

4、Hello World探究

(1) POM文件

pom.xml 文件中默认有两个模块：

spring-boot-starter 核心模块，包括自动配置支持、日志和YAML，如果引入了 spring-boot-starter-web web模块可以去掉此配置，因为 spring-boot-starter-web 自动依赖了spring-boot-starter。

spring-boot-starter-test 测试模块，包括JUnit、Hamcrest、Mockito。

父项目

```
1 <parent>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-parent</artifactId>
4   <version>1.5.9.RELEASE</version>
5 </parent>
6
7 //他的父项目是
8 <parent>
9   <groupId>org.springframework.boot</groupId>
10  <artifactId>spring-boot-dependencies</artifactId>
11  <version>1.5.9.RELEASE</version>
12  <relativePath>../../spring-boot-dependencies</relativePath>
13 </parent>
14 //他来真正管理Spring Boot应用里面的所有依赖版本；
```

复制

启动器

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

spring-boot-starter: spring-boot场景启动器；帮我们导入了web模块正常运行所依赖的组件；

Spring Boot将所有的功能场景都抽取出来，做成一个个的starters（启动器），只需要在项目里面引入这些starter相关场景的所有依赖都会导入进来。要什么功能就导入什么场景的启动器。

(2) 主程序类，主入口类

```
1  /**
2   * @SpringBootApplication 来标注一个主程序类，说明这是一个Spring Boot应用
3   */
4  @SpringBootApplication
5  public class HelloWorldMainApplication {
6      public static void main(String[] args) {
7          // Spring应用启动起来
8          SpringApplication.run(HelloWorldMainApplication.class,args);
9      }
10 }
```

@SpringBootApplication: Spring Boot应用标注在某个类上说明这个类是SpringBoot的主配置类，SpringBoot就应该运行这个类的main方法来启动SpringBoot应用；

```
1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Inherited
5  @SpringBootConfiguration
6  @EnableAutoConfiguration
7  @ComponentScan(excludeFilters = {
8      @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
9      @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
10 public @interface SpringBootApplication {
11 }
```

- SpringBootConfiguration:Spring Boot的配置类；

标注在某个类上，表示这是一个Spring Boot的配置类；

- @Configuration:配置类上来标注这个注解；

配置类 ----- 配置文件；配置类也是容器中的一个组件；@Component

- @EnableAutoConfiguration：开启自动配置功能；

以前我们需要配置的东西，Spring Boot帮我们自动配置；@EnableAutoConfiguration告诉SpringBoot开启自动配置功能；这样自动配置才能生效；

```
1  @AutoConfigurationPackage
2  @Import(EnableAutoConfigurationImportSelector.class)
3  public @interface EnableAutoConfiguration {
4  }
```

@AutoConfigurationPackage: 自动配置包

@Import(AutoConfigurationPackages.Registrar.class):

Spring的底层注解@Import, 给容器导入一个组件; 导入的组件由AutoConfigurationPackages.Registrar.class;

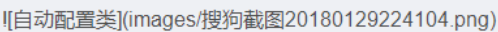
==将主配置类 (@SpringBootApplication标注的类) 的所在包及下面所有子包里面的所有组件扫描到Spring容器; ==

@Import(EnableAutoConfigurationImportSelector.class);

给容器中导入组件?

EnableAutoConfigurationImportSelector: 导入哪些组件的选择器;

将所有需要导入的组件以全类名的方式返回; 这些组件就会被添加到容器中;

会给容器中导入非常多的自动配置类 (xxxAutoConfiguration) ; 就是给容器中导入这个场景需要的所有组件, 并配置好这些组件; 

有了自动配置类, 免去了我们手动编写配置注入功能组件等的工作;

SpringFactoriesLoader.loadFactoryNames(EnableAutoConfiguration.class,classLoader);

Spring Boot在启动的时候从类路径下的META-INF/spring.factories中获取EnableAutoConfiguration指定的值, 将这些值作为自动配置类导入到容器中, 自动配置类就生效, 帮我们进行自动配置工作; 以前我们需要自己配置的东西, 自动配置类都帮我们;

J2EE的整体整合解决方案和自动配置都在spring-boot-autoconfigure-1.5.9.RELEASE.jar;

2、YAML语法：

(1) 基本语法

k:(空格)v: 表示一对键值对（空格必须有）；

以**空格**的缩进来控制层级关系；只要是左对齐的一列数据，都是同一个层级的

```
1 | server:
2 |     port: 8081
3 |     path: /hello
```

属性和值也是大小写敏感；

(2) 值的写法

- 字面量：普通的值（数字，字符串，布尔）

k: v: 字面直接来写；

字符串默认不用加上单引号或者双引号；

""：双引号；不会转义字符串里面的特殊字符；特殊字符会作为本身想表示的意思

name: "zhangsan \n lisi": 输出；zhangsan 换行 lisi

"：单引号；会转义特殊字符，特殊字符最终只是一个普通的字符串数据

name: 'zhangsan \n lisi': 输出；zhangsan \n lisi

- 对象、Map（属性和值）（键值对）：

k: v: 在下一行来写对象的属性和值的关系；注意缩进

对象还是k: v的方式

```
1 | friends:
2 |     lastName: zhangsan
3 |     age: 20
```

- 数组（List、Set）：

用- 值表示数组中的一个元素

```
1 | pets:
2 | - cat
3 | - dog
4 | - pig
```

3、配置文件值注入

(1) yml配置文件

```
1 person:
2   lastName: hello
3   age: 18
4   boss: false
5   birth: 2017/12/12
6   maps: {k1: v1,k2: 12}
7   lists:
8     - lisi
9     - zhaoliu
10  dog:
11    name: 小狗
12    age: 12
```

(2) javaBean:

```
1 /**
2  * 将配置文件中配置的每一个属性的值，映射到这个组件中
3  * @ConfigurationProperties：告诉SpringBoot将本类中的所有属性和配置文件中相关的配置进行绑定；
4  *   prefix = "person"：配置文件中哪个下面的所有属性进行一一映射
5  *
6  * 只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；
7  *
8  */
9 @Component
10 @ConfigurationProperties(prefix = "person")
11 public class Person {
12
13     private String lastName;
14     private Integer age;
15     private Boolean boss;
16     private Date birth;
17
18     private Map<String,Object> maps;
19     private List<Object> lists;
20     private Dog dog;
```

我们可以导入配置文件处理器，以后编写配置就有提示了

```
1 <!-- 导入配置文件处理器，配置文件进行绑定就会有提示-->
2 <dependency>
3     groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-configuration-processor</artifactId>
5     <optional>true</optional>
6 </dependency>
```

(3) @Value获取值和@ConfigurationProperties获取值比较

配置文件yml还是properties他们都能获取到值；

如果说，我们只是在某个业务逻辑中需要获取一下配置文件中的某项值，使用@Value；

如果说，我们专门编写了一个javaBean来和配置文件进行映射，我们就直接使用@ConfigurationProperties；

(4) 配置文件注入值数据校验

```
1 @Component
2 @ConfigurationProperties(prefix = "person")
3 @Validated
4 public class Person {
5
6     /**
7      * <bean class="Person">
8      *     <property name="lastName" value="字面量/${key} 从环境变量、配置文件中获取值/#{SpEL}"></property>
9      * </bean/>
10    */
11
12    //lastName必须是邮箱格式
13    @Email
14    //@Value("${person.last-name}")
15    private String lastName;
16    //@Value("#{11*2}")
17    private Integer age;
18    //@Value("true")
19    private Boolean boss;
20
21    private Date birth;
22    private Map<String, Object> maps;
23    private List<Object> lists;
24    private Dog dog;
```

(5) @PropertySource&@ImportResource&@Bean

@**PropertySource**: 加载指定的配置文件;

```
1  /**
2   * 将配置文件中配置的每一个属性的值，映射到这个组件中
3   * @ConfigurationProperties：告诉SpringBoot将本类中的所有属性和配置文件中相关的配置进行绑定；
4   *      prefix = "person"：配置文件中哪个下面的所有属性进行一一映射
5   *
6   * 只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；
7   * @ConfigurationProperties(prefix = "person")默认从全局配置文件中获取值；
8   *
9   */
10 @PropertySource(value = {"classpath:person.properties"})
11 @Component
12 @ConfigurationProperties(prefix = "person")
13 //@Validated
14 public class Person {
15
16     /**
17      * <bean class="Person">
18      *      <property name="lastName" value="字面量/${key}从环境变量、配置文件中获取值/#{SpEL}"></property>
19      * </bean>
20      */
21
22     //lastName必须是邮箱格式
23     // @Email
24     //@Value("${person.last-name}")
25     private String lastName;
26     //@Value("#{11*2}")
27     private Integer age;
28     //@Value("true")
29     private Boolean boss;
```

@**ImportResource**: 导入Spring的配置文件，让配置文件里面的内容生效;

Spring Boot里面没有Spring的配置文件，我们自己编写的配置文件，也不能自动识别;

想让Spring的配置文件生效，加载进来; @**ImportResource**标注在一个配置类上

```
1  @ImportResource(locations = {"classpath:beans.xml"})
2  导入Spring的配置文件让其生效
```

不来编写Spring的配置文件

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
5
6         <bean id="helloService" class="com.atguigu.springboot.service.HelloService"></bean>
7  </beans>
```

SpringBoot推荐给容器中添加组件的方式；推荐使用全注解的方式

- 配置类**@Configuration**----->Spring配置文件
- 使用**@Bean**给容器中添加组件

```
1  /**
2   * @Configuration: 指明当前类是一个配置类；就是来替代之前的Spring配置文件
3   *
4   * 在配置文件中用<bean></bean>标签添加组件
5   *
6   */
7  @Configuration
8  public class MyAppConfig {
9
10     // 将方法的返回值添加到容器中；容器中这个组件默认id就是方法名
11     @Bean
12     public HelloService helloService02(){
13         System.out.println("配置类@Bean给容器中添加组件了...");
14         return new HelloService();
15     }
16 }
```

4、配置文件占位符

(1) 随机数

```
1  ${random.value}、${random.int}、${random.long}
2  ${random.int(10)}、${random.int[1024,65536]}
```

(2) 占位符获取之前配置的值，如果没有可以用:指定默认值

```
1  person.last-name=张三${random.uuid}
2  person.age=${random.int}
3  person.birthday=2017/12/15
4  person.boss=false
5  person.maps.k1=v1
6  person.maps.k2=14
7  person.lists=a,b,c
8  person.dog.name=${person.hello:hello}_dog
9  person.dog.age=15
```


5、Profile

(1) 多Profile文件

我们在主配置文件编写的时候，文件名可以是 `application-{profile}.properties/yml`

默认使用`application.properties`的配置；

(2) yml支持多文档块方式

```
1  server:
2    port: 8081
3  spring:
4    profiles:
5      active: prod
6
7  ---
8  server:
9    port: 8083
10 spring:
11   profiles: dev
12
13
14 ---
15
16 server:
17   port: 8084
18 spring:
19   profiles: prod #指定属于哪个环境
```

(3) 激活指定profile

- 在配置文件中指定 `spring.profiles.active=dev`
- 命令行：

```
java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev;
```

可以直接在测试的时候，配置传入命令行参数

- 虚拟机参数；

```
-Dspring.profiles.active=dev
```

6、配置文件加载位置

springboot 启动会扫描以下位置的application.properties或者application.yml文件作为Spring boot的默认配置文件

`--file:./config/`

`--file:./`

`--classpath:/config/`

`--classpath:/`

优先级由高到底，高优先级的配置会覆盖低优先级的配置；

SpringBoot会从这四个位置全部加载主配置文件；**互补配置**；

==我们还可以通过spring.config.location来改变默认的配置文件的配置位置==

项目打包好以后，我们可以使用命令行参数的形式，启动项目的时候来指定配置文件的新位置；指定配置文件和默认加载的这些配置文件共同起作用形成互补配置；

`java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --spring.config.location=G:/application.properties`

7、外部配置加载顺序

SpringBoot也可以从以下位置加载配置； 优先级从高到低；高优先级的配置覆盖低优先级的配置，所有的配置会形成互补配置

(1) 命令行参数

所有的配置都可以在命令行上进行指定

`java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --server.port=8087 --server.context-path=/abc`

多个配置用空格分开；`--配置项=值`

(2) 来自java:comp/env的JNDI属性

(3) Java系统属性 (System.getProperties())

(4) 操作系统环境变量

(5) RandomValuePropertySource配置的random.*属性值

由jar包外向jar包内进行寻找；优先加载带profile

(6) jar包外部的application-{profile}.properties或application.yml(带spring.profile)配置文件

(7) jar包内部的application-{profile}.properties或application.yml(带spring.profile)配置文件

再来加载不带profile

(8) jar包外部的application.properties或application.yml(不带spring.profile)配置文件

(9) jar包内部的application.properties或application.yml(不带spring.profile)配置文件

(10) @Configuration注解类上的@PropertySource

(11) 通过SpringApplication.setDefaultProperties指定的默认属性

8、自动配置原理

配置文件到底能写什么？怎么写？自动配置原理；

[配置文件能配置的属性参照]

(1) SpringBoot启动的时候加载主配置类，开启了自动配置功能 `==@EnableAutoConfiguration==`

(2) `@EnableAutoConfiguration` 作用：

- 利用`EnableAutoConfigurationImportSelector`给容器导入一些组件？
- 可以查看`selectImports()`方法的内容；
- `List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);`获取候选的配置

```
1 SpringFactoriesLoader.loadFactoryNames()  
2 扫描所有jar包类路径下 META-INF/spring.factories  
3 把扫描到的这些文件的内容包装成properties对象  
4 从properties中获取到EnableAutoConfiguration.class类（类名）对应的值，然后把他们添加在容器中
```

将 类路径下 META-INF/spring.factories 里面配置的所有`EnableAutoConfiguration`的值加入到了容器中；

每一个这样的 `xxxAutoConfiguration`类都是容器中的一个组件，都加入到容器中；用他们来做自动配置；

(3) 每一个自动配置类进行自动配置功能；

(4) 以`**HttpEncodingAutoConfiguration`（Http编码自动配置）**为例解释自动配置原理：

`@Configuration` //表示这是一个配置类，以前编写的配置文件一样，也可以给容器中添加组件

`@EnableConfigurationProperties(HttpEncodingProperties.class)` //启动指定类的`ConfigurationProperties`功能；将配置文件中对应的值和`HttpEncodingProperties`绑定起来；并把`HttpEncodingProperties`加入到IOC容器中

`@ConditionalOnWebApplication` //spring底层`@Conditional`注解，根据不同的条件，如果满足指定的条件，整个配置类里面的配置就会生效；判断当前应用是否是web类，如果是，当前配置类生效

`@ConditionalOnClass(CharacterEncodingFilter.class)` //判断当前项目有没有这个类`CharacterEncodingFilter`；

SpringMVC中进行乱码解决的过滤器；

`@ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled", matchIfMissing = true)` //判断配置文件中是否存在某个配置`spring.http.encoding.enabled`；如果不存在，判断也是成立的

//即使我们配置文件中不配置`spring.http.encoding.enabled=true`，也是默认生效的；

```
public class HttpEncodingAutoConfiguration {  
    //已经和springboot配置文件映射了  
    private final HttpEncodingProperties properties;  
    //只有一个有参构造器的情况下，参数的值就会从容器中拿  
    public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {  
        this.properties = properties;  
    }  
}
```

`@Bean` //给容器中添加一个组件，这个组件的某些值需要从`properties`中回去

`@ConditionalOnEncodingFilter(CharacterEncodingFilter.class)` //判断容器没有这个

组件?

```
public CharacterEncodingFilter characterEncodingFilter(){
    CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();
    filter.setEncoding(this.properties.getCharset().name());

    filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));

    filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));
    return filter;
}
}
```

根据当前不同的条件判断，决定这个配置类是否生效?

—但这个配置类生效；这个配置类就会给容器中添加各种组件；这些组件的属性是从对应的properties类中获取的，这些类里面的每一个属性又是和配置文件绑定的；

(5) 所有在配置文件中能配置的属性都是在xxxxProperties类中封装着；配置文件能配置什么就可以参照某个功能对应的这个属性类

```
1 @ConfigurationProperties(prefix = "spring.http.encoding") // 从配置文件中获取指定的值和bean的属性进行绑定
2 public class HttpEncodingProperties {
3
4     public static final Charset DEFAULT_CHARSET = Charset.forName("UTF-8");
```

****精髓****

- springboot启动会加载大量的自动配置类
- 我们看我们需要的功能有没有springboot默认写好的自动配置类；
- 我们再来看这个自动配置类中到底配置了哪些组件（只要我们要用的组件有，我们就不需要再来配置了）
- 给容器中自动配置类添加组件的时候，会动从properties类中获取某些属性，我们就可以在配置文件中指定这些属性的值；

xxxxAutoConfigurartion：自动配置类；

给容器中添加组件

xxxxProperties:封装配置文件中相关属性；

9、细节

@Conditional派生注解（Spring注解版原生的@Conditional作用）

作用：必须是@Conditional指定的条件成立，才给容器中添加组件，配置里面的所有内容才生效；

@Conditional扩展注解	作用（判断是否满足当前指定条件）
@ConditionalOnJava	系统的java版本是否符合要求
@ConditionalOnBean	容器中存在指定Bean；
@ConditionalOnMissingBean	容器中不存在指定Bean；
@ConditionalOnExpression	满足SpEL表达式指定
@ConditionalOnClass	系统中有指定的类
@ConditionalOnMissingClass	系统中没有指定的类
@ConditionalOnSingleCandidate	容器中只有一个指定的Bean，或者这个Bean是首选Bean
@ConditionalOnProperty	系统中指定的属性是否有指定的值
@ConditionalOnResource	类路径下是否存在指定资源文件
@ConditionalOnWebApplication	当前是web环境
@ConditionalOnNotWebApplication	当前不是web环境
@ConditionalOnJndi	JNDI存在指定项

自动配置类必须在一定的条件下才能生效；

我们可以通过启用 debug=true属性；来让控制台打印自动配置报告==，这样我们就可以很方便的知道哪些自动配置类生效；

我们可以通过启用 debug=true属性；来让控制台打印自动配置报告==，这样我们就可以很方便的知道哪些自动配置类生效；

```
=====
AUTO-CONFIGURATION REPORT
=====
```

Positive matches: (自动配置类启用的)

```
DispatcherServletAutoConfiguration matched:
- @ConditionalOnClass found required class
'org.springframework.web.servlet.DispatcherServlet'; @ConditionalOnMissingClass
did not find unwanted class (OnClassCondition)
- @ConditionalOnWebApplication (required) found
StandardServletEnvironment (OnWebApplicationCondition)
```

Negative matches: (没有启动，没有匹配成功的自动配置类)

```
ActiveMQAutoConfiguration:
Did not match:
```

- @ConditionalOnClass did not find required classes
'javax.jms.ConnectionFactory', 'org.apache.activemq.ActiveMQConnectionFactory'
(OnClassCondition)

AopAutoConfiguration:

Did not match:

- @ConditionalOnClass did not find required classes
'org.aspectj.lang.annotation.Aspect', '三、快速入门'

日志分类

日志抽象层

- JCL : jakarta commons logging
- SLF4j : simple logging facade for java
- jboss-logging

日志实现层

- log4j
- log4j2
- logback
- JUL: java.util.logging

SLF4j, log4j, logback是同一个人写的

JUL: java.util.logging 是jdk1.4自带的, 为了防止被log4j占有市场, 所以临时出的, 功能不咋地

log4j2是apache的

五、日志

1、SpringBoot选用 SLF4j和logback

以后开发的时候，日志记录方法的调用，不应该来直接调用日志的实现类，而是调用日志抽象层里面的方法；

给系统里面导入slf4j的jar和 logback的实现jar。

```
1 import org.slf4j.Logger;
2 import org.slf4j.LoggerFactory;
3
4 public class HelloWorld {
5     public static void main(String[] args) {
6         Logger logger = LoggerFactory.getLogger(HelloWorld.class);
7         logger.info("Hello World");
8     }
9 }
```

2、遗留问题

a (slf4j+logback) : Spring (commons-logging) 、Hibernate (jboss-logging) 、MyBatis、xxxx

统一日志记录，即使是别的框架和我一起统一使用slf4j进行输出？

如何让系统中所有的日志都统一到slf4j？

- (1) 将系统中其它日志框架先排除出去
- (2) 用中间包来替换原有的日志框架
- (3) 导入slf4j其它的实现

3、SpringBoot日志关系

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

SpringBoot使用它来做日志功能；

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

4、底层依赖关系

- (1) Spring Boot底层也是使用slf4j+logback的方式进行日志记录
- (2) Spring Boot也把其他的日志都替换成了slf4j；
- (3) 中间替换包？

```

@SuppressWarnings("rawtypes")
public abstract class LogFactory {

    static String UNSUPPORTED_OPERATION_IN_JCL_OVER_SLF4J =
"http://www.slf4j.org/codes.html#unsupported_operation_in_jcl_over_slf4j";

    static LogFactory logFactory = new SLF4JLogFactory();

```

Spring框架用的commons-logging:

```

1  <dependency>
2      <groupId>org.springframework</groupId>
3      <artifactId>spring-core</artifactId>
4      <exclusions>
5          <exclusion>
6              <groupId>commons-logging</groupId>
7              <artifactId>commons-logging</artifactId>
8          </exclusion>
9      </exclusions>
10 </dependency>

```

****=SpringBoot能自动适配所有的日志，而且底层使用slf4j+logback的方式记录日志，引入其他框架的时候，只需要把这个框架依赖的日志框架排除掉即可；****

5、日志使用

(1) 默认配置

Spring Boot默认帮我们配置好了日志

//记录器

```
Logger logger = LoggerFactory.getLogger(getClass());
```

@Test

```
public void contextLoads() {
```

```
    //System.out.println();
```

```
    //日志的级别;
```

```
    //由低到高 trace<debug<info<warn<error
```

```
    //可以调整输出的日志级别; 日志就只会在这个级别以以后的高级别生效
```

```
    logger.trace("这是trace日志...");
```

```
    logger.debug("这是debug日志...");
```

//SpringBoot默认给我们使用的是info级别的，没有指定级别的就用SpringBoot默认规定的级别; root级别

```
    logger.info("这是info日志...");
```

```
    logger.warn("这是warn日志...");
```

```
    logger.error("这是error日志...")
```

```
}
```



```
logging.level.com.atguigu=trace
```

```
#logging.path=
```

```
# 不指定路径在当前项目下生成springboot.log日志
```

```
# 可以指定完整的路径;
```

```
#logging.file=G:/springboot.log
```

```
# 在当前磁盘的根路径下创建spring文件夹和里面的log文件夹; 使用 spring.log 作为默认文件
```

```
logging.path=/spring/log
```

```
# 在控制台输出的日志的格式
```

```
logging.pattern.console=%d{yyyy-MM-dd} [%thread] %-5level %logger{50} - %msg%n
```

```
# 指定文件中日志输出的格式
```

```
logging.pattern.file=%d{yyyy-MM-dd} === [%thread] === %-5level === %logger{50} === %msg%n
```

(2) 指定配置

给类路径下放上每个日志框架自己的配置文件即可; SpringBoot就不使用他默认配置的了。

logback.xml: 直接就被日志框架识别了;

logback-spring.xml: 日志框架就不直接加载日志的配置项, 由SpringBoot解析日志配置, 可以使用SpringBoot的高级Profile功能;

```
1 <springProfile name="staging">
2   <!-- configuration to be enabled when the "staging" profile is active -->
3   可以指定某段配置只在某个环境下生效
4 </springProfile>
```

```
<appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
  <!--
```

日志输出格式:

%d表示日期时间,

%thread表示线程名,

%-5level: 级别从左显示5个字符宽度

%logger{50} 表示logger名字最长50个字符, 否则按照句点分割。

%msg: 日志消息,

%n是换行符

```
-->
```

```
<layout class="ch.qos.logback.classic.PatternLayout">
```

```
  <springProfile name="dev">
```

```
    <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} ----> [%thread] ---> %-5level
```

```
%logger{50} - %msg%n</pattern>
```

```
  </springProfile>
```

```
  <springProfile name="!dev">
```

```
    <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} ==== [%thread] ==== %-5level
```

```
%logger{50} - %msg%n</pattern>
    </springProfile>
</layout>
</appender>
```

如果使用logback.xml作为日志配置文件，还要使用profile功能，会有以下错误：

```
no applicable action for [springProfile]
```

6、切换日志框架

可以按照slf4j的日志适配图，进行相关的切换；

slf4j+log4j的方式：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <artifactId>logback-classic</artifactId>
      <groupId>ch.qos.logback</groupId>
    </exclusion>
    <exclusion>
      <artifactId>log4j-over-slf4j</artifactId>
      <groupId>org.slf4j</groupId>
    </exclusion>
  </exclusions>
</dependency>
```

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
</dependency>
```

切换为log4j2:

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4   <exclusions>
5     <exclusion>
6       <artifactId>spring-boot-starter-logging</artifactId>
7       <groupId>org.springframework.boot</groupId>
8     </exclusion>
9   </exclusions>
10 </dependency>
11
12 <dependency>
13   <groupId>org.springframework.boot</groupId>
14   <artifactId>spring-boot-starter-log4j2</artifactId>
15 </dependency>
```

六、总结

使用 Spring Boot 可以非常方便、快速搭建项目，使我们不用关心框架之间的兼容性，适用版本等各种问题，我们想使用任何东西，仅仅添加一个配置就可以，所以使用 Spring Boot 非常适合构建微服务。