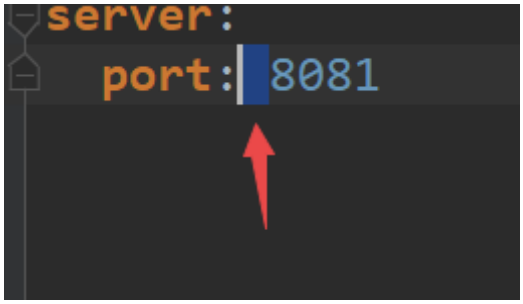
配置文件

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

```
# SpringBoot这个配置文件中到底可以配置哪些东西？

# 【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

```
package com.kuang.pojo;

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

@Component
@PropertySource(value = "classpath:geekst.properties")//加载指定的配置文件
public class People {
    //SPEL表达式取出配置文件的值
    @Value("${name}")
    private String name;

    ...getter、setter、toString()、无参构造，全参构造...
}
```

  - geekst.properties：

```
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

```yaml
Student:
  name: Geekst
  age: 2
```

  - 对比Properties

```properties
Student.name=Geekst
Student.age=2
```

- 行内写法

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

【properties无法存】

- 数组

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

# 行内写法
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

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

- 准备两个实体类：

```java
package com.kuang.pojo;

import org.springframework.stereotype.Component;

@Component //组件，扫描到Spring容器中
public class Dog {
    private String name;
    private int age;

    ...getter、setter、toString()、无参构造，全参构造...
}
```

```java
package com.kuang.pojo;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.List;
import java.util.Map;

/**
 * @ConfigurationProperties作用： 将配置文件中配置的每一个属性的值，映射到这个组件中；
 * 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
 * 参数 prefix = “person” : 将配置文件中的person下面的所有属性一一对应
 * 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
 */
@Component
@ConfigurationProperties(prefix = "person")
public class Person {
    private String name;
    private int age;
    private Boolean happy;
    private Date birth;
    private Map<String, Object> maps;
    private List<Object> lists;
    private Dog dog;

    ...getter、setter、toString()、无参构造，全参构造...
}
```

- application.yaml：

```yaml
person:
  name: Geekst
  age: 3
  happy: true
  birth: 2020/2/21
  maps: {k1: v1,k2: v2,k3: v3}
  lists:
    - code
    - music
    - girl
  dog:
    name: Geekst
    age: 3
```

核心：@ConfigurationProperties(prefix = "person")

- 测试运行

```java
@SpringBootTest
class SpringbootO2ConfigApplicationTests {
    @Autowired
    private Dog dog;

    @Autowired
    private Person person;

    @Test
    void contextLoads() {
        System.out.println(person);
    }

}
```

控制台打印：

```
Person{
    name='Geekst',
    age=3,
    happy=true,
    birth=Fri Feb 21 00:00:00 CST 2020,
    maps={k1=v1, k2=v2, k3=v3},
    lists=[code, music, girl],
    dog=Dog{name='Geekst', age=3}
}
```

2.3 使用一些表达式

```yaml
person:
  name: Geekst${random.uuid}
  age: ${random.int}
  happy: true
  birth: 2020/2/21
  maps: {k1: v1,k2: v2,k3: v3}
  lists:
    - code
    - music
    - girl
  dog:
    name: ${person.hello:nohello}_旺财
    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：

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

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， 这个就是我们可以在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，yml中可以封装对象， 使用@value就不支持
  - 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

```java
package com.kuang.pojo;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component //组件，扫描到Spring容器中，注册Bean
@ConfigurationProperties(prefix = "dog")
public class Dog {
    private String firstName;
    private int age;

    ...getter、setter、toString()、无参构造，全参构造...
}
```

```yaml
dog:
  first-name: firstname
  age: 8
```

这里的first-name和驼峰命名firstName能够对应：

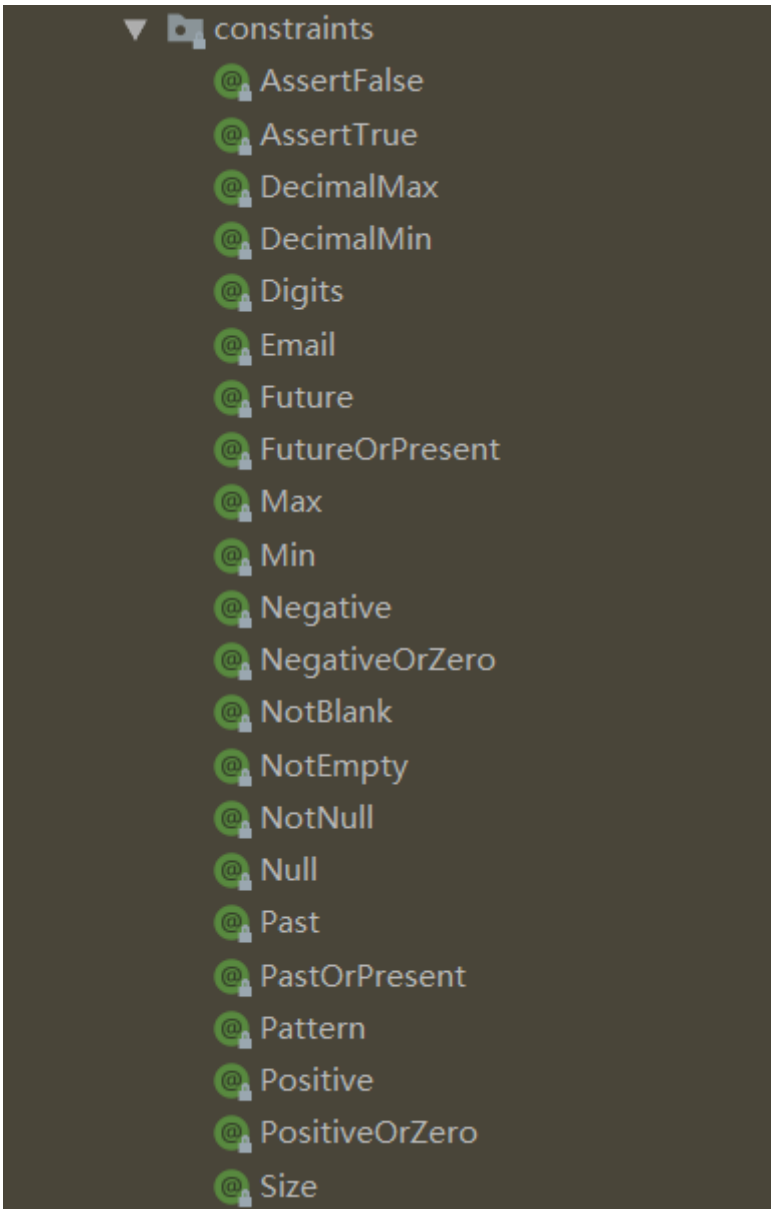
```
Dog{firstName='firstname', age=8}
```

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：

```
@Validated //数据校验
public class Person {
    @Email(message = "邮箱格式错误")
    private String name;

    。。。。省略。。。
```

```yaml
person:
  name: Geekst${random.uuid}
```

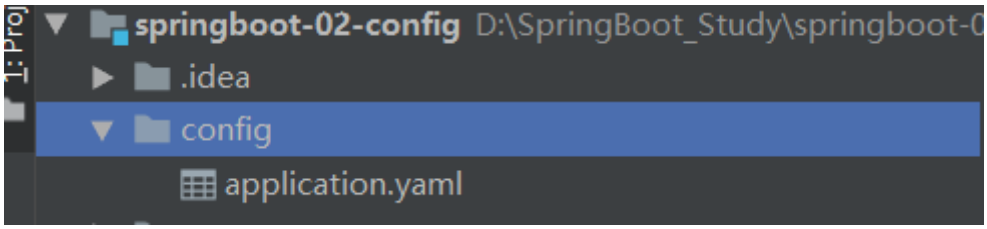
```
Property: person.name
Value: Geekst11f30151-5de5-4744-95f5-05eec838dd88
Origin: class path resource [application.yaml]:18:9
Reason: 邮箱格式错误
```

message用于出错时的提示。

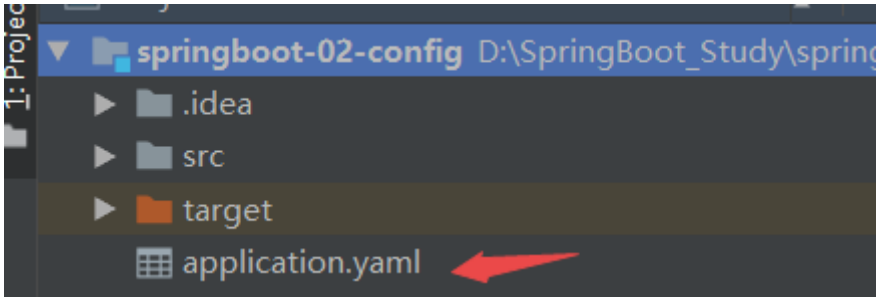
3 多环境配置

3.1 有四种地方可以配置application.yaml

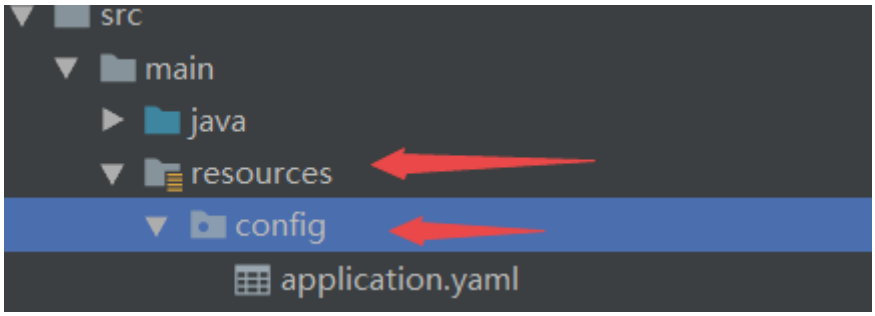
- 优先级最高：项目路径下config文件夹下



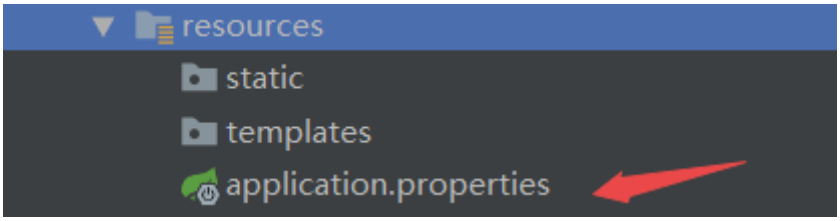
- 优先级第二：项目路径下



- 优先级第三：classpath的config下



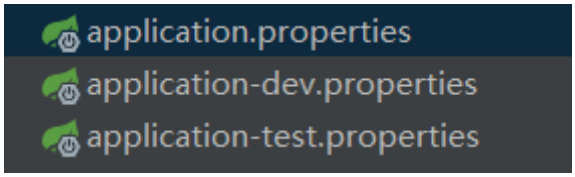
- 优先级最低（也就是默认的）：classpath下



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：

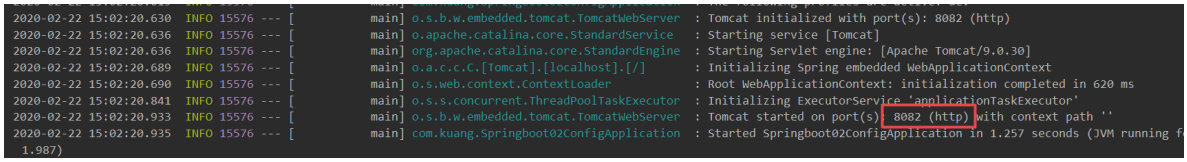


- dev中：server.port=8082
- test中：server.port=8081
- 而默认的application.properties已经是默认8080，就不需要写了。

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

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

这里我们只需要写 - 后面的名字（test、dev）就好。



【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

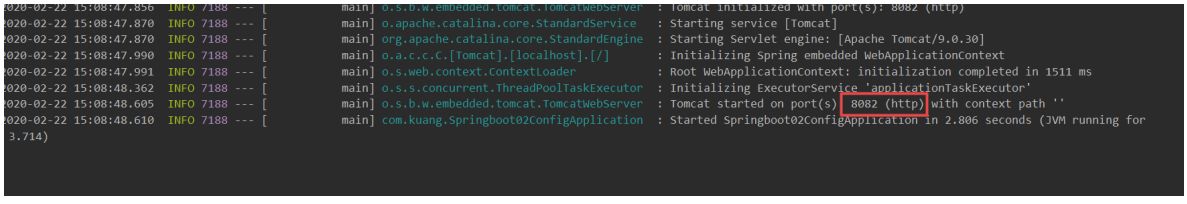
全都写在application.yaml中：

```
server:
  port: 8081

spring:
  profiles:
    active: dev
---
server:
  port: 8082
spring:
  profiles: dev
---
server:
  port: 8083
spring:
  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。



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

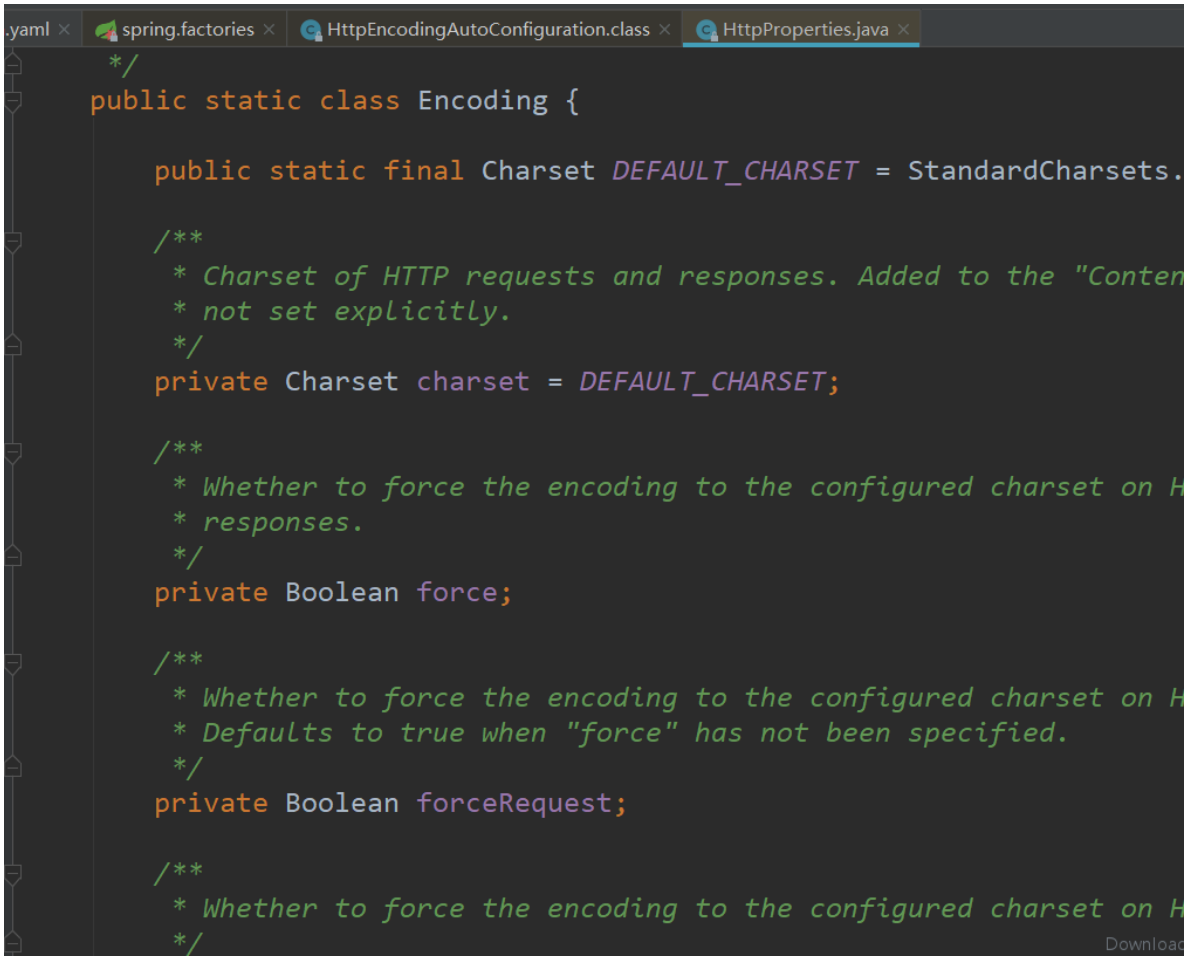
```
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\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

```
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,…
```

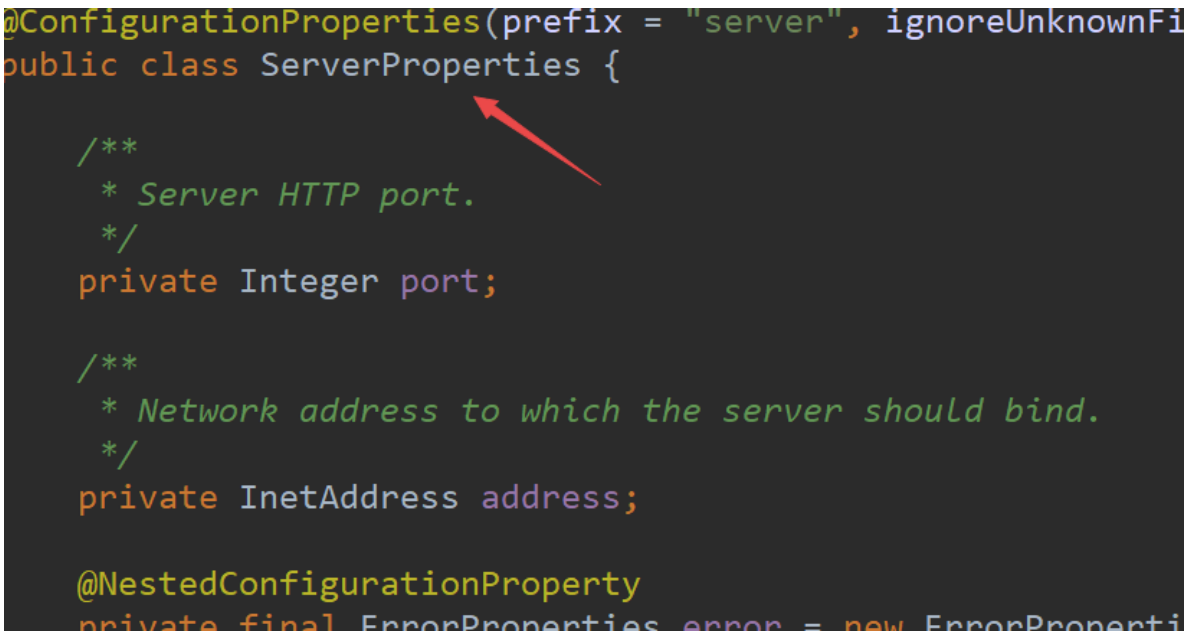
```
private final Encoding properties;

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

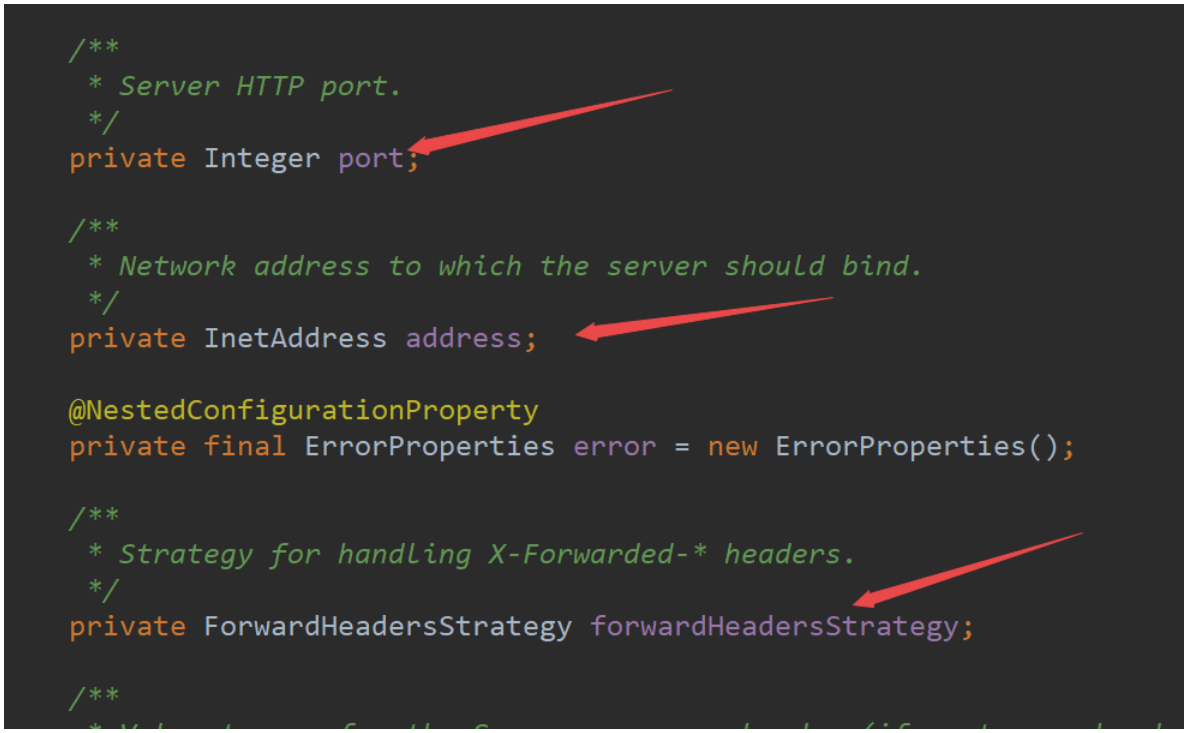


4.2 举个例子

再看我们经常配置的server.port的源码：



有一个类 ServerProperties，里面有各种配置：



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：

```yaml
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）