

第4章 Spring Boot原理分析

广东财经大学信息学院

罗 东 俊 博士

ZSUJONE@126.COM

(内部资料，请勿外传)



目的和要求

- 熟悉**Spring Boot**全局属性的配置以及自定义配置。
- 掌握**Spring Boot**配置文件属性值注入。
- 掌握**Profile**多环境配置。
- 了解随机值设置以及参数间引用。
- 掌握自定义依赖启动器的过程。
- 了解**Spring Boot**的自动化配置原理以及执行流程。



主要内容

- 4.1 Spring Boot核心配置
- 4.2 Spring Boot自动配置
- 4.3 Spring Boot依赖管理
- 4.4 Spring Boot项目执行流程



4.1 Spring Boot核心配置

- 4.1.1 Spring Boot全局属性配置
- 4.1.2 Spring Boot自定义配置



4.1.1 Spring Boot全局属性配置

- 1. 全局配置文件
- 2. 配置文件属性值的注入
- 3. Profile多环境配置
- 4. 随机值设置以及参数间引用



1. 全局配置文件

■ 全局配置文件能够对一些默认属性值进行修改。
有两种文件形式：

✧ ① **application.properties** 配置文件

✧ ② **application.yaml** 配置文件

■ 全局配置文件存放路径：

✧ **src/main/resource** 目录或者类路径的 **/config**

■ **application.properties** 配置文件的优先级高于 **application.yaml** 配置文件

✧ 相同属性的配置值优先从 **application.properties** 文件中读取



application.properties文件示例

```
server.address=80  
server.port=8443  
spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
spring.config.additional-location=  
spring.config.location=  
spring.config.name=application
```



application.yaml文件示例

#对实体类对象Person进行属性配置

person:

id: 2

name: 张三

hobby: [sing,read,sleep]

family: [father,mother]

map: {k1: v1,k2: v2}

pet: {type: cat, name: tom}



YAML文件语法

- **YAML**文件格式是**Spring Boot**支持的一种**JSON**超集文件格式。
- 相较于传统的**.properties**配置文件，**YAML**文件以数据为核心，是一种更为直观且容易被电脑识别的数据序列化格式。
- **YAML**文件中的键值语法格式：
 - ✧ **key**: (空格) **value**
 - ✧ **value**值类型不同，写法不同



value值的不同写法

■ ①value的值为普通数据类型

```
server:  
  port: 8081  
  path: /hello
```



value值的不同写法

■ ②value的值为数组或单列集合类型

缩进式写法

```
person:  
  hobby:  
    - play  
    - read  
    - sleep
```

行内式写法

```
person:  
  hobby: [play,read,sleep]
```



value值的不同写法

■ ③value的值为Map集合或对象类型

缩进式写法

```
person:  
  map:  
    k1: v1  
    k2: v2
```

行内式写法

```
person:  
  map: {k1: v1,k2: v2}
```



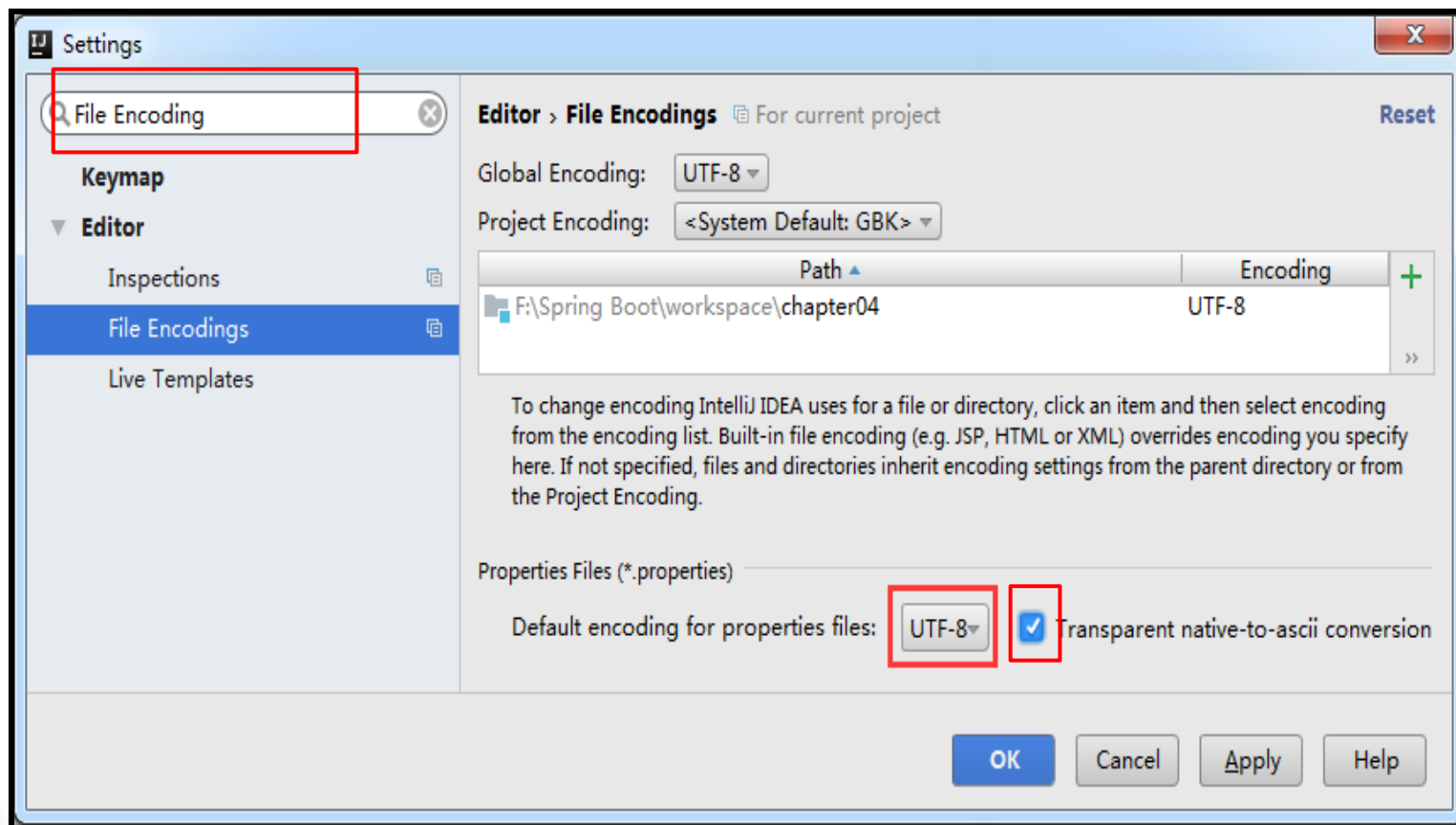
属性配置文件编码设置

■ 属性配置文件中，中文字符往往会出现乱码，此时需要设置**IDEA**

✧ 打开的**IDEA**开发工具中，依次选择【**File**】->【**Settings**】选项打开项目设置窗口，在窗口中搜索“**File Encodings**”关键字，将**Default encoding for properties files** 设置为**UTF-8**



属性配置文件编码设置



2. 配置文件属性值的注入

■ 在Spring Boot的全局配置文件中，

- ✧ 如果配置的属性是Spring Boot默认提供的属性，如server.port，那么Spring Boot内部会自动扫描并读取属性值。
- ✧ 如果配置的属性是用户自定义属性，则必须使用以下两种方式之一注入属性值：
 - ①使用@ConfigurationProperties注入
 - ②使用@Value注入



使用 @ConfigurationProperties 注入

```
@Component
```

```
@ConfigurationProperties(prefix = "person")
```

```
public class Person {  
    private int id;  
    public void setId(int id) {  
        this.id = id;  
    }  
}
```



注意：使用@Component和@ConfigurationProperties注解批量注入属性值时，要保证配置文件中的属性与对应实体类的属性**一致**，否则无法正确获取并注入属性值。

@Component：将一个自定义类作为Bean添加到Spring容器中。此外还可使用@Controller、@Service、@Repository、@Configuration等注解。

@ConfigurationProperties(prefix = "person")：将配置文件中前缀为 " person" 属性的配置值注入到Person实体类的对应属性中。



使用 @Value注入

@Component

```
public class Person {
```

```
    @Value("${person.id}")
```

```
    private int id;
```

```
}
```

使用@Component和@Value注解对每一个属性注入设置，免去了属性
setXX()方法



两种注解的对比分析

对比点	@ConfigurationProperties	@Value
底层框架	Spring Boot	Spring
功能	批量注入配置文件中的属性	单个注入
属性setXX()方法	需要	不需要
复杂类型属性注入	支持	不支持
松散绑定	支持	不支持
JSR303数据校验	支持	不支持
SpEL表达式	不支持	支持



松散绑定语法

<code>person.firstName=james</code>	// 标准写法, 对应Person类属性名
<code>person.first-Name=james</code>	// 使用横线 "-" 分隔多个单词
<code>person.first_Name=james</code>	//使用下划线 "_" 分隔多个单词
<code>person.FIRST_NAME=james</code>	// 使用大小写格式, 推荐常量属性配置



JSR303数据校验示例

@Component

@ConfigurationProperties(prefix = "person")

@Validated //引入Spring框架支持的数据校验规则

```
public class Person {
```

@Email //对属性进行规则匹配

```
    private String email;
```

```
    public void setEmail(String email) {
```

```
        this.email = email;
```

```
    }
```

```
}
```



SpEL表达式语法

■ SpEL表达式语法: #{xx}

@Component

```
public class Person {
```

```
    @Value("#{5*2}")
```

//不使用配置文件，直接为属性注入值

```
    private int id;
```

```
}
```



3. Profile多环境配置

- 在实际开发中，应用程序通常要部署到不同的运行环境，如开发环境、测试环境、生成环境。
- **Spring Boot**提供了两种多环境配置方式：
 - ✧ ①使用**Profile**文件
 - ✧ ②使用 **@Profile**注解



①使用Profile文件

■ 使用**Profile**文件进行多环境配置时，该**Profile**文件名必须满足如下格式：

✧ **application-{profile}.properties**

- **{profile}**对应具体的环境标识，如**dev**（开发环境）、**test**（测试环境）、**prod**（生产环境）
- **Profile**文件放在项目的**resources**目录下

■ 激活指定环境的方式：

✧ 通过命令行方式激活指定环境的配置文件

- 如：**java -jar xxx.jar --spring.profiles.active=dev**

✧ 在全局配置文件设置**spring.profiles.active**属性激活

- 如：**spring.profiles.active=dev**



示例代码

■ application-dev.properties:

```
server.port=8081
```

■ application-test.properties:

```
server.port=8082
```

■ application-prod.properties:

```
server.port=8083
```



②使用 @Profile注解

■ **@Profile**作用于配置类上，然后通过**value**属性值标识不同环境配置（等同于**Profile**文件名称中的**profile**值）

✧如： **@Profile("dev")**

■ 激活指定环境的方式

✧在全局配置文件中配置**spring.profiles.active**属性激活

• 如： **spring.profiles.active=dev**



示例代码

```
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Profile;
```

@Configuration

@Profile("dev") // 标识开发环境配置

```
public class DevDBConnector implements DBConnector {  
    @Override  
    public void configure() {  
        System.out.println("数据库配置环境dev");  
    }  
}
```



4. 随机值设置以及参数间引用

- 在**Spring Boot**配置文件中设置属性时，除了可以设置固定的属性值外，还可使用随机值和参数间引用。



①随机值设置

- 随机值设置使用了**Spring Boot**内嵌的**RandomValuePropertySource**类，对一些隐秘属性值或者测试用例值进行随机值注入
- 随机值设置语法格式：**`${random.xx}`**
 - ◇ **xx**表示需要指定生成的随机数类型和范围



示例代码

```
my.secret=${random.value}
```

```
my.number=${random.int}
```

```
my.bignumber=${random.long}
```

```
my.uuid=${random.uuid}
```

```
my.number.less.than.ten=${random.int(10)}
```

```
my.number.in.range=${random.int[1024,65536]}
```

之间的随机整数

//配置随机字符串

//配置随机的整数

//配置随机long类型数

//配置随机UUID类型数

//配置小于10的随机整数

//配置范围在[1024,65536]之



②参数间引用

- 参数间的引用，即先前定义的属性可以被其他属性引用。
- 参数间引用语法格式为： **`${xx}`**
 - ✧ **xx**表示先前在配置文件中已经配置过的属性名。
- 示例代码：

```
app.name=MyApp  
app.description=${app.name} is a Spring Boot application
```



4.1.2 Spring Boot 自定义配置

- 1. 自定义配置文件
- 2. 自定义配置类
- 3. 自定义自动配置类



1. 自定义配置文件

- 全局配置文件可以被**Spring Boot**自动加载，并且几乎所有属性配置都可以写在全局配置文件中。
- 但在模块开发中，通常也需要自定义配置文件，此时就需要手动加载。
- 手动加载自定义配置文件相关注解：
 - ✧ **@PropertySource**: 加载自定义属性配置文件
 - 在属性配置实体类上添加该注解，指定所需加载的属性配置文件位置和名称
 - 该注解默认不支持自定义的**YAML**文件加载
 - ✧ **@ImportResource**: 加载自定义**XML**配置文件
 - 在项目启动类上添加该注解，指定所需加载的**XML**配置文件位置和名称
 - **Spring Boot**默认不再使用**XML**文件配置项目



1) 自定义属性配置文件案例

■ 本案例实现自定义属性配置文件加载，并为实体类注入属性值。

■ 搭建步骤：

- ✧ ① 创建**Spring Boot**项目
- ✧ ② 引入配置处理器依赖
- ✧ ③ 创建实体类
- ✧ ④ 编写自定义属性配置文件
- ✧ ⑤ 效果测试



①创建Spring Boot项目

■ 使用Spring Initializr方式创建一个Spring Boot项目chapter04，在Dependencies依赖选择中选择Web依赖。

The image displays two screenshots of the Spring Initializr 'New Project' wizard interface.

Left Screenshot: Project Metadata

- Group:
- Artifact:
- Type:
- Language:
- Packaging:
- Java Version:
- Version:
- Name:
- Description:
- Package:

Right Screenshot: Dependencies

Spring Boot: 2.3.7.RELEASE

Dependencies List:

- ☒ Spring Web
- ☐ Spring Reactive Web
- ☐ Rest Repositories
- ☐ Spring Session
- ☐ Rest Repositories HAL Explorer
- ☐ Rest Repositories HAL Browser
- ☐ Spring HATEOAS
- ☐ Spring Web Services
- ☐ Jersey
- ☐ Vaadin

Selected Dependencies:

- Web
- Spring Web

Spring Web Description:

使用Spring MVC构建Web（包括RESTful）应用程序，使用Apache Tomcat作为默认的嵌入式容器。

[Building a RESTful Web Service](#)

[Serving Web Content with Spring MVC](#)

[Building REST services with Spring](#)

[Reference doc](#)

②引入配置处理器依赖

- 在pom.xml中引入配置处理器依赖，这个依赖会在编写属性配置文件时给出自动代码提示。

```
<!-- Spring Boot提供的配置处理器依赖 -->  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-configuration-processor</artifactId>  
    <optional>true</optional>  
</dependency>
```

③创建实体类

- 在chapter04项目下新建一个 **com.itheima.domain** 包，并在该包下新建一个普通实体类 **Pet** 和一个配置文件映射实体类 **Person**，提供 **person.properties** 自定义配置文件中对应的属性。



普通实体类Pet

```
public class Pet {  
    private String type;  
    private String name;  
  
    public String getType() { return type;}  
    public void setType(String type) {this.type = type;}  
    public String getName() {return name;}  
    public void setName(String name) {this.name = name;}  
}
```



配置文件映射实体类Person

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.PropertySource;
import java.util.*;

@Component
@PropertySource("classpath:person.properties")
@ConfigurationProperties(prefix = "person")
public class Person {
    private int id;           //id
    private String name;      //名称
    private List hobby;       //爱好
    private String[] family;  //家庭成员
    private Map map;
    private Pet pet;          //宠物
    .....
}
```


④编写自定义属性配置文件

- 在项目的**resources**目录下新建一个**person.properties**自定义属性配置文件，在该配置文件中编写需要设置的配置属性。

#对实体类对象Person进行属性配置

person.id=1

person.name=tom

person.hobby=play,read,sleep

person.family=father,mother

person.map.k1=v1

person.map.k2=v2

person.pet.type=dog

person.pet.name=kity

⑤效果测试

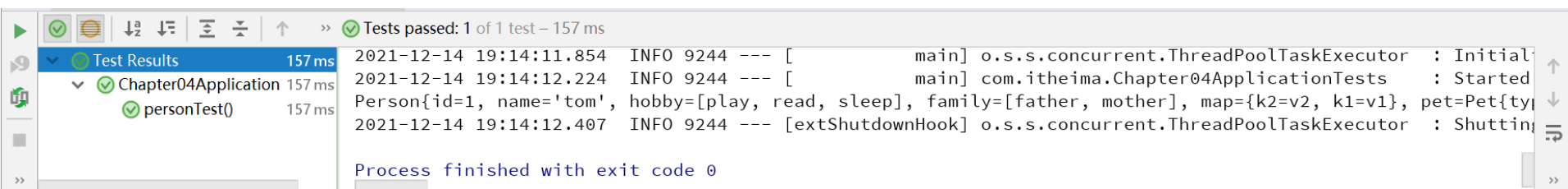
- 在测试类中引入**Person**实体类**Bean**对象，并新增一个测试方法进行输出测试。

```
@Autowired  
private Person person;  
  
@Test  
public void personTest() {  
    System.out.println(person);  
}
```



测试结果

■ 测试结果，如图所示



The screenshot shows an IDE's test results panel. On the left, a tree view displays the test hierarchy: 'Test Results' (157 ms), 'Chapter04Application' (157 ms), and 'personTest()' (157 ms), all marked with green checkmarks. The main area shows the test execution log with the following content:

```
2021-12-14 19:14:11.854 INFO 9244 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initial
2021-12-14 19:14:12.224 INFO 9244 --- [main] com.itheima.Chapter04ApplicationTests : Started
Person{id=1, name='tom', hobby=[play, read, sleep], family=[father, mother], map={k2=v2, k1=v1}, pet=Pet{ty
2021-12-14 19:14:12.407 INFO 9244 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting
Process finished with exit code 0
```



自定义YAML文件的加载支持

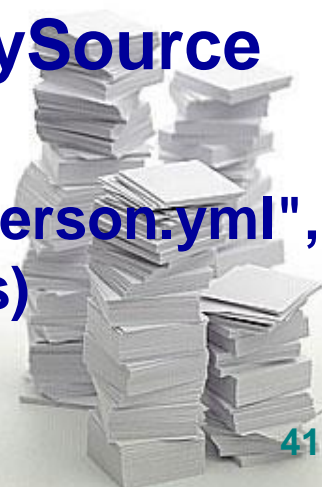
■ **@PropertySource**默认不支持自定义YAML文件的加载。

■ 解决办法：

✧ 构建新的**PropertySource**工厂类（如**YamlPropertySourceFactory**），继承默认的**DefaultPropertySourceFactory**，实现YAML解析

✧ 然后，在需要支持YAML加载的**@PropertySource**注解上指定**factory**即可

- 如：**@PropertySource(value = "classpath:person.yml", factory = YamlPropertySourceFactory.class)**



YamlPropertySourceFactory类

```
import org.springframework.boot.env.YamlPropertySourceLoader;
import org.springframework.core.env.PropertySource;
import org.springframework.core.io.support.DefaultPropertySourceFactory;
import org.springframework.core.io.support.EncodedResource;
import java.io.IOException;
import java.util.List;
public class YamlPropertySourceFactory extends DefaultPropertySourceFactory {
    @Override
    public PropertySource<?> createPropertySource(String name, EncodedResource resource)
throws IOException {
        List<PropertySource<?>> sources = new
YamlPropertySourceLoader().load(resource.getResource().getFilename(),resource.getResource());
        return sources.get(0);
    }
}
```

2) 自定义XML配置文件案例

- 本案例实现自定义XML配置文件加载，并根据配置文件中<bean>标签的设置让Spring容器创建相应业务类的Bean实例。
- 搭建步骤：
 - ✧ ①创建业务类
 - ✧ ②编写自定义XML配置文件
 - ✧ ③加载自定义XML配置文件
 - ✧ ④效果测试



①创建业务类

- 在chapter04项目下新建一个 **com.itheima.service** 包，并在该包下新创建一个业务类 **MyService**，该类中不需要编写任何代码。

```
public class MyService {  
}
```



②编写自定义XML配置文件

- 在**resources**目录下新建一个名为**beans.xml**的XML自定义配置文件，在该配置文件中通过**<beans>**标签向**Spring**容器中添加**MyService**类的对象（**Bean**实例）。

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
<!--将指定类配置给Spring容器，让Spring容器创建一个该类的id为myService的Bean实例-->
```

```
<bean id="myService" class="com.itheima.service.MyService" />
```

```
</beans>
```

③加载自定义XML配置文件

- 在项目启动类上添加 **@ImportResource** 注解加载指定 **XML** 配置文件。

```
@ImportResource("classpath:beans.xml")
```



④效果测试

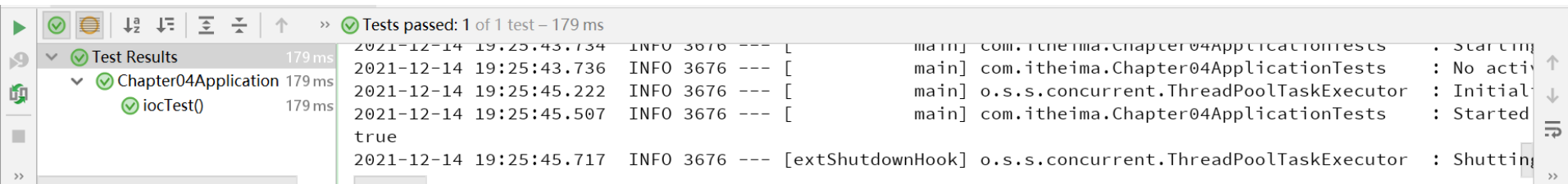
- 在测试类中注入**ApplicationContext**类的**Bean**实例，并新增一个测试方法进行输出测试。

```
@Autowired
private ApplicationContext applicationContext;
@Test
public void iocTest() {
    System.out.println(applicationContext.containsBean("myService"));
}
```



测试结果

■ 测试结果如图所示。



2. 自定义配置类

- 配置类本质上就是传统**Spring**框架中对应的**XML**配置文件
- **Spring Boot** “约定大于配置” 思想更推荐使用配置类代替**XML**配置文件对业务类进行配置（给**Spring**容器）。
- 相关注解：
 - ✧ **@Configuration**：定义一个配置类，并添加为**Spring**容器组件
 - ✧ **@Bean**：将方法返回的对象作为**Bean**实例注入**Spring**容器，方法名即为该**Bean**实例id



自定义配置类案例

■ 本案例实现自定义配置类，实现向**Spring**容器注入业务类的**Bean**实例。

■ 搭建步骤：

✧①创建业务类

- 使用前文的**MyService**类

✧②编写自定义配置类

✧③效果测试

- 使用前文的**iocTest()**方法，测试结果相同
- 测试前事先停止**beans.xml**配置文件的加载



编写自定义配置类

- 在`com.itheima.config`包下新创建一个类 `MyConfig`，使用 `@Configuration` 注解将该类声明为一个配置类。

```
@Configuration
public class MyConfig {
    @Bean
    public MyService myService(){
        return new MyService();
    }
}
```



3.自定义自动配置类

- 自动配置类是指满足特定条件时才被自动启用其配置项或被自动初始化的配置类。
- 主要通过**Spring Boot**的条件注解来实现。



Spring Boot的条件注解

注解名	条件实现类	条件
@ConditionalOnBean	OnBeanCondition	Spring容器中存在指定的实例Bean
@ConditionalOnClass	OnClassCondition	类加载器（类路径）中存在对应的类
@ConditionalOnCloudPlatform	OnCloudPlatformCondition	是否在云平台
@ConditionalOnExpression	OnExpressionCondition	判断SpEL 表达式是否成立
@ConditionalOnJava	OnJavaCondition	指定Java版本是否符合要求
@ConditionalOnJndi	OnJndiCondition	在JNDI（Java命名和目录接口）存在的条件下查找指定的位置
@ConditionalOnMissingBean	OnBeanCondition	Spring容器中不存在指定的实例Bean
@ConditionalOnMissingClass	OnClassCondition	类加载器（类路径）中不存在对应的类
@ConditionalOnNotWebApplication	OnWebApplicationCondition	当前应用程序不是Web程序
@ConditionalOnProperty	OnPropertyCondition	应用环境中属性是否存在指定的值
@ConditionalOnResource	OnResourceCondition	是否存在指定的资源文件。
@ConditionalOnSingleCandidate	OnBeanCondition	Spring容器中是否存在且只存在一个对应的实例Bean
@ConditionalOnWebApplication	OnWebApplicationCondition	当前应用程序是Web程序



自动配置类示例

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;  
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;  
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;  
import org.springframework.boot.context.properties.EnableConfigurationProperties;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
```

```
@EnableConfigurationProperties(HelloServiceProperties.class)
```

```
@ConditionalOnProperty(prefix="hello",value="enabled",matchIfMissing=true)
```

```
@ConditionalOnClass(HelloService.class)
```



自动配置类示例

```
public class HelloServiceAutoConfiguration {  
    @Autowired  
    private HelloServiceProperties helloServiceProperties;  
    @Bean  
    @ConditionalOnMissingBean(HelloService.class)  
    public HelloService helloService(){  
        System.out.println("Execute Create New Bean");  
        HelloService helloService = new HelloService();  
        helloService.setMsg(helloServiceProperties.getMsg());  
        return helloService;  
    }  
}
```

自动配置类HelloServiceAutoConfiguration实现对业务类HelloService进行自动配置。

相关注解

- **@EnableConfigurationProperties(HelloServiceProperties.class)**: 开启对配置文件映射实体类 **HelloServiceProperties** 属性注入
- **@ConditionalOnProperty(prefix="hello",value="enabled",matchIfMissing=true)**: 存在对应属性配置信息（属性前缀为“hello”，启用，缺失检查）时初始化该配置类。
- **@ConditionalOnClass(HelloService.class)**: 当类路径下存在 **HelloService** 类时初始化该配置类。
- **@ConditionalOnMissingBean(HelloService.class)**: 当 **Spring** 容器内不存在 **HelloService** 类的 **Bean** 实例时，初始化 **HelloService** 类。



4.2 Spring Boot自动配置

- Spring Boot应用的启动入口是 `@SpringBootApplication` 注解标注类中的 `main()` 方法;
- `@SpringBootApplication` 能够扫描Spring组件并自动配置Spring Boot。

`@SpringBootApplication`

```
public class Chapter04Application {  
    public static void main(String[] args){  
        SpringApplication.run(Chapter04Application.class,args);  
    }  
}
```

Spring Boot自动配置实现原理

■ **@SpringBootApplication**注解是一个组合注解，包含三个核心注解：

✧ **@SpringBootConfiguration**：标识一个Spring Boot配置类，且该配置类可被组件扫描器扫描

✧ **@EnableAutoConfiguration**：开启自动配置功能

✧ **@ComponentScan**：将指定包中的组件自动装配到Spring容器中



Spring Boot自动配置实现原理

- 所以， Spring Boot自动配置主要是由 **@SpringBootApplication** 通过使用 **@EnableAutoConfiguration** 来实现的。
- **@EnableAutoConfiguration** 也是一个组合注解，包含两个核心注解：
 - ✧ **@AutoConfigurationPackage**: 获取项目启动类所在根目录，即为后续组件扫描器要扫描的包位置
 - ✧ **@Import({AutoConfigurationImportSelector.class})**: 从Spring Boot提供的自动配置类中筛选出当前项目需要启动的自动配置类，从而构建当前项目运行所需的环境

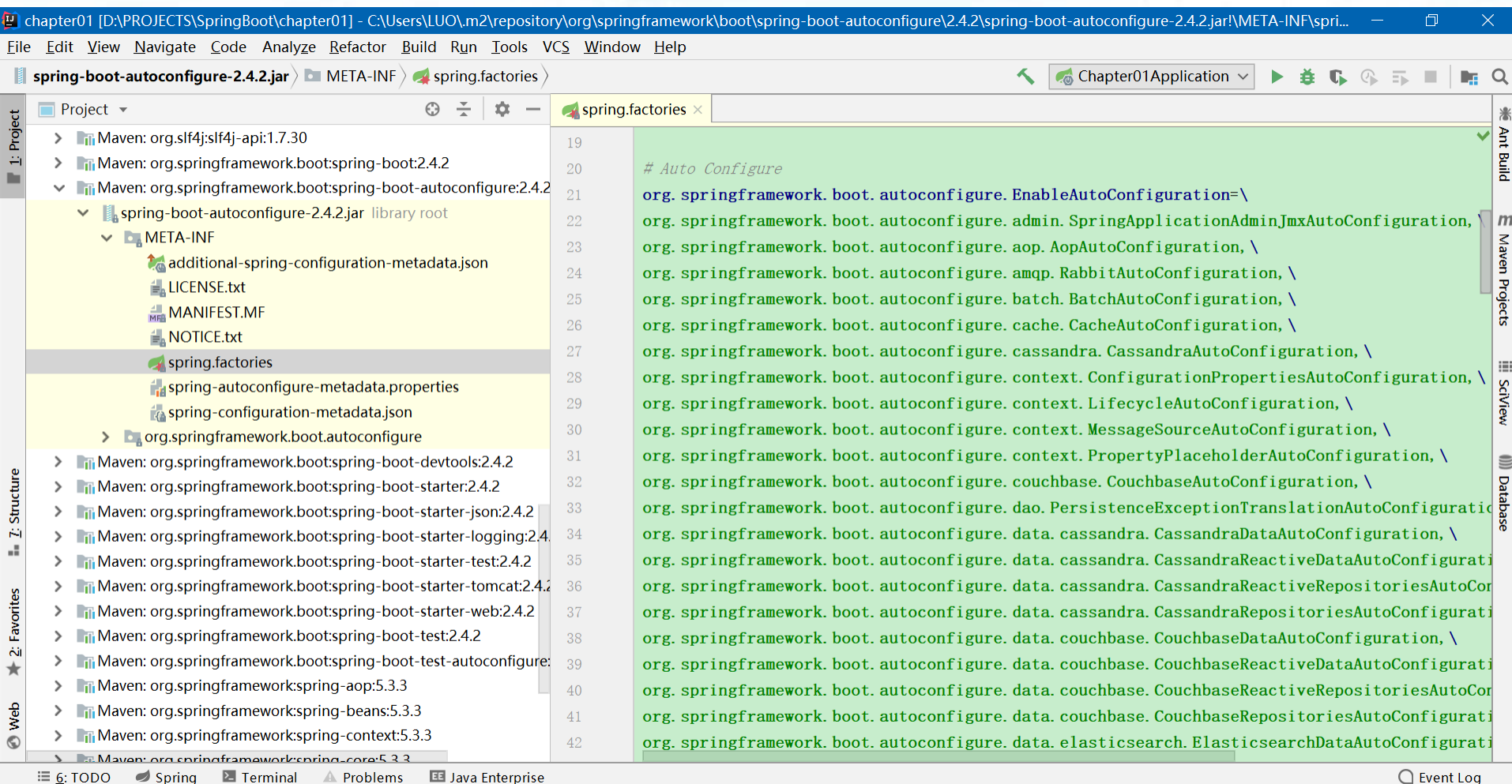
Spring Boot自动配置类的加载

■ 由 **@EnableAutoConfiguration** 实现 **Spring Boot** 自动配置类的加载。

✧ 从 **classpath** 中搜索所有 **META-INF/spring.factories** 配置文件，并将其中 **org.springframework.boot.autoconfigure.EnableAutoConfiguration** 对应的配置项（自动配置类）通过 **Java** 反射机制进行实例化，然后汇总并加载到 **Spring** 容器。



Spring Boot提供的自动配置类



The screenshot displays an IDE window titled "chapter01 [D:\PROJECTS\SpringBoot\chapter01] - C:\Users\LUO\m2\repository\org\springframework\boot\spring-boot-autoconfigure\2.4.2\spring-boot-autoconfigure-2.4.2.jar\META-INF\spring.factories". The left sidebar shows the project structure, with the "spring.factories" file selected under the "META-INF" directory of the "spring-boot-autoconfigure-2.4.2.jar" library root. The main editor shows the content of the "spring.factories" file, which is a list of auto-configuration classes that are enabled by default in Spring Boot. The classes are listed in a single line, separated by commas, and are all fully qualified class names.

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration, \
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration, \
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration, \
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration, \
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration, \
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration, \
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration, \
org.springframework.boot.autoconfigure.context.LifecycleAutoConfiguration, \
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration, \
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration, \
org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration, \
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration, \
org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration, \
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveDataAutoConfiguration, \
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveRepositoriesAutoConfiguration, \
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration, \
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration, \
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveDataAutoConfiguration, \
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositoriesAutoConfiguration, \
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration, \
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration,
```


@SpringBootApplication 内部源码

.....

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan(

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

), @Filter(

 type = FilterType.CUSTOM,
 classes = {AutoConfigurationExcludeFilter.class}

))

)

public @interface SpringBootApplication {

.....

}

@EnableAutoConfiguration 内部源码

.....

```
@AutoConfigurationPackage
```

```
@Import({AutoConfigurationImportSelector.class})
```

```
public @interface EnableAutoConfiguration {
```

```
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
```

```
    Class<?>[] exclude() default {};
```

```
    String[] excludeName() default {};
```

```
}
```



4.3 Spring Boot依赖管理

<!-- 引入Spring Boot依赖的父包，为项目提供统一的子依赖版本管理-->

<parent>

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

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

<version>2.1.3.RELEASE</version>

父依赖启动器

</parent>

<dependencies>

<!-- 引入Web场景依赖启动器 -->

<dependency>

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

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

</dependency>

Web依赖启动器

</dependencies>



Spring Boot依赖管理

- **spring-boot-starter-parent**的主要作用是通过<**properties**>标签对一些常用技术框架的依赖文件进行了统一版本号管理。
- **spring-boot-starter-web**的主要作用是提供**Web**开发场景所需的底层所有依赖文件，它对**Web**开发场景所需的依赖文件进行了统一管理。
 - ✧通过该依赖启动器实现了**Spring Boot**与**web**开发技术框架的整合



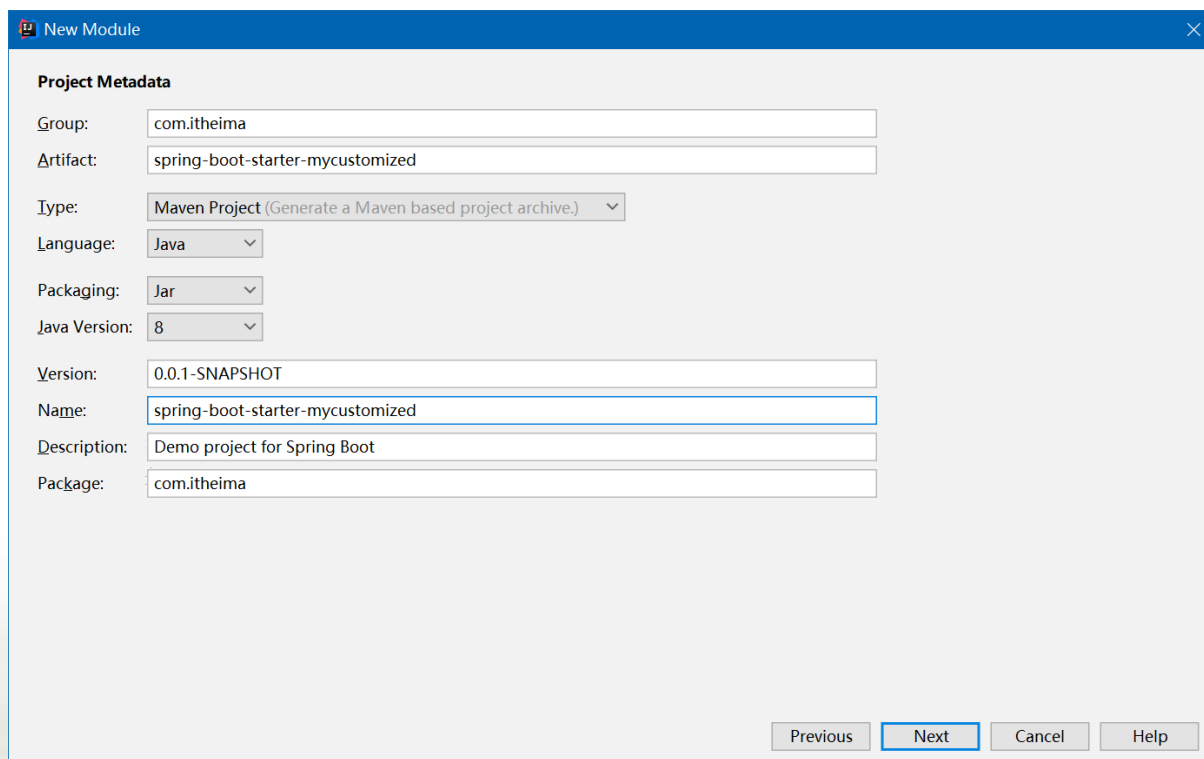
1.自定义依赖启动器

- **Spring Boot**官方为许多开发场景提供了依赖启动器来整合这些开发场景的技术框架。
- 开发者也可自定义依赖启动器（官方命名格式为：**spring-boot-starter-{name}**）来与**Spring Boot**框架进行整合。
- 自定义依赖启动器搭建步骤：
 - ✧ ①新建**Spring Boot**模块
 - ✧ ②修改**pom.xml**文件
 - ✧ ③创建配置文件映射实体类
 - ✧ ④创建业务类
 - ✧ ⑤创建自动配置类
 - ✧ ⑥注册自动配置类



①创建Spring Boot模块

- 在chapter04项目中，使用Spring Initializr方式创建一个Spring Boot模块myspringbootstarter。



New Module

Project Metadata

Group: com.itheima

Artifact: spring-boot-starter-mycustomized

Type: Maven Project (Generate a Maven based project archive.)

Language: Java

Packaging: Jar

Java Version: 8

Version: 0.0.1-SNAPSHOT

Name: spring-boot-starter-mycustomized

Description: Demo project for Spring Boot

Package: com.itheima

Previous Next Cancel Help

②修改pom.xml文件

■ 修改myspringbootstarter模块的pom.xml文件，增加Spring Boot的自动配置依赖。

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6     <parent>
7         <!-- 引入父依赖启动器，为项目提供统一的子依赖版本管理 -->
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-parent</artifactId>
10        <version>2.4.2</version>
11        <relativePath/> <!-- lookup parent from repository -->
12    </parent>
13    <groupId>com.itheima</groupId>
14    <artifactId>spring-boot-starter-mycustomized</artifactId>
15    <version>1.0-SNAPSHOT</version>
16    <packaging>jar</packaging>
17    <properties>
18        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
19    </properties>
```

修改pom.xml文件

```
20 <dependencies>
21     <!-- spring boot 自动配置需要的包 -->
22     <dependency>
23         <groupId>org.springframework.boot</groupId>
24         <artifactId>spring-boot-autoconfigure</artifactId>
25     </dependency>
26     <!-- Spring Boot提供的配置处理器依赖 -->
27     <dependency>
28         <groupId>org.springframework.boot</groupId>
29         <artifactId>spring-boot-configuration-processor</artifactId>
30         <optional>true</optional>
31     </dependency>
32 </dependencies>
33 </project>
```



③创建配置文件映射实体类

- 在myspringbootstarter模块的com.itheima包下新创建一个配置文件映射实体类HelloServiceProperties。



HelloServiceProperties类

```
import org.springframework.boot.context.properties.ConfigurationProperties;
```

```
//在引用项目的application.properties中通过"hello.msg="设置属性值
```

```
@ConfigurationProperties(prefix="hello")
```

```
public class HelloServiceProperties {
```

```
    //设置msg的默认值
```

```
    private String msg = "World";
```

```
    public String getMsg() {
```

```
        return msg;
```

```
    }
```

```
    public void setMsg(String msg) {
```

```
        this.msg = msg;
```

```
    }
```

```
}
```

④创建业务类

■ 在myspringbootstarter模块的com.itheima包下新创建一个业务类HelloService。

```
public class HelloService {  
    private String msg;  
    public String haloHello(){  
        return "Hello Starter =====>>>>" + msg;  
    }  
    public String getMsg() {  
        return msg;  
    }  
    public void setMsg(String msg) {  
        this.msg = msg;  
    };  
}
```

⑤创建自动配置类

- 在myspringbootstarter模块的com.itheima包下新创建一个自动配置类
HelloServiceAutoConfiguration。



HelloServiceAutoConfiguration类

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;  
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;  
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;  
import org.springframework.boot.context.properties.EnableConfigurationProperties;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
```

```
@EnableConfigurationProperties(HelloServiceProperties.class)
```

```
@ConditionalOnProperty(prefix="hello",value="enabled",matchIfMissing=true)
```

```
@ConditionalOnClass(HelloService.class)
```



HelloServiceAutoConfiguration类

```
public class HelloServiceAutoConfiguration {  
    @Autowired  
    private HelloServiceProperties helloServiceProperties;  
    @Bean  
    @ConditionalOnMissingBean(HelloService.class)  
    public HelloService helloService(){  
        System.out.println("Execute Create New Bean");  
        HelloService helloService = new HelloService();  
        helloService.setMsg(helloServiceProperties.getMsg());  
        return helloService;  
    }  
}
```

自动配置类实现对业务类进行自动配置。



⑥注册自动配置类

- 在模块的**src/main/resources**目录下，创建**META-INF/spring.factories**文件，内容如下

```
# Auto Configure  
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\ncom.itheima.HelloServiceAutoConfiguration
```

- 上述文件内容中，若有多个自动配置类，则使用“,”分开。



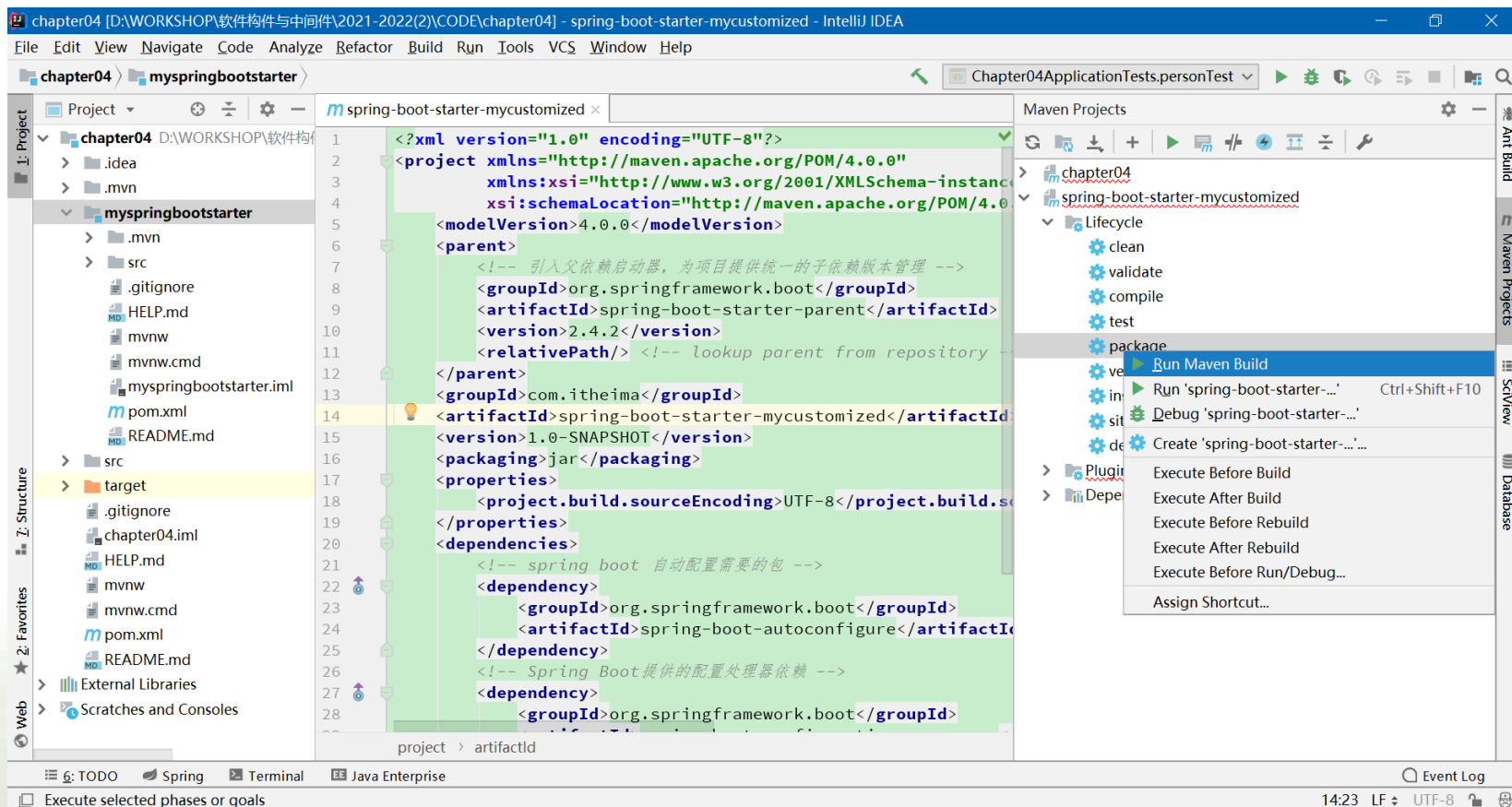
2.安装自定义依赖启动器

- 自定义依赖启动器完成后，就可将其安装到Maven本地仓库了。
- 搭建步骤：
 - ✧①生成模块jar包
 - ✧②执行mvn install安装



①生成模块jar包

■ 按如图操作，jar包会生成在myspringbootstarter模块的target目录下。



生成的jar包

chapter04 [D:\WORKSHOP\软件构件与中间件\2021-2022(2)\CODE\chapter04] - spring-boot-starter-mycustomized - IntelliJ IDEA

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

chapter04 myspringbootstarter target spring-boot-starter-mycustomized-1.0-SNAPSHOT.jar Chapter04ApplicationTests.personTest

Project

- myspringbootstarter
 - .mvn
 - src
 - target
 - classes
 - generated-sources
 - generated-test-sources
 - maven-archiver
 - maven-status
 - test-classes
- spring-boot-starter-mycustomized-1.0-SNAPSHOT.jar
- .gitignore
- HELP.md

Run: spring-boot-starter-mycustomized [package] x

```
[INFO] --- maven-jar-plugin:3.2.0:jar (default-jar) @ spring-boot-starter-mycustomized ---
[INFO] Building jar: D:\WORKSHOP\软件构件与中间件\2021-2022(2)\CODE\chapter04\myspringbootstarter\target\spring-boot-starter-mycustomized-1.0-SNAPSHOT.jar
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.333 s
[INFO] Finished at: 2021-12-14T21:42:48+08:00
[INFO] Final Memory: 27M/213M
[INFO] -----

Process finished with exit code 0
```

spring-boot-starter-mycustomized x

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6     <parent>
7         <!-- 引入父依赖启动器，为项目提供统一的子依赖版本管理 -->
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-parent</artifactId>
10        <version>2.4.2</version>
11        <relativePath/> <!-- lookup parent from repository -->
12    </parent>
13    <groupId>com.itheima</groupId>
14    <artifactId>spring-boot-starter-mycustomized</artifactId>
15    <version>1.0-SNAPSHOT</version>
16    <packaging>jar</packaging>
17    <name>spring-boot-starter-mycustomized</name>
18    <description>spring-boot-starter-mycustomized</description>
19    <url>http://maven.apache.org</url>
20    <licenses>
21        <license>
22            <name>The Apache Software License, Version 2.0</name>
23            <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
24        </license>
25    </licenses>
26    <developers>
27        <developer>
28            <id>spring</id>
29            <name>Spring</name>
30            <email>spring@spring.io</email>
31            <organization>Spring</organization>
32            <organizationUrl>http://spring.io</organizationUrl>
33        </developer>
34    </developers>
35    <scm>
36        <url>http://maven.apache.org</url>
37        <connection>scm:svn:http://maven.apache.org/svn</connection>
38        <developerConnection>scm:svn:http://maven.apache.org/svn</developerConnection>
39    </scm>
40    <properties>
41        <java.version>1.8</java.version>
42        <spring.version>5.3.12</spring.version>
43    </properties>
44    <dependencies>
45        <dependency>
46            <groupId>org.springframework.boot</groupId>
47            <artifactId>spring-boot-starter</artifactId>
48            <scope>compile</scope>
49        </dependency>
50        <dependency>
51            <groupId>org.springframework.boot</groupId>
52            <artifactId>spring-boot-starter-test</artifactId>
53            <scope>test</scope>
54        </dependency>
55    </dependencies>
56    <build>
57        <plugins>
58            <plugin>
59                <groupId>org.springframework.boot</groupId>
60                <artifactId>spring-boot-maven-plugin</artifactId>
61            </plugin>
62        </plugins>
63    </build>
64    <profiles>
65        <profile>
66            <id>dev</id>
67            <activation>
68                <activeByDefault>true</activeByDefault>
69                <os>
70                    <name>Windows</name>
71                </os>
72            </activation>
73            <properties>
74                <spring.profiles.active>dev</spring.profiles.active>
75            </properties>
76        </profile>
77    </profiles>
78    <reporting>
79        <plugins>
80            <plugin>
81                <groupId>org.apache.maven.plugins</groupId>
82                <artifactId>maven-project-info-reports-plugin</artifactId>
83            </plugin>
84        </plugins>
85    </reporting>
86    <test>
87        <resources>
88            <resource>
89                <directory>src/test/resources</directory>
90            </resource>
91        </resources>
92        <suites>
93            <suite>
94                <name>JUnit</name>
95                <test>org.junit.runner.JUnit4</test>
96            </suite>
97        </suites>
98    </test>
99    <repositories>
100    </repositories>
101</project>
```

Ant Build Maven Projects ScView Database

4: Run 6: TODO Spring Terminal Java Enterprise

Event Log 14:23 LF UTF-8

②执行mvn install安装

■在cmd界面中，执行以下命令后，自定义依赖启动器即将会安装到事先设置好的Maven本地仓库中

✧**mvn install:install-file -Dfile=[你的模块工作目录]\target\spring-boot-starter-mycustomized-1.0-SNAPSHOT.jar -DgroupId=com.itheima -DartifactId=spring-boot-starter-mycustomized -Dversion=1.0-SNAPSHOT -Dpackaging=jar**



3.测试自定义依赖启动器

■ 搭建步骤:

- ✧①引入自定义依赖启动器
- ✧②编写测试类
- ✧③查看测试结果



①引入自定义依赖启动器

■在chapter04项目的pom.xml中引入自定义依赖启动器

```
<!--引入自定义starter-->
```

```
<dependency>
```

```
    <groupId>com.itheima</groupId>
```

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

```
    <version>1.0-SNAPSHOT</version>
```

```
</dependency>
```



②编写测试类

- 在chapter04项目下新建一个 **com.itheima.controller**包，并在该包下新建一个控制器类**HelloController**。
- 然后在该项目的全局配置文件 **application.properties**中设置 **hello.msg=Welcome**



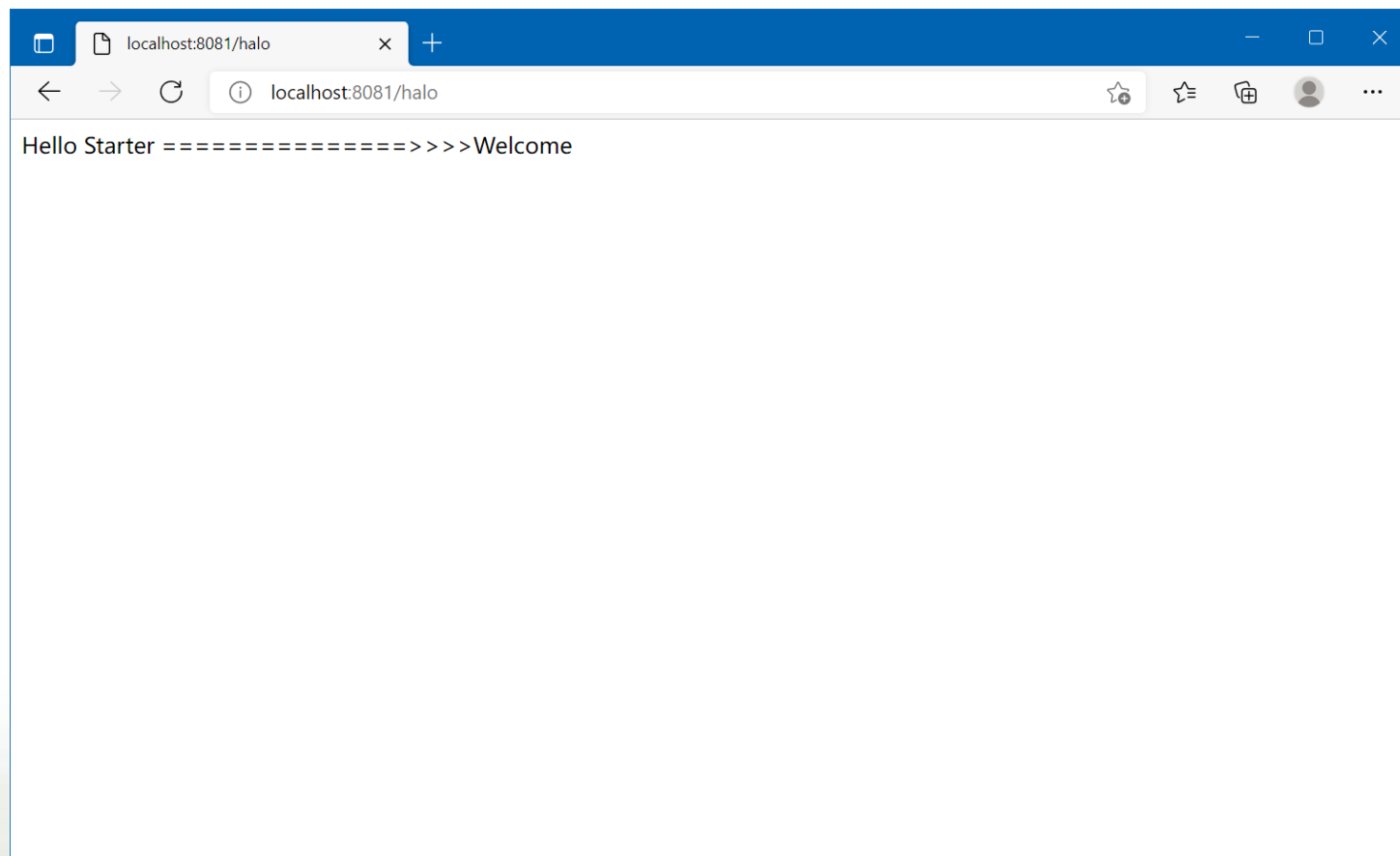
HelloController类

```
import com.itheima.HelloService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {
    @Autowired
    private HelloService helloService;
    @GetMapping("/halo")
    public String index(){
        return helloService.haloHello();
    }
}
```

③查看测试结果

■ “dev”环境下测试结果，如图所示



4.4 Spring Boot项目执行流程

`@SpringBootApplication`

标记该类为项目启动类

```
public class Chapter04Application {  
    public static void main(String[] args){
```

```
        SpringApplication.run(Chapter04Application.class,args);
```

```
    }
```

```
}
```

SpringApplication.run()方法启动项目启动类



SpringApplication 内部源码

```
.....  
public class SpringApplication {  
.....  
    public static ConfigurableApplicationContext run(Class<?> primarySource, String... args) {  
        return run(new Class[]{primarySource}, args);  
    }  
    public static ConfigurableApplicationContext run(Class<?>[] primarySources, String[] args){  
        return (new SpringApplication(primarySources)).run(args);  
    }  
.....  
}
```



Spring Boot项目执行流程

■ Spring Boot项目的执行流程主要分为两步：

✧ 1. 初始化SpringApplication实例

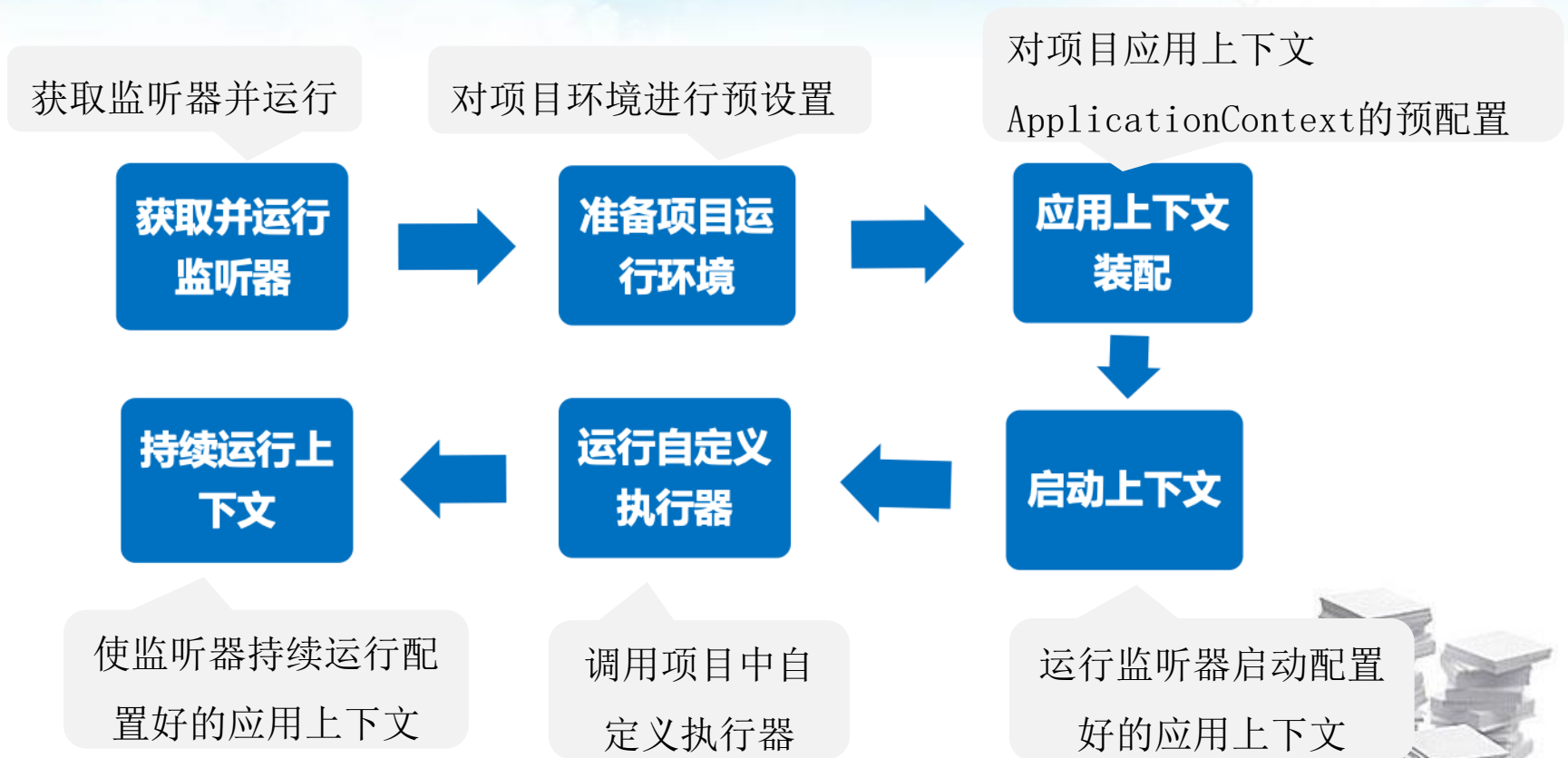
✧ 2. 调用run()启动项目



1.初始化SpringApplication实例



2.调用run()启动项目



本章小结

■本章具体讲解了：

✧4.1 Spring Boot核心配置

- Spring Boot全局属性配置
- Spring Boot自定义配置

✧4.2 Spring Boot自动配置

✧4.3 Spring Boot依赖管理

✧4.4 Spring Boot项目执行流程



