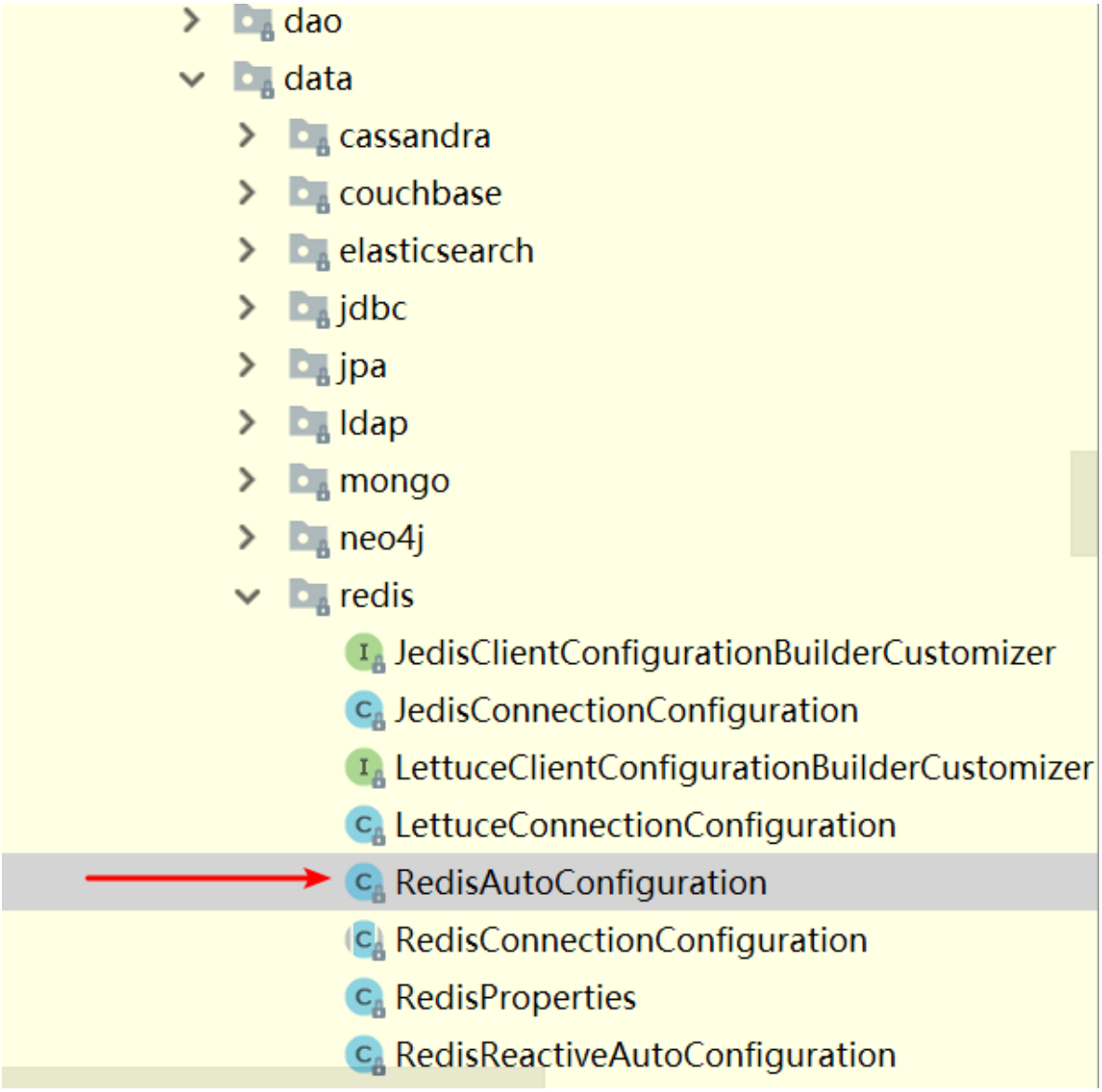
测试：

在启动类中创建一个Jedis

```
@Bean
public Jedis jedis(){
    return new Jedis();
}
```

启动的时候可以看到当Jedis存在时则不再加载。

2.10、查看redis源jar包



三、SpringBoot监听机制

3.1、JAVA的监听机制

SpringBoot 的监听机制，其实是对Java提供的事件监听机制的封装。

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

3.2、Springboot监听器

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。

一共有四种实现方法：

- ApplicationContextInitializer
- SpringApplicationRunListener
- CommandLineRunner
- ApplicationRunner

3.3、创建Listener模块

3.3.1、创建MyApplicationContextInitializer

创建MyApplicationContextInitializer，实现ApplicationContextInitializer接口

```
@Component
public class MyApplicationContextInitializer implements
ApplicationContextInitializer {
    @Override
    public void
initialize(ConfigurableApplicationContext
configurableApplicationContext) {

 System.out.println("ApplicationContextInitializer...ini
tialize");
    }
}
```

3.3.2、创建MySpringApplicationRunListener

创建MySpringApplicationRunListener，实现SpringApplicationRunListener接口

```
@Component
public class MySpringApplicationRunListener implements
SpringApplicationRunListener {
    @Override
    public void starting() {

 System.out.println("SpringApplicationRunListener...正在
启动");
    }

    @Override
    public void
environmentPrepared(ConfigurableEnvironment environment)
{

 System.out.println("SpringApplicationRunListener...环境
准备中");
    }

    @Override
    public void
contextPrepared(ConfigurableApplicationContext context)
{
```

```
 System.out.println("SpringApplicationRunListener...上下文准备");
    }

    @Override
    public void contextLoaded(ConfigurableApplicationContext context) {

 System.out.println("SpringApplicationRunListener...上下文加载");
    }

    @Override
    public void started(ConfigurableApplicationContext context) {

 System.out.println("SpringApplicationRunListener...已经启动");
    }

    @Override
    public void running(ConfigurableApplicationContext context) {

 System.out.println("SpringApplicationRunListener...正在启动中");
    }

    @Override
    public void failed(ConfigurableApplicationContext context, Throwable exception) {

 System.out.println("SpringApplicationRunListener...启动失败");
    }
}
```

3.3.3、创建MyCommandLineRunner

创建MyCommandLineRunner，实现CommandLineRunner接口

```
@Component
public class MyCommandLineRunner implements
CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        System.out.println("CommandLineRunner...run");
    }
}
```

3.3.4、创建MyApplicationRunner

创建MyApplicationRunner，实现ApplicationRunner接口

```
@Component
public class MyApplicationRunner implements
ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws
Exception {
        System.out.println("ApplicationRunner...run");
    }
}
```

3.3.5、运行启动类

```
2019-12-06 16:58:38.571  INFO 13528 --- [           main] c.i.l.Springbo
 (JVM running for 1.098)
ApplicationRunner...run
CommandLineRunner...run

Process finished with exit code 0
```

只有MyApplicationRunner和MyCommandLineRunner运行了监听。

3.3.6、修改监听类

修改MyApplicationRunner和MyCommandLineRunner打印args

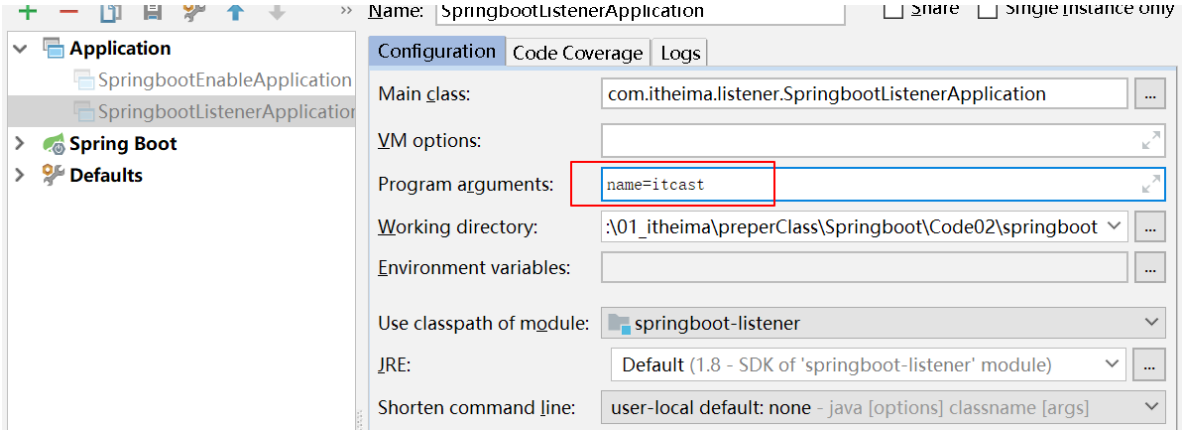
MyApplicationRunner：

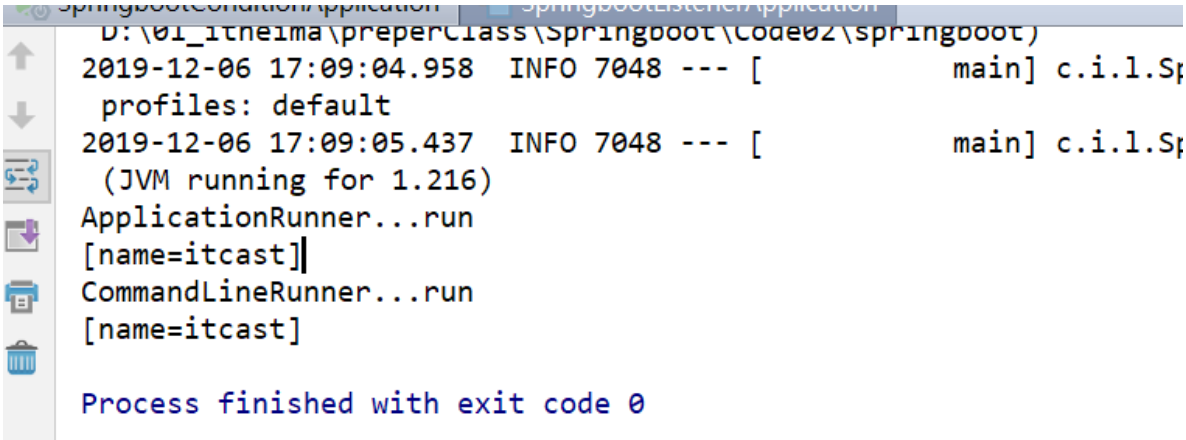
```
@Override
public void run(ApplicationArguments args) throws
Exception {
    System.out.println("ApplicationRunner...run");

 System.out.println(Arrays.asList(args.getSourceArgs()))
;
}
```

MyCommandLineRunner:

```
@Override
public void run(String... args) throws Exception {
    System.out.println("CommandLineRunner...run");
    System.out.println(Arrays.asList(args));
}
```

在运行时可以将配置信息加载进来



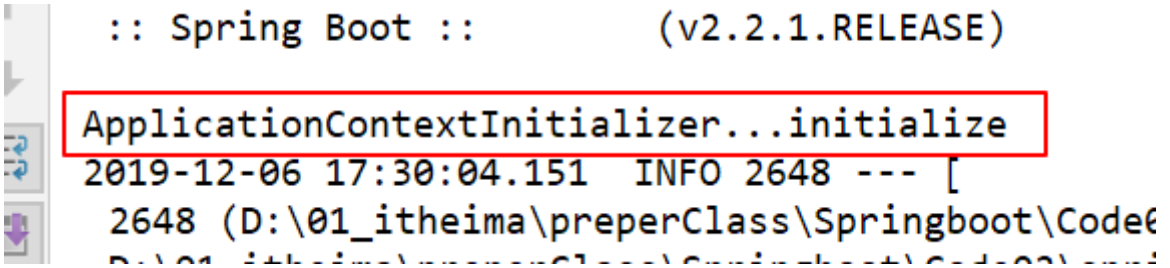


所以，这两个监听ApplicationRunner和CommandLineRunner是一样的。

3.4、配置MyApplicationContextInitializer

在模块中增加 META-INF/spring.factories 配置文件

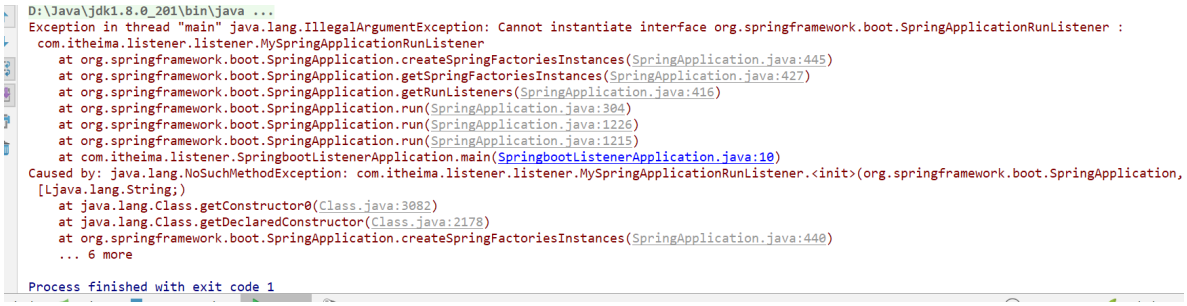
```
org.springframework.context.ApplicationContextInitializer=com.itheima.listener.listener.MyApplicationContextInitializer
```



3.5、配置MySpringApplicationRunListener

修改 META-INF/spring.factories 配置文件

```
org.springframework.boot.SpringApplicationRunListener=com.itheima.listener.listener.MySpringApplicationRunListener
```



运行提示：没有MySpringApplicationRunListener的init方法

所以需要查看SpringApplicationRunListener这个接口的实现类

```
public class EventPublishingRunListener implements SpringApplicationRunListener, Ordered {
    private final SpringApplication application;
    private final String[] args;
    private final SimpleApplicationEventMulticaster initialMulticaster;

    public EventPublishingRunListener(SpringApplication application, String[] args) {
        this.application = application;
        this.args = args;
        this.initialMulticaster = new SimpleApplicationEventMulticaster();
        Iterator var3 = application.getListeners().iterator();

        while(var3.hasNext()) {
            ApplicationListener<?> listener = (ApplicationListener)var3.next();
            this.initialMulticaster.addApplicationListener(listener);
        }

    }
```

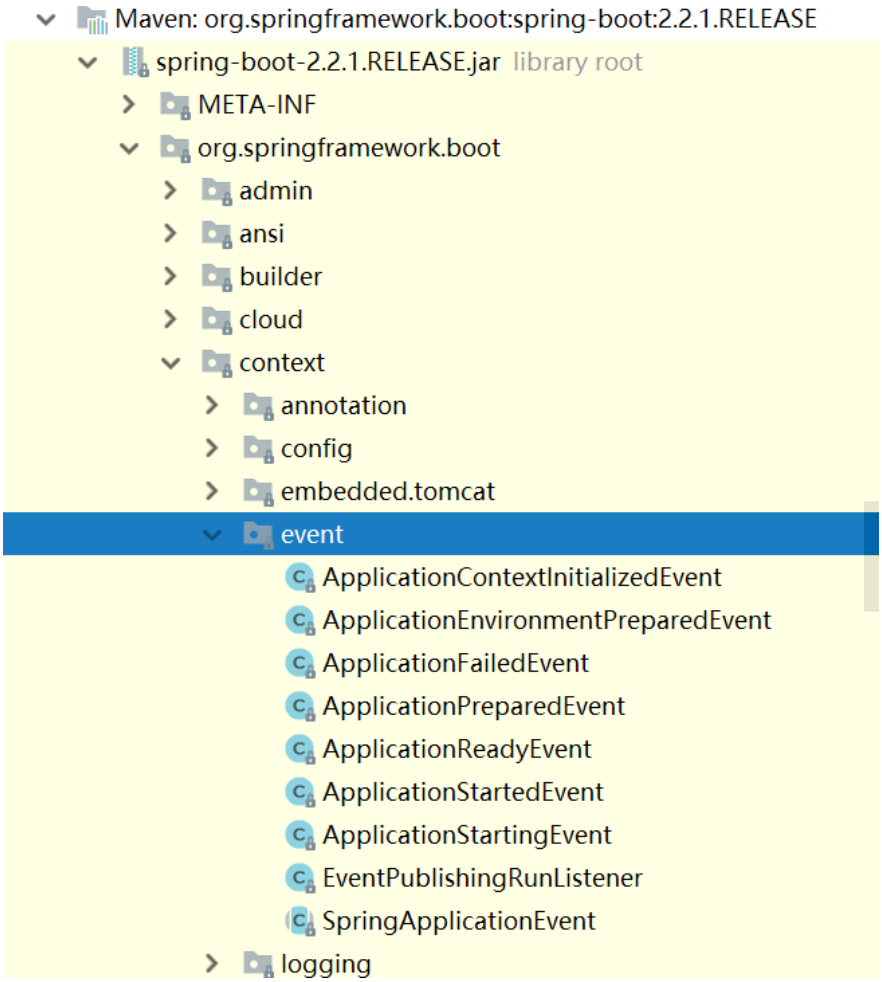
需要SpringApplication和String[]两个参数进行构造，因此修改MySpringApplicationRunListener实现类，增加构造方法

```
public MySpringApplicationRunListener(SpringApplication application, String[] args) {}
```

并且去掉@Component注解

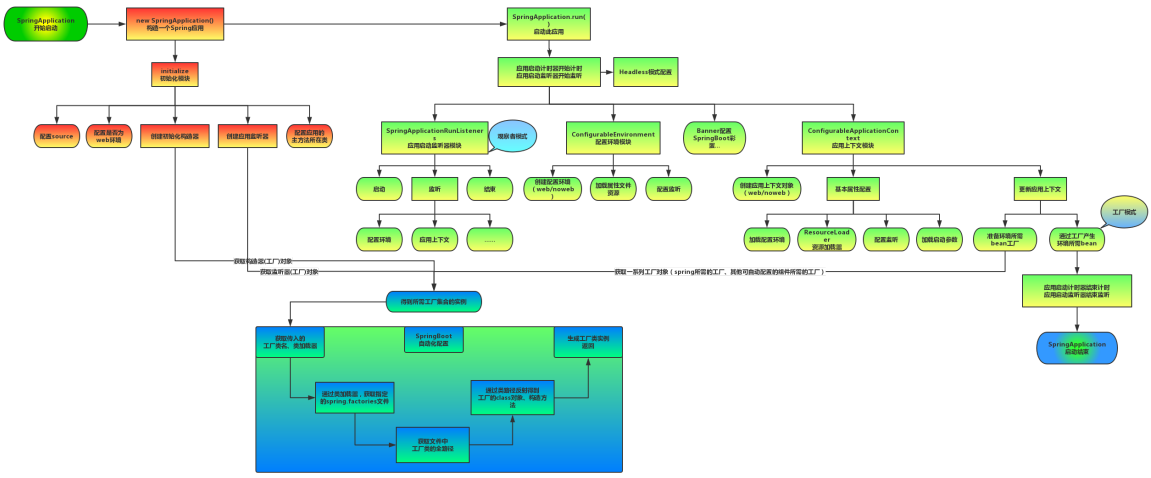
```
D:\Java\jdk1.8.0_201\bin\java ...
SpringApplicationRunListener...正在启动
SpringApplicationRunListener...环境准备中


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.2.1.RELEASE)

ApplicationContextInitializer...initialize
SpringApplicationRunListener...上下文准备
2019-12-06 17:36:11.628  INFO 16868 --- [           m
 16868 (D:\01_itheima\preperClass\Springboot\Code02\s
 D:\01_itheima\preperClass\Springboot\Code02\springbo
2019-12-06 17:36:11.630  INFO 16868 --- [           m
 profiles: default
SpringApplicationRunListener...上下文加载
2019-12-06 17:36:11.945  INFO 16868 --- [           m
 (JVM running for 0.828)
SpringApplicationRunListener...已经启动
ApplicationRunner...run
[name=itcast]
CommandLineRunner...run
[name=itcast]
SpringApplicationRunListener...正在启动中

Process finished with exit code 0
```

此时启动正常。



ApplicationStartedEvent继承自SpringApplicationEvent
SpringApplicationEvent继承自ApplicationEvent
ApplicationEvent继承自EventObject
证明：Springboot里的监听事件是对java监听事件的一个封装

3.6、SpringBoot运行流程分析



四、SpringBoot监控

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。

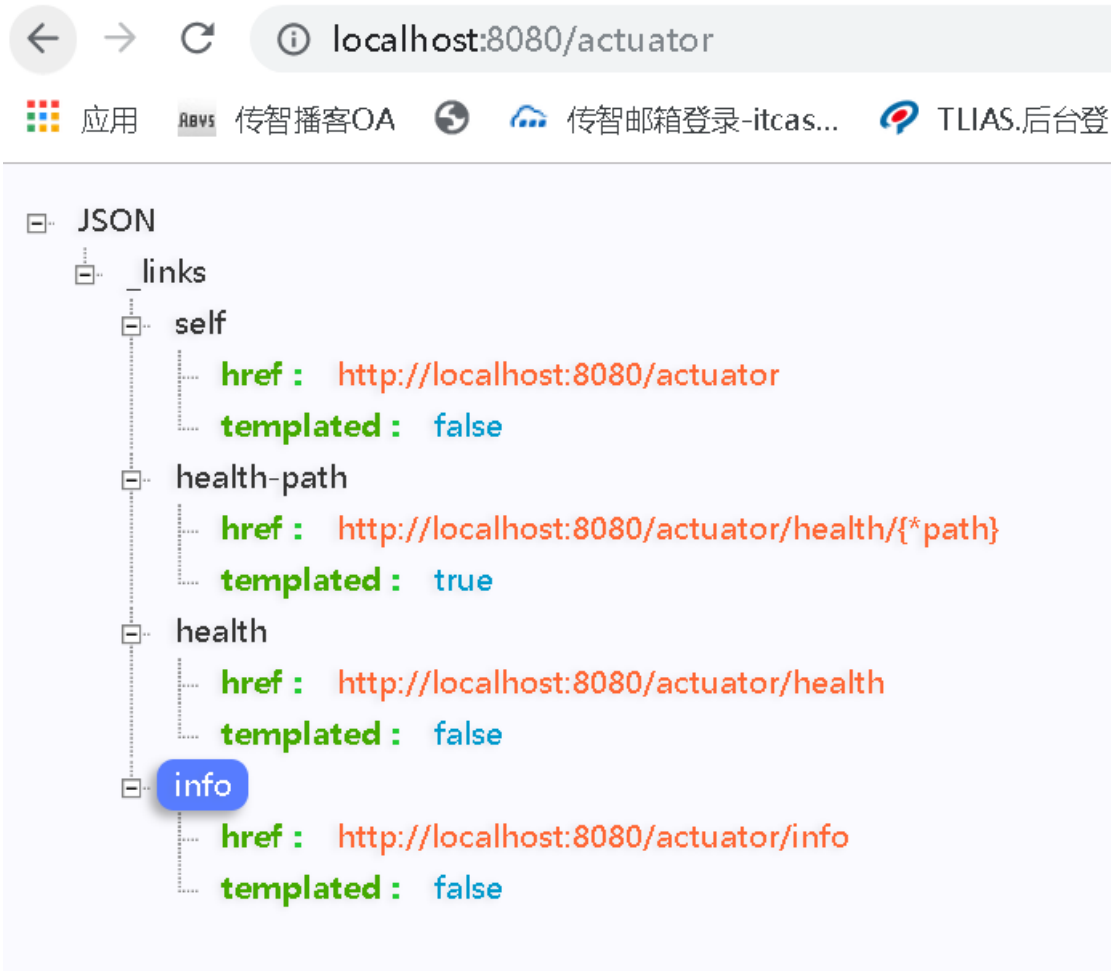
4.1、创建模块

需要勾选web和ops下的SpringBootActuator

坐标自动引入

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

直接访问： http://localhost:8080/actuator



4.2、查看info

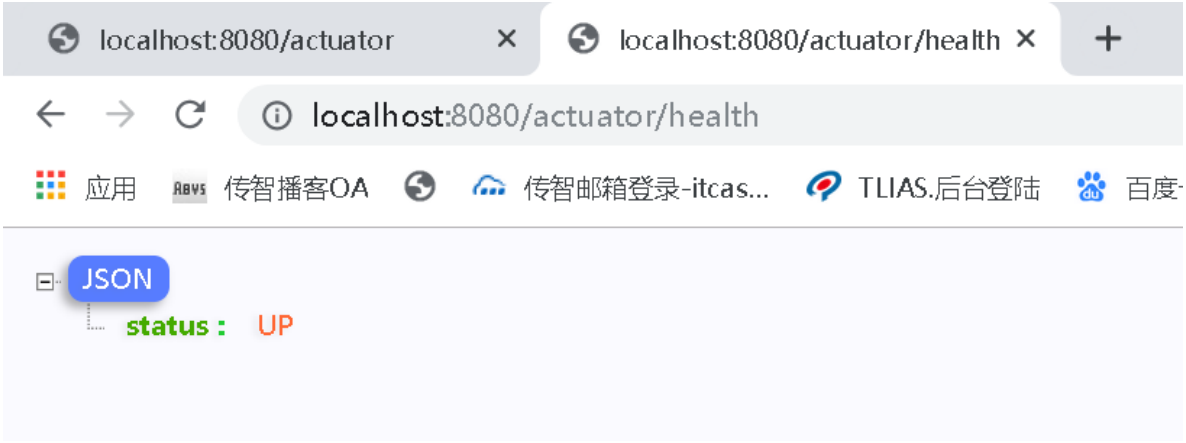
info获取的是配置文件以info开头的信息：

修改properties

```
info.name=zhangsan
info.age=20
```

4.3、查看所有信息

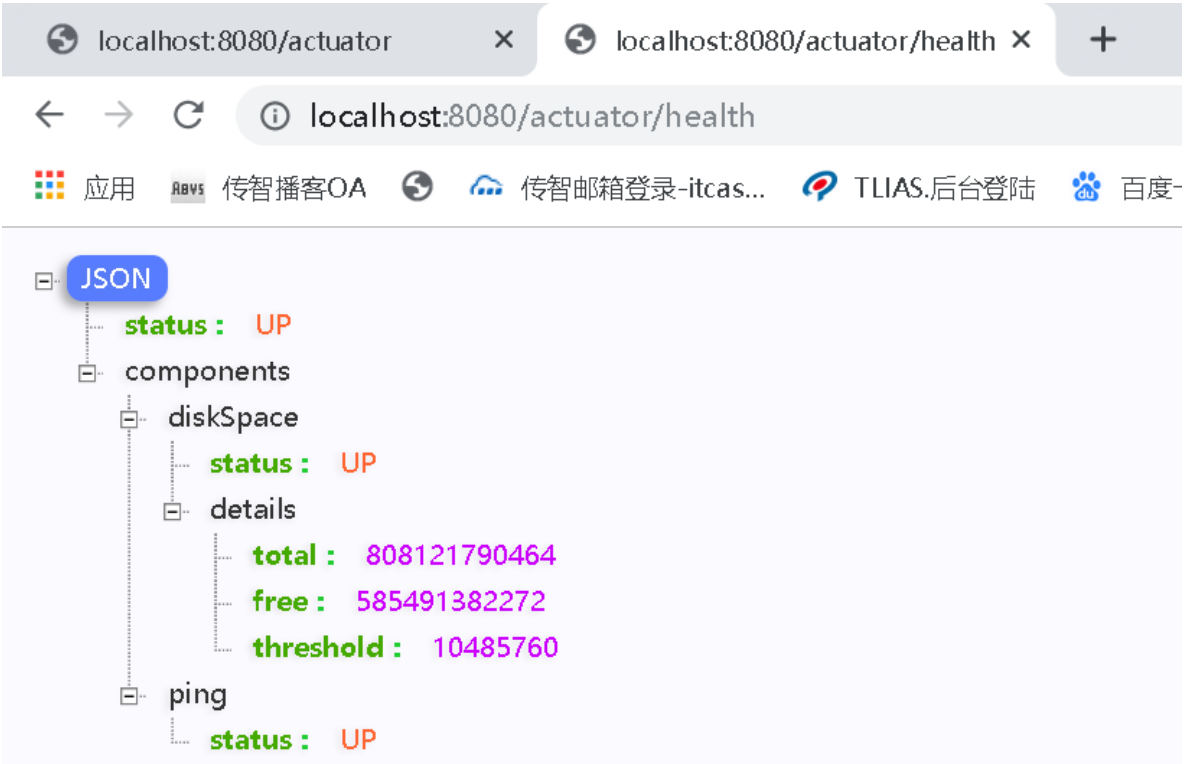
未开启所有明细状态：



修改properties

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

开启所有明细状态：



4.4、暴露所有的信息

修改properties：

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


localhost:8080/actuator
localhost:8080/actuator/health
localhost:8080/actuator
应用
传智播客OA
传智邮箱登录-itcas...
TLIAS.后台登陆
JSON
_links
self
href : http://localhost:8080/actuator
templated : false
beans
href : http://localhost:8080/actuator/beans
templated : false
caches-cache
href : http://localhost:8080/actuator/caches/{cache}
templated : true
caches
href : http://localhost:8080/actuator/caches
templated : false
health-path
href : http://localhost:8080/actuator/health/{*path}
templated : true
health
href : http://localhost:8080/actuator/health
templated : false
info
href : http://localhost:8080/actuator/info
templated : false
conditions
href : http://localhost:8080/actuator/conditions