

1. 目的

创建一个springboot应用程序， 浏览器发送hello， 服务器接受请求并处理， 响应hello world字符串给浏览器

2. 工程

POM依赖

```
<!--导入springBoot相关的依赖-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.9.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

主程序

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

controller

```
@Controller  
public class HelloController {  
  
    @ResponseBody  
    @RequestMapping("/hello")  
    public String hello(){  
        return "Hello World!";  
    }  
}
```

打包可执行jar

引入新依赖

```
<!--这个插件，可以将应用打包成一个可执行的jar包-->
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

使用命令打包

```
mvn package。会生成对应的可运行的jar包（如：springBootHello-1.0-SNAPSHOT.jar）
```

```
java -jar springBootHello-1.0-SNAPSHOT.jar
```

即使目标环境没有tomcat也没关系，因为打包的时候已经将相关环境都打包进来了[可以解压看看呀。。。]

3. Hello探究

1. POM文件

父项目以前做依赖管理的。。。

```
<!--导入springBoot相关的依赖-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.9.RELEASE</version>
</parent>
```

父项目：

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>1.5.9.RELEASE</version>
  <relativePath>../../spring-boot-dependencies</relativePath>
</parent>
```

来真正管理springBoot应用里面的所有依赖版本：Spring boot的版本仲裁中心，以后我们导入依赖默认是不需要写版本：（没有dependencies里面的管理的依赖自然需要声明版本号）

2. 启动器

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

spring-boot-starter-web：

Spring-boot-starter:场景启动器，帮我们导入了web相关的依赖。Spring-boot有很多的场景启动器，如：spring-boot-starter-web/spring-boot-starter-email

Spring-boot将所有的功能都抽取出来了，做出一个个的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，要用什么功能就导入什么场景的启动器

3. 主程序类

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

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

```
@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 {
```

@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: Spring的配置类，参考下文，配置类也是组件（@Component）

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {
```

@EnableAutoConfiguration: 开启自动配置功能；以前需要配置的东西，Spring Boot帮我们自动配置，就是告诉Spring Boot开启自动配置功能，这样自动配置才能生效啊！！！

```
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

@AutoConfigurationPackage: 自动配置包

```
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {
```

@Import: Spring的底层注解，给容器中导入一些组件。具体内容由对应的类指定

所有这里的主要目的是将主配置类（@AutoConfigurationPackage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容器中；

Annotations:

- 标注的注解 (Annotated annotation): `type = Class@1676 "interface org.springframework.boot.autoconfigure.SpringBootApplication"`
- 标注的类 (Annotated class): `introspectedClass = Class@475 "class com.luoyu.HelloWorldMainApplication"`
- Evaluate计算结果 (Evaluate calculation result): `new PackageImport(metadata).getPackageName()` result: `com.luoyu`

`registerBeanDefinitions`: 注册一些bean编译信息

`AnnotationMetadata`: 注解标注的一些元信息

`@Import(EnableAutoConfigurationImportSelector.class)`, 给容器中导入一些组件

(`EnableAutoConfigurationImportSelector.class`/开启自动配置导入选择器, 确定导入哪些组件的选择器)

```
public class EnableAutoConfigurationImportSelector
    extends AutoConfigurationImportSelector {
    @Override
    protected boolean isEnabled(AnnotationMetadata metadata) {
        if (getClass().equals(EnableAutoConfigurationImportSelector.class)) {
            return getEnvironment().getProperty(
                EnableAutoConfiguration.ENABLED_OVERRIDE_PROPERTY, Boolean.class,
                defaultValue: true);
        }
        return true;
    }
}
```

```
private ResourceLoader resourceLoader;
@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    try {
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
            .loadMetadata(this.beanClassLoader);
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        List<String> configurations = getCandidateConfigurations(annotationMetadata,
            attributes);
        configurations = removeDuplicates(configurations);
        configurations = sort(configurations, autoConfigurationMetadata);
    }
}
```

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去

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    try {
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
            .loadMetadata(this.beanClassLoader);
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        List<String> configurations = getCandidateConfigurations(annotationMetadata,
            attributes);
    }
}
```

Variables

- ↳ this = {EnableAutoConfigurationImportSelector@2196}
- ↳ annotationMetadata = {StandardAnnotationMetadata@2197}
- ↳ annotations = {Annotation[1]@2811}
- ↳ 0 = {\$Proxy7@2813}
 - ↳ h = {AnnotationInvocationHandler@2814}
 - ↳ type = {Class@1706} "interface org.springframework.boot.autoconfigure.SpringBootApplication" ... Navigate
 - ↳ memberValues = {LinkedHashMap@2815} size = 4
 - ↳ memberMethods = {Method[4]@2816}
- ↳ nestedAnnotationsAsMap = true
- ↳ introspectedClass = {Class@475} "class com.luoyu.HelloWorldMainApplication" ... Navigate

```

public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    try {
        AutoConfigurationMetadata autoConfigurationMetadata =
            AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
        configurations = removeDuplicates(configurations);
        configurations = sort(configurations, autoConfigurationMetadata);
        Set<String> exclusions = getExclusions(annotationMetadata, attributes);
        checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
    }
}

```

Variables

```

+ <--> this = {EnableAutoConfigurationImportSelector@2185}
+ <--> annotationMetadata = {StandardAnnotationMetadata@2186}
+ <--> autoConfigurationMetadata = {AutoConfigurationMetadataLoader$PropertiesAutoConfigurationMetadata@2810}
+ <--> attributes = {AnnotationAttributes@2813} size = 2
+ <--> configurations = {ArrayList@2818} size = 96
    > <--> 0 = "org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration"
    > <--> 1 = "org.springframework.boot.autoconfigure.aop.AopAutoConfiguration"
    > <--> 2 = "org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration"
    > <--> 3 = "org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration"
    > <--> 4 = "org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration"
    > <--> 5 = "org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration"
    > <--> 6 = "org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration"

```

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。。。如何获取到的呢？

```

SpringFactoriesLoader.loadFactoryNames(
    EnableAutoConfiguration.class, getClassLoader());

```

```

public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {
    String factoryClassName = factoryClass.getName();
    try {
        Enumeration<URL> urls = (classLoader != null ? classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        List<String> result = new ArrayList<String>();
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            Properties properties = PropertiesLoaderUtils.loadProperties(new UrlResource(url));
            String factoryClassNames = properties.getProperty(factoryClassName);
            result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(factoryClassNames)));
        }
    }
    return result;
}

```

使用classLoader类加载器获取一个资源，然后将资源当成properties文件，然后从properties中拿出来factoryClassNames。

Spring Boot在启动的时候从类路径下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，然后自动配置类就会生效，帮我们进行自动配置工作。。。

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。。。

4. 使用Spring Initializer创建

使用idea 的spring initializr进行创建。向导会联网创建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文件目录结构：
 - static：保存所有静态资源：js/css/images
 - templates：保存所有的模版页面
 - application.properties：Spring Boot应用的配置文件

4.1 配置文件

SpringBoot使用一个全局的配置文件，配置文件名是固定的
(application.properties/application.yml)

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了

YAML (YAML Ain't Markup Language) ，以数据为中心，比json/xml等更适合做配置文件

- YAML A Markup Language:是一个标记语言
- YAML isn't Markup Language:不是一个标记语言

```
server:  
  port: 8081
```

4.2 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。

字面量：普通的值（数字，字符串，布尔）

- 字符串默认不用加单引号或者双引号
- 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
- 单引号：会转义字符，特殊字符就是一个普通的字符串数据

对象/Map（属性和值）（键值对）：

```
server:  
  port: 8081  
  address: /abc/def
```

行内写法：

```
server: {port: 8081, address: /abc/def}
```

数据（List/Set）：用-表示数组中的一个元素

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

4.3 配置文件注入

配置文件

```
person:  
    lastName: zhangsan  
    age: 18  
    boss: false  
    birth: 2017/12/12  
    maps: {k1: v1, k2: v2}  
    lists:  
        - lisi  
        - zhaoliu  
dog:  
    name: xiaogou  
    age: 2
```

Person

```
/**  
 * 将配置文件中配置的每一个属性的值，映射到这个组建中  
 * @ConfigurationProperties:告诉SpringBoot将本类中的所有属性和配置文件中的相关  
配置进行绑定；  
 *           prefix = "person" :配置文件中那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，才能使用容器提供的功能@ConfigurationProperties  
 **/  
 @Component  
 @ConfigurationProperties(prefix = "person")  
 public class Person {  
  
    private String lastName;  
    private String age;  
    private Boolean boss;  
    private Date birth;
```

```
    private Map<String, Object> maps;
    private List<Object> list;
    private Dog dog;
}
```

单元测试和配置文件处理器

```
<!--Spring Boot进行单元测试的模块-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<!--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>
```

单元测试验证

```
/**
 * SpringBoot单元测试
 * 可以在测试期间很方便的类似编码一样进行自动注入等容器的功能
 */
@SpringBootTest
class SpringInitializrDemoApplicationTests {

    @Autowired
    Person person;

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

```
}
```

```
}
```

4.4 配置文件乱码

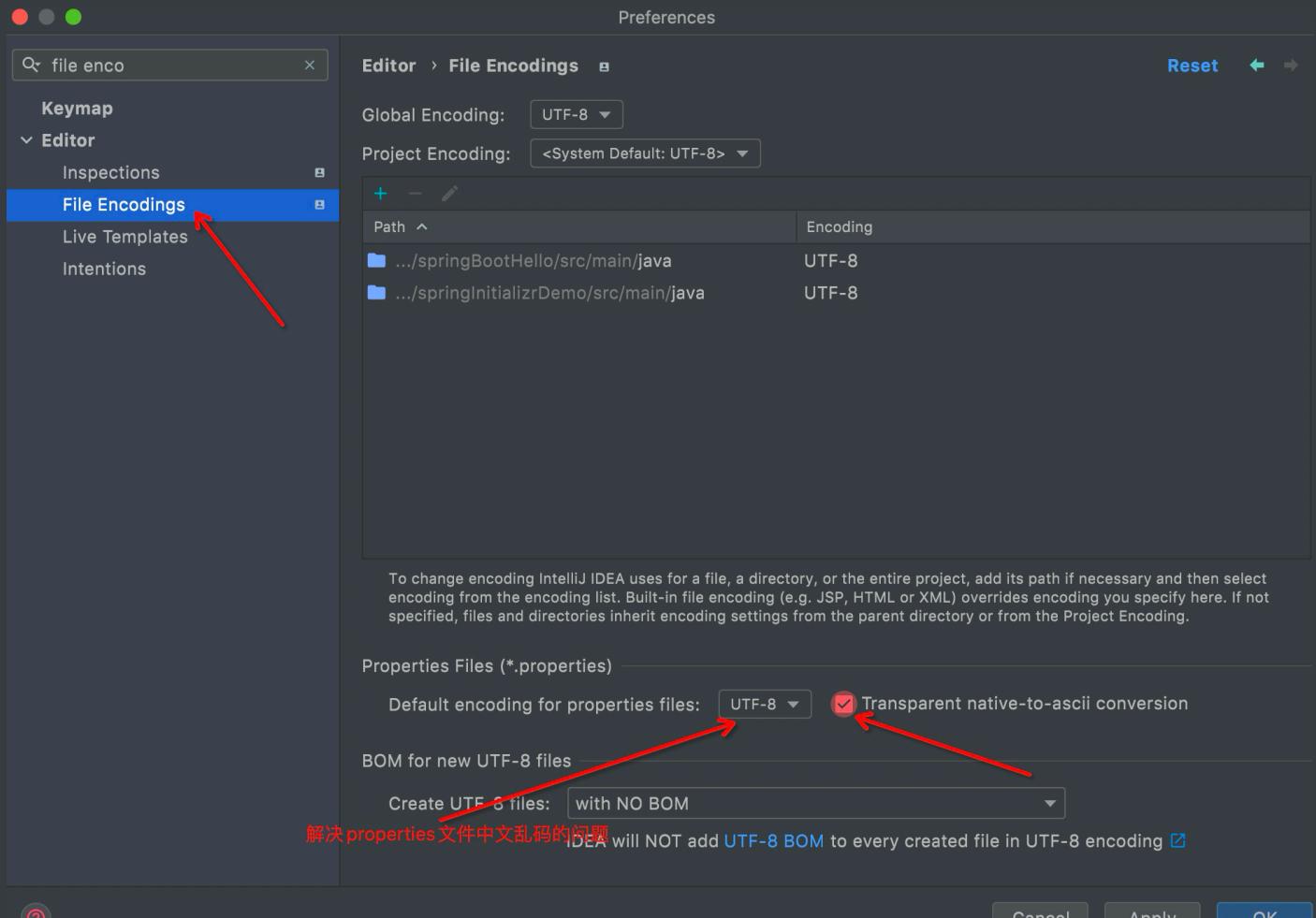
The screenshot shows an IDE interface with several windows:

- Project Explorer:** Shows files like `application.properties` and `application.yml`.
- Code Editor:** Displays the `application.properties` file with the following content:

```
#idea , properties编码默认不是uft-8
#配置person信息
person.last-name=张三
person.age=18
person.birth=2017/12/12
person.boss=false
person.list=a,b,c
person.maps.k1=v1
person.maps.k2=12
person.dog.name=dog
person.dog.age=14
```
- Test Runner:** Shows a test named `SpringInitializrDemoApplicationTests` with a green success icon and the message "Tests passed: 1 of 1 test - 162 ms".
- Terminal:** Shows the command-line output of the application:

```
:: Spring Boot ::          (v2.5.4)
2021-08-21 14:16:55.273  INFO 38188 --- [           main] l.s.SpringInitializrDemoApplicationTests : Starting SpringInitializrDemoApplicationTests ...
2021-08-21 14:16:55.276  INFO 38188 --- [           main] l.s.SpringInitializrDemoApplicationTests : No active profile has been configured, falling back to default profiles: default
2021-08-21 14:16:56.718  INFO 38188 --- [           main] l.s.SpringInitializrDemoApplicationTests : Started SpringInitializrDemoApplicationTests in 0.01 seconds (process info)
Person{lastName='??', age='18', boss=false, birth=Tue Dec 12 00:00:00 CST 2017, maps={k1=v1, k2=12}, list=[a, b, c], }
```

A red arrow points from the terminal output to the line `Person{lastName='??', age='18', ...}`, highlighting the encoding issue.



```
@Component  
// @ConfigurationProperties(prefix = "person")  
public class Person {  
  
    /**  
     * <bean class="Person">  
     *     <property name="lastName" value="?" />  
     * </bean>  
     * 分析下: value支持什么? 字面量/${key}从环境变量、配置文件中获取/#{SpEL}  
    **/  
    @Value("${person.lastName}")  
    private String lastName;
```

作用是一样的。Component 和 bean 都是将其放入容器中

The screenshot shows an IDE interface with two main sections. The top section displays a Java file named `Person.java` with code related to Spring configuration. The bottom section shows the test results for a test named `SpringInitializrDemoApplicationTests`.

```
24 ① @Component
25 // @ConfigurationProperties(prefix = "person")
26 ② public class Person {
27
28     /**
29      * <bean class="Person">
30      *     <property name="lastName" value="?" />
31      * </bean>
32      * 分析下: value支持什么? 字面量/${key}从环境变量、配置文件中获取/#{SpEL}
33      */
34     @Value("${person.last-name}")
35     private String lastName;
36     @Value("#{11*2}")
37     private String age; ③
38     @Value("false") ④
39     private Boolean boss;
```

The code uses `@Value` annotations to inject values into the `Person` class. Annotations ③ and ④ are highlighted with red arrows pointing to the test output below.

Tests passed: 1 of 1 test - 254 ms

```
:: Spring Boot :: (v2.5.4)

2021-08-21 14:33:14.231 INFO 38310 --- [           main] l.s.SpringInitializrDemoApplicationTests
2021-08-21 14:33:14.233 INFO 38310 --- [           main] l.s.SpringInitializrDemoApplicationTests
2021-08-21 14:33:15.858 INFO 38310 --- [           main] l.s.SpringInitializrDemoApplicationTests
Person{lastName='张三', age='22', boss=false, birth=null, maps=null, list=null, dog=null}
```

4.5 @Value和 @ConfigurationProperties区别

	@ConfigurationProperties	@Value
功能	批量注入文件中的属性	一个个指定
支持松散绑定Relax Binding (lastName是属性) 配置文件中可以写：person.lastName/person.last-name/person.last_name/PERSON_LAST_NAME	支持	不支持
SpEL	不支持	支持
JSR303校验	支持	不支持
复杂类型的封装	支持	不支持

```
/**
 * 将配置文件中配置的每一个属性的值，映射到这个组建中
 * @ConfigurationProperties:告诉SpringBoot将本类中的所有属性和配置文件中的相关
配置进行绑定；
 *
 *           prefix = "person" :配置文件中那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，才能使用容器提供的功能ConfigurationProperties
 */
@Component
@ConfigurationProperties(prefix = "person")
//@Validated
public class Person {

    /**
     * <bean class="Person">
     *      <property name="lastName" value="?" />
     * </bean>
     * 分析下：value支持什么？字面量/${key}从环境变量、配置文件中获取/#${SpEL}
     */
    //WithValue("${person.last-name}")
    //结合Validated，判断当前lastName的填充内容是否是邮箱
    //@Email
    private String lastName;
    //WithValue("#${11*2}")
}
```

```
    private String age;  
    // @Value("false")  
    private Boolean boss;  
    private Date birth;  
  
    private Map<String, Object> maps;  
}
```

4.5 @PropertySource @ImportResource

@PropertySource:加载指定的配置文件

```
/**  
 * 将配置文件中配置的每一个属性的值，映射到这个组建中  
 * @ConfigurationProperties:告诉SpringBoot将本类中的所有属性和配置文件中的相关  
 * 配置进行绑定；  
 *           prefix = "person" :配置文件中那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，才能使用容器提供的功能ConfigurationProperties  
 * @ConfigurationProperties(prefix = "person"): 默认从全局配置中获取值  
 **/  
  
 @PropertySource(value = {"classpath:person.properties"})  
 @Component  
 @ConfigurationProperties(prefix = "person")  
 // @Validated  
 public class Person {
```

@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配置文件生效，那么加载进来：将@ImportResource标注在配置类上

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

    public static void main(String[] args) {
        SpringApplication.run(SpringInitializrDemoApplication.class,
args);
    }

}
```

```
<?xml version="1.0" encoding="GB2312"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-
2.0.xsd">

    <bean id="helloServices"
          class="com.luoyu.springinitializrdemo.services.HelloServices">
    </bean>
</beans>
```

- Spring Boot推荐给容器中添加组件的方式：全注解的方式
 - 配置类====》 Spring配置文件
 - 使用@Bean给容器中添加组件

```
/**
 * @Configuration:指明当前类是一个配置类，就是类替代之前的Spring配置文件
 * 在配置文件中，是使用<bean><bean/>来添加组件的
 */
```

```
@Configuration  
public class AppConfig {  
  
    //将方法的返回值添加到容器中，容器中这个组件默认的id就是方法名  
    @Bean  
    public HelloServices helloServices(){  
        System.out.println("@Bean给容器中添加组件来");  
        return new HelloServices();  
    }  
}
```

4.6 配置文件的占位符

- RandomVaulePropertySource:配置文件中可以使用随机数
 - \${random.value},\${random.int},\${random.long},\${random.int(10)},{\${random.int[1024,65536]}}
- 属性配置占位符
 - app.name=MyApp
 - app.desribe=\${app.name} is a Spring Boot Application
 - 如上：可以在配置文件中引用前面配置过的属性
 - \${app.name:xxx}，用来表示找不到指定属性的值

```
#idea , properties编码默认不是uft-8  
#配置person信息  
person.last-name=落渔${random.uuid}  
person.age=${random.int}  
person.birth=2017/12/12  
person.boss=false  
person.list=a,b,c  
person.maps.k1=v1  
person.maps.k2=12  
person.dog.name=${person.hello:hello}_dog  
person.dog.age=14
```

4.7 Profile

Profile是Spring对不同环境提供不同配置功能的支持，可以通过激活/指定参数等方式快速切换

1. 多profile文件形式

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

默认使用application.properties

- application-{profile}.properties:application-dev.properties/application-prod.properties

2. YAML多文档块

```
server:  
  port: 8081  
  
spring:  
  profiles:  
    active: prod  
---  
  
server:  
  port: 8082  
  
spring:  
  profiles: dev  
---  
  
server:  
  port: 8083  
  
spring:  
  profiles: prod
```

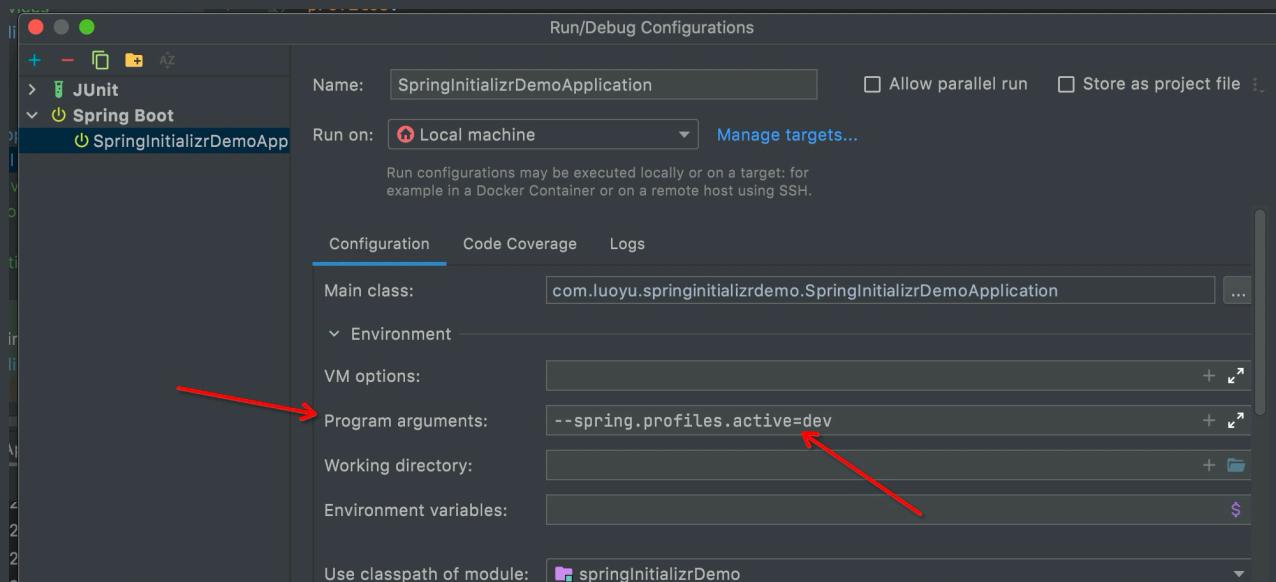
3. 激活profile,

- 在配置文件中指定

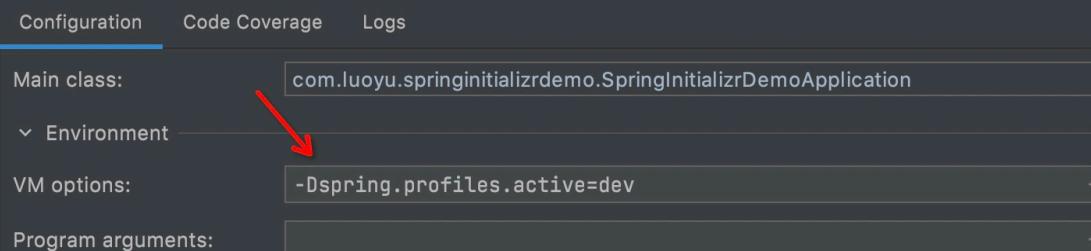
```
spring.profiles.active=prod
```

■ 命令行方式激活

```
--spring.profiles.active=dev
```



■ 虚拟机参数



4.8 配置文件加载位置

SpringBoot启动会扫描以下位置的application.properties/yml作为SpringBoot默认配置文件

```
-file:./config/  
-file:./  
-classpath:./config/  
-classpath:./
```

以上是按照优先级从高到低的顺序，所有文件都会被加载

项目打包好后，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；指定的配置文件会和默认加载的配置文件一起起作用呢：形成互补配置

4.9 外部加载配置顺序

4.10 自动配置原理

配置文件到底可以写什么？为什么？

自动配置原理：

- SpringBoot启动的时候，加载主配置类，并且开启自动配置功能
 @EnableAutoConfiguration
- @EnableAutoConfiguration作用：
 - 利用EnableAutoConfigurationImportSelector给容器中导入一些组件。详细
 可以查看selectImports里面的内容

```
List<String> configurations =
getCandidateConfigurations(annotationMetadata,
                           attributes);
//获取候选的配置
```

```
SpringFactoriesLoader.loadFactoryNames(
    getSpringFactoriesLoaderFactoryClass(),
    getBeanClassLoader());
//扫描所有jar包路径下: META-INF/spring.factories
//把扫描到的内容包装成properties对象
//从properties中获取EnableAutoConfiguration.class类名对应的
//值，然后把他们添加到容器中
```

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration
```

每一个xxxAutoConfiguration类都是容器中的一个组件，都加入到容器中，用它们来做自动配置

- 每一个自动配置类，进行自动配置功能
- 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，跟以前的配置文件一样，可以给容器中添加组件
@EnableConfigurationProperties(HttpEncodingProperties.class)
    //启用指定类HttpEncodingProperties.class的
ConfigurationProperties功能,
    //将配置文件的值跟HttpEncodingProperties属性绑定起来了
@ConditionalOnWebApplication//Spring底层@Confitional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，判断当前是否是web应用
@ConditionalOnClass(CharacterEncodingFilter.class)//判断当前项目是否有这个类【CharacterEncodingFilter，是SpringMVC里面的乱码解决过滤器】
@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，如果不存在也认为这个判断成立。)
public class HttpEncodingAutoConfiguration {
```

根据当前不同的条件，决定这个配置类是否生效！！！

```
@Bean//给容器中添加组件，这个组件的某些值需要从properteis中获取
@ConditionalOnMissingBean(CharacterEncodingFilter.class)
public CharacterEncodingFilter characterEncodingFilter() {
    CharacterEncodingFilter filter = new
OrderedCharacterEncodingFilter();
    filter.setEncoding(this.properties.getCharset().name());

    filter.setForceRequestEncoding(this.properties.shouldForce(T
pe.REQUEST));

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

- 所有在配置文件中能配置的属性都是在xxxProperties类中封装着，所有配置文件中可以配置什么？就可以参考某个功能对应的属性类

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

4. 精髓：

- SpringBoot启动会加载大量配置类
- 我们看我们需要的功能有没有SpringBoot默认写好的配置类
- 我们再来看这个自动配置类到底配置了哪些组件
- 给容器中配置组件的时候，会从properties中获取某些属性，我们就可以在配置文件中配置对应的值

xxxAutoConfiguration: 自动配置类

给容器中添加组件

xxxproperties: 封装配置文件中相关的参数

--自动配置类必须在一定的条件下生效？我们怎么知道哪些自动配置类生效？

可以通过debug=true，让控制台打印自动控制报告，这样就可以很方便的看到哪些生效。

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

Positive matches:

AopAutoConfiguration matched:

```
- @ConditionalOnProperty (spring.aop.auto=true) matched  
(OnPropertyCondition)
```

```
AopAutoConfiguration.ClassProxyingConfiguration matched:  
  - @ConditionalOnMissingClass did not find unwanted class  
'org.aspectj.weaver.Advice' (OnClassCondition)  
  - @ConditionalOnProperty (spring.aop.proxy-target-  
class=true) matched (OnPropertyCondition)
```

Negative matches:

ActiveMQAutoConfiguration:

Did not match:

```
  - @ConditionalOnClass did not find required class  
'javax.jms.ConnectionFactory' (OnClassCondition)
```

AopAutoConfiguration.AspectJAutoProxyingConfiguration:

Did not match:

```
  - @ConditionalOnClass did not find required class  
'org.aspectj.weaver.Advice' (OnClassCondition)
```

5. 日志

5.1 日志框架

JUL/JCL/Jboss-Logging/logback/log4j/log4j2/slf4j

日志门面：

- JCL (~~jakarta commons logging~~)、Slf4j (Simple logging Facade for java) 、 Jboss-Logging

日志实现：

- Log4j、JUL (java.util.logging) 、Log4j2、logback

所以通常的做法是：门面选一个，实现选一个

SpringBoot：底层是spring框架，Spring框架默认是JCL。同时SpringBoot选用SLF4j和Logback

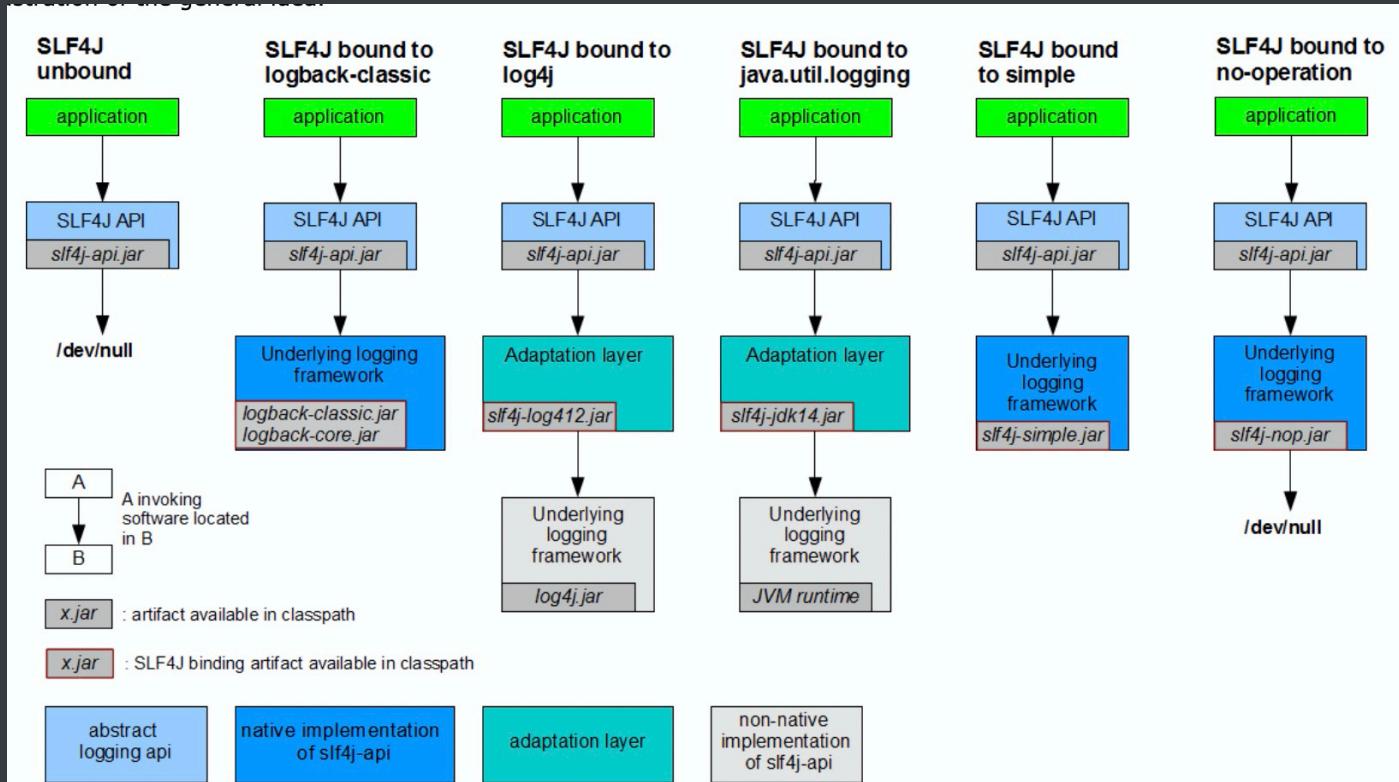
5.2 SLF4J的使用

如何在系统使用slf4j？

以后开发的时候，日志记录方法的调用，不应该调用实现类，而是应该调用抽象层里面的方法

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld {
    public static void main(String[] args) {
        Logger logger = LoggerFactory.getLogger(HelloWorld.class);
        logger.info("Hello World");
    }
}
```

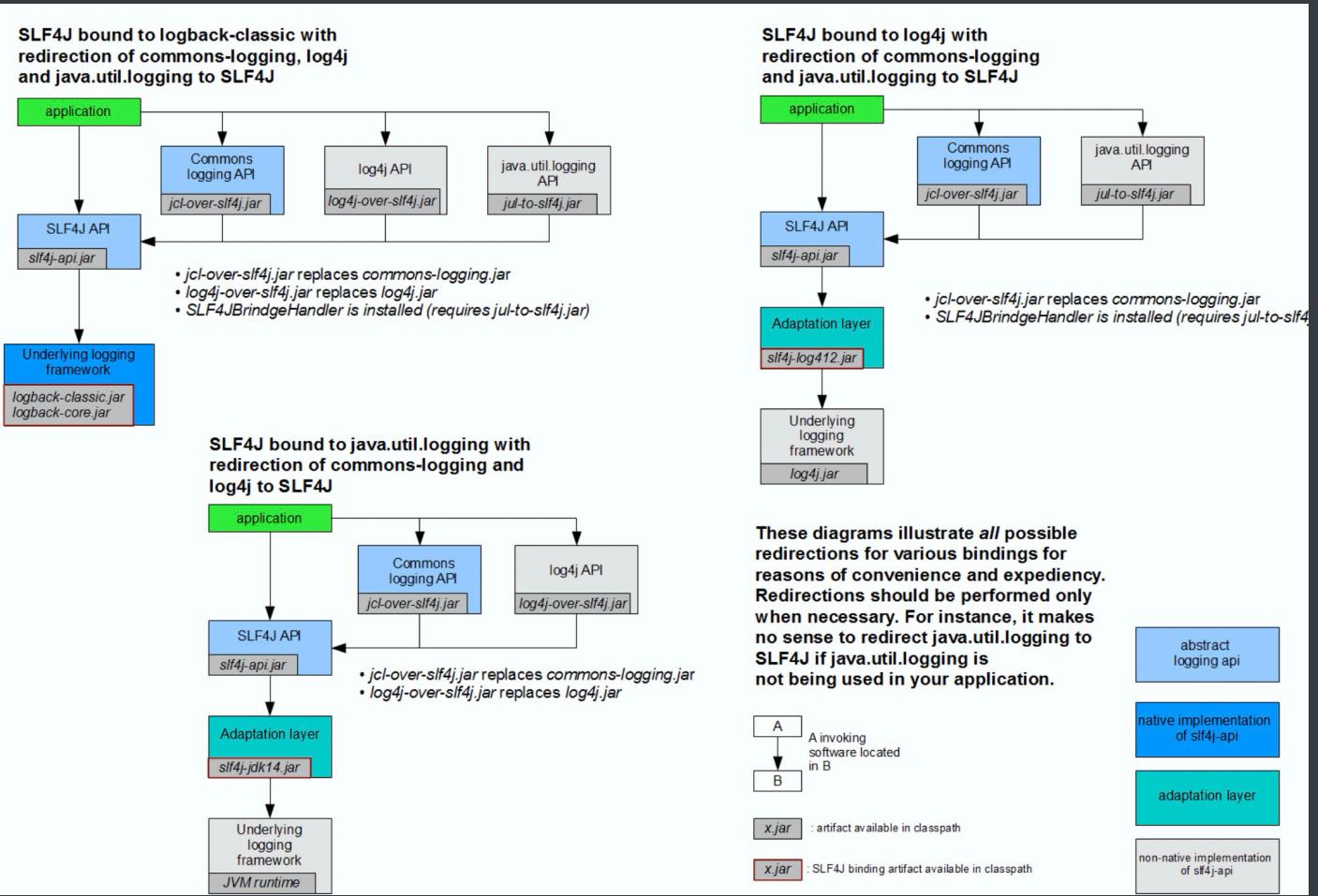


每一个日志的实现框架，都有自己的配置文件，使用slf4以后，配置文件还是用实现日志的配置

5.3 统一日志记录

A (Slf4j+logback) +Spring (commons-logging) +Hiberate (jboss-logging) +Mybatis

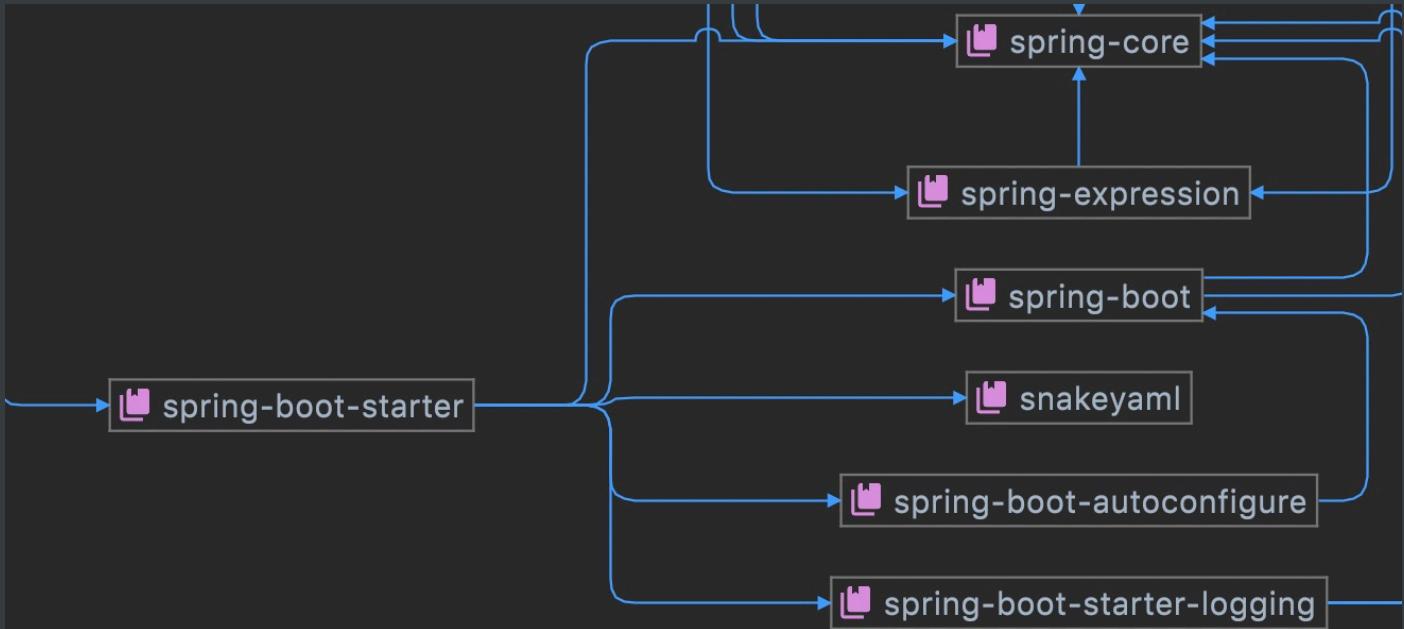
统一日志记录，别的框架统一使用Slf4j进行输出



- 将系统中其他日志框架排除出去
- 用中间包来替换原有的日志框架
- 然后导入slf4j其他的实现

5.4 SpringBoot日志关系

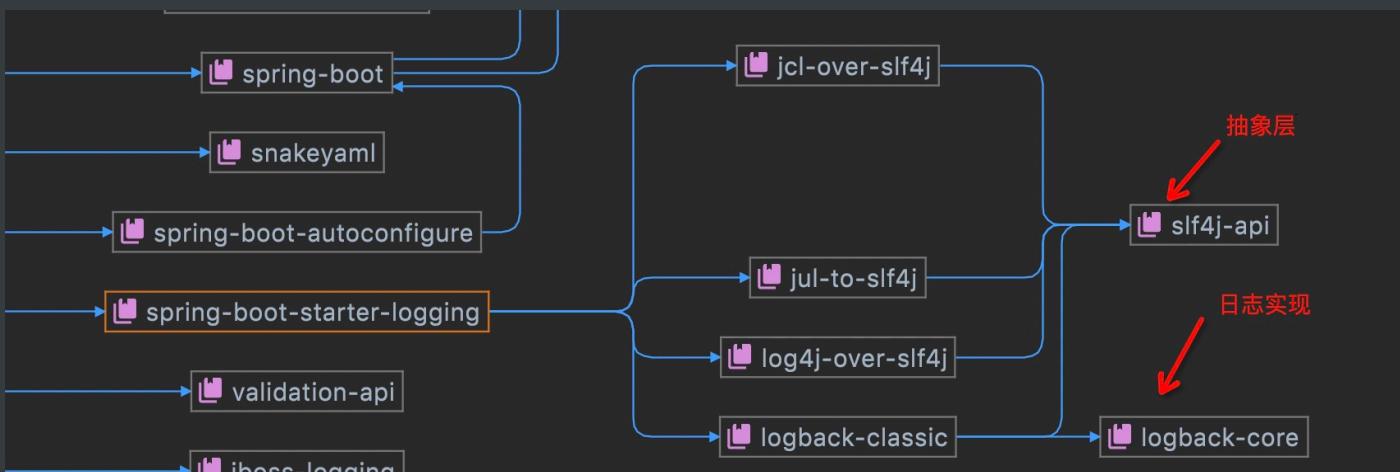
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
```



如上：SpringBoot使用spring-boot-starter-logging来做日志框架

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
</dependency>
  
```



底层依赖关系——总结：

- SpringBoot底层也是使用logback+slf4j实现日志
- SpringBoot也把其他的日志都替换成SLF4J
- 中间替换包是什么东西呢？

```

36 /rawtypes/
37 public abstract class LogFactory {
38
39     static String UNSUPPORTED_OPERATION_IN_JCL_OVER_SLF4J = "http://www..";
40
41     static LogFactory logFactory = new SLF4JLogFactory();
42

```

The name (priority) of the key in the config file used to specify the priority of that particular config file. The associated value is a floating-point number; higher values take priority over lower values.

- 如果我们引入了其他框架，那么一定需要把默认的日志框架排除掉

- Spring框架使用的是commons-logging

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <exclusions>
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>

```

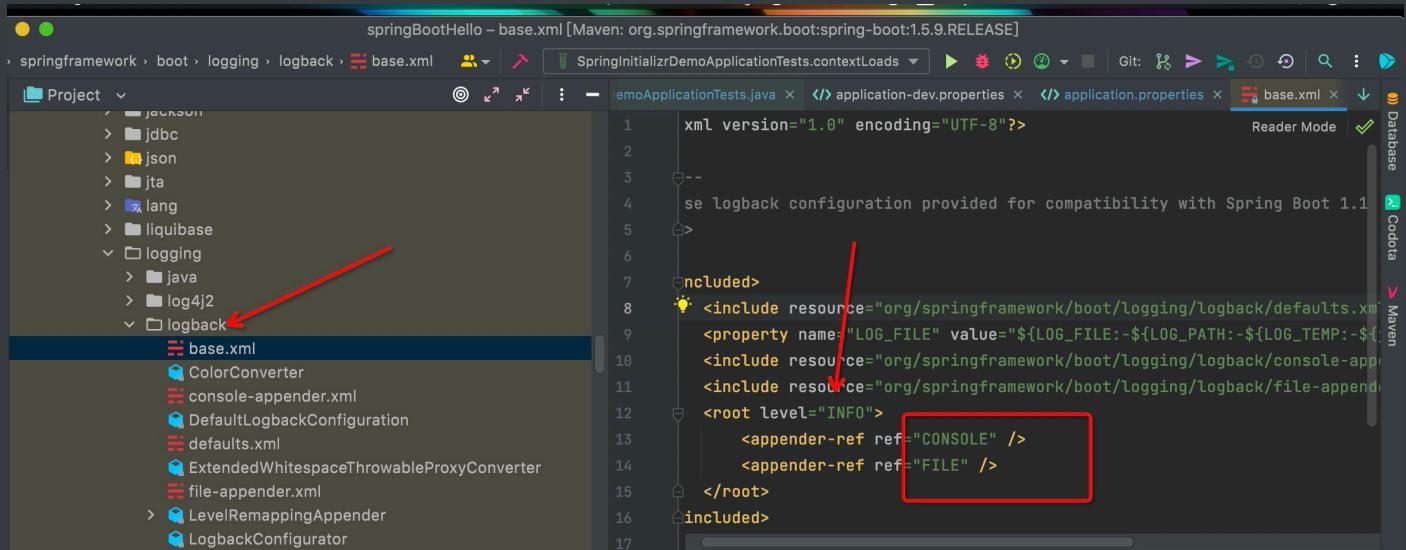
SpringBoot能自动适配所有的日志，而且底层使用的是slf4j+logback。唯一需要的就是引入其他框架的时候，排除掉默认的日志框架

5.5 日志使用

默认配置：SpringBoot默认已经配置好了日志

- %d表示日期时间
- %thread表示线程名
- %-5level级别从左显示5个字符的宽度
- %logger{50}表示logger名字最长50个字符，否则按照句点分割
- %msg表示日志消息
- %n是换行符

```
%d{yyyy-mm-dd HH:mm:ss.SSS} {%-5level %logger{50} -%msg%n
```



```
xml version="1.0" encoding="UTF-8"?>
<!--
    see logback configuration provided for compatibility with Spring Boot 1.1
-->
<!--
    included
-->
<include resource="org/springframework/boot/logging/logback/defaults.xml" />
<property name="LOG_FILE" value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-${java.io.tmpdir}}}/${PROJECT_NAME}.log}" />
<include resource="org/springframework/boot/logging/logback/console-appender.xml" />
<include resource="org/springframework/boot/logging/logback/file-appender.xml" />
<root level="INFO">
    <appender-ref ref="CONSOLE" />
    <appender-ref ref="FILE" />
</root>
<!--
    included
-->
```

其他

1. 查看ssh公钥方法:`cd ~/.ssh cat id_rsa.pub`

- 2.