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： key=value
- application.yml
    - 语法： key:空格value

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

> server.prot=8081

---

yaml概述

YAML是 "YAML Ain't a Markup Language" （YAML不是一种标记语言）的递归缩写。在开发的这种语言时，YAML 的意思其实是："Yet Another Markup Language"（仍是一种标记语言）

不管他是不是标记语言，反正这种语言以数据作为中心，而不是以标记语言为重点！

以前的配置文件，大多数都是使用xml来配置；比如一个简单的端口配置，我们来对比下yaml和xml

传统xml配置：

```
<server>
    <port>8081<port>
</server>
```

yaml配置：

```
server：
  prot: 8080
```

---

yml基础语法

说明：语法要求严格！

1、空格不能省略

2、以缩进来控制层级关系，只要是左边对齐的一列数据都是同一个层级的。

3、属性和值的大小写都是十分敏感的。

字面量：普通的值 [数字，布尔值，字符串]

字面量直接写在后面就可以 ， 字符串默认不用加上双引号或者单引号；

```
k: v
```

注意：

- “ ” 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；

  比如 ：name: "kuang \n shen" 输出 ：kuang 换行 shen
- ' 单引号，会转义特殊字符 ， 特殊字符最终会变成和普通字符一样输出

  比如 ：name: ‘kuang \n shen’ 输出 ：kuang \n shen

对象、Map（键值对）

```
#对象、Map格式
k:
    v1:
    v2:
```

在下一行来写对象的属性和值得关系，注意缩进；比如：

```
student:
    name: qinjiang
    age: 3
```

行内写法

```
student: {name: qinjiang,age: 3}
```

数组（ List、set ）

用 - 值表示数组中的一个元素,比如：

```
pets:
    - cat
    - dog
    - pig
```

行内写法

```
pets: [cat,dog,pig]
```

修改SpringBoot的默认端口号

配置文件中添加，端口号的参数，就可以切换端口；

```
server:
  port: 8082
```

注入配置文件：给实体类注入匹配值

原来是可以用@Value注解来注入：

```
@Component //注册bean
public class Dog {
    @Value("阿黄")
    private String name;
    @Value("18")
    private Integer age;
}
```

现在用yaml注入：

- 实体类

```
@Component
@Data
@ConfigurationProperties(value = "person")
public class Person {
    String name;
    Integer age;
    boolean happy;
    Date birthday;
    Map<String, Double> salary;
    List<Object> hobbies;
    Dog dog;
}
```

- application.properties

```
person:
  name: liu
  age: 3
  happy: false
  birth: 2000/01/01
  salary: {1: 10000,2: 8000}
  hobbies:
    - code
    - girl
    - music
  dog:
    name: 旺财
    age: 3
```

- 测试代码：

```
@SpringBootTest
class ApplicationTest {

    @Autowired
    private Dog dog;
    @Autowired
    private Person person;

    @Test
    public void contextLoads() {
```

```
        System.out.println(person);
    }

}
```

需要注意的是，使用配置注入属性还需要导入配置注解处理器的依赖：

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

1. 可以看到，person里面的birthDay在配置文件里是没有的，所以输出的是null
2. 注解@ConfigurationProperties从默认配置文件中去查找值

加载指定的配置文件

@PropertySource：加载指定的配置文件

1. 新建yml文件

```
@Component
@Data
@PropertySource(value = "classpath:person.yml")
public class Person {
    @Value("${name}")
    String name;
    Integer age;
    boolean happy;
    Date birthday;
    Map<String, Double> salary;
    List<Object> hobbies;
    Dog dog;
}
```

注意：@PropertySource需要跟@Value组合使用，比如：

```
@PropertySource(value = "classpath:person.yml")
@ConfigurationProperties(prefix = "person")
```

@Value 对比 @ConfigurationProperties，@Value这个使用起来并不友好！我们需要为每个属性单独注解赋值，比较麻烦；我们来看个功能对比图

	@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

1、@ConfigurationProperties只需要写一次即可 ， @Value则需要每个字段都添加

2、松散绑定：这个什么意思呢? 比如我的yml中写的last-name，这个和lastName是一样的， - 后面跟着的字母默认是大写的。这就是松散绑定。可以测试一下

3、JSR303数据校验 ， 这个就是我们可以在字段是增加一层过滤器验证 ， 可以保证数据的合法性

4、复杂类型封装，yml中可以封装对象 ， 使用value就不支持

结论：

配置yml和配置properties都可以获取到值 ， 强烈推荐 yml；

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来和配置文件进行一一映射，就直接@configurationProperties，不要犹豫！

配置文件占位符

1、随机数

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

2、占位符获取之前配置的值，如果没有可以是用:指定默认值

```
person.last-name=张三${random.uuid}
person.age=${random.int}
person.birth=2017/12/15
person.boss=false
person.maps.k1=v1
person.maps.k2=14
person.lists=a,b,c
person.dog.name=${person.hello:hello}_dog
person.dog.age=15
```

多环境配置

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

例如：

application-test.properties 代表测试环境配置

application-dev.properties 代表开发环境配置

但是Springboot并不会直接启动这些配置文件，它**默认使用application.properties主配置文件**；

我们需要通过一个配置来选择需要激活的环境：

```
#比如在配置文件中指定使用dev环境，我们可以通过设置不同的端口号进行测试；
#我们启动SpringBoot，就可以看到已经切换到dev下的配置了；
spring.profiles.active=dev
```

2、yml的多文档块方式

```
server:
  port: 8081
spring:
  profiles:
    active: prod

---
server:
  port: 8083
spring:
  profiles: dev


---

server:
  port: 8084
spring:
  profiles: prod  #指定属于哪个环境
```

3、激活指定profile

1、在配置文件中指定 spring.profiles.active=dev

2、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

3、虚拟机参数；

-Dspring.profiles.active=dev

注意：如果yml和properties同时都配置了端口，并且没有激活其他环境 ， 默认会使用properties配置文件的！

外部配置文件

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：

```
优先级
1：项目路径下的config文件夹配置文件优先级
2：项目路径下配置文件优先级
3：资源路径下的config文件夹配置文件优先级
4：资源路径下配置文件
```

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；

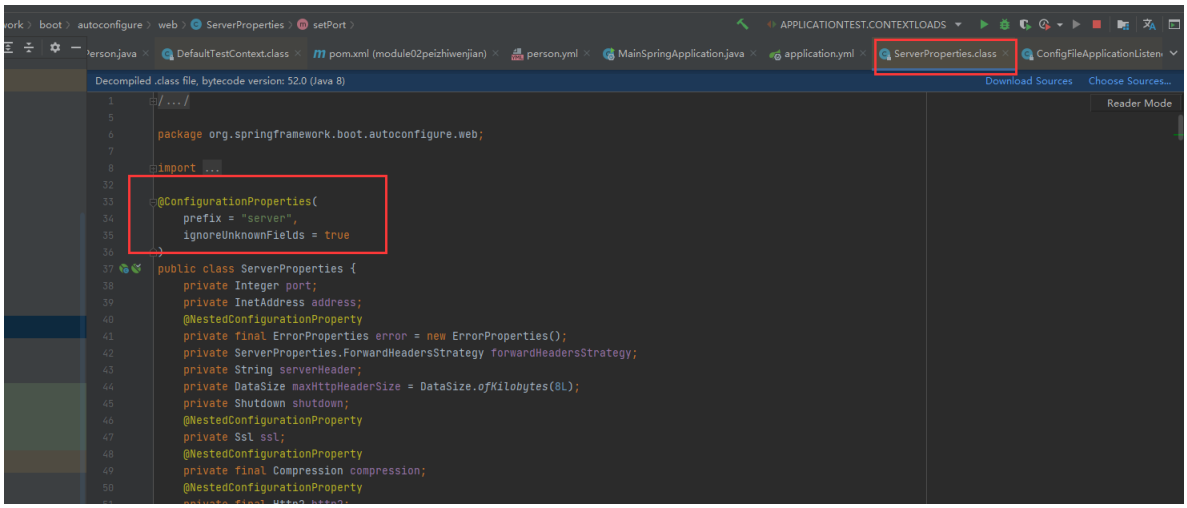
我们在最低级的配置文件中设置一个项目访问路径的配置来测试互补问题；

```
#配置项目的访问路径
server.servlet.context-path=/kuang
```

自动配置原理

配置文件能写什么？

利用配置文件到属性值得映射，spring可以通过大量的properties.class配置类，自动装配默认属性，进而被更多的地方引用



我们以**HttpEncodingAutoConfiguration（Http编码自动配置）**为例解释自动配置原理；

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

//启动指定类的ConfigurationProperties功能；
  //进入这个HttpProperties查看，将配置文件中对应的值和HttpProperties绑定起来；
  //并把HttpProperties加入到ioc容器中
@EnableConfigurationProperties({HttpProperties.class})

//Spring底层@Conditional注解
  //根据不同的条件判断，如果满足指定的条件，整个配置类里面的配置就会生效；
```

```
  //这里的意思就是判断当前应用是否是web应用，如果是，当前配置类生效
@ConditionalOnWebApplication(
    type = Type.SERVLET
)

//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；
@ConditionalOnClass({CharacterEncodingFilter.class})

//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pring.http.encoding.enabled=true，也是默认生效的；
@ConditionalOnProperty(
    prefix = "spring.http.encoding",
    value = {"enabled"},
    matchIfMissing = true
)

public class HttpEncodingAutoConfiguration {
    //他已经和SpringBoot的配置文件映射了
    private final Encoding 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(HttpProperties properties) {
        this.properties = properties.getEncoding();
    }

    //给容器中添加一个组件，这个组件的某些值需要从properties中获取
    @Bean
    @ConditionalOnMissingBean //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(org.springframework.
boot.autoconfigure.http.HttpProperties.Encoding.Type.REQUEST));

 filter.setForceResponseEncoding(this.properties.shouldForce(org.springframework
.boot.autoconfigure.http.HttpProperties.Encoding.Type.RESPONSE));
        return filter;
    }
    //。。。。。。。
}
```

总结

1、SpringBoot启动会加载大量的自动配置类

2、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；

3、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）

4、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；

@Conditional

了解完自动装配的原理后，我们来关注一个细节问题，**自动配置类必须在一定的条件下才能生效；**

@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，配置配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

那么多的自动配置类，必须在一定的条件下才能生效；也就是说，我们加载了这么多的配置类，但不是所有的都生效了。

我们怎么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；

```
#开启springboot的调试类
debug=true
```

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）