Spring Boot的配置文件和自动配置原理

1.使用Spring Initializer快速创建Spring Boot项目

1、IDEA：使用 Spring Initializer快速创建项目

IDE都支持使用Spring的项目创建向导快速创建一个Spring Boot项目；

选择我们需要的模块；向导会联网创建Spring Boot项目；

默认生成的Spring Boot项目；

- 主程序已经生成好了，我们只需要我们自己的逻辑
- resources文件夹中目录结构
  - **static**：保存所有的静态资源； js css images；
  - **templates**：保存所有的模板页面；（Spring Boot默认jar包使用嵌入式的Tomcat，默认不支持JSP页面）；可以使用模板引擎（freemarker、thymeleaf）
  - **application.properties**：Spring Boot应用的配置文件；可以修改一些默认设置；

2.自定义SpringApplication

如果SpringApplication默认设置不符合您的喜好，则可以创建一个本地实例并对其进行自定义。例如，要关闭横幅，您可以编写：

```
public static void main(String[] args) {
 SpringApplication app = new SpringApplication(MySpringConfiguration.class);
 app.setBannerMode(Banner.Mode.OFF);
 app.run(args);
}
```

- **通过构造者模式流式构造SpringApplication:**

```
new SpringApplicationBuilder()
 .bannerMode(Banner.Mode.OFF)
 .run(args);
```

3.配置文件的使用

3.1 配置文件介绍

SpringBoot使用一个全局的配置文件 核心配置文件，配置文件名**在约定的情况下 名字**是固定的；

配置文件的作用：修改SpringBoot自动配置的默认值；SpringBoot在底层都给我们自动配置好；

- **application.properties**
- **application.yml**
- **application.yaml**

YAML（YAML Ain't Markup Language）

YAML A Markup Language：是一个标记语言

YAML isn't Markup Language：不是一个标记语言；

两种配置文件的格式

在springboot框架中，resource文件夹里可以存放配置的文件有两种：properties和yml。

1、application.properties的用法：扁平的k/v格式。

```
server.port=8081
server.servlet.context-path=/tuling
```

2、application.yml的用法：树型结构。

```
server:
 port: 8088
 servlet:
 context-path: /tuling
```

两种前者是，而后者是yml的，建议使用后者，因为它的可读性更强。 可以看到要转换成YML我们只需把properies里按.去拆分即可。

3.2 yml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

属性和值也是大小写敏感；

如果有特殊字符% & 记得用单引号（ ‘）包起来

3.3 配置文件的加载顺序：

```
<includes>
 <include>**/application*.yml</include>
 <include>**/application*.yaml</include>
 <include>**/application*.properties</include>
</includes>
```

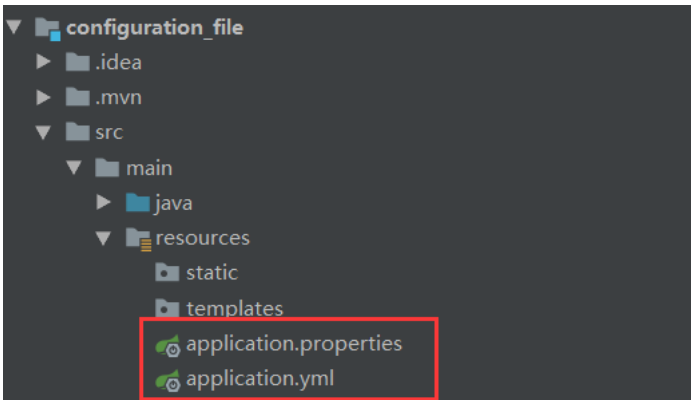
如果同时存在不同后缀的文件按照这个顺序加载主配置文件；互补配置；

3.3.4 外部约定配置文件加载顺序：

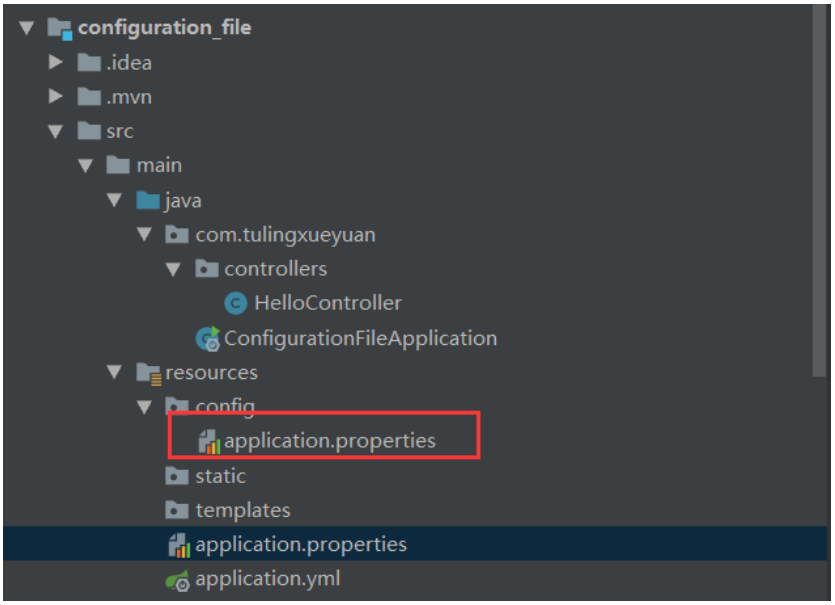
springboot 启动还会扫描以下位置的application.properties或者application.yml文件作为Spring boot的默认配置文件

低↓

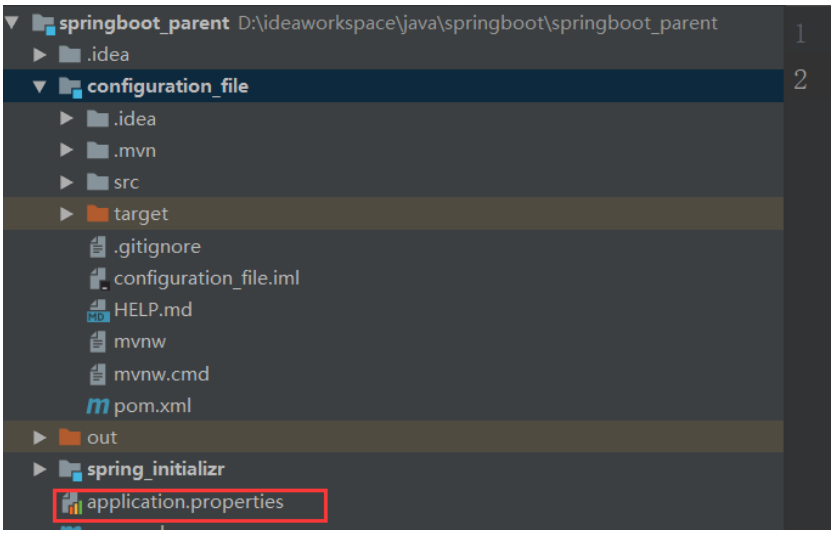
1. classpath根目录下的



2. classpath根config/

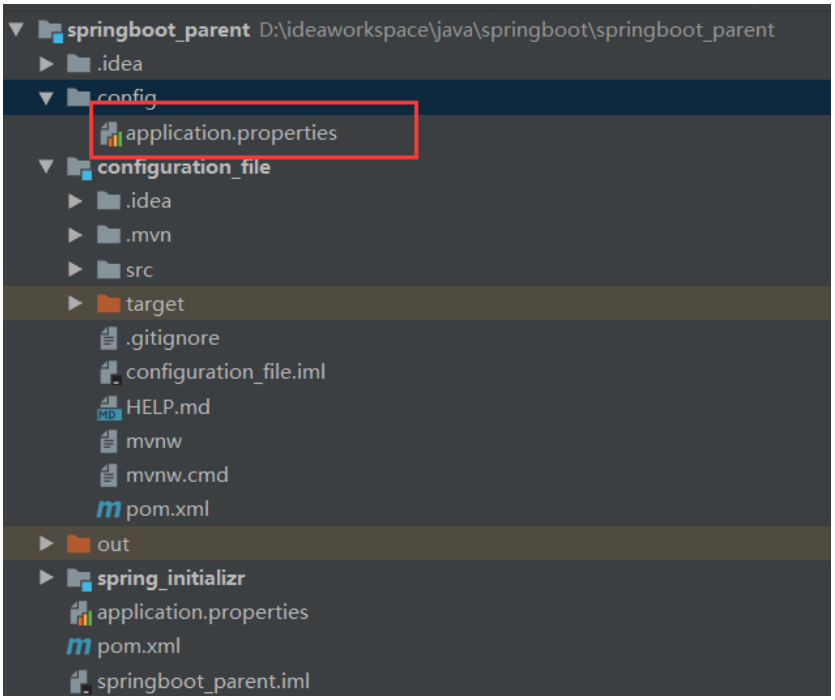


3. 项目根目录



如果当前项目是继承/耦合 关系maven项目的话，项目根目录=父maven项目的根目录

4. 项目根目录/config



5. 直接子目录/config

```
java -jar configuration_file-0.0.1-SNAPSHOT.jar --spring.config.location=D:\config/
```

高↓

优先级由底到高，高优先级的配置会覆盖低优先级的配置；互补配置；

官网：

```
optional:classpath:/
optional:classpath:/config/
optional:file:./
optional:file:./config/*/
optional:file:./config/
optional:classpath:custom-config/ --spring.config.location
optional:file:./custom-config/ --spring.config.location
```

3.5 Profile文件的加载

Profile的意思是配置，对于应用程序来说，不同的环境需要不同的配置。

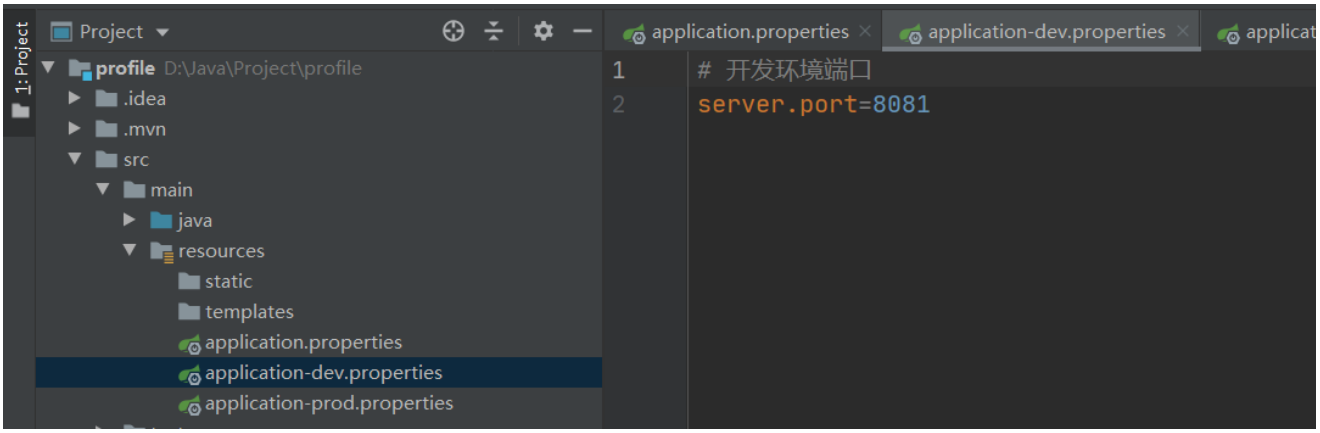
SpringBoot框架提供了多profile的管理功能，我们可以使用profile功能来区分不同环境的配置。

1、多Profile文件

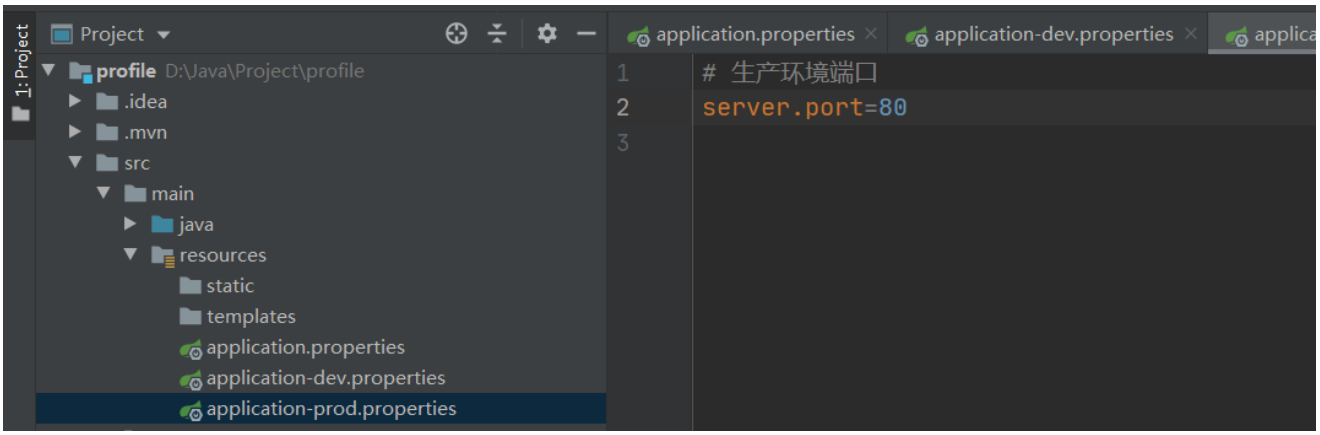
1.Spring官方给出的语法规则是application-{profile}.properties（.yaml/.yml）。

2.如果需要创建自定义的的properties文件时，可以用**application-xxx.properties**的命名方式， 根据实际情况，我创建了一个开发环境下使用的properties文件和一个生产环境下使用的properties文件，其中只对端口进行了配置，如下图所示：

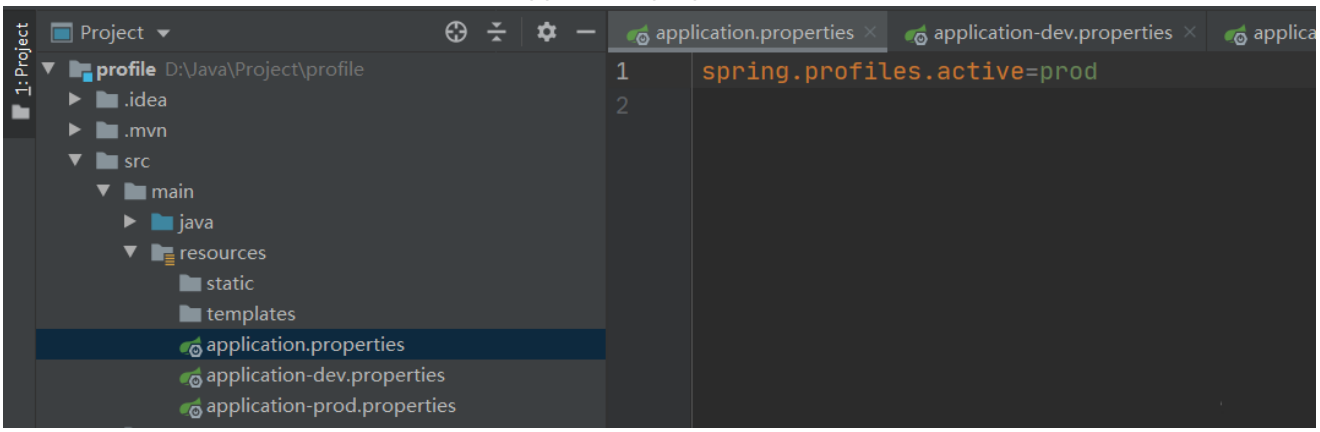
a.开发环境如下：



b.生产环境如下：



3.若我们需要在两种环境下进行切换，只需要在application.properties中加入如下内容即可。




```
先按照位置来读取优先级， 在同一位置下profile优先级最高， 如果没有指定profile，先yml--yaml--properties
```

2、激活指定profile

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

```
java -jar configuration_file-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev;
```

- **还可以通过spring.config.location来改变默认的配置文件**

使用spring.config.location 指定的配置文件， 是不会进行互补。

```
java -jar configuration_file-0.0.1-SNAPSHOT.jar --spring.config.location=D:/application.properties
```

- **还可以通过spring.config.name来改变默认的配置文件**
- **是不会进行互补。**

```
java -jar configuration_file-0.0.1-SNAPSHOT.jar --spring.config.name=application-prod
```

3.6 所有配置文件按以下顺序考虑： 优先级从低到高

1. 打包在jar中配置文件
2. 打包在jar中profile
3. 打包的jar之外的配置文件
4. 打包的jar之外的profile

```
java -jar configuration_file-0.0.1-SNAPSHOT.jar

jar包之外的配置文件 yml-->yaml-->properties
optional:classpath:/config/ yml-->yaml-->properties
optional:classpath:/ yml-->yaml-->properties


```

```
java -jar configuration_file-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev

jar包之外的配置文件 profile-dev --> yml-->yaml-->properties
optional:classpath:/config/ profile-dev --> yml-->yaml-->properties
optional:classpath:/ profile-dev --> yml-->yaml-->properties


```

```
java -jar configuration_file-0.0.1-SNAPSHOT.jar --spring.config.location=D:/application.properties

优先级最大， 因为指定了具体的配置文件。 所以不会和默认的约定配置文件进行互补

```

3.7配置文件读取方式 低---高

1. `@PropertySource``@Configuration`类上的注释。请注意，`Environment`在刷新应用程序上下文之前，不会将此类属性源添加到中。现在配置某些属性（如`logging.*`和`spring.main.*`在刷新开始之前先读取）为时已晚。
    a. 会和约定的配置文件形成互补
    b. 一定要指定.properties配置

```
@PropertySource("classpath:appSource.properties")
```

2. 默认属性（通过设置指定`SpringApplication.setDefaultProperties`）。
    a. 会和约定的配置文件形成互补

```
public static void main(String[] args) throws IOException {
 SpringApplication springApplication = new SpringApplication(ExternConfigurationApplication.class);

 // 创建Properties
 Properties properties = new Properties();
 // 通过当前类的ClassLoader
 InputStream is= ExternConfigurationApplication.class.getClassLoader()
 .getResourceAsStream("app.properties");
 // 将输入流读取成properties
  properties.load(is);

  springApplication.setDefaultProperties(properties);
  springApplication.run(args);
```

```
}
```

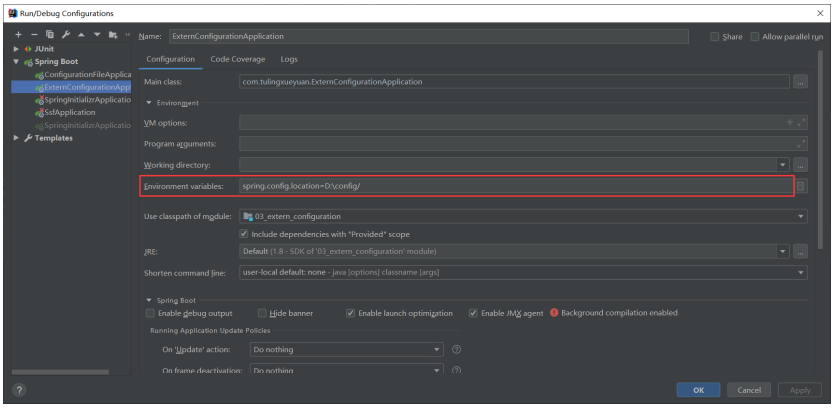
3. 配置数据（例如`application.properties`文件）

    a. 约定配置文件

4. 操作系统环境变量。

    a. 会使约定配置文件失效

    b. 1.idea



    c. 2.windows

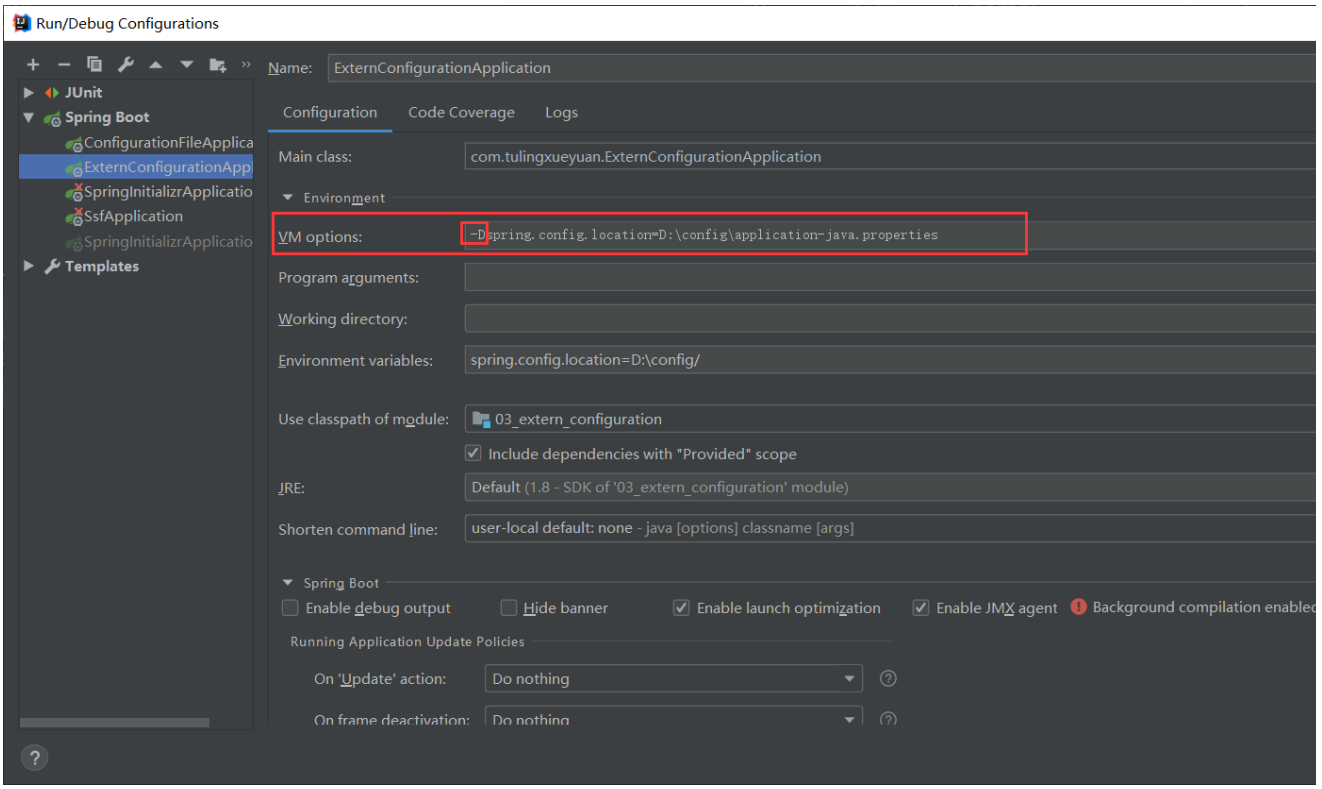
```
set spring.config.location=D:\config/
java -jar 03_extern_configuration-0.0.1-SNAPSHOT.jar

```

5. Java系统属性（`System.getProperties()`）。

    a. 会使约定配置文件失效

    b. idea



    c. 命令行java属性

```
java -Dspring.config.location=D:\config\application-java.properti
es -jar 03_extern_configuration-0.0.1-SNAPSHOT.jar
```

6. 的JNDI属性`java:comp/env`。

7. `ServletContext` 初始化参数。

```
ServletContext 的配置标签需要写到 web-app （根标签）中 ，具体如下：
 <context-param>
 <param-name>spring.config.location</param-name>
 <param-value>xxx.properties</param-value>
 </context-param>
```

8. `ServletConfig` 初始化参数。

```
ServletConfig 的配置标签需要写到 Servlet 标签中，标签如下：
<init-param>
 <param-name>spring.config.location</param-name>
 <param-value>xxx.properties</param-value>
 </init-param>
```

9. 来自的属性`SPRING_APPLICATION_JSON`（嵌入在环境变量或系统属性中的嵌入式JSON）。

10. 命令行参数。

    a. 会使约定配置文件失效

```
java -jar configuration_file-0.0.1-SNAPSHOT.jar --spring.config.location=D:/application.properties
```

11. `properties`测试中的属性。可用于测试应用程序的特定部分`@SpringBootTest`的测试注释和注释。

12. `@TestPropertySource` 测试中的注释。

    a. 用在单元测试上的

```
@TestPropertySource("classpath:appSource.properties")
```

13. `$HOME/.config/spring-boot`当devtools处于活动状态时，目录中的Devtools全局设置属性。

4、配置文件值注入

将YAML映射到属性

- **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

```
friends:
 lastName: zhangsan
 age: 20
```

行内写法：

```
friends: {lastName: zhangsan,age: 18}
```

- **数组（List、Set）：**

用- 值表示数组中的一个元素

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

行内写法

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

配置文件

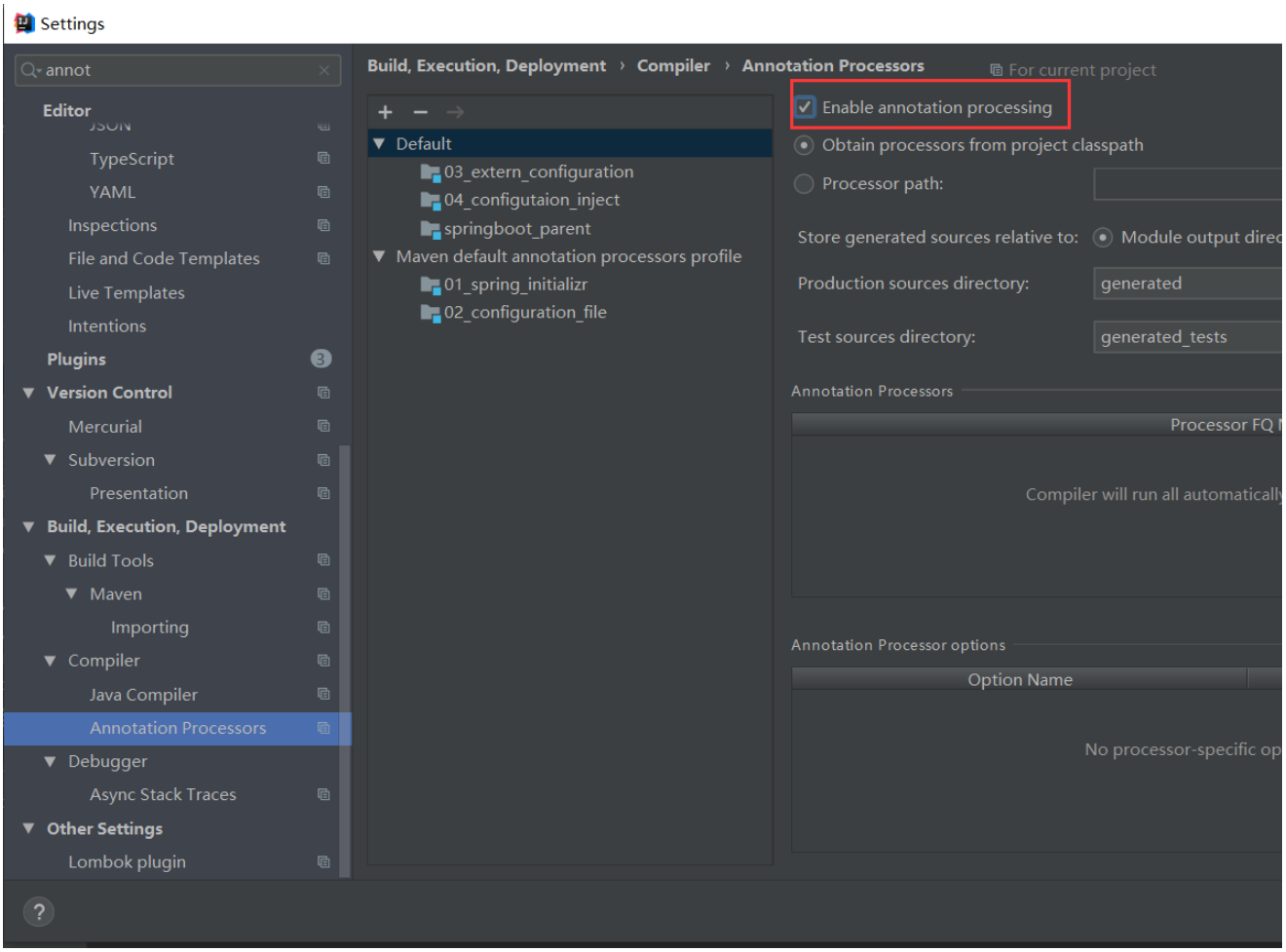
```yaml
person:
 lastName: hello
 age: 18
 boss: false
 birth: 2017/12/12
 maps: {k1: v1,k2: 12}
 lists:
 - lisi
 - zhaoliu
  dog:
  name: 小狗
  age: 12
```

javaBean:

```
/**
* 将配置文件中配置的每一个属性的值，映射到这个组件中
* @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；
* prefix = "person"：配置文件中哪个下面的所有属性进行一一映射
*
* 只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；
*
*/
@Component
 @ConfigurationProperties(prefix = "person")
public class Person {
private String lastName;
private Integer age;
private Boolean boss;
private Date birth;
private Map<String,Object> maps;
private List<Object> lists;
private Dog dog;
```

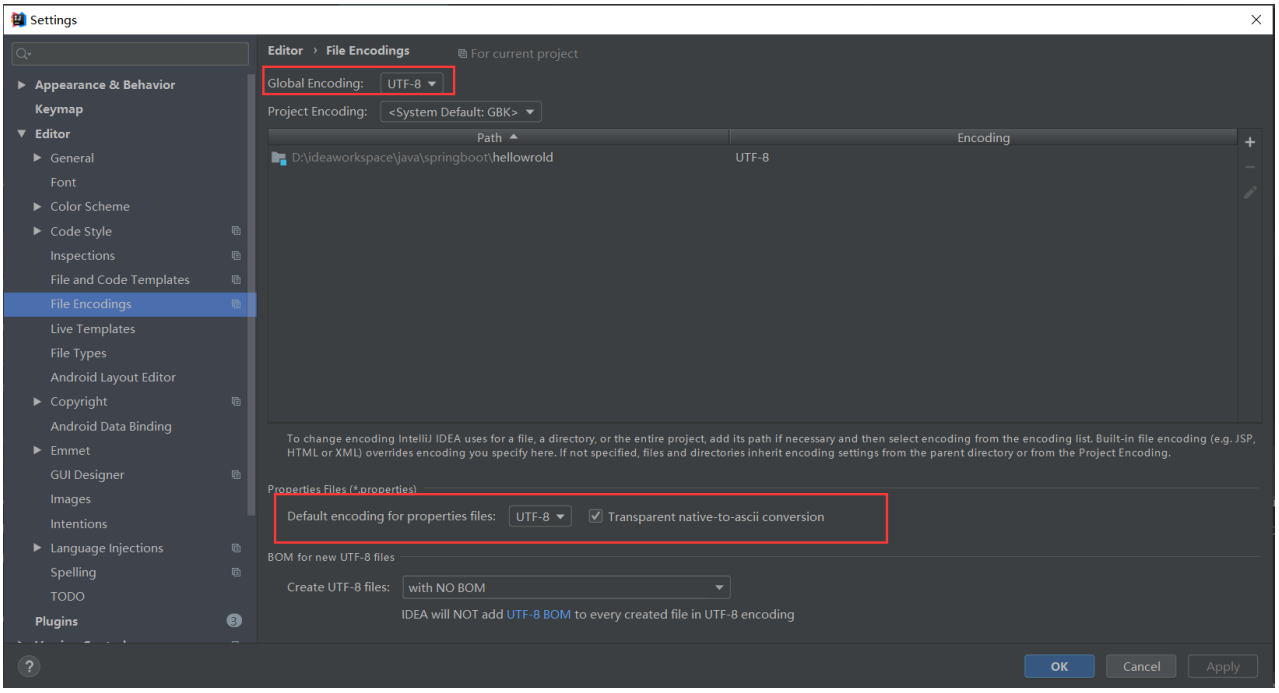
我们可以导入配置文件处理器，以后编写配置就有提示了

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



properties配置文件在idea中默认utf-8可能会乱码

调整



松散绑定：

```yaml
user:
  USERNAME: 徐庶
user:
```

```
  userName: 徐庶
  user:
  user_name: 徐庶
user:
  user-name: 徐庶

以上4种命名是可以自动绑定bean属性 User.username
```

@Value获取值和@ConfigurationProperties获取值比较

	@ConfigurationProperties	@Value
绑定	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	支持有限
SpEL	不支持	支持
自动提示	支持	不支持

配置文件yml还是properties他们都能获取到值；

使用场景

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；

3、配置文件注入值数据校验

```
@Component
@ConfigurationProperties(prefix = "person")
@Validated
public class Person {
/**
* <bean class="Person">
* <property name="lastName" value="字面量/${key}从环境变量、配置文件中获取值/#
{SpEL}"></property>
* <bean/>
 */
 //lastName必须是邮箱格式
 @Email
 //@Value("${person.last-name}")
 private String lastName;
 //@Value("#{11*2}")
 private Integer age;
 //@Value("true")
 private Boolean boss;
 private Date birth;
 private Map<String,Object> maps;
 private List<Object> lists;
 private Dog dog;
```

- **@PropertySource**：加载指定的配置文件；

```java
/**
 * 将配置文件中配置的每一个属性的值，映射到这个组件中
 * @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；
 * prefix = "person"：配置文件中哪个下面的所有属性进行一一映射
 *
 * 只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；
 * @ConfigurationProperties(prefix = "person")默认从全局配置文件中获取值；
 *
 */
@PropertySource(value = {"classpath:person.properties"})
@Component
@ConfigurationProperties(prefix = "person")
//@Validated
public class Person {
/**
 * <bean class="Person">
 * <property name="lastName" value="字面量/${key}从环境变量、配置文件中获取值/#
{SpEL}"></property>
 * <bean/>
 */
//lastName必须是邮箱格式
// @Email
//@Value("${person.last-name}")
private String lastName;
//@Value("#{11*2}")
private Integer age;
//@Value("true")
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

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

4.1、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
```

5. Spring Boot的配置和自动配置原理

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]

.@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需要运行这个类的main方法来启动SpringBoot应用；

- ### SpringBootApplication

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

`@Target(ElementType.TYPE)` 设置当前注解可以标记在哪

`@Retention(RetentionPolicy.RUNTIME)` 当注解标注的类编译以什么方式保留

`RetentionPolicy.RUNTIME` 会被jvm加载

`@Documented` java doc 会生成注解信息

`@Inherited` 是否会被继承

> `@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，会帮我们自动去加载 自动配置类**
> `@ComponentScan ：` 扫描包 相当于在spring.xml 配置中<context:comonent-scan> 但是并没有指定basepackage，如果没有指定spring底层会自动扫描当前配置类所有在的包
> `TypeExcludeFilter` springboot对外提供的扩展类， 可以供我们去按照我们的方式进行排除
> `AutoConfigurationExcludeFilter` 排除所有配置类并且是自动配置类中里面的其中一个
> **动配置功能；这样自动配置才能生效；.**

这个注解里面，最主要的就是`@EnableAutoConfiguration`，这么直白的名字，一看就知道它要开启自动配置，SpringBoot要开始骚了，于是默默进去`@EnableAutoConfiguration`的源码。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
  // 略
```

```
}
```

@AutoConfigurationPackage

将当前配置类所在包保存在BasePackages的Bean中。供Spring内部使用

@AutoConfigurationPackage

```
@Import(AutoConfigurationPackages.Registrar.class) // 保存扫描路径， 提供给spring-data-jpa 需要扫描 @Entity
public @interface AutoConfigurationPackage {
```

- 就是注册了一个保存当前配置类所在包的一个Bean

@Import(EnableAutoConfigurationImportSelector.class) 关键点!

可以看到，在@EnableAutoConfiguration注解内使用到了@import注解来完成导入配置的功能，而EnableAutoConfigurationImportSelector 实现了DeferredImportSelectorSpring内部在解析@Import注解时会调用getAutoConfigurationEntry方法，这块属于Spring的源码，有点复杂，我们先不管它是怎么调用的。 下面是2.3.5.RELEASE实现源码：

getAutoConfigurationEntry方法进行扫描具有META-INF/spring.factories文件的jar包。

```
protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
 if (!isEnabled(annotationMetadata)) {
 return EMPTY_ENTRY;
 }
 AnnotationAttributes attributes = getAttributes(annotationMetadata);
 //
 List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
 configurations = removeDuplicates(configurations);
 Set<String> exclusions = getExclusions(annotationMetadata, attributes);
 checkExcludedClasses(configurations, exclusions);
 configurations.removeAll(exclusions);
 configurations = getConfigurationClassFilter().filter(configurations);
 fireAutoConfigurationImportEvents(configurations, exclusions);
 return new AutoConfigurationEntry(configurations, exclusions);
}
```

任何一个springboot应用，都会引入spring-boot-autoconfigure，而spring.factories文件就在该包下面。spring.factories文件是Key=Value形式，多个Value时使用,隔开，该文件中定义了关于初始化，监听器等信息，而真正使自动配置生效的key是org.springframework.boot.autoconfigure.EnableAutoConfiguration，如下所示：
等同于

@Import({

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
...省略
org.springframework.boot.autoconfigure.websocket.WebSocketMessagingAutoConfiguration,\
org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration
```

})

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；
所有自动配置类表

- 每一个自动配置类进行自动配置功能；

后续： @EnableAutoConfiguration注解通过@SpringBootApplication被间接的标记在了Spring Boot的启动类上。在SpringApplication.run(...)的内部就会执行selectImports()方法，找到所有JavaConfig自动配置类的全限定名对应的class，然后将所有自动配置类加载到Spring容器中

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

```
1
2  @Configuration(proxyBeanMethods = false)
3  @EnableConfigurationProperties(ServerProperties.class)
4  @ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.SERVLET)
5  @ConditionalOnClass(CharacterEncodingFilter.class)
6  @ConditionalOnProperty(prefix = "server.servlet.encoding", value = "enabled", matchIfMissing = true)
7  public class HttpEncodingAutoConfiguration {
8
9   private final Encoding properties;
10
11   public HttpEncodingAutoConfiguration(ServerProperties properties) {
12   this.properties = properties.getServlet().getEncoding();
13   }
14
15   @Bean
16   @ConditionalOnMissingBean
17   public CharacterEncodingFilter characterEncodingFilter() {
18   CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();
19   filter.setEncoding(this.properties.getCharset().name());
20   filter.setForceRequestEncoding(this.properties.shouldForce(Encoding.Type.REQUEST));
21   filter.setForceResponseEncoding(this.properties.shouldForce(Encoding.Type.RESPONSE));
22   return filter;
23   }
```

@Configuration(proxyBeanMethods = false)

- 标记了@Configuration Spring底层会给配置创建cglib动态代理。 作用：就是防止每次调用本类的Bean方法而重新创建对象，Bean是默认单例的

@EnableConfigurationProperties(ServerProperties.class)

- 启用可以在配置类设置的属性 对应的类

- **@xxxConditional**根据当前不同的条件判断，决定这个配置类是否生效?

@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

我们怎么知道哪些自动配置类生效；

我们可以通过设置配置文件中：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；

```
=========================
CONDITIONS EVALUATION REPORT
=========================


Positive matches:---**表示自动配置类启用的**
-----------------
...省略...

Negative matches:---**没有匹配成功的自动配置类**
-----------------
...省略...

```

2、细节

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；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
    - 还记得这个过滤器吧， 对，就是我们以前设置编码的， 我们现在不需要去web.xml配置过滤器了， 只需要往容器中注入该过滤器。它的值都是通过properties设置的

```
public CharacterEncodingFilter characterEncodingFilter() {
    CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();
    filter.setEncoding(this.properties.getCharset().name());
    filter.setForceRequestEncoding(this.properties.shouldForce(Encoding.Type.REQUEST));
    filter.setForceResponseEncoding(this.properties.shouldForce(Encoding.Type.RESPONSE
    return filter;
}
```

比如需要配置编码那就可以推算出

```
properties.getServlet().getEncoding().setCharset(UTF_8);
```

就等于如下配置：

`properties`= server 为什么呢？ 因为 `ServerProperties` 上面有个注解

`@ConfigurationProperties(prefix = "server")` 已经设置名为server

`getServlet()`= servlet

`getEncoding()`=encoding

`setCharset(UTF_8)`= charset=UTF-8

```
server.servlet.encoding.charset=UTF-8
```

所以只有知道了自动配置的原理及源码 才能灵活的配置SpringBoot

请介绍自动配置的原理