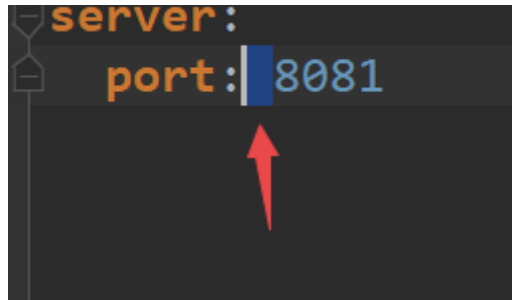


配置文件

SpringBoot使用一个全局的配置文件，配置文件名称是固定的

- application.properties
 - 语法结构：key=value
- application.yml
 - 语法结构：key: 空格 value
 -



之前是server.post，这里yaml中server为父元素。

- **配置文件的作用：** 修改SpringBoot自动配置的默认值，因为SpringBoot在底层都给我们自动配置好了

1 Properties —— application.properties

```
1 # SpringBoot这个配置文件中到底可以配置哪些东西？
2
3 # 【不推荐】官方配置文档：https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/htmlsingle/#common-application-properties
```

- 给实体类赋值
 - 实体类

```
1 package com.kuang.pojo;
2
3 import org.springframework.beans.factory.annotation.Value;
4 import org.springframework.context.annotation.PropertySource;
5 import org.springframework.stereotype.Component;
6
7 @Component
8 @PropertySource(value = "classpath:geekst.properties")//加载指定的配置文件
9 public class People {
10     //SPEL表达式取出配置文件的值
11     @Value("${name}")
12     private String name;
13
14     ...getter、setter、toString()、无参构造，全参构造...
15 }
```

- geekst.properties:

```
1 name=Geekst
```

- 控制台打印

```
People{name='Geekst'}
```

2 yaml【官方推荐】—— application.yaml

2.1 基本语法

对空格的要求十分高

- 注释: #
- 普通的key-value
 - key:(空格)value name: Geekst
 - 对比Propertis: name=Geekst
- 对象 (例如Student) —— 注意空格

```
1 student:
2   name: Geekst
3   age: 2
```

- 对比Properties

```
1 student.name=Geekst
2 student.age=2
```

- 行内写法

```
1 student: {name: Geekst, age: 2}
```

【properties无法存】

- 数组

```
1 pets:
2   - cat
3   - dog
4   - pig
5
6 # 行内写法
7 pets2: [cat,dog,pig]
```

【properties无法存】

2.2 给实体类赋值

以前我们可以在属性上用@Value进行赋值。

- 导入依赖

```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-configuration-processor</artifactId>
4   <optional>true</optional>
5 </dependency>

```

- 准备两个实体类:

```

1 package com.kuang.pojo;
2
3 import org.springframework.stereotype.Component;
4
5 @Component //组件, 扫描到Spring容器中
6 public class Dog {
7     private String name;
8     private int age;
9
10    ...getter、setter、toString()、无参构造, 全参构造...
11 }

```

```

1 package com.kuang.pojo;
2
3 import org.springframework.boot.context.properties.ConfigurationProperties;
4 import org.springframework.stereotype.Component;
5
6 import java.util.Date;
7 import java.util.List;
8 import java.util.Map;
9
10 /**
11  * @ConfigurationProperties作用: 将配置文件中配置的每一个属性的值, 映射到这个组件
12  * 中;
13  * 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
14  * 参数 prefix = "person" : 将配置文件中的person下面的所有属性一一对应
15  * 只有这个组件是容器中的组件, 才能使用容器提供的@ConfigurationProperties功能
16  */
17 @Component
18 @ConfigurationProperties(prefix = "person")
19 public class Person {
20     private String name;
21     private int age;
22     private Boolean happy;
23     private Date birth;
24     private Map<String, Object> maps;
25     private List<Object> lists;
26     private Dog dog;
27
28    ...getter、setter、toString()、无参构造, 全参构造...
29 }

```

- application.yaml:

```

1  person:
2    name: Geekst
3    age: 3
4    happy: true
5    birth: 2020/2/21
6    maps: {k1: v1,k2: v2,k3: v3}
7    lists:
8      - code
9      - music
10     - girl
11  dog:
12    name: Geekst
13    age: 3

```

核心: @ConfigurationProperties(prefix = "person")

- 测试运行

```

1  @SpringBootTest
2  class Springboot02ConfigApplicationTests {
3      @Autowired
4      private Dog dog;
5
6      @Autowired
7      private Person person;
8
9      @Test
10     void contextLoads() {
11         System.out.println(person);
12     }
13
14 }

```

控制台打印:

```

1  Person{
2    name='Geekst',
3    age=3,
4    happy=true,
5    birth=Fri Feb 21 00:00:00 CST 2020,
6    maps={k1=v1, k2=v2, k3=v3},
7    lists=[code, music, girl],
8    dog=Dog{name='Geekst', age=3}
9  }

```

2.3 使用一些表达式

```

1 person:
2   name: Geekst${random.uuid}
3   age: ${random.int}
4   happy: true
5   birth: 2020/2/21
6   maps: {k1: v1,k2: v2,k3: v3}
7   lists:
8     - code
9     - music
10    - girl
11  dog:
12    name: ${person.hello:nohello}_旺财
13    age: 3

```

- \${random.uuid} —— 随机的uuid
- \${random.int} —— 随机数
- \${person.hello:nohello} —— 如果存在person中有hello这个属性，则赋予hello属性的值；如果没有，则赋予：后面的值nohello

控制台打印：

```

1 Person{
2   name='Geekstf97bdb5a-1500-446b-bb5e-9427d4af200e',
3   age=3,
4   happy=true,
5   birth=Fri Feb 21 00:00:00 CST 2020,
6   maps={k1=v1, k2=v2, k3=v3},
7   lists=[code, music, girl],
8   dog=Dog{name='nohello_旺财', age=3}
9 }

```

2.4 对比properties

	@ConfigurationProperties	@Value
功能	批量注入配置文件中的属性	一个个指定
松散绑定（松散语法）	支持	不支持
SpEL	不支持	支持
JSR303数据校验	支持	不支持
复杂类型封装	支持	不支持

可以看出：properties除了可以支持SpEL以外，对如今使用得到的一些技术都无法支持，所以优先选择使用yaml

- 小结：
 - ConfigurationProperties只需要写一次即可，value则需要每个字段都添加
 - JSR303数据校验，这个就是我们可以再字段是增加一层过滤器验证，可以保证数据的合法性
 - 复杂类型封装，yaml中可以封装对象，使用@value就不支持
 - 配置yaml和配置properties都可以获取到值，强烈推荐yaml
 - 如果我们在某个业务中，只需要获取配置文件中的某个值，可以使用一下@value
 - 如果说，我们专门编写了一个JavaBean来和配置文件进行映射，就直接使用@configurationProperties

- 什么是松散绑定？

比如我的yml中写的first-name，这个和firstName是一样的，-后面跟着的字母默认是大写的。这就是松散绑定

```
1 package com.kuang.pojo;
2
3 import org.springframework.boot.context.properties.ConfigurationProperties;
4 import org.springframework.stereotype.Component;
5
6 @Component //组件，扫描到Spring容器中，注册Bean
7 @ConfigurationProperties(prefix = "dog")
8 public class Dog {
9     private String firstName;
10    private int age;
11
12    ...getter、setter、toString()、无参构造，全参构造...
13 }
```

```
1 dog:
2   first-name: firstname
3   age: 8
```

这里的first-name和驼峰命名firstName能够对应：

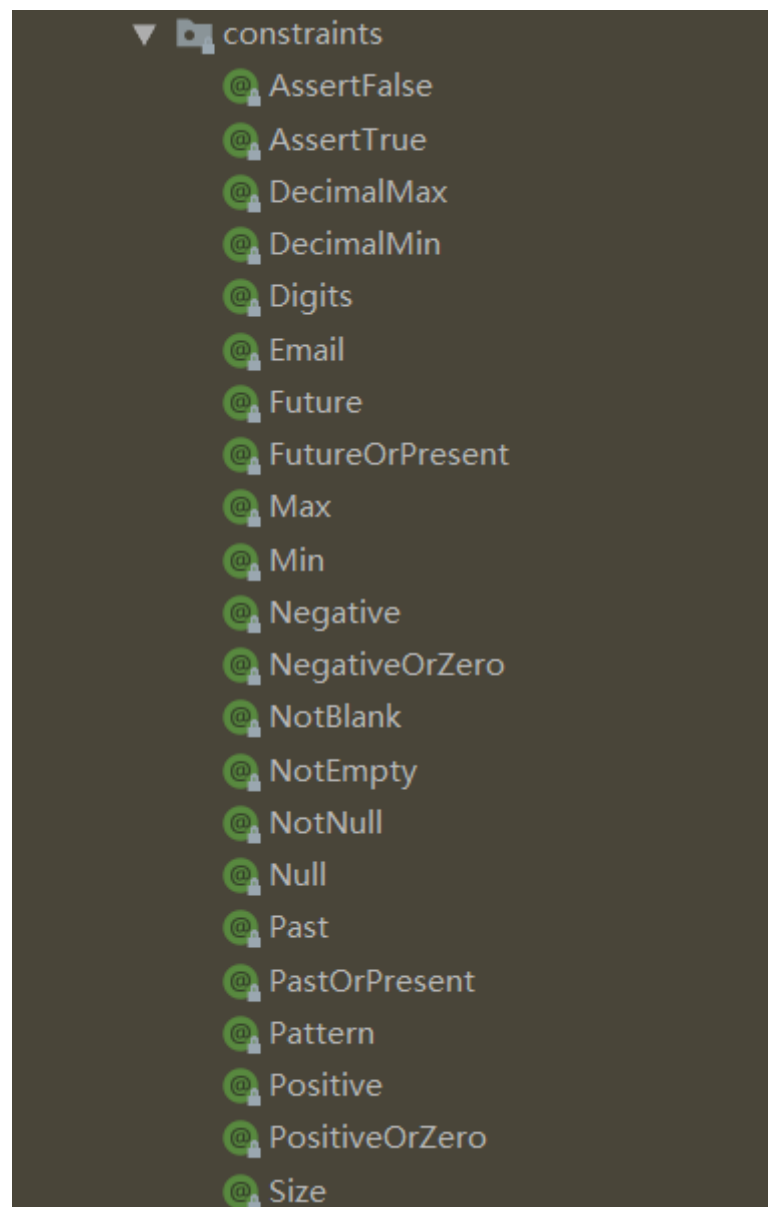
```
Dog{firstName='firstname', age=8}
```

2.5 JSR303校验

- 先在类上使用注解：@Validated //数据校验
- 之后属性（字段）上可进行的校验（也是使用注解）：

Constraint	详细信息
@Null	被注释的元素必须为 null
@NotNull	被注释的元素必须不为 null
@AssertTrue	被注释的元素必须为 true
@AssertFalse	被注释的元素必须为 false
@Min(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@Max(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@DecimalMin(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@DecimalMax(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@Size(max, min)	被注释的元素的大小必须在指定的范围内
@Digits (integer, fraction)	被注释的元素必须是一个数字，其值必须在可接受的范围内
@Past	被注释的元素必须是一个过去的日期
@Future	被注释的元素必须是一个将来的日期
@Pattern(value)	被注释的元素必须符合指定的正则表达式

Constraint	详细信息
@Email	被注释的元素必须是电子邮箱地址
@Length	被注释的字符串的大小必须在指定的范围内
@NotEmpty	被注释的字符串的必须非空
@Range	被注释的元素必须在合适的范围内



例如：

```

1  @Validated //数据校验
2  public class Person {
3      @Email(message = "邮箱格式错误")
4      private String name;
5
6      ... 省略...

```

```
1 person:
2   name: Geekst${random.uuid}
```

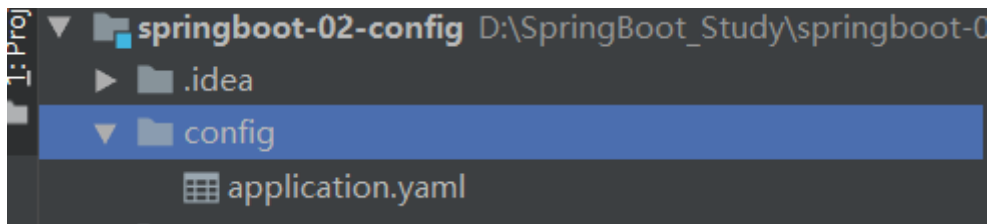
```
Property: person.name
Value: Geekst11f30151-5de5-4744-95f5-05eec838dd88
Origin: class path resource [application.yaml]:18:9
Reason: 邮箱格式错误
```

message用于出错时的提示。

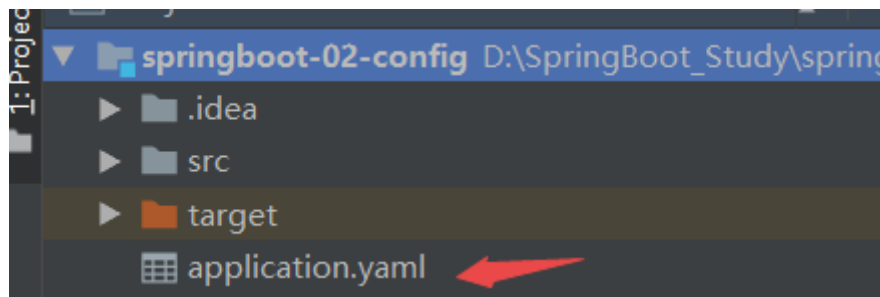
3 多环境配置

3.1 有四种地方可以配置application.yaml

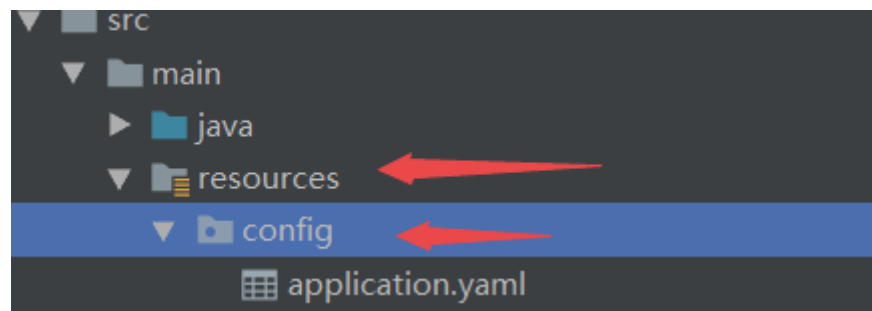
- 优先级最高：项目路径下config文件夹下



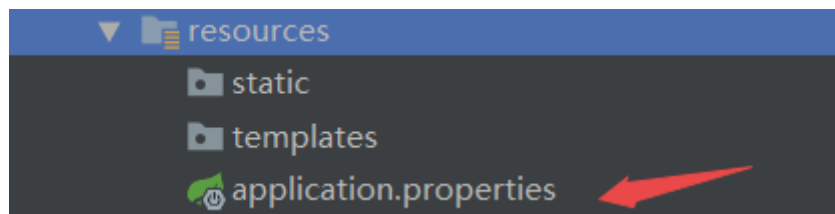
- 优先级第二：项目路径下



- 优先级第三：classpath的config下



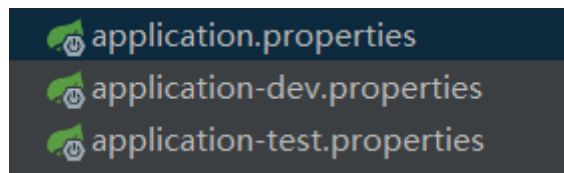
- 优先级最低（也就是默认的）：classpath下



所以，系统给我们的默认文件是优先级最低的，我们可以在外面建，直接覆盖。

3.1 多环境之Properties配置

假设有三个环境：



- dev中: server.port=8082
- test中: server.port=8081
- 而默认的应用程序.properties已经是默认8080, 就不需要写了。

如果我们想用dev的环境, 就在application.properties中进行如下的配置:

```
1 # SpringBoot的多环境配置, 可以选择激活哪一个配置文件 (环境)
2 spring.profiles.active=dev
```

这里我们只需要写 - 后面的名字 (test、dev) 就好。

```
2020-02-22 15:02:20.630 INFO 15576 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8082 (http)
2020-02-22 15:02:20.636 INFO 15576 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-02-22 15:02:20.636 INFO 15576 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.30]
2020-02-22 15:02:20.689 INFO 15576 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-02-22 15:02:20.690 INFO 15576 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 620 ms
2020-02-22 15:02:20.841 INFO 15576 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-02-22 15:02:20.933 INFO 15576 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s) 8082 (http) with context path ''
2020-02-22 15:02:20.935 INFO 15576 --- [main] com.kuang.Springboot02ConfigApplication : Started Springboot02ConfigApplication in 1.257 seconds (JVM running f
1.987)
```

【所以, 我们的一些配置文件的命名一定要规范! 】

3.2 多环境之yaml配置

全都写在application.yaml中:

```
1 server:
2   port: 8081
3
4 spring:
5   profiles:
6     active: dev
7   ---
8 server:
9   port: 8082
10  spring:
11    profiles: dev
12  ---
13 server:
14   port: 8083
15  spring:
16    profiles: test
```

- --- 就是将三个“模块”进行分割, 最上面的就是默认环境。
- spring:
 profiles:
 active: dev
 是进行选择环境。

```
020-02-22 15:08:47.856 INFO 7188 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8082 (http)
020-02-22 15:08:47.870 INFO 7188 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
020-02-22 15:08:47.870 INFO 7188 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.30]
020-02-22 15:08:47.990 INFO 7188 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
020-02-22 15:08:47.991 INFO 7188 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1511 ms
020-02-22 15:08:48.362 INFO 7188 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
020-02-22 15:08:48.605 INFO 7188 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s) 8082 (http) with context path ''
020-02-22 15:08:48.610 INFO 7188 --- [main] com.kuang.Springboot02ConfigApplication : Started Springboot02ConfigApplication in 2.806 seconds (JVM running for
3.714)
```

4 配置文件再理解

4.1 spring.factories

通过查看spring.factories和其中的值对应的类源码可以发现一个规律：

xxxAutoConfiguratio的默认值：xxxProperties文件 和 配置文件绑定，我们就可以进行自定义的配置

```
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfigurat
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\
```

```
prefix = "spring.http.encoding",
value = {"enabled"},
```

```
Ⓟ spring.http.encoding.charset=UTF... Charset
Ⓟ spring.http.encoding.enabled=true Boolean
Ⓟ spring.http.encoding.force (Whet... Boolean
Ⓟ spring.http.encoding.force-request Boole...
Ⓟ spring.http.encoding.force-response Bool...
Ⓟ spring.http.encoding.mapping Map<Locale,...
Ⓟ spring.http.log-request-details Boolean
```

```
private final Encoding properties;

public HttpEncodingAutoConfiguration(HttpProperties properties) {
    this.properties = properties.getEncoding();
}
```

```
.yaml x spring.factories x HttpEncodingAutoConfiguration.class x HttpProperties.java x
*/
public static class Encoding {

    public static final Charset DEFAULT_CHARSET = StandardCharsets.

    /**
     * Charset of HTTP requests and responses. Added to the "Content"
     * not set explicitly.
     */
    private Charset charset = DEFAULT_CHARSET;

    /**
     * Whether to force the encoding to the configured charset on H
     * responses.
     */
    private Boolean force;

    /**
     * Whether to force the encoding to the configured charset on H
     * Defaults to true when "force" has not been specified.
     */
    private Boolean forceRequest;

    /**
     * Whether to force the encoding to the configured charset on H
     */
}
```

4.2 举个例子

再看我们经常配置的server.port的源码：

```
@ConfigurationProperties(prefix = "server", ignoreUnknownFields = true)
public class ServerProperties {

    /**
     * Server HTTP port.
     */
    private Integer port;

    /**
     * Network address to which the server should bind.
     */
    private InetAddress address;

    @NestedConfigurationProperty
    private final ErrorProperties error = new ErrorProperties();

    /**
     */
}

@ConfigurationProperties(prefix = "server", ignoreUnknownFields = true)
public class ServerProperties {
    /**
     */
}
```

有一个类 ServerProperties，里面有各种配置：

```

/**
 * Server HTTP port.
 */
private Integer port;

/**
 * Network address to which the server should bind.
 */
private InetAddress address;

@NestedConfigurationProperty
private final ErrorProperties error = new ErrorProperties();

/**
 * Strategy for handling X-Forwarded-* headers.
 */
private ForwardHeadersStrategy forwardHeadersStrategy;

/**
 *
 */

```

4.3 总结

- 根据当前不同的条件判断，决定这个配置类是否生效！
- 一但这个配置类生效；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**；
- 所有在配置文件中能配置的属性都是在xxxxProperties类中封装的；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

5 自动装配的精髓【终结版】

- 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在SpringBoot默认写好的自动配置类当中
- 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存在在其中，我们就不需要再手动配置了
- 给容器中自动配置类添加组件的时候，会从properties类中获取某些属性。我们只需要在配置文件中指定这些属性的值即可
 - 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；【我们通过yml等配置文件对其进行修改】

6 @Conditional

- 了解完自动装配的原理后，我们知道：自动配置类必须在一定的条件下才能生效。
- 使用Conditional注解后，必须是@Conditional指定的条件成立，才给容器中添加组件，配置配里面的所有内容才生效。
- @Conditional派生注解：

@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

标注在类上或者方法上。

7 查看哪些自动配置类生效

yaml中：

```
1 | debug: true
```

分为三个等级：

- Positive matches:（自动配置类启用的：正匹配）
- Negative matches:（没有启动，没有匹配成功的自动配置类：负匹配）
- Unconditional classes:（没有条件的类）

```
Positive matches:
```

```
-----
```

```
Negative matches:
```

```
-----
```

```
Unconditional classes:
```

```
-----
```