Spring Boot 2.0 的配置详解（图文教程）

Spring Boot 配置，包括自动配置和外化配置。本文先实现自定义属性工程，将属性外化配置在 application.properties 应用配置文件，然后在工程中获取该属性值。接着会详细介绍属性的获取方式、外化配置和自动配置。最后会介绍利用自动配置自定义 Start 组件。

快速入门工程

HelloBookController 控制层中，在代码中以硬编码的方式使用字符串表示书信息。下面把书的信息作为属性，外化配置在 application.properties。好处是将应用参数、业务参数或第三方参数等统一配置在应用配置文件中，避免配置侵入业务代码，达到可配置的方式，方便及时调整修改。

配置属性

新建工程命名为 chapter-2-spring-boot-config ，在 application.properties 中配置书名和作者，配置如下：

```
## 书信息
demo.book.name=[Spring Boot 2.x Core Action]
demo.book.writer=BYSocket
```

.properties 文件的每行参数被存储为一对字符串，即一个存储参数名称，被称为键；另一个为值。一般称为键值对配置。井号（#）或者英文状态下的叹号（!）作为第一行中第一个非空字符来表示该行的文本为注释。另外，反斜杠（\）用于转义字符。

Spring Boot 支持并推荐使用 YAML 格式的配置文件，将 application.properties 文件替换成 application.yml 文件，并配置相同的属性，配置如下：

```
## 书信息
demo:
    book:
        name: 《Spring Boot 2.x 核心技术实战 - 上 基础篇》
        writer: 泥瓦匠BYSocket
```

YAML 是一个可读性高，用来表达数据序列的格式。表示键值对格式时，注意键和值由冒号及空白字符分开。强调下，空白字符是必须的，IDE 一般也会提示。两种配置方式都

非常便捷，在开发中选择 .properties 或 .yml 文件配置。但如果两种配置文件同时存在的时候，默认优先使用 .properties 配置文件。YAML 与 .properties 配置文件对比如图 1 所示。

配置文件	可读性	便捷性	是否支持嵌套对象
YAML	☑	☑	☑
properties	☑	☑	☐

图 1 YAML 与 .properties 配置文件对比

注意：在 application.properties 配置中文值，读取时会出现中文乱码问题。因为 Java .properties 文件默认编码方式是 iso-8859 ，Spring Boot 应用以 UTF-8 的编码方式读取，就导致出现乱码问题。

官方 Issue 中的解决方法是，将 .properties 文件中配置的中文值转义成 Unicode 编码形式。例如“demo.book.writer= 泥瓦匠”应该配置成“demo.book.writer=\u6ce5\u74e6\u5320”。利用 IDEA properties 插件或利用 Java 文件转码工具 native2ascii 来快速地进行转义。该工具有在线版实现，点击这里获得访问地址。

创建属性类

在工程中新建包目录 demo.springboot.config，并在目录中创建名为 BookProperties 的属性类，代码如下：

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

/**
 * 书属性
 */
@Component
public class BookProperties {

    /**
     * 书名
     */
    @Value("${demo.book.name}")
    private String name;

    /**
     * 作者
     */
    @Value("${demo.book.writer}")
    private String writer;
```

```
    // ... 省略 getter / setter 方法
}
```

利用 @Component 注解定义了书的属性 Bean，并通过 @Value 注解为该 Bean 的成员变量（或者方法参数）自动注入 application.properties 文件的属性值。@Value 注解是通过“${propName}”的形式引用属性，propName 表示属性名称。

核心注解的知识点有以下几个。

- @Component 注解。

@Component 对类进行标注，职责是泛指组件 Bean，应用启动时会被容器加载并加入容器管理。常见的 @Controller、@Service、@Repository 是 @Component 的分类细化组件，分别对应控制层、服务层、持久层的 Bean。

- @Value 注解。

@Value 对 Bean 的字段或者方法参数进行标注，职责是基于表达式给字段或方法参数设置默认属性值。通常格式是注解 + SpEL 表达式，如 @Value(“SpEL 表达式”)。

使用 @Vlaue 注解来引用属性值时，确保所引用的属性值在 application.properties 文件存在并且相对应匹配，否则会造成 Bean 的创建错误，引发 java.lang.IllegalArgumentException 非法参数异常。

获取属性

修改原有的 HelloBookController 类，通过注入的方式获取书属性 Bean 并返回。代码如下。

```
import demo.springboot.config.BookProperties;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloBookController {

    @Autowired
    BookProperties bookProperties;

    @GetMapping("/book/hello")
    public String sayHello() {
        return "Hello,  " + bookProperties.getWriter() + " is
writing "
                + bookProperties.getName() + " ! ";
    }
}
```

通过 @Autowired 注解标记在 BookProperties 字段，控制层自动装配属性 Bean 并使用。默认情况下要求被注解的 Bean 必须存在，需要允许 NULL 值，可以设置其 required 属性为 false，即 @Autowired(required = false)。

运行工程

执行 ConfigApplication 类启动，在控制台看到成功运行的输出后，打开浏览器访问 /book/hello 地址，可以看到如图 2 所示的返回结果：

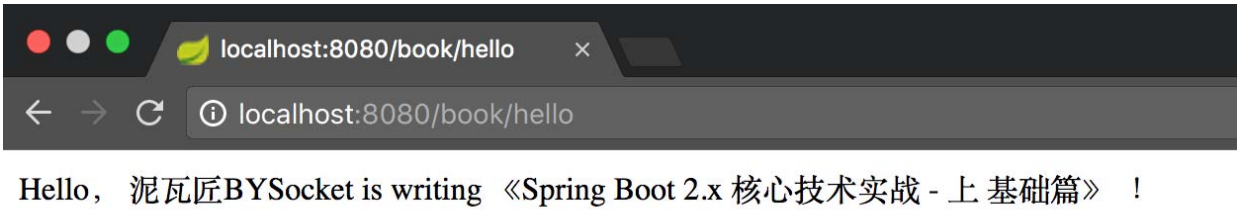


图 2 Hello Book 页面

也可以通过单元测试的方式验证属性获取是否成功。单元测试代码如下：


```
import demo.springboot.config.BookProperties;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class ConfigApplicationTests {
    @Autowired
    BookProperties bookProperties;

    @Test
    public void testBookProperties() {
        Assert.assertEquals(bookProperties.getName(),"'Spring
Boot 2.x Core Action'");

Assert.assertEquals(bookProperties.getWriter(),"BYSocket");
    }
}
```

配置属性的获取方式

配置属性的常用获取方式有基于 @Value 和 @ConfigurationProperties 注解两种方式。两种方式适合的场景不同，下面具体介绍其使用方法和场景。

@Value 注解

@Value 注解对 Bean 的变量或者方法参数进行标注，职责是基于表达式给字段或方法参数设置默认属性值。通常格式是注解 + SpEL 表达式，如 @Value(“SpEL 表达式”)，并标注在对应的字段或者方法上方，且必须对变量一一标注。这种方式适用于小而不复杂的属性结构。属性结构复杂，字段很多的情况下，这种方式会比较繁琐，应该考虑使用 @ConfigurationProperties 注解。

另外通过 @PropertySource 注解引入对应路径的其他 .properties 文件。将书信息重新配置在 classpath 下新的 book.properties 配置文件后，读取新配置文件的代码如下：

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.PropertySource;
import org.springframework.stereotype.Component;

/**
 * 书属性
 */
@Component
@PropertySource("classpath:book.properties")
public class BookProperties {

    /**
     * 书名
     */
    @Value("${demo.book.name}")
    private String name;

    /**
     * 作者
     */
    @Value("${demo.book.writer}")
    private String writer;

    // ... 省略 getters / setters 方法
}
```


@ConfigurationProperties 注解

在包目录 demo.springboot.config 中创建名为 BookComponent 的属性类，并使用 @ConfigurationProperties 注解获取属性，代码如下：

```
import
org.springframework.boot.context.properties.ConfigurationProperti
es;
import org.springframework.stereotype.Component;

/**
 * 书属性
 *
 */
@Component
@ConfigurationProperties(prefix = "demo.book")
public class BookComponent {

    /**
     * 书名
     */
    private String name;

    /**
     * 作者
     */
    private String writer;

    // ... 省略 getters / setters 方法
}
```


类似 @Value 注解方式，使用 @ConfigurationProperties(prefix = “demo.book”) 注解标注在类上方可以达到相同的效果。@ConfigurationProperties 注解的 prefix 是指定属性的参数名称。会匹配到配置文件中“demo.book.”*结构的属性，星号“”*是指会一一对应匹配 BookComponent 类的字段名。例如，字段 name 表示书名，会匹配到 demo.book.name 属性值。

@Value 注解方式强制字段必须对应在配置文件，@ConfigurationProperties 注解方式则不是必须的。一般情况下，所有字段应该保证一一对应在配置文件。如果没有属性值对应的话，该字段默认为空，@ConfigurationProperties 注解方式也不会引发任何异常，Spring Boot 推荐使用 @ConfigurationProperties 注解方式获取属性。

同样使用单元测试验证获取属性是否成功。单元测试代码如下：

```
@Autowired
BookComponent bookComponent;

@Test
public void testBookComponent() {
    Assert.assertEquals(bookComponent.getName(),"'Spring Boot 2.x
Core Action'");
    Assert.assertEquals(bookComponent.getWriter(),"BYSocket");
}
```

API org.springframework.boot.context.properties.ConfigurationProperties 注解参数有如下几个。

- prefix：字符串值，绑定该名称前缀的属性对象。
- value：字符串值，功能同 prefix 参数。
- ignoreInvalidFields：布尔值，默认 false。绑定对象时，忽略无效字段。
- ignoreUnknownFields：布尔值，默认 true。绑定对象时，忽略未知字段。

@ConfigurationProperties 数据验证

@ConfigurationProperties 注解方式支持验证功能，即当属性类被 @Validated 注解标注时，Spring Boot 初始化时会验证类的字段。在类的字段上添加 JSR-303 约束注解，进行数据验证。下面为书属性字段添加非 NULL 和字符串非空约束，代码如下：

```
import
org.springframework.boot.context.properties.ConfigurationProperti
es;
import org.springframework.stereotype.Component;
import org.springframework.validation.annotation.Validated;

import javax.validation.constraints.NotEmpty;
import javax.validation.constraints.NotNull;

/**
 * 书属性
 *
 */
@Component
@ConfigurationProperties(prefix = "demo.book")
@Validated
public class BookComponent {

    /**
     * 书名
     */
    @NotEmpty
    private String name;

    /**
     * 作者
     */
    @NotNull
    private String writer;

    // ... 省略 getters / setters 方法
}
```

通过 @Validated 注解开启对 BookComponent 类字段的数据验证，如果 name 字段为 NULL 或者为空字符串时，会引发 BindValidationException 绑定数据验证异常。数据验证常用在邮箱格式或者有长度限制的属性字段。另外，验证嵌套属性的值，必须在嵌套对象字段上方标注 @Valid 注解，用来触发其验证。例如，在书属性中新增嵌套对象出版社 Publishing，就需要在该对象上方标注 @Valid 注解，来开启对 Publishing 对象的数据验证。综上，两种属性获取方式各有优缺点，对比如图 3 所示：

#	@ConfigurationProperties	@Value
适用于成组 / 复杂的结构化对象	☑	☐
是否支持数据校验	☑	☐
支持 SpEL 表达式	☐	☐

图 3 @ConfigurationPropertiesd vs @Value

外化配置

Spring Boot 可以将配置外部化，即分离存储在 classpath 之外，这种模式叫做 “外化配置”。常用在不同环境中，将配置从代码中分离外置，只要简单地修改下外化配置，可以依旧运行相同的应用代码。外化配置表现形式不单单是 .properties 和 .yml 属性文件，还可以使用环境变量和命令行参数等来实现。那么，多处配置了相同属性时，Spring Boot 是通过什么方式来控制外化配置的冲突呢？答案是外化配置优先级。

外化配置优先级

用命令行配置去覆盖 .properties 文件配置方法很简单。正常情况下利用 Java 命令运行工程，代码如下：

```
// chapter-2-spring-boot-config 目录下运行
java -jar target/chapter-2-spring-boot-config-1.0.jar
```

下面将书的作者信息 BYSocket 改成 Jeff，通过命令行配置覆盖属性，代码如下：

```
java -jar target/chapter-2-spring-boot-config-1.0.jar --
demo.book.writer=Jeff
```

在命令行配置中，设置属性值的格式是用两个连续的减号“-”标志属性。在控制台看到成功运行的输出后，打开浏览器，访问 /book/hello 地址，可以看到如图 4 所示的返回结果：

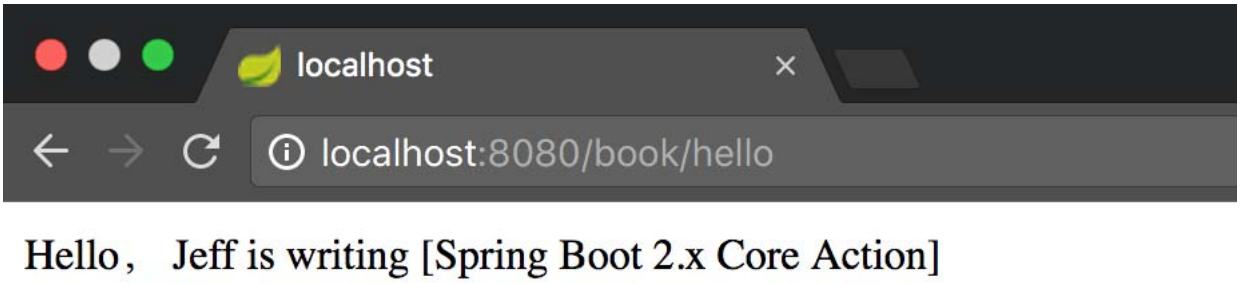


图 4 书信息被覆盖页面

通过命令行配置覆盖属性提供了非常大的作用与便利性，常见于使用 shell 脚本运行工程时，可以方便地修改工程运行的配置。

但是这就引发了一个问题，岂不让工程很有侵入性，如果开放这个功能，导致未知的安全问题。所以 Spring Boot 提供了屏蔽命令行属性值设置，在应用启动类中设置 setAddCommandLineProperties 方法为 false，用于关闭命令行配置功能，代码如下：

```
SpringApplication.setAddCommandLineProperties(false);
```


命令行配置属性的优先级是第四。外化配置获取属性时，会按优先级从高到低获取。如果高优先级存在属性，则返回属性，并忽略优先级低的属性。优先级如下：

1. 本地 Devtools 全局配置
2. 测试时 @TestPropertySource 注解配置
3. 测试时 @SpringBootTest 注解的 properties 配置
4. 命令行配置
5. SPRING_APPLICATION_JSON 配置
6. ServletConfig 初始化参数配置
7. ServletContext 初始化参数配置
8. Java 环境的 JNDI 参数配置
9. Java 系统的属性配置
10. OS 环境变量配置
11. 只能随机属性的 RandomValuePropertySource 配置
12. 工程 jar 之外的多环境配置文件（application-{profile}.properties 或 YAML）
13. 工程 jar 之内的多环境配置文件（application-{profile}.properties 或 YAML）
14. 工程 jar 之外的应用配置文件（application.properties 或 YAML）
15. 工程 jar 之内的应用配置文件（application.properties 或 YAML）
16. @Configuration 类中的 @PropertySource 注解配置
17. 默认属性配置（SpringApplication.setDefaultProperties 指定）

属性引用

在 application.properties 中配置属性时，属性之间可以直接通过“${propName}”的形式引用其他属性。比如新增书的描述 description 属性，代码如下：

```
## 书信息
demo.book.name=[Spring Boot 2.x Core Action]
demo.book.writer=BYSocket
demo.book.description=${demo.book.writer}'s${demo.book.name}
```

demo.book.description 属性引用了前面定义的 demo.book.name 和 demo.book.writer 属性，其值为 BYSocket's[Spring Boot 2.x Core Action]。一方面可以使相同配置可以复用，另一方面增强了配置的阅读性。

使用随机数

在 application.properties 中配置属性时，可以使用随机数配置，例如注入某些密钥、UUID 或者测试用例，需要每次不是一个固定的值。RandomValuePropertySource 类随机提供整形、长整形数、UUID 或者字符串。使用代码如下：

```
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]}
```


多环境配置

多环境是指不同配置的生产服务器使用同一工程代码部署，比如开发环境、测试环境、预发环境、生产环境等。各个环境的工程端口、数据库配置、Redis 配置、日志配置等都会不同，传统模式下需要修改配置，工程重新编译打包，并部署到指定环境服务器。结果是容易发生配置错误，导致开发部署效率低下。Spring Boot 使用多环境配置去解决这个问题。

多环境配置，类似 Maven 构建配置文件的思路，即配置多个不同环境的配置文件，再通过 spring.profiles.active 命令去指定读取特定配置文件的属性。多环境配置文件是不同于 application.properties 应用配置文件。多环境配置文件的约定命名格式为 application-{profile}.properties。多环境配置功能默认为激活状态，如果其他配置未被激活，则 {profile} 默认为 default，会加载 application-default.properties 默认配置文件，没有该文件就会加载 application.properties 应用配置文件。

多环境配置文件的属性读取方式和从 application.properties 应用配置文件读取方式一致。不管多环境配置文件在工程 jar 包内还是包外，按照配置优先级覆盖其他配置文件。

在微服务实践开发中，经常会使用一个类似 deploy 工程去管理配置文件和打包其他业务工程。

在 application.properties 同级目录中，新建 application-dev.properties 作为开发环境配置文件，配置如下：

```
## 书信息
demo.book.name=[Spring Boot 2.x Core Action]  From Dev
demo.book.writer=BYSocket
```

新建 application-prod.properties 作为生产环境配置文件，代码如下：

```
## 书信息
demo.book.name=<Spring Boot 2.x Core Action Dev> From Prod
demo.book.writer=BYSocket
```

通过命令行指定读取 dev 环境配置文件并运行工程，代码如下：

```
java -jar target/chapter-2-spring-boot-config-1.0.jar --
spring.profiles.active=dev
```


在多个环境配置中，通过命令“–spring.profiles.active=dev”指定读取某个配置文件，将 dev 更改成 prod ，轻松切换读取生产环境配置。也可以在控制台的日志中确定配置读取来自 dev：

```
2017-11-09 12:10:52.978  INFO 72450 --- [           main]
demo.springboot.ConfigApplication        : The following profiles
are active: dev
```

最后打开浏览器，访问 /book/hello 地址，可以看到如图 5 所示的返回结果：

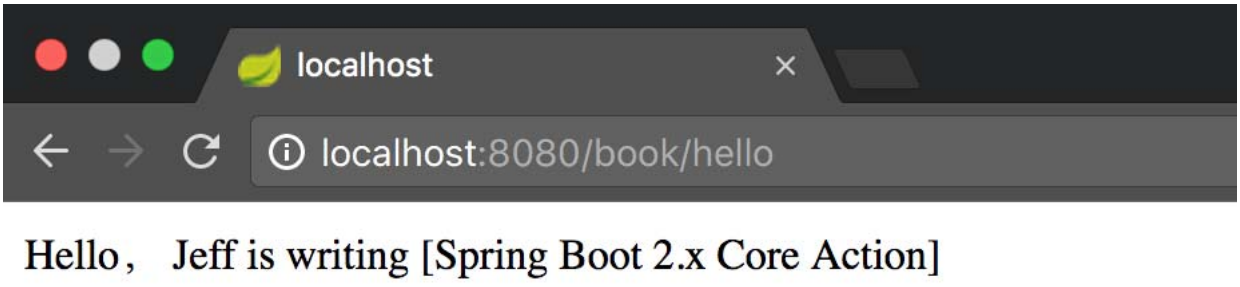


图 5 dev 环境书信息页面

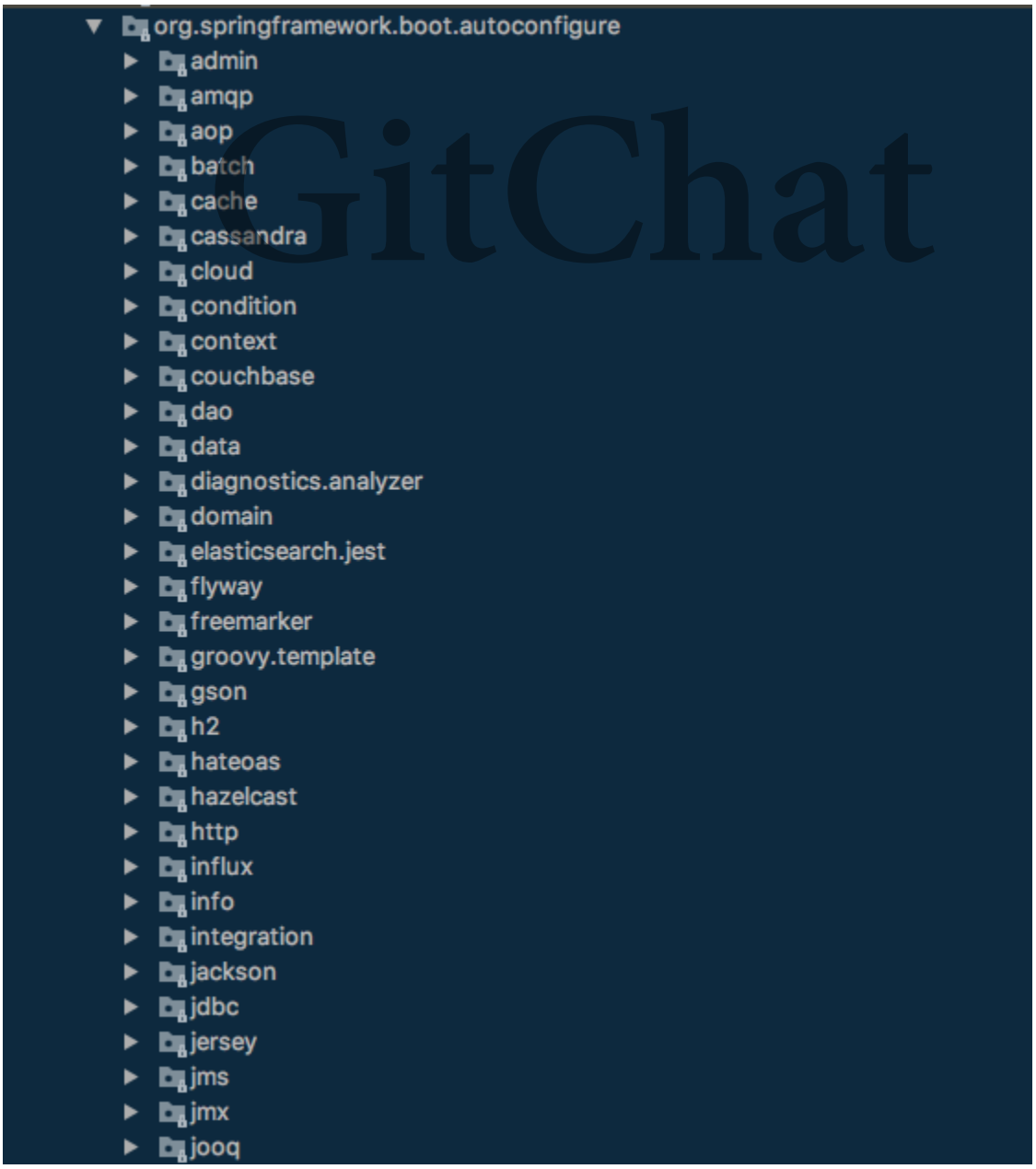
4 自动配置

Spring Boot `spring-boot-autoconfigure` 依赖实现了默认的配置项，即应用默认值。这种模式叫做“自动配置”。Spring Boot 自动配置会根据添加的依赖，自动加载依赖相关的配置属性并启动依赖。例如默认用的内嵌式容器是 Tomcat ，端口默认设置为 8080。

为什么需要自动配置？顾名思义，自动配置的意义是利用这种模式代替了配置 XML 繁琐模式。以前使用 Spring MVC，需要进行配置组件扫描、调度器、视图解析器等，使用 Spring Boot 自动配置后，只需要添加 MVC 组件即可自动配置所需要的 Bean。所有自动配置的实现都在 spring-boot-autoconfigure 依赖中，包括 Spring MVC 、 Data 和其它框架的自动配置。

4.1 spring-boot-autoconfigure 依赖

spring-boot-autoconfigure 依赖，是 Spring Boot 实现自动配置的核心 Starter 组件。其实现源码包结构如图 6 所示：



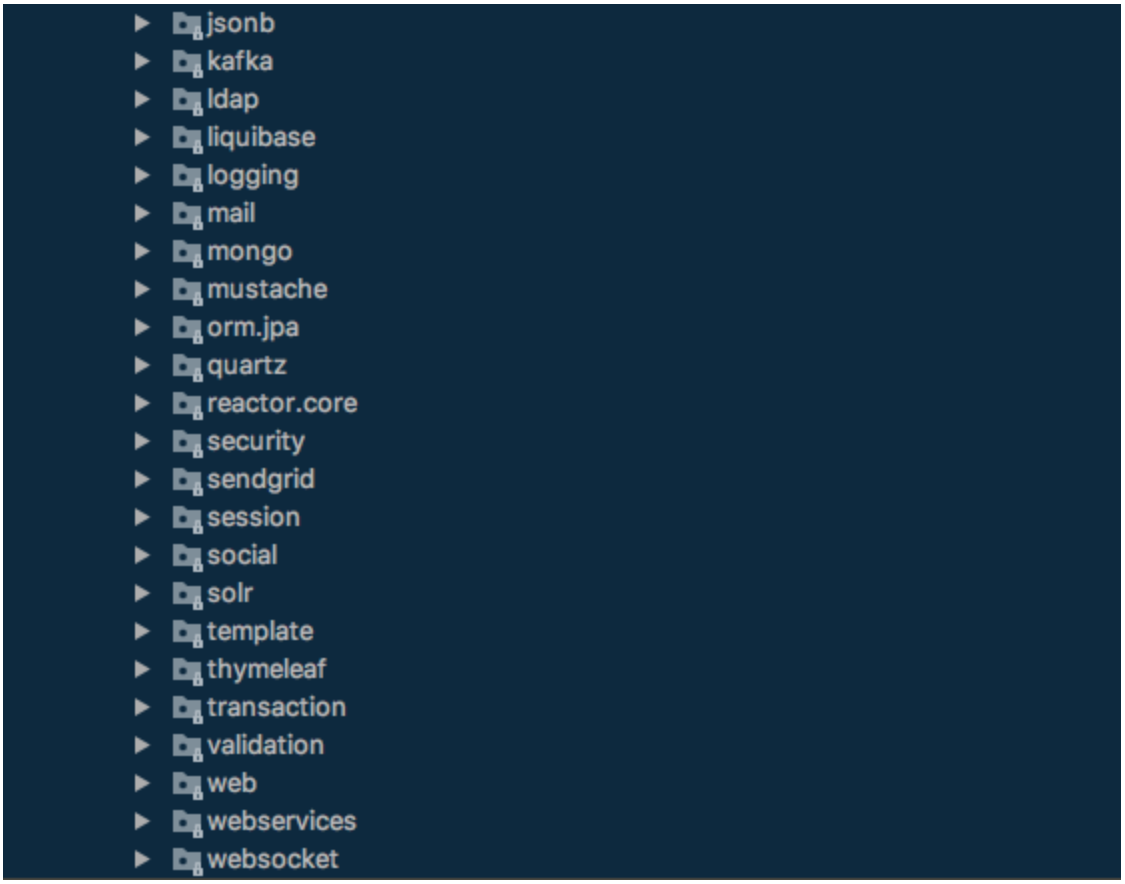


图 6 spring-boot-autoconfigure 依赖包目录

从图中可以看出，其中常见核心的包如下：


```
org.springframework.boot.autoconfigure
org.springframework.boot.autoconfigure.data.jpa
org.springframework.boot.autoconfigure.thymeleaf
org.springframework.boot.autoconfigure.web.servlet
org.springframework.boot.autoconfigure.web.reactive
……省略
```

在各自包目录下有对应的自动配置类，代码如下：

```
JpaRepositoriesAutoConfiguration
ThymeleafAutoConfiguration
WebMvcAutoConfiguration
WebFluxAutoConfiguration
……省略
```

上面自动配置类依次是 Jpa 自动配置类、Thymeleaf 自动配置类、Web MVC 自动配置类和 WebFlux 自动配置类。

spring-boot-autoconfigure 职责是通过 @EnableAutoConfiguration 核心注解，扫描 ClassPath 目录中自动配置类对应依赖，并按一定规则获取默认配置并自动初始化所需要

的 Bean。在 application.properties 配置文件也可以修改默认配置项，常用配置清单地址如下：

https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

4.2 @EnableAutoConfiguration 注解

自动配置工作机制是通过 @EnableAutoConfiguration 注解中 @Import 的 AutoConfigurationImportSelector 自动配置导入选择器类实现的。查阅源码可得具体流程如下：

- AutoConfigurationImportSelector 通过 SpringFactoriesLoader.loadFactoryNames() 核心方法读取 ClassPath 目录下面的 META-INF/spring.factories 文件。
- spring.factories 文件中配置的 Spring Boot 自动配置类，例如常见的 WebMvcAutoConfiguration Web MVC 自动配置类和 ServletWebServerFactoryAutoConfiguration 容器自动配置类。
- spring.factories 文件和 application.properties 文件都属于配置文件，配置的格式均为键值对。里面配置的每个自动配置类都会定义相关 Bean 的实例配置，也会定义什么条件下自动配置和哪些 Bean 被实例化。
- 当 pom.xml 添加某 Starter 依赖组件的时候，就会自动触发该依赖的默认配置。

例如添加 spring-boot-starter-web 依赖后，启动应用会触发容器自动配置类。容器自动配置类 ServletWebServerFactoryAutoConfiguration 的部分代码如下：

```
package org.springframework.boot.autoconfigure.web.servlet;

@Configuration
@ConditionalOnClass({ServletRequest.class})
@ConditionalOnWebApplication(
    type = Type.SERVLET
)
@EnableConfigurationProperties({ServerProperties.class})
@Import({ServletWebServerFactoryAutoConfiguration.BeanPostProcess
orsRegistrar.class})
public class ServletWebServerFactoryAutoConfiguration {
……省略
}
```

上面代码中，@ConditionalOnClass 注解表示对应的 ServletRequest 类在 ClassPath 目录下面存在，并且 @ConditionalOnWebApplication 注解表示该应用是 Servlet Web 应用时，才会去启动容器自动配置，并通过 ServerProperties 类默认设置了端口为 8080。Type.SERVLET 枚举代表 Servlet Web 应用，Type.REACTIVE 枚举代表响应式 WebFlux 应用。

自动配置，是一把双刃剑。用好了就像天下武功唯快不破一样。但要注意一些自动化配置造成的问题。常见的问题有：

- Spring Boot 工程添加某些 Starter 组件依赖，又不需要触发组件自动配置；
- Spring Boot 配置多个不同数据源配置时，使用 XML 配置多数据源，但其默认数据源配置会触发自动配置出现问题。

类似场景下，解决方式是通过 exclude 属性指定并排除自动配置类，代码如下：

```
@SpringBootApplication(exclude =
{DataSourceAutoConfiguration.class})
```

也等价于配置 @EnableAutoConfiguration 注解，代码如下：

```
@SpringBootApplication
@EnableAutoConfiguration(exclude =
{DataSourceAutoConfiguration.class})
```

自动配置会最大的智能化，只有配置了 exclude 属性时，Spring Boot 优先初始化用户定义的 Bean ，然后再进行自动配置。

4.3 利用自动配置自定义 Starter 组件

当公司需要共享或者开源 Spring Boot Starter 组件依赖包，就可以利用自动配置自定义 Starter 组件。一个完整的 Starter 组件包括以下两点：

- 提供自动配置功能的自动配置模块。
- 提供依赖关系管理功能的组件模块，即封装了组件所有功能，开箱即用。

实现自定义 Starter 组件，并不会将这两点严格区分，可以将自动配置功能和依赖管理结合在一起实现。下面利用自动配置实现自定义 Starter 组件：spring-boot-starter-swagger 组件是用来快速生成 API 文档，简化原生使用 Swagger2 。

spring-boot-starter-swagger 组件为 Spring For All 社区（ spring4all.com ）开源项目，源代码地址是： https://github.com/SpringForAll/spring-boot-starter-swagger。

什么是 Swagger2

Swagger2 是 API 最大的开发框架，基于 OpenAPI 规范（ OAS ），管理了 API 整个生命周期，即从 API 设计到文档，从测试到部署。具体更多了解见其官网：https://swagger.io。

spring-boot-starter-swagger 组件依赖

创建一个新的 Spring Boot 工程，命名为 spring-boot-starter-swagger。在 pom.xml 配置如下。

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-
processor</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger-ui</artifactId>
            <version>${version.swagger}</version>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger2</artifactId>
            <version>${version.swagger}</version>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-bean-validators</artifactId>
            <version>${version.swagger}</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.16.12</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
```


配置中添加了 spring-boot-starter 组件依赖用于自动配置特性，springfox-swagger2 依赖是 Swagger2 框架。

Swagger2 属性配置类 SwaggerProperties

新建名为 SwaggerProperties 的 Swagger2 属性配置类，包含了所有默认属性值。使用该组件时，可以在 application.properties 配置文件配置对应属性项，进行覆盖默认配置。代码如下：

```
import lombok.Data;
import lombok.NoArgsConstructor;
import
org.springframework.boot.context.properties.ConfigurationProperti
es;
```

```
import springfox.documentation.schema.ModelRef;

import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

@Data
@ConfigurationProperties("swagger")
public class SwaggerProperties {

    /**是否开启swagger**/
    private Boolean enabled;

    /**标题**/
    private String title = "";
    /**描述**/
    private String description = "";
    /**版本**/
    private String version = "";
    /**许可证**/
    private String license = "";
    /**许可证URL**/
    private String licenseUrl = "";
    /**服务条款URL**/
    private String termsOfServiceUrl = "";

    private Contact contact = new Contact();

    /**swagger会解析的包路径**/
    private String basePackage = "";

    /**swagger会解析的url规则**/
    private List<String> basePath = new ArrayList<>();
    /**在basePath基础上需要排除的url规则**/
    private List<String> excludePath = new ArrayList<>();

    /**分组文档**/
    private Map<String, DocketInfo> docket = new LinkedHashMap<>
();

    /**host信息**/
    private String host = "";

    /**全局参数配置**/
    private List<GlobalOperationParameter>
globalOperationParameters;

     ... 省略，具体代码见 GitHub
}
```

用 @ConfigurationProperties(prefix = “swagger”) 标注在类上方是指定属性的参数名称为 swagger。会对应匹配到配置文件中“swagger.*”结构的属性，例如，字段标题 title 表示标题，会匹配到 swagger.title 属性值。

Swagger2 自动配置类 SwaggerAutoConfiguration

新建名为 SwaggerAutoConfiguration 的 Swagger2 自动配置类，提供 Swagger2 依赖关系管理功能和自动配置功能。代码如下：

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnMis
singBean;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnPro
perty;

@Configuration
@ConditionalOnProperty(name = "swagger.enabled", matchIfMissing =
true)
@Import({
        Swagger2DocumentationConfiguration.class,
        BeanValidatorPluginsConfiguration.class
})
public class SwaggerAutoConfiguration implements BeanFactoryAware
{

    private BeanFactory beanFactory;

    @Bean
    @ConditionalOnMissingBean
    public SwaggerProperties swaggerProperties() {
        return new SwaggerProperties();
    }

    @Bean
    @ConditionalOnMissingBean
    @ConditionalOnProperty(name = "swagger.enabled",
matchIfMissing = true)
    public List<Docket> createRestApi(SwaggerProperties
swaggerProperties) {
        ... 省略，具体代码见 GitHub
    }

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws
BeansException {
        this.beanFactory = beanFactory;
```

```
    }
}
```

上面代码实现流程如下：

1. @Configuration 注解标注在类上方，表明该类为配置类。
2. @Import 注解引入 Swagger2 提供的配置类 Swagger2DocumentationConfiguration 和 Bean 数据验证插件配置类 BeanValidatorPluginsConfiguration。
3. @ConditionalOnMissingBean 注解标注了两处方法，当 Bean 没有被创建时会执行被标注的初始化方法。第一处被标记方法是 swaggerProperties()，用来实例化属性配置类 SwaggerProperties；第二处被标记方法是 createRestApi()，用来实例化 Swagger2 API 映射的 Docket 列表对象。
4. @ConditionalOnProperty 注解标注在 createRestApi() 方法，name 属性会去检查环境配置项 swagger.enabled。默认情况下，属性存在且不是 false 的情况下，会触发该初始化方法。matchIfMissing 属性默认值为 false，这里设置为 true，表示如果环境配置项没被设置，也会触发。

Swagger2 启动注解类 EnableSwagger2Doc

新建名为 EnableSwagger2Doc 的 Swagger2 启动注解类，用于开关 spring-boot-starter-swagger 组件依赖。代码如下：


```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import({SwaggerAutoConfiguration.class})
public @interface EnableSwagger2Doc {


}
```

上面代码 @Import 注解引入 Swagger2 自动配置类 SwaggerAutoConfiguration。当将该注解配置在应用启动类上方，即可开启 Swagger2 自动配置及其功能。

使用 spring-boot-starter-swagger 组件依赖

上面简单介绍了 spring-boot-starter-swagger 组件的核心代码实现，同样使用方式也很简单。在 chapter-2-spring-boot-config 工程的 Maven 配置中添加对应的依赖配置，目前支持 1.5.0.RELEASE 以上版本，配置如下：

```
<!-- 自定义 swagger2 Starter 组件依赖 -->
<dependency>
    <groupId>com.spring4all</groupId>
    <artifactId>spring-boot-starter-swagger</artifactId>
```

```
        <version>2.0</version>
    </dependency>
```

另外，需要在 ConfigApplication 应用启动类上方配置启动注解类 EnableSwagger2Doc，代码如下：

```
import com.spring4all.swagger.EnableSwagger2Doc;
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;

@EnableSwagger2Doc // 开启 Swagger
@SpringBootApplication
public class ConfigApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigApplication.class, args);
    }
}
```

执行 ConfigApplication 类启动，在控制台看到成功运行的输出后，打开浏览器访问 localhost:8080/swagger-ui.html 地址，可以看到自动生成的 Swagger API 文档，如图 7 所示：

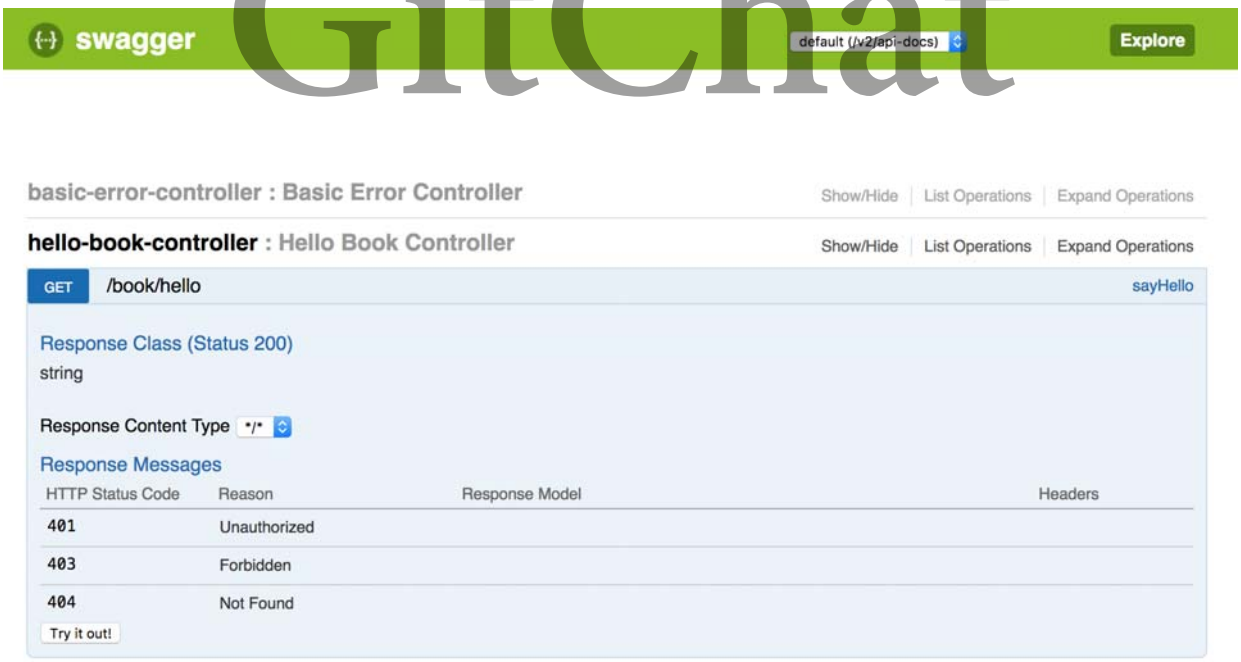


图 7 Swagger API 文档

API org.springframework.boot.autoconfigure.EnableAutoConfiguration 注解参数

- exclude：Class 数组，排除特定的自动配置类。
- excludeName：字符串数组，排除特定名称的自动配置类。

API org.springframework.boot.autoconfigure.ConditionalOnProperty 注解参数

- havingValue：字符串，属性期望值是否匹配。
- matchIfMissing：布尔值，如果该属性值未设置，则匹配。
- name：字符串数组，要测试的属性名。
- prefix：字符串，属性前缀名。
- value：字符串，功能同 name。

API org.springframework.boot.autoconfigure.ConditionalOnClass 注解参数

- name：字符串数组，类名必须存在。
- value：Class 数组，类必须存在。

API org.springframework.boot.autoconfigure.ConditionalOnMissingBean 注解参数

- annotation：注解 Class 数组，匹配注解装饰的 Bean。
- ignored：Class 数组，匹配时，忽略该类型的 Bean。
- ignoredType：字符串数组，匹配时，忽略该类型名称的 Bean。
- name：字符串数组，匹配要检查的 Bean 名称。
- search：SearchStrategy 对象，通过 SearchStrategy 来决定程序的上下文策略。
- type：字符串数组，匹配要检查的 Bean 类型名称。
- value：Class 数组，匹配要检查的 Bean 类型。

API org.springframework.boot.autoconfigure.ConditionalOnWebApplication 注解参数

- type：ConditionalOnWebApplication.Type 对象，匹配对应的 Web 应用程序类型。

小结

本文从自定义属性快速入门工程出发，介绍了两种配置文件以及属性的获取方式，然后讲解了外化配置的优先级、属性引用、随机式使用和多环境配置，最后讲解了自动配置的原理、核心注解以及利用自动配置实现了自定义 Starter 组件。

示例代码地址：https://github.com/JeffLi1993/springboot-core-action-book-demo/tree/master/chapter-2-spring-boot-config