

spring-boot

1. SpringBoot 入门

- 1.1. SpingBoot简介
- 1.2. 微服务
- 1.3. 环境准备
- 1.4. Spring Boot HelloWorld
 - 1.4.1. 创建一个Maven工程
 - 1.4.2. 导入Spring Boot相关的依赖
 - 1.4.3. 编写一个主程序；启动Spring Boot应用
 - 1.4.4. 编写相关的Controler、Service
 - 1.4.5. 运行主程序测试
 - 1.4.6. 简化部署
- 1.5. Hello World探究
 - 1.5.1. POM文件
 - 1.5.1.1. 父项目
 - 1.5.1.2. 启动器
 - 1.5.2. 入口类和@SpringBootApplication
 - 1.5.3. 自动配置
- 1.6. 使用Spring Initialzer快速创建Spring Boot项目
 - 1.6.1. STS使用Spring Starter Project快速创建项目

2. Spring Boot 配置

- 2.1. 配置文件
- 2.2. YAML语法
 - 2.2.1. 基本语法
 - 2.2.2. 值的写法
 - 2.2.2.1. 字面量
 - 2.2.2.2. 对象、Map
 - 2.2.2.3. 数组、List、Set
- 2.3. 配置文件注入
 - 2.3.1. @ConfigurationProperties获取值
 - 2.3.2. @Value获取值
 - 2.3.3. 获取值比较
 - 2.3.3.1. 获取值比较
 - 2.3.3.2. 获取值使用场景
 - 2.3.4. 加载指定配置文件
 - 2.3.4.1. @PropertySource
 - 2.3.4.2. @ImportResource
 - 2.3.5. SpringBoot推荐方式
- 2.4. 配置文件占位符
 - 2.4.1. 随机数
 - 2.4.2. 占位符
- 2.5. 多环境Profile
- 2.6. 配置文件加载位置

- 2.7. 外部配置加载顺序
- 2.8. 自动配置原理
 - 2.8.1. 配置文件属性参照
 - 2.8.2. 自动配置原理
- 3. Spring Boot 日志
 - 3.1. 日志框架
 - 3.2. SLF4j使用
 - 3.3. 遗留问题
 - 3.4. SpringBoot 日志关系
 - 3.5. SpringBoot 日志默认配置
 - 3.6. SpringBoot 日志指定配置
- 4. SpringBoot Web开发
 - 4.1. 使用SpringBoot
 - 4.2. 自动配置原理
 - 4.3. 静态资源映射规则
 - 4.4. Thymeleaf 模版引擎
 - 4.4.1. 整合Thymeleaf
 - 4.4.2. Thymeleaf使用&语法
 - 4.4.2.1. Thymeleaf使用
 - 4.4.2.2. Thymeleaf语法
 - 4.4.3. SpringMVC自动配置
 - 4.4.3.1. SpringMVC自动配置
 - 4.4.3.2. 扩展SpringMVC
 - 4.4.4. 如何修改默认配置
 - 4.4.5. RestfulCRUD
 - 4.4.5.1. 配置首页
 - 4.4.5.2. 引入静态资源
 - 4.4.5.3. 国际化
 - 4.4.5.4. 登录
 - 4.4.5.5. CRUD-员工列表
 - 4.4.5.5.1. 要求
 - 4.4.5.5.1. 实现
 - 4.4.6. SpringBoot错误处理
 - 4.4.6.1. SpringBoot默认的错误处理
 - 4.4.7. 配置嵌入式Servlet容器
 - 4.4.7.1. 定制和修改Servlet容器的相关配置
 - 4.4.7.2. 注册Servlet三大组件 (Servlet、Filter、Listener)
 - 4.4.7.3. 替换为其他嵌入式Servlet容器
 - 4.4.7.4. 嵌入式Servlet容器自动配置原理
 - 4.4.7.5. 嵌入式Servlet容器启动原理
 - 4.4.8. 配置外部servlet容器
 - 5. Docker
 - 6. SpringBoot与数据访问
 - 7. 自定义starter
 - 8. 更多SpringBoot整合示例

spring-boot

1. SpringBoot 入门

1.1. SpringBoot简介

- 简化Spring应用开发的一个框架；
- 整个Spring技术站的一个大集合；
- SpringBoot是J2EE的一站式解决方案；
- SpringCloud是分布式整体解决方法。
- 优点
 - 快速创建独立运行的Spring项目以及与主流框架集成
 - 使用嵌入式的Servlet容器，应用无需打成WAR包
 - starters自动依赖与版本控制
 - 大量的自动配置，简化开发，也可以修改默认值
 - 无需配置XML，无代码生成，开箱即用
 - 准生产环境的运行时应用监控
 - 与云计算的天然集成

1.2. 微服务

- 微服务；架构风格（服务微化）
 - 一个应用应该是一组小型服务；可以通过HTTP的方式进行互通。
 - 每一个功能元素最终都是一个可以独立替换和独立升级的软件单元。
- 单体应用
 - ALL IN ONE

1.3. 环境准备

```
1 jdk:java version "1.8.0_40"
2 maven:Apache Maven 3.5.0
3 ide:Spring Tool Suite 3.7.3.RELEASE
4 sprign-boot:1.5.16.RELEASE
```

1.4. Spring Boot HelloWorld

1 | 一个功能：浏览器发送hello请求，服务器接收请求并处理，响应Hello world字符串；

1.4.1. 创建一个Maven工程

1.4.2. 导入Spring Boot相关的依赖

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4     http://maven.apache.org/xsd/maven-4.0.0.xsd">
5
6   <modelVersion>4.0.0</modelVersion>
7   <groupId>cn.colg</groupId>
8   <artifactId>spring-boot-01-helloworld</artifactId>
9   <version>0.0.1-SNAPSHOT</version>
10  <name>spring-boot-01-helloworld :: spring-boot 初识</name>
11
12  <parent>
13    <groupId>org.springframework.boot</groupId>
14    <artifactId>spring-boot-starter-parent</artifactId>
15    <version>1.5.16.RELEASE</version>
16  </parent>
17
18  <dependencies>
19    <dependency>
20      <groupId>org.springframework.boot</groupId>
21      <artifactId>spring-boot-starter-web</artifactId>
22    </dependency>
23  </dependencies>
24
25</project>
```

1.4.3. 编写一个主程序；启动Spring Boot应用

```
1 package cn.colg;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 /**
7  * Spring Boot 启动类
8  *
9  * <pre>
10 * '@SpringBootApplication' : 标注一个主程序类，说明这是一个Spring Boot的主
配置类，SpringBoot就应该运行这个类的main方法来启动SpringBoot应用
11</pre>
```

```
11 * </pre>
12 *
13 * @author colg
14 */
15 @SpringBootApplication
16 public class SpringBoot01HelloWorldApplication {
17
18     public static void main(String[] args) {
19         SpringApplication.run(SpringBoot01HelloWorldApplication.class,
20         args);
21     }
21 }
```

1.4.4. 编写相关的Controller、Service

```
1 package cn.colg.controller;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 /**
7  * HelloController
8  *
9  * @author colg
10 */
11 @RestController
12 public class HelloController {
13
14     @GetMapping("/hello")
15     public String hello() {
16         return "hello world";
17     }
18 }
```

1.4.5. 运行主程序测试

1.4.6. 简化部署

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

将这个应用打成jar包，直接使用 `java -jar spring-boot-01-helloworld-0.0.1-SNAPSHOT.jar` 命令执行

1.5. Hello World探究

1.5.1. POM文件

1.5.1.1. 父项目

```
1 <!--
2   它的父项目是：
3   <parent>
4     <groupId>org.springframework.boot</groupId>
5     <artifactId>spring-boot-dependencies</artifactId>
6     <version>1.5.13.RELEASE</version>
7     <relativePath>../../spring-boot-
dependencies</relativePath>
8   </parent>
9   Spring Boot应用里面的所有依赖版本；
10 以后导入依赖默认是不需要写版本；（没有在dependencies里面管理的依赖自然需要
声明版本号）
11  -->
12  <parent>
13    <groupId>org.springframework.boot</groupId>
14    <artifactId>spring-boot-starter-parent</artifactId>
15    <version>1.5.16.RELEASE</version>
16  </parent>
```

1.5.1.2. 启动器

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4 </dependency>
```

- **spring-boot-starter-web**

```
1 | spring-boot-starter: spring-boot场景启动器；帮我们导入了web模块正常运行所依赖的组件；
```

- Spring Boot将所有功能场景都抽取出来，做成一个个的starters（启动器），只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

1.5.2. 入口类和@SpringBootApplication

- 程序从main方法开始运行
- 使用SpringApplicaiton.run()加载主程序类
- 主程序类需要标注@SpringBootApplication
- @EnableAutoConfiguration是核心注解
- @Import导入所有的自动配置场景
- @AutoConfigurationPackage定义默认的包扫描规则
- 程序启动扫描加载主程序类所在的包以及下面所有子包的组件。

1.5.3. 自动配置

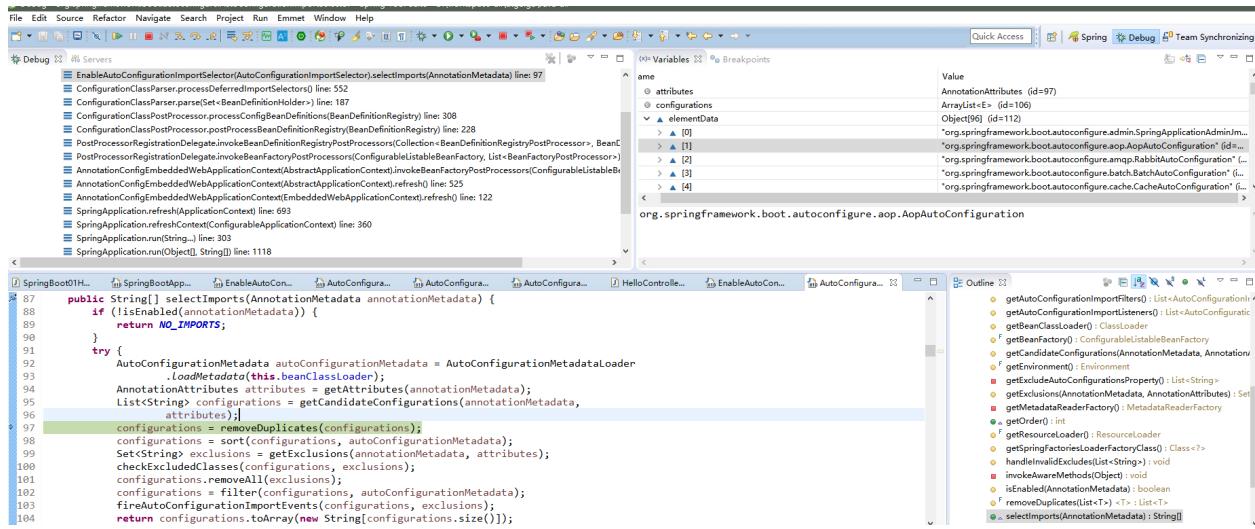
```
1 | /**
2 |  * Spring Boot 启动类
3 |  *
4 |  * <pre>
5 |  * '@SpringBootApplication'： 标注一个主程序类，说明这是一个Spring Boot的主
6 |  * 配置类，SpringBoot就应该运行这个类的main方法来启动SpringBoot应用
7 |  * </pre>
8 |  *
9 |  * @author colg
10| */
11@SpringBootApplication
12public class SpringBoot01HelloWorldApplication {
13
14    public static void main(String[] args) {
15        SpringApplication.run(SpringBoot01HelloWorldApplication.class,
16        args);
17    }
18}
```

```
1 @Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Documented
4 @Inherited
5 @SpringBootConfiguration
6 @EnableAutoConfiguration
7 @ComponentScan(excludeFilters = {
8     @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class),
9     @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
10 public @interface SpringBootApplication {
```

- **@SpringBootConfiguration** : SpringBoot的配置类
 - 标注在某个类上，表示这是一个SpringBoot的配置类；
 - **Configuration**: 配置类上来标注这个注解
 - 配置类 --- 配置文件
- **@EnableAutoConfiguration** : 开启自动配置功能
 - 以前需要配置的东西，SpringBoot帮我们自动配置；

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

- **@AutoConfigurationPackage** : 自动配置包
 - **@Import(AutoConfigurationPackages.Registrar.class)** : Spring的底层注解
@import，给容器中导入一个组件。将主配置类(@SpringBootApplication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。
 - **@Import(EnableAutoConfigurationImportSelector.class)** : 给容器中导入组件的选择器，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。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。



- 有了自动配置，免去了我们手动编写配置注入功能组件等工作；

- ```

■ List<String> configurations =
SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactor
yClass(), getClassLoader());

```
- ```

■ Enumeration<URL> urls = (classLoader != null ?
classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));

```

- Spring Boot在启动的时候从类路径下的 `META-INF/spring.factories` 中获取 `EnableAutoConfiguration` 指定的值 (`spring-boot-autoconfigure-1.5.16.RELEASE.jar` 中包含了所有场景的自动配置类代码)，将这些值作为自动配置类导入到容器中，自动配置类就生效，帮我们进行自动配置工作；以前我们需要自己配置的东西，自动配置类都帮我们做了。
- 这些自动配置类是Spring Boot进行自动配置的精髓

1.6. 使用Spring Initializer快速创建Spring Boot项目

1.6.1. STS使用Spring Starter Project快速创建项目

默认生成的spring boot项目：

- 主程序已经生成好了，测试基础类已经生成好了
- resources文件夹目录结构
 - static：保存所有的静态资源；js、css、images；
 - templates：保存所有的模版页面；（spring boot默认jar包使用嵌入式的tomcat，默认不支持jsp）；可以使用模版引擎（freemarker、thymeleaf）；
 - application.properties：spring boot应用的配置文件，可以修改一些默认设置。

2. Spring Boot 配置

2.1. 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；

- properties:

```
1 | server.port=8001
```

- yml: 以数据为中心，比json、xml等更适合做配置文件

```
1 | server:  
2 |   port: 8001
```

- xml:

```
1 | <server>  
2 |   <port>8001</port>  
3 | </server>
```

2.2. YAML语法

2.2.1. 基本语法

```
1 | k:(空格)v: 表示一对键值对(空格必须有);  
2 | 以空格的缩进来控制层级关系;只要在左对齐的一列数据,都是同一个层级的,属性和值也是大小写  
  敏感;
```

2.2.2. 值的写法

2.2.2.1. 字面量

普通的值：数字、字符串、布尔

```
1 | k: v
2 | 字面值直接写 ;
3 | 字符串默认不用加上单引号或者双引号 ;
4 | ""： 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达的意思
5 |     name: "Jack \n Rose" ; 输出：Jack 换行 Rose
6 |
7 | ''： 单引号；会转义特殊字符，特殊字符最终知识一个普通的字符串数据
8 |     name: 'Jack \n Rose' ; 输出：Jack \n Rose
```

2.2.2.2. 对象、Map

```
1 | k: v
2 | 在下一行写对象的属性和值的关系；注意缩进
3 | 对象还是k: v的方式
```

- 普通写法

```
1 | friends:
2 |   lastName: Jack
3 |   age: 20
```

- 行内写法：

```
1 | friends: {lastName: Jack, age: 20}
```

2.2.2.3. 数组、List、Set

- 用-值来表示数组中的一个元素

- 普通写法

```
1 | pets:
2 |   - cat
3 |   - dog
4 |   - pig
```

- 行内写法

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

2.3. 配置文件注入

2.3.1. @ConfigurationProperties获取值

- 配置文件

- 普通写法：

```
1 person:  
2   last-name: Jack  
3   age: 18  
4   boss: false  
5   birth: 2017/12/12  
6   maps:  
7     k1: v1  
8     k2: v2  
9   lists:  
10    - Jack  
11    - Rose  
12   dog:  
13     name: 小狗  
14     age: 3
```

- 行内写法：

```
1 person:  
2   last-name: Jack  
3   age: 18  
4   boss: false  
5   birth: 2017/12/12  
6   maps: {k1: v1, k2: v2}  
7   lists: [Jack, Rose]  
8   dog: {name: 小狗, age: 3}
```

javaBean：

```
1 /**
2  * Person 实体 `@ConfigurationProperties` 获取值
3  *
4  * <pre>
5  * 将配置文件中配置的每一个属性的值，映射到这个组件中
6  * `@ConfigurationProperties`：告诉SpringBoot将本类中的所有属性和配置文件中相
7  * 关的配置进行绑定，默认从全局配置文件中获取值
8  *       `(prefix = "person")`：配置文件中哪个与下面的所有属性进行一一映射
9  *
10 * 只有这个组件是容器中的组件，才能使用容器提供的`@ConfigurationProperties`组件
11 * </pre>
12 *
13 * @author colg
```

```

13  /*
14  @ConfigurationProperties(prefix = "person")
15  @Component
16  @Data
17  public class Person implements Serializable {
18
19      private static final long serialVersionUID = 1L;
20
21      private String lastName;
22      private Integer age;
23      private Boolean boss;
24      private Date birth;
25
26      private Map<String, Object> maps;
27      private List<Object> lists;
28
29      private Dog dog;
30  }

```

- spring-boot-configuration-processor依赖：

```

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

```

```

1 person : Person(lastName=Jack, age=18, boss=false, birth=Tue Dec 12
00:00:00 CST 2017, maps={k1=v1, k2=v2}, lists=[Jack, Rose], dog=Dog(name=
小狗, age=3))

```

2.3.2. @Value获取值

```

1 cat:
2     lastName: black
3     age: 3
4     birth: 2018/03/15

```

```

1 /**
2  * Cat 实体 '@value' 获取值
3  *
4  * @author colg
5 */

```

```

6  @Component
7  @Data
8  public class Cat implements Serializable {
9
10  /*
11   * colg [xml配置]
12   * <bean id="cat" class="cn.colg.bean.Cat">
13   *     <property name="lastName" value="black" />
14   *     <property name="age" value="3" />
15   *     <property name="birth" value="2018/03/15" />
16   *   </bean>
17   */
18
19  private static final long serialVersionUID = 1L;
20
21  @Value("${cat.lastName}")
22  private String lastName;
23  @Value("${cat.age}")
24  private Integer age;
25  @Value("${cat.birth}")
26  private Date birth;
27 }

```

```
1 | cat : Cat(lastName=black, age=3, birth=Thu Mar 15 00:00:00 CST 2018)
```

2.3.3. 获取值比较

2.3.3.1. 获取值比较

	@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
复杂数据类型	支持	不支持

2.3.3.2. 获取值使用场景

配置文件yml还是properties他们都能获取到值；

- 如果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`

```

1  /**
2  * HelloController
3  *
4  * @author colg
5  */
6 @RestController
7 public class HelloController {
8
9     @Value("${person.last-name}")
10    private String name;
11
12    @GetMapping("/hello")
13    public String hello() {
14        return "hello " + name;
15    }
16
17    @GetMapping("/hello2")
18    public String hello2(@Value("${person.last-name}") String name2)
19    {
20        return "hello " + name2;
21    }

```

- 如果专门编写了一个javaBean来和配置文件进行映射，使用 `@ConfigurationProperties`

2.3.4. 加载指定配置文件

2.3.4.1. `@PropertySource`

```

1 color.red: 红色
2 color.blue: 蓝色
3 color.createDate: 2018/01/01
4 color.maps.k1: v1
5 color.maps.k2: v2
6 color.lists: Jack, Rose

```

```

1 /**
2  * Color 实体 '@PropertySource' 加载指定的配置文件
3  *
4  * @author colg
5  */
6 @ConfigurationProperties(prefix = "color")
7 @PropertySource(value = {"classpath:color.properties"})
8 @Component
9 @Data

```

```
10 public class Color implements Serializable {  
11  
12     private static final long serialVersionUID = 1L;  
13  
14     private String red;  
15     private String blue;  
16     private Date createDate;  
17  
18     private Map<String, Object> maps;  
19     private List<String> lists;  
20 }
```

```
1 color : Color(red=红色, blue=蓝色, createDate=Mon Jan 01 00:00:00 CST  
2018, maps={k2=v2, k1=v1}, lists=[Jack, Rose])
```

2.3.4.2. @ImportResource

```
1 <?xml version="1.0" encoding="UTF-8"?>  
2 <beans xmlns="http://www.springframework.org/schema/beans"  
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
4      xsi:schemaLocation="http://www.springframework.org/schema/beans  
5          http://www.springframework.org/schema/beans/spring-beans.xsd">  
6  
6     <bean id="phone" class="cn.colg.bean.Phone">  
7         <property name="name" value="oppo" />  
8         <property name="color" value="green" />  
9         <property name="age" value="2" />  
10        <property name="maps">  
11            <map>  
12                <entry key="k1" value="v1" />  
13                <entry key="k2" value="v2" />  
14            </map>  
15        </property>  
16        <property name="lists">  
17            <list>  
18                <value>Jack</value>  
19                <value>Rose</value>  
20            </list>  
21        </property>  
22    </bean>  
23  
24 </beans>
```

```
1 /**  
2  * Phone 实体 '@ImportResource' 导入spring的配置文件
```

```
3  *
4  * @author colg
5  */
6 @ImportResource(locations = {"classpath:beans.xml"})
7 @Component
8 @Data
9 public class Phone implements Serializable {
10
11     private static final long serialVersionUID = 1L;
12
13     private String name;
14     private String color;
15     private Integer age;
16
17     private Map<String, Object> maps;
18     private List<String> lists;
19 }
```

```
1 phone : Phone(name=oppo, color=green, age=2, maps={k1=v1, k2=v2}, lists=[Jack, Rose])
```

2.3.5. SpringBoot推荐方式

```
1 /**
2  * 配置类；'@Configuration'： 指定当前类是一个配置类；就是来替代之前的Spring配置文件
3  *
4  * @author colg
5  */
6 @Configuration
7 public class AppConfig {
8
9     /**
10      * 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
11      *
12      * @return
13      */
14     @Bean
15     public Person person() {
16         return new Person();
17     }
18
19     @Bean
20     public Cat cat() {
21         return new Cat();
```

```
22     }
23
24     @Bean
25     public Color color() {
26         return new Color();
27     }
28
29     @Bean
30     public Phone phone() {
31         return new Phone();
32     }
33 }
```

```
1 person : Person(lastName=Jack, age=18, boss=false, birth=Tue Dec 12
00:00:00 CST 2017, maps={k1=v1, k2=v2}, lists=[Jack, Rose], dog=Dog(name=
小狗, age=3))
2 cat : Cat(lastName=black, age=3, birth=Thu Mar 15 00:00:00 CST 2018)
3 color : Color(red=红色, blue=蓝色, createDate=Mon Jan 01 00:00:00 CST
2018, maps={k2=v2, k1=v1}, lists=[Jack, Rose])
4 phone : Phone(name=oppo, color=green, age=2, maps={k1=v1, k2=v2}, lists=
[Jack, Rose])
```

2.4. 配置文件占位符

2.4.1. 随机数

```
1 ${random.value}、 ${random.int}、 ${random.long}
2 ${random.int(10)}、 ${random.int[1024,65536]}
```

2.4.2. 占位符

- 可以在配置文件中引用前面配置过的属性（优先级前面配置过的这里都能用）
- \${app.name:默认值} 来指定找不到属性时的默认值

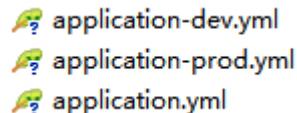
```
1 taxi:
2   name: 丰田卡罗拉${random.uuid}
3   plate-number: 鄂FPN18${random.int[10]}
4   age: ${random.int[1,3]}
5   dog:
6     name: ${taxi.hello:小狗}
7     age: ${taxi.age}
```

```
1 | taxi : Taxi(name=丰田卡罗拉f2454df0-b702-44e5-a63b-b31f9662aa8d,  
| plateNumber=鄂FPN181, age=1, dog=Dog(name=小狗, age=1))
```

2.5. 多环境Profile

Profile是Spring对不同环境提 指定参数等方式快速切换环境

- 多Profile文件
 - 格式：application-{profile}.properties/yml



application-dev.yml
application-prod.yml
application.yml

- 多Profile文档块模式

```
1 | spring:  
2 |   profiles:  
3 |     active:  
4 |       - dev          # 激活dev环境  
5 |  
6 | # 三个短横线分隔多个profile区 ( 文档块 )  
7 | ---  
8 | server:  
9 |   port: 8002  
10 | spring:  
11 |   profiles: dev  
12 | ---  
13 | ---  
14 | server:  
15 |   port: 8003  
16 | spring:  
17 |   profiles: prod  
18 | ---  
19 | # profiles: default 表示未指定默认配置  
20 | ---  
21 | server:  
22 |   port: 8001  
23 | spring:  
24 |   profiles: default
```

- 激活方式
 - 配置文件

```
1 | spring:  
2 |   profiles:  
3 |     active:  
4 |       - dev # 激活dev环境
```

- 命令行

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

- jvm参数

```
1 | -Dspring.profiles.active=dev
```

2.6. 配置文件加载位置

- Spring Boot 启动会扫描以下位置的application.yml/properties作为Spring Boot的默认配置文件
 - file:/config/
 - file:./
 - classpath:/config/
 - classpath:/
- 优先级由高到低，高优先级的配置会覆盖低优先级的配置；
- Spring Boot会从这4个位置全部加载主配置文件；互补配置；
- **spring.config.location:** F:/application.yml 手动指定配置文件位置。

2.7. 外部配置加载顺序

- Spring Boot 支持多种外部配置方式，这些方式优先级如下：

1. 命令行参数
2. 来自java:comp/env的JNDI属性
3. Java系统属性 (System.getProperties())
4. 操作系统环境变量
5. RandomValuePropertySource配置的random.*属性值
6. **jar包外部的application-{profile}.properties或application.yml(带spring.profile)配置文件**
7. **jar包内部的application-{profile}.properties或application.yml(带spring.profile)配置文件**
8. **jar包外部的application.properties或application.yml(不带spring.profile)配置文件**
9. **jar包内部的application.properties或application.yml(不带spring.profile)配置文件**
10. @Configuration注解类上的@PropertySource

11. 通过SpringApplication.setDefaultProperties指定的默认属性

- 所有支持的配置加载来源
 - 官方文档，外部配置：<https://docs.spring.io/spring-boot/docs/1.5.16.RELEASE/reference/htmlsingle/#boot-features-external-config>

2.8. 自动配置原理

2.8.1. 配置文件属性参照

<https://docs.spring.io/spring-boot/docs/1.5.16.RELEASE/reference/htmlsingle/#common-application-properties>

2.8.2. 自动配置原理

- SpringBoot启动的时候记在主配置类，开启了自动配置功能 `@EnableAutoConfiguration`
 - `@EnableAutoConfiguration` 作用：
 - `@Import(EnableAutoConfigurationImportSelector.class)`：利用 `EnableAutoConfigurationImportSelector` 给容器中导入一些组件
 - 可以查看 `AutoConfigurationImportSelector.selectImports(AnnotationMetadata)` 方法中的内容；
 - `List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);` 获取候选的位置

```
1 List<String> configurations =
2 SpringFactoriesLoader.loadFactoryNames(SpringFactoriesLoade
rFactoryClass(), getClassLoader());
3 // 扫描所有jar包类路径下 "META-INF/spring.factories"
4
5 Properties properties =
6 PropertiesLoaderUtils.loadProperties(new UrlResource(url));
7 // 把扫描到的这些文件的内容包装成properties对象
8 // 从properties中获取到EnableAutoConfiguration.class类（类名）对应
9 // 的值，然后把他们添加到容器中
```

- 将类路径下 `META-INF/spring.factories` 里面配置的所有 `EnableAutoConfiguration` 的值加入到了容器中；
- 每一个自动配置类进行自动配置功能

- 以 `org.springframework.boot.autoconfigure.web.HttpEncodingAutoConfiguration` 为例解释自动配置原理：

```

1  @Configuration // 表示这是一个配置类，类似于编写的配置文件，也可以给容器中添加
2  组件
3  @EnableConfigurationProperties(HttpEncodingProperties.class) // 启用
4  指定类的ConfigurationProperties功能，将配置文件中对应的值和
5  HttpEncodingProperties进行绑定；并把HttpEncodingProperties加入到ioc容器中
6
7  @ConditionalOnWebApplication // Spring底层@Conditional注解，根据不同的条
8  件，如果满足指定的条件，整个配置类里面的配置就会生效； 判断当前应用是否是web应用
9  ，如果是，当前配置类生效
10
11 @ConditionalOnClass(CharacterEncodingFilter.class) // 判断当前项目有没有
12  指定类，CharacterEncodingFilter：SpringMVC中进行乱码解决的过滤器
13
14 @ConditionalOnProperty(prefix = "spring.http.encoding", value =
15  "enabled", matchIfMissing = true) // 判断配置文件中是否存在某个配置
16  spring.http.encoding，matchIfMissing = true：如果不存在没，判断也是成立的
17  // 即使配置文件中不配置 spring.http.encoding.enabled = true，也是默认生效的
18  public class HttpEncodingAutoConfiguration {
19
20      // 已经和SpringBoot的配置文件映射了
21      private final HttpEncodingProperties properties;
22
23      // 只有一个有参构造器的情况下，参数的值就会从容器中拿
24      public HttpEncodingAutoConfiguration(HttpEncodingProperties
25          properties) {
26          this.properties = properties;
27      }
28
29      @Bean // 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
30      @ConditionalOnMissingBean(CharacterEncodingFilter.class)
31      public CharacterEncodingFilter characterEncodingFilter() {
32          CharacterEncodingFilter filter = new
33          OrderedCharacterEncodingFilter();
34          filter.setEncoding(this.properties.getCharset().name());
35
36          filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQU
37          EST));
38
39          filter.setForceResponseEncoding(this.properties.shouldForce(Type.RES
40          PONSE));
41
42          return filter;

```

- 根据当前不同的条件判断，决定这个配置类是否生效，一旦这个配置类生效；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xxxProperties类中封装着，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

```

1 @ConfigurationProperties(prefix = "spring.http.encoding") // 从配置文件
2   中获取指定的值和bean的属性进行绑定
3
4   public class HttpEncodingProperties {
5
6     public static final Charset DEFAULT_CHARSET =
7       Charset.forName("UTF-8");

```

- 精髓：

- SpringBoot启动会加载大量的自动配置类
- 看需要的功能有没有SpringBoot默认写好的自动配置类
- 再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有，就不需要再来配置了）
- 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。我们就可以在配置文件中指定这些属性的值

- 通用模式

- xxxAutoConfiguration：自动配置类，给容器中添加组件
- xxxProperties：属性配置类，封装配置文件中相关属性

```

1 # 能配置的属性都是来源于这个功能的xxxProperties类
2 spring:
3   http:
4     encoding:
5       enabled: true
6       charset: UTF-8
7       force: true

```

- @Conditional 扩展

@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

- 自动配置类必须在一定条件下才能生效，可以通过启用 `debug: true` 属性，来让控制台打印自动配置报告，这样就可以很方便的指定哪些配置类生效

```

1 # 开启springBoot的debug，查看详细的自动配置报告
2 debug: true

```

```

1 =====
2 AUTO-CONFIGURATION REPORT // 自动配置报告
3 =====
4
5
6 Positive matches: // 自动配置类启用的
7 -----
8
9     DispatcherServletAutoConfiguration matched:
10         - @ConditionalOnClass found required class
11           'org.springframework.web.servlet.DispatcherServlet';
12           @ConditionalOnMissingClass did not find unwanted class
13             (OnClassCondition)

```

```
11      - @ConditionalOnWebApplication (required) found
12      StandardServletEnvironment (OnWebApplicationCondition)
13
14      DispatcherServletAutoConfiguration.DispatcherServletConfiguration
15      matched:
16          - @ConditionalOnClass found required class
17          'javax.servlet.ServletRegistration'; @ConditionalOnMissingClass did
18          not find unwanted class (OnClassCondition)
19              - Default DispatcherServlet did not find dispatcher servlet
20              beans
21      (DispatcherServletAutoConfiguration.DefaultDispatcherServletCondition)
22
23
24
25
26
27
28
29
30  Negative matches: // 自动配置类未启用的
31  -----
32
33      ActiveMQAutoConfiguration:
34          Did not match:
35              - @ConditionalOnClass did not find required classes
36              'javax.jms.ConnectionFactory',
37              'org.apache.activemq.ActiveMQConnectionFactory' (OnClassCondition)
38
39
40      AopAutoConfiguration:
41          Did not match:
42              - @ConditionalOnClass did not find required classes
43              'org.aspectj.lang.annotation.Aspect',
44              'org.aspectj.lang.reflect.Advice' (OnClassCondition) // 缺少 Aspect,
45              Advice类
46
47
48
49
50  Exclusions: // 排除的自动配置类
51  -----
52
53
54
55
56  Unconditional classes: // 无条件启用的自动配置类
57  -----
58
59
60      org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration
61
62
```

41

```
org.springframework.boot.autoconfigure.web.WebClientAutoConfiguration
```

3. Spring Boot 日志

3.1. 日志框架

- 案例

```
1 小张，开发一个大型系统：  
2 1、System.out.println("");；将关键数据打印在控制台；去掉？写在一个文件？  
3 2、框架来记录系统的一些运行信息；日志框架；zhanglogging.jar  
4 3、高大上的几个功能？异步模式？自动归档？xxxx？zhanglogging-good.jar  
5 4、将以前框架卸下来？换上新的框架，重新修改之前相关的API；zhanglogging-  
6 project.jar  
7 5、JDBC---数据库驱动：  
8 写了一个统一的接口层；日志门面（日志的一个抽象层）；logging-  
abstract.jar  
9 给项目中导入具体日志实现就行了；之前的日志框架都是实现的抽象层
```

- 市面上的日志框架

- JUL (java.util.logging) 、 JCL (Apache Commons Logging) 、 Log4j 、 Log4j2 、 Logback 、 SLF4j 、 jboss-logging 等

- SpringBoot使用的日志框架

- SpringBoot底层是Spring框架，Spring框架默认使用JCL，spirng-boot-starter-logging采用了slf4j+logback形式，Spring Boot也能自动适配 (jul、log4j2、logback) 并简化配置

日志门面（日志的抽象层）	日志实现
JCL (Jakarta Commons Logging) SLF4j (Simple Logging Facade for Java) jboss-logging	Log4j JUL (java.util.logging) Log4j2 Logback

- 左边选一个门面 (SLF4j) 、右边选一个实现 (Logback)

3.2. SLF4j使用

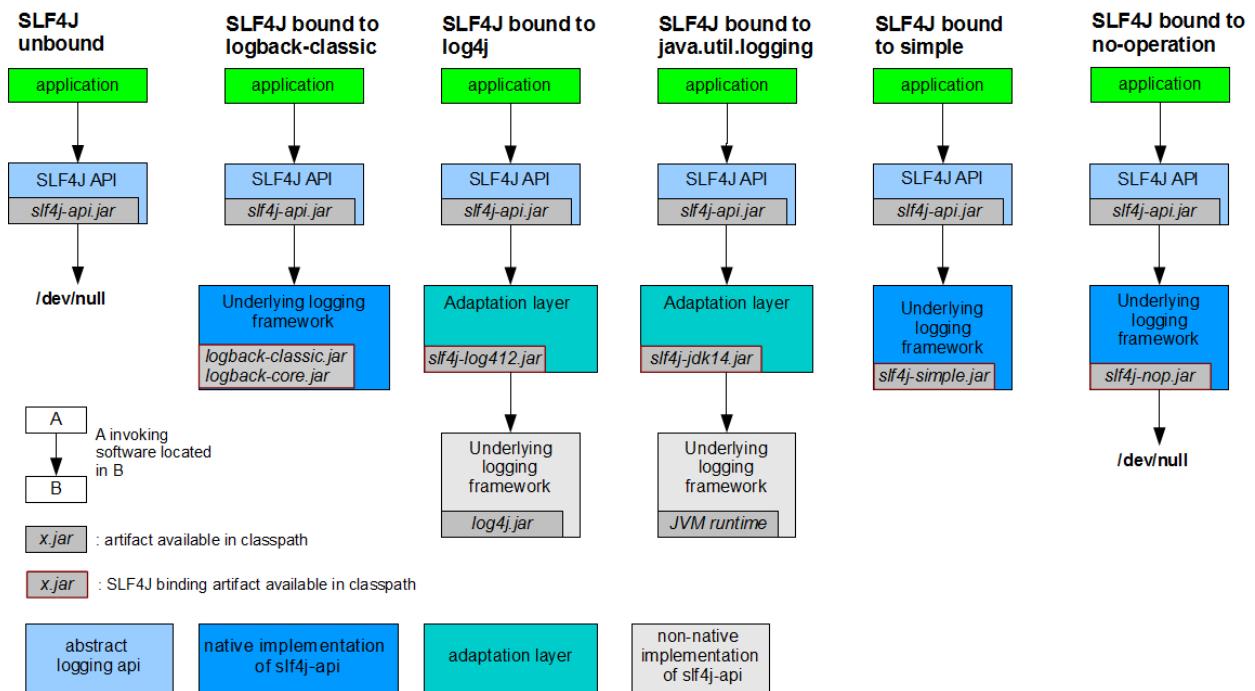
- 如何在系统中使用SLF4j

- 以后开发的时候，日志记录方法的调用，不应该来直接调用日志的实现类，而是调用日志抽象层里面的方法；给系统里导入slf4j的jar和logback的实现jar

```

1 import org.slf4j.Logger;
2 import org.slf4j.LoggerFactory;
3
4 /**
5  * 使用SLF4J
6  *
7  * @author colg
8  */
9 public class Helloworld {
10
11     public static void main(String[] args) {
12         Logger logger = LoggerFactory.getLogger(Helloworld.class);
13         logger.info("Hello World");
14     }
15
16 }
```

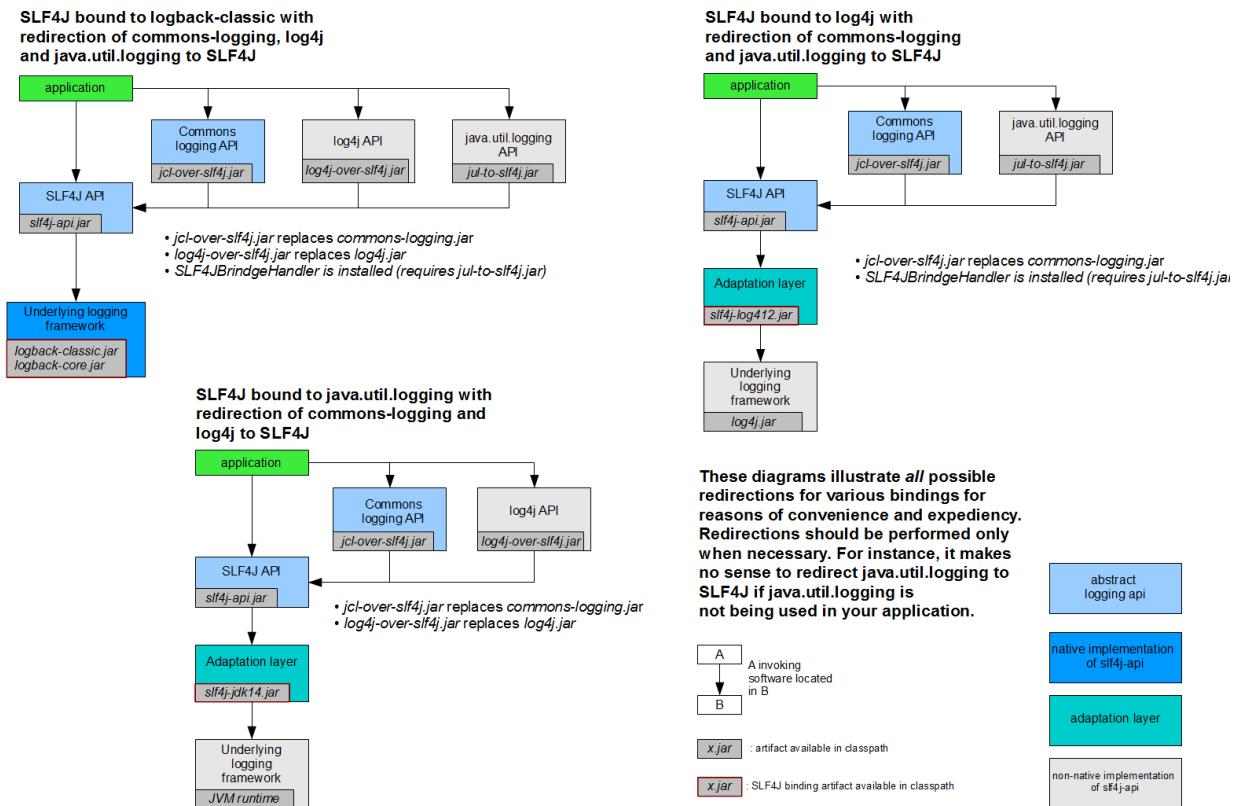
图示：



每一个日志的实现框架都有自己的配置文件。使用SLF4J以后，配置文件还是做成日志实现框架自己本身的配置文件；

3.3. 遗留问题

- a (slf4j+logback) : Spring (commons-logging) 、 Hibernate (jboss-logging) 、 Mybatis、 xxx
 - 问题：统一日志记录，即使是别的框架也要统一使用slf4j进行输出？



- 如何让系统中所有的日志都统一到slf4j：
 - 将系统中其他日志框架先排除出去；
 - 用中间包来替换原有的日志框架；
 - 导入slf4j其他的实现。

3.4. SpirngBoot 日志关系

- pom.xml

```

1   <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter</artifactId>
4   </dependency>
  
```

- SpirngBoot用它来做日志功能：

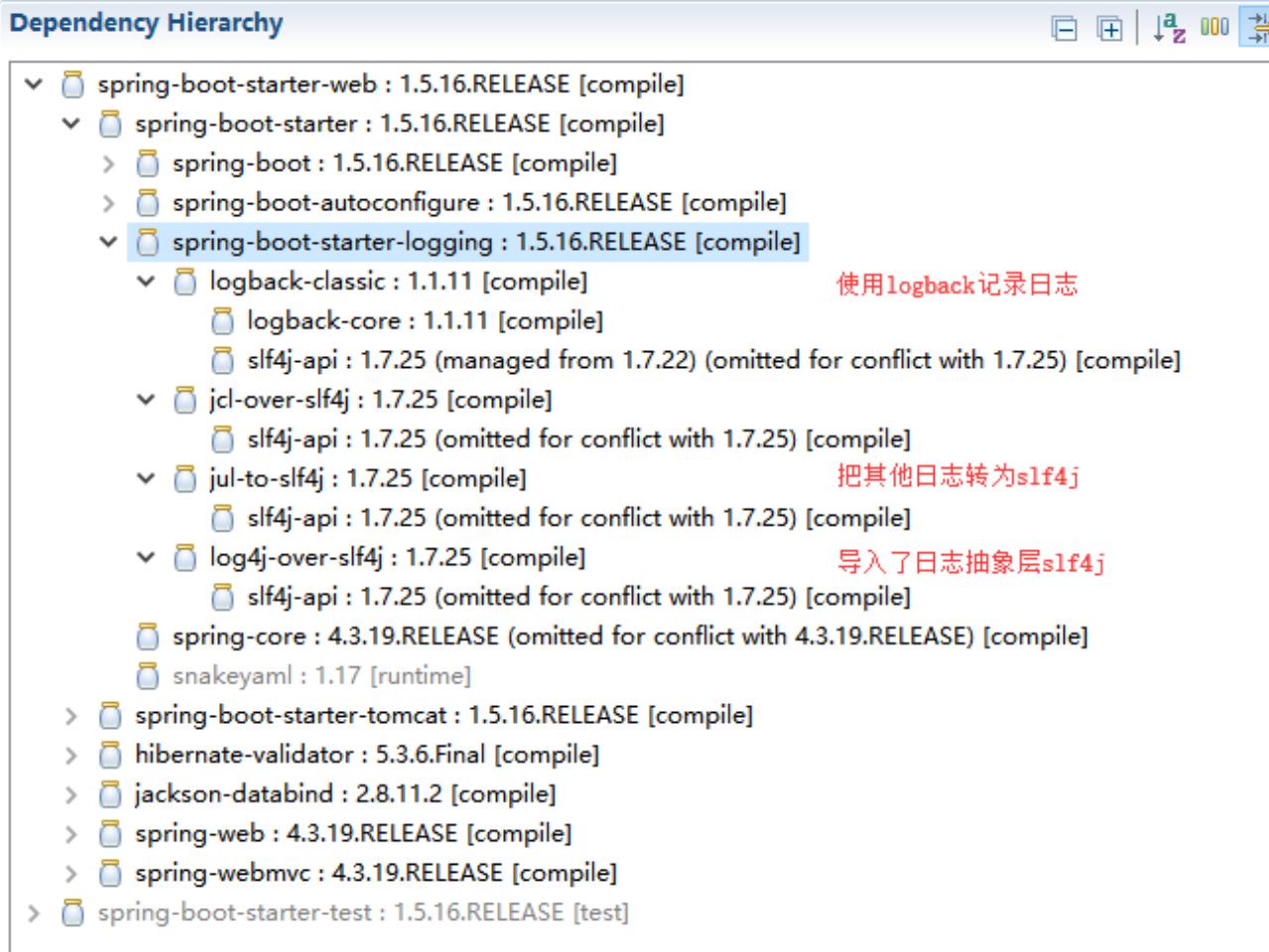
```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-logging</artifactId>
4 </dependency>

```

- 底层依赖关系：

Dependency Hierarchy [test]



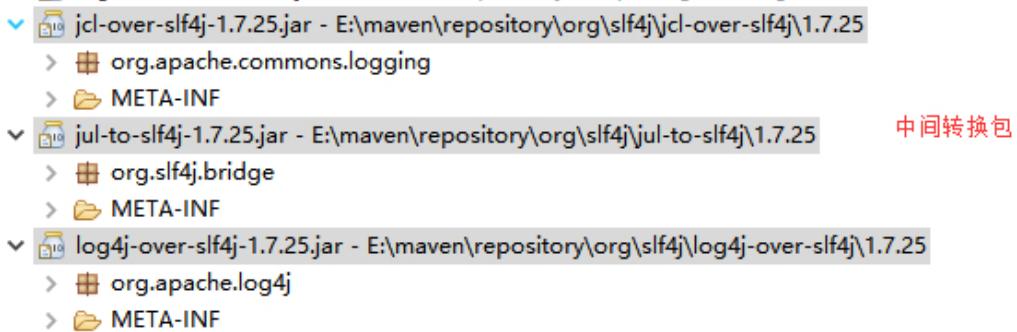
- 总结：

- SpringBoot底层也是使用slf4+logback的方式进行日志记录
- SpringBoot也把其他的日志都替换成slf4j
- 中间替换包：jcl-over-slf4j

```

1 public abstract class LogFactory {
2
3     static String UNSUPPORTED_OPERATION_IN_JCL_OVER_SLF4J =
4         "http://www.slf4j.org/codes.html#unsupported_operation_in_jcl_over_slf4j";
5     static LogFactory logFactory = new SLF4JLogFactory();

```



- 如果要引入其他框架，一定要把这个框架的默认日志依赖移除掉

```

1 <dependency>
2   <groupId>org.springframework</groupId>
3   <artifactId>spring-core</artifactId>
4   <exclusions>
5     <exclusion>
6       <groupId>commons-logging</groupId>
7       <artifactId>commons-logging</artifactId>
8     </exclusion>
9   </exclusions>
10  </dependency>

```

- Spring Boot能自动适配所有的日志，而且底层使用slf4j+logback的方式记录日志，引入其他框架的时候，只需要把这个框架依赖的日志框架排除掉。

3.5. SpringBoot 日志默认配置

```

1 /**
2 * Spring Boot 日志默认配置 测试
3 *
4 * @author colg
5 */
6 public class LoggingTest extends SpringBoot06LoggingApplicationTests {
7
8   /** 日志记录器 */
9   Logger logger = LoggerFactory.getLogger(LoggingTest.class);
10
11  @Test
12  public void loggerTest() {
13    /*
14     * colg [日志级别]
15     * 由低到高: trace < debug < info < warn < error
16     * 可以调整输出的日志级别；日志就会只在这个级别以后的高级别生效
17     */
18    logger.trace("这是trace日志...");
```

```

19     logger.debug("这是debug日志...");  

20     // Spring Boot 默认使用的是info级别的  

21     logger.info("这是info日志...");  

22     logger.warn("这是warn日志...");  

23     logger.error("这是error日志...");  

24 }

```

```

1 logging:  

2   level:  

3     cn.colg.logging: trace  

4   # 日志文件名：当前项目下生成springboot.log日志，也可以指定完整的路径  

5   # file: springboot.log  

6   file: D:/workspace-all/atguigu/Java-all/JavaEE/spring-boot/spring-  

boot-doc/springboot.log  

7  

8   # 日志文件的位置：当前磁盘的根路径下创建spring文件夹和log文件夹；使用spring.log作为  

9   # path: /spring/log  

10  path: D:/workspace-all/atguigu/Java-all/JavaEE/spring-boot/spring-  

boot-doc/spring/log  

11  

12 # logging.file和logging.path同时指定时，以logging.file为准

```

logging.file	logging.path	Example	Description
(none)	(none)	只在控制台输出	
指定文件名	(none)	my.log	输出日志到my.log文件
(none)	指定目录	/var/log	输出到指定目录的spring.log文件中

3.6. SpringBoot 日志指定配置

给类路径下放上每个日志框架自己的日志文件即可；SpringBoot就不使用默认的配置了。

- 自定义日志规则

Logging System	Customization
Logback	<code>logback-spring.xml</code> , <code>logback-spring.groovy</code> , <code>logback.xml</code> or <code>logback.groovy</code>
Log4j2	<code>log4j2-spring.xml</code> or <code>log4j2.xml</code>
JDK (Java Util Logging)	<code>logging.properties</code>

- Logback使用的日志文件
 - `logback.xml` : 直接就被日志框架识别了
 - `logback-spring.xml` : 日志框架就不直接加载日志的配置项，由SpringBoot解析日志配置，可以使用SpringBoot的高级Profile功能

```

1  <!-- 可以指定某段配置只在某个环境下生效 -->
2  <springProfile name="staging">
3      <!-- configuration to be enabled when the "staging" profile is
4          active -->
5  </springProfile>
6
6  <springProfile name="dev, staging">
7      <!-- configuration to be enabled when the "dev" or "staging"
8          profiles are active -->
9  </springProfile>
10
10 <springProfile name="!production">
11     <!-- configuration to be enabled when the "production" profile
12         is not active -->
12 </springProfile>
```

4. SpringBoot Web开发

4.1. 使用SpringBoot

- 创建SpringBoot应用，选中需要的模块
- SpringBoot已经默认将这些场景配置好了，只需要在配置文件中指定少量配置就可以运行起来
- 自己编写业务代码

4.2. 自动配置原理

- 这个场景SpringBoot帮我们配置了什么？能不能修改？能修改哪些配置？能不能扩展？xxx

```
1 | webMvcAutoConfiguration: 给容器中自动配置组件 ;
2 | WebMvcProperties: 配置类来封装配置文件的内容 ;
```

4.3. 静态资源映射规则

```
1 | @ConfigurationProperties(prefix = "spring.resources", ignoreUnknownFields
2 | = false)
3 | public class ResourceProperties implements ResourceLoaderAware,
4 | InitializingBean {
5 |     // 可以设置和资源有关的参数，缓存时间
```

```
1 |     // 配置静态资源映射
2 |     @Override
3 |     public void addResourceHandlers(ResourceHandlerRegistry
4 | registry) {
5 |         if (!this.resourceProperties.isAddMappings()) {
6 |             logger.debug("Default resource handling disabled");
7 |             return;
8 |         }
9 |         // webjars 资源映射
10 |         Integer cachePeriod =
11 |             this.resourceProperties.getCachePeriod();
12 |         if (!registry.hasMappingForPattern("/webjars/**")) {
13 |             customizeResourceHandlerRegistration(registry
14 |                 .addResourceHandler("/webjars/**")
15 |                 .addResourceLocations("classpath:/META-
16 | INF/resources/webjars/")
17 |                 .setCachePeriod(cachePeriod));
18 |         }
19 |         // 自定义资源映射
20 |         String staticPathPattern =
21 |             this.mvcProperties.getStaticPathPattern();
22 |         if (!registry.hasMappingForPattern(staticPathPattern)) {
23 |             customizeResourceHandlerRegistration(
24 |                 registry.addResourceHandler(staticPathPattern)
25 |                 .addResourceLocations(
26 |
27 |             this.resourceProperties.getStaticLocations())
28 |                 .setCachePeriod(cachePeriod));
29 |         }
30 |     }
31 |
32 |     // 配置欢迎页映射
33 |     @Bean
```

```
29     public welcomePageHandlerMapping welcomePageHandlerMapping(
30         ResourceProperties resourceProperties) {
31         return new
32             welcomePageHandlerMapping(resourceProperties.getwelcomePage(),
33                 this.mvcProperties.getstaticPathPattern());
34         }
35
36         // 配置图标
37         @Configuration
38         @ConditionalOnProperty(value = "spring.mvc.favicon.enabled",
39             matchIfMissing = true)
40         public static class FaviconConfiguration {
41
42             private final ResourceProperties resourceProperties;
43
44             public FaviconConfiguration(ResourceProperties
45             resourceProperties) {
46                 this.resourceProperties = resourceProperties;
47             }
48
49             @Bean
50             public SimpleUrlHandlerMapping faviconHandlerMapping() {
51                 SimpleUrlHandlerMapping mapping = new
52                     SimpleUrlHandlerMapping();
53                 mapping.setOrder(Ordered.HIGHEST_PRECEDENCE + 1);
54                 // 所有 **/favicon.ico
55
56                 mapping.setUrlMap(Collections.singletonMap("/**/favicon.ico",
57                     faviconRequestHandler()));
58                 return mapping;
59             }
60
61             @Bean
62             public ResourceHttpRequestHandler faviconRequestHandler() {
63                 ResourceHttpRequestHandler requestHandler = new
64                     ResourceHttpRequestHandler();
65                 requestHandler
66                     .setLocations(this.resourceProperties.getFaviconLocations());
67                 return requestHandler;
68             }
69         }
```

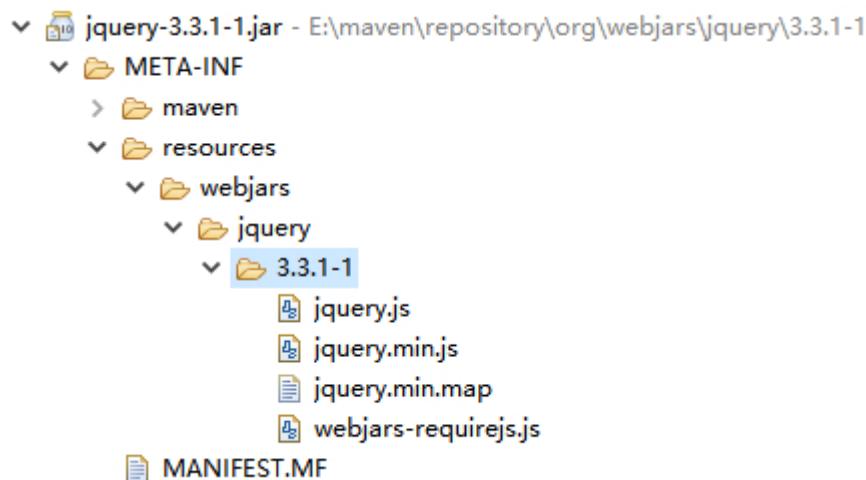
```

1  private String[] getStaticWelcomePageLocations() {
2      String[] result = new String[this.staticLocations.length];
3      for (int i = 0; i < result.length; i++) {
4          String location = this.staticLocations[i];
5          if (!location.endsWith("/")) {
6              location = location + "/";
7          }
8          result[i] = location + "index.html";
9      }
10     return result;
11 }

```

- `/webjars/**` , 都去 `classpath:/META-INF/resources/webjars/` 找资源;

- webjars : 以jar包的方式引入静态资源 <https://www.webjars.org/>



```

1  <!-- 引入jquery-webjars -->
2  <dependency>
3      <groupId>org.webjars</groupId>
4      <artifactId>jquery</artifactId>
5      <version>3.3.1-1</version>
6  </dependency>

```

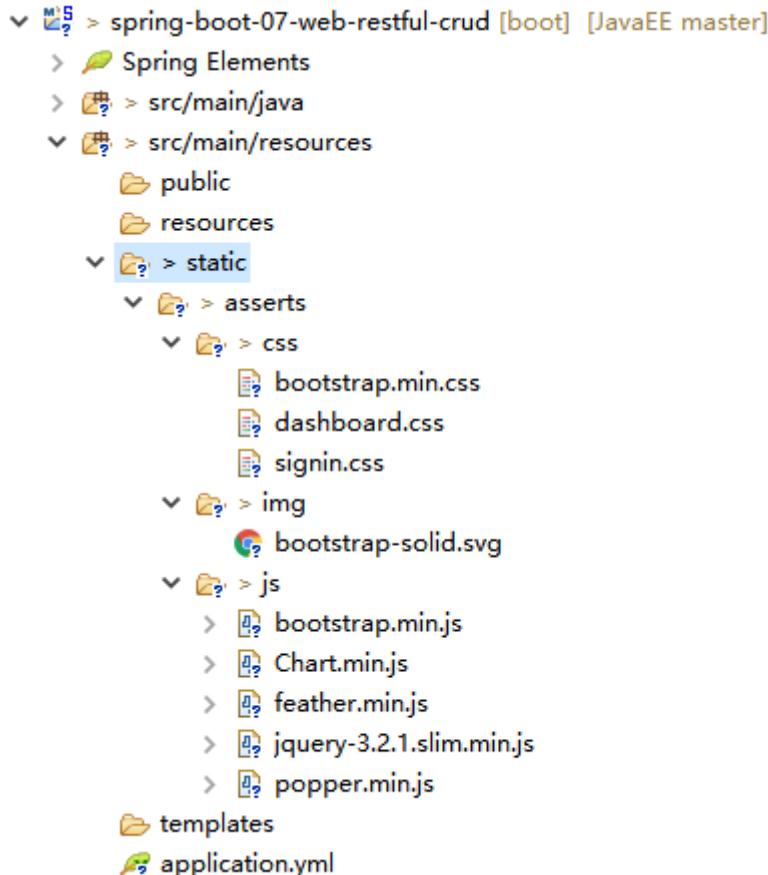
```

1 # 访问资源
2 http://localhost:8080/webjars/jquery/3.3.1-1/jquery.js

```

- `/**` , 访问当前项目的任何资源 , (静态资源的文件夹)

```
1 "classpath:/META-INF/resources/",
2 "classpath:/resources/",
3 "classpath:/static/",
4 "classpath:/public/"
5 "/"：当前项目的根路径
```



```
1 # 访问资源
2 http://localhost:8080/assets/img/bootstrap-solid.svg
3 http://localhost:8080/assets/css/bootstrap.min.css
4 http://localhost:8080/assets/js/jquery-3.2.1.slim.min.js
```

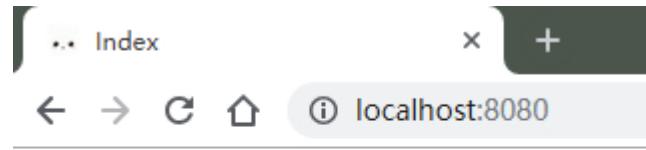
- index.html，欢迎页，静态资源文件夹下的所有index.html页面；被/**`映射

```
1 # 欢迎页，找index.html页面
2 http://localhost:8080/
```

← → C ⌂ ⓘ localhost:8080

Spring Boot 欢迎页

- **/favicon.ico，图标都是在静态资源换文件下找



Spring Boot 欢迎页

4.4. Thymeleaf 模版引擎

4.4.1. 整合Thymeleaf

- 引入thymeleaf

```
1      <!-- 覆盖 thymeleaf 版本号 -->
2      <thymeleaf.version>3.0.9.RELEASE</thymeleaf.version>
3      <!-- 布局功能的支持程序，thymeleaf3主程序适配layout2以上版本 -->
4      <thymeleaf-layout-dialect.version>2.3.0</thymeleaf-layout-
dialect.version>
5      <!-- 引入thymeleaf，默认为2.16版本，推荐修改为3.x版本 -->
6      <dependency>
7          <groupId>org.springframework.boot</groupId>
8          <artifactId>spring-boot-starter-thymeleaf</artifactId>
9      </dependency>
```

4.4.2. Thymeleaf使用&语法

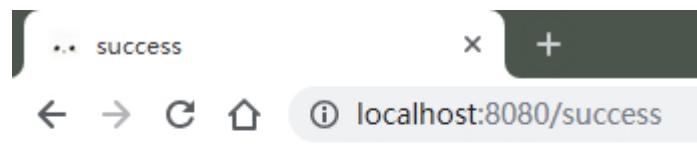
4.4.2.1. Thymeleaf使用

```
1  @ConfigurationProperties(prefix = "spring.thymeleaf")
2  public class ThymeleafProperties {
3
4      private static final Charset DEFAULT_ENCODING =
Charset.forName("UTF-8");
5
6      private static finalMimeType DEFAULT_CONTENT_TYPE =
MimeType.valueOf("text/html");
7
8      public static final String DEFAULT_PREFIX = "classpath:/templates/";
9
10     public static final String DEFAULT_SUFFIX = ".html";
11     // 只要把HTML页面放在classpath:/templates/，thymeleaf就能自动渲染；
```

只要把HTML页面放在 `classpath:/templates/`，thymeleaf就能自动渲染

```
1  /***
```

```
2 * ThymeleafController  
3 *  
4 * @author colg  
5 */  
6 @Controller  
7 public class ThymeleafController {  
8  
9     @GetMapping("/success")  
10    public String success() {  
11        // classpath:/templates/success.html  
12        return "success";  
13    }  
14 }  
15 }
```



Spring Boot Success

- 使用：
 - 导入thymeleaf的名称空间

```
1 <html lang="en" xmlns:th="http://www.thymeleaf.org">
```

- 使用thymeleaf语法

```
1 <!DOCTYPE html>  
2 <html lang="en" xmlns:th="http://www.thymeleaf.org">  
3 <head>  
4 <meta charset="UTF-8">  
5 <title>success2</title>  
6 </head>  
7 <body>  
8     <h1>Spring Boot Success2</h1>  
9     <!-- th:text 将div里面的文本内容设置为 -->  
10    <div th:text="${hello}">这是显示欢迎信息</div>  
11 </body>  
12 </html>
```

4.4.2.2. Thymeleaf语法

```

1 # 简单表达
2 ${...}: 变量表达式
3 *{...}: 选择变量表达式，和${...}在功能上是一样的
4 #{...}: 获取国际化消息
5 @{...}: url表达式
6 ~{...}: 片段引用表达式
7 # 字面
8 文本文字: 'one text' , 'Another one!' , ...
9 号码文字: 0 , 34 , 3.0 , 12.3 , ...
10 布尔文字: true, false
11 空字面: null
12 文字标记: one, sometext, main, ...
13 # 文字操作
14 字符串连接: +
15 文字替换: |The name is ${name}|
16 # 算术运算
17 二元运算符: + , - , * , / , %
18 减号(一元运算符): -
19 布尔运算:
20 二元运算符: and, or
21 布尔否定(一元运算符): !, not
22 # 比较运算
23 比较: > , < , >= , <= ( gt , lt , ge , le )
24 相等: == , != ( eq , ne )
25 # 条件运算
26 IF-THEN: (if) ? (then)
27 IF-THEN-ELSE: (if) ? (then) : (else)
28 默认: (value) ?: (defaultValue)
29 # 特殊
30 无操作: _

```

4.4.3. SpringMVC自动配置

4.4.3.1. SpringMVC自动配置

<https://docs.spring.io/spring-boot/docs/1.5.16.RELEASE/reference/htmlsingle/#boot-features-developing-web-applications>

Spring Boot自动配置好了SpringMVC

以下是SpringBoot对SpringMVC的默认配置：

- 自动配置了ViewResolver（视图解析器：根据方法的返回值得到视图对象（View），视图对象决定如何渲染（转发？重定向））
 - `ContentNegotiatingViewResolver`：组合所有的视图解析器；
 - 如何定制：可以给容器中添加一个视图解析器，自动将其组合起来

- 静态资源文件夹路径、 webjars
- 自动注册了 `Converter` , `GenericConverter` , `Formatter`
 - `Converter` : 转换器 ; `public String hello(User user);` 类型转换器使用 `Converter`
 - `Formatter` : 格式化器 ; `2018/09/28 == Date;`
 - 自己添加的转换器只需要放在容器中即可
- `HttpMessageConverters` : SpringMVC用来转换Http请求和响应的 ; `user=json`
 - `HttpMessageConverters` : 是从容器中确定的 ; 获取所有的 `HttpMessageConverter`

4.3.3.2. 扩展SpringMVC

```

1   <mvc:view-controller path="/success" view-name="success"/>
2   <mvc:interceptors>
3     <mvc:interceptor>
4       <mvc:mapping path="success"/>
5       <bean></bean>
6     </mvc:interceptor>
7   </mvc:interceptors>

```

编写一个配置类 (`@Configuration`) , 是 `WebMvcConfigurerAdapter` 类型 ; 不能标注 `EnableWebMvc`

4.4.4. 如何修改默认配置

- 模式
 - SpringBoot在自动配置很多组件的时候 , 先看容器中有没有用户自己配置的 (`@Bean`、 `@Component`) , 如果有就用用户配置的 , 如果没有 , 才自动配置 ; 如果有些组件可以有多个 (`ViewResolver`) 将用户配置的和自己默认的组合起来 ;

```

1 /**
2 * <pre>
3 * SpringMVC 扩展
4 * `WebMvcConfigurerAdapter` : 扩展SpringMVC的功能
5 * </pre>
6 *
7 * @author colg
8 */
9 @Configuration
10 public class MyMvcConfig extends WebMvcConfigurer{
11
12 /**
13 * 视图映射
14 *

```

```

15     * @param registry
16     */
17     @Override
18     public void addViewControllers(ViewControllerRegistry registry)
19     {
20         // 浏览器发送 /colg 请求来到 success 页面
21         // 请求: http://localhost:8080/colg
22         // 跳转: classpath:/templates/success.html
23         registry.addViewController("/colg").setViewName("success");
24     }

```

- 既保留了所有的自动配置，也能用扩展的配置

- 原理：

- `WebMvcAutoConfiguration`：SpringMVC的自动配置类
- `@Import(EnableWebMvcConfiguration.class)`：在做其他自动配置时会导入 `EnableWebMvcConfiguration`

```

1  @Configuration
2  public static class EnableWebMvcConfiguration extends
DelegatingWebMvcConfiguration {

```

```

1  // 从容器中获取所有的WebMvcConfigurer
2  @Autowired(required = false)
3  public void setConfigurers(List<WebMvcConfigurer> configurers) {
4      if (!CollectionUtils.isEmpty(configurers)) {
5          this.configurers.addWebMvcConfigurers(configurers);
6          // 一个参考实现；将所有的WebMvcConfigurer相关配置都来一起调用
7          /*
8              @Override
9              protected void addViewControllers(ViewControllerRegistry
registry) {
10                  this.configurers.addViewControllers(registry);
11              }
12          */
13      }
14  }

```

- 容器中所有的 WebMvcConfigurer都会一起起作用
- 自己写的配置类也会被调用
 - 效果：SpringMVC的自动配置和扩展配置都会起作用
- **全面接管SpringMVC**

- SpringBot对SprngMVC的自动配置就不需要了，所有的都是自己配置；所有的SpringMVC自动配置都时效
- 在配置类中添加 @EnablewebMvc 即可
- @EnablewebMvc 原理

```

1 | @Import(DelegatingWebMvcConfiguration.class)
2 | public @interface EnablewebMvc {
3 |

```

```

1 | @Configuration
2 | public class DelegatingWebMvcConfiguration extends
   | WebMvcConfigurationSupport {

```

- 自动配置类

```

1 | @Configuration
2 | @ConditionalOnWebApplication
3 | @ConditionalOnClass({ Servlet.class, DispatcherServlet.class,
4 |                     WebMvcConfigurerAdapter.class })
5 | // 容器中没有这个组件的时候，这个自动配置类才生效
6 | @ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
7 | @AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
8 | @AutoConfigureAfter({ DispatcherServletAutoConfiguration.class,
9 |                      ValidationAutoConfiguration.class })
10 | public class WebMvcAutoConfiguration {

```

- @EnablewebMvc 将 WebMvcConfigurationSupport 组件导入进来了，导入了 SpringMVC最基本的功能
- 在SpringBoot中会有非常多的 xxxConfigurer 帮助扩展配置

4.4.5. RestfulCRUD

4.4.5.1. 配置首页

- 方式一

```

1 | /**
2 |  * LoginController
3 |  *
4 |  * @author colg
5 |  */
6 | @Controller
7 | public class LoginController {
8 |

```

```
9  /**
10  * 使用空的controller跳转登录页
11  *
12  * @return
13  * @author colg
14  */
15 @GetMapping({"/", "/index", "index.html"})
16 public String login() {
17     return "login";
18 }
19
20 }
```

- 方式二

```
1 /**
2 * <pre>
3 * SpringMVC 扩展
4 * `WebMvcConfigurerAdapter`：扩展SpringMVC的功能
5 * </pre>
6 *
7 * @author colg
8 */
9 // @EnableWebMvc 不要接管SpringMVC
10 @Configuration
11 public class MyMvcConfig extends WebMvcConfigurerAdapter {
12
13 /**
14 * 视图映射 <br>
15 * 所有的WebMvcConfigurerAdapter组件都会一起起作用
16 *
17 * @param registry
18 */
19 @Override
20 public void addViewControllers(ViewControllerRegistry registry)
{
21     // 浏览器发送 /colg 请求来到 success 页面
22     // 请求: http://localhost:8080/colg 跳转:
23     classpath:/templates/success.html
24     registry.addViewController("/colg").setViewName("success");
25
26     // 跳转: classpath:/templates/login.html
27     registry.addViewController("/").setViewName("login");
28     registry.addViewController("/index").setViewName("login");
29 }
```

```
28     registry.addViewController("/index.html").setViewName("login");
29 }
30
31 }
```

4.4.5.2. 引入静态资源

```
1 <!-- Bootstrap core CSS -->
2 <link href="asserts/css/bootstrap.min.css"
      th:href="@{/asserts/css/bootstrap.min.css}" rel="stylesheet">
3 <!-- Custom styles for this template -->
4 <link href="asserts/css/signin.css" th:href="@{/asserts/css/signin.css}"
      rel="stylesheet">
```

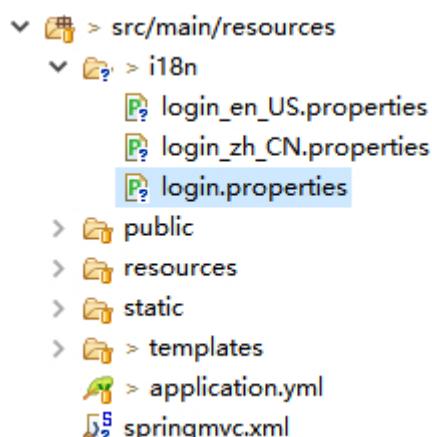
```
1 server:
2   context-path: /crud
```

```
1 <!-- Bootstrap core CSS -->
2 <link href="/crud/asserts/css/bootstrap.min.css" rel="stylesheet">
3 <!-- Custom styles for this template -->
4 <link href="/crud/asserts/css/signin.css" rel="stylesheet">
```

修改server.context-path后，不受影响

4.4.5.3. 国际化

- SpringMVC国际化步骤
 - 编写国际化配置文件
 - 使用ResourceBundleMessageSource管理国际化资源文件
 - 在页面使用fmt:message取出国际化内容
- SpringBoot国际化步骤
 - 编写国际化配置文件，抽取页面需要显示的国际化消息



```
1 login.tip=请登录  
2 login.username=用户名  
3 login.password=密码  
4 login.remember=记住我  
5 login.btn=登录  
6 login.chinese=中文  
7 login.english=英文
```

- SpringBoot自动配置好了管理国际化资源文件的组件；

- SpringBoot自动配置国际化

```
1 @ConfigurationProperties(prefix = "spring.messages")  
2 public class MessageSourceAutoConfiguration {  
3     // 配置文件可以直接放在类路径下叫messages.properties  
4     private String basename = "messages";  
5  
6     @Bean  
7     public MessageSource messageSource() {  
8         ResourceBundleMessageSource messageSource = new  
ResourceBundleMessageSource();  
9         if (StringUtils.hasText(this.basename)) {  
10             // 设置国际化资源文件的基础名(去掉语言国家代码)  
11             messageSource.setBasename(StringUtils.commaDelimitedListToStringArray(  
12                     StringUtils.trimAllWhitespace(this.basename)));  
13         }  
14         if (this.encoding != null) {  
15             messageSource.setDefaultEncoding(this.encoding.name());  
16         }  
17         messageSource.setFallbackToSystemLocale(this.fallbackToSystem  
Locale);  
18         messageSource.setCacheSeconds(this.cacheSeconds);  
19         messageSource.setAlwaysUseMessageFormat(this.alwaysUseMessage  
Format);  
20         return messageSource;  
21     }
```

- 修改配置文件名称

```
1 spring:  
2   messages:  
3     basename: i18n.login
```

- 在页面获取国际化内容

```
1 <body class="text-center">  
2   <form class="form-signin" action="dashboard.html">  
3       
6     <h1 class="h3 mb-3 font-weight-normal">[[#{login.tip}]]  
7     </h1>  
8     <label class="sr-only">[[#{login.username}]]</label>  
9     <input type="text" class="form-control"  
10    placeholder="Username" th:placeholder="#{login.username}"  
11    required="" autofocus="">  
12     <label class="sr-only">[[#{login.password}]]</label>  
13     <input type="password" class="form-control"  
14    placeholder="Password" th:placeholder="#{login.password}"  
15    required="">  
16     <div class="checkbox mb-3">  
17       <label>  
18         <input type="checkbox" value="remember-me"> [[#  
19           {login.remember}]]  
20       </label>  
21     </div>  
22     <button class="btn btn-lg btn-primary btn-block"  
23       type="submit">[[#{login.btn}]]</button>  
24     <p class="mt-5 mb-3 text-muted">© 2017-2018</p>  
25     <a class="btn btn-sm">[[#{login.chinese}]]</a>  
26     <a class="btn btn-sm">[[#{login.english}]]</a>  
27   </form>  
28 </body>
```

- 国际化原理：

- Locale (区域信息对象) ; LocaleResolver (获取区域信息对象)

```

1      @Bean
2          @ConditionalOnMissingBean
3          @ConditionalOnProperty(prefix = "spring.mvc", name =
4              "locale")
4          public LocaleResolver localeResolver() {
5              if (this.mvcProperties
6                  .getLocaleResolver() ==
7                      webMvcProperties.LocaleResolver.FIXED) {
8                  return new
9                      FixedLocaleResolver(this.mvcProperties.getLocale());
10             }
11             // 如果没有设置，就根据请求头"Accept-Language"获取区域信息进
12             行国际化
13             AcceptHeaderLocaleResolver localeResolver = new
14                 AcceptHeaderLocaleResolver();
15             localeResolver.setDefaultLocale(this.mvcProperties.getLocale());
16             return localeResolver;
17         }

```

■ 点击链接实现国际化

```

1 <a class="btn btn-sm"
2     th:href="@{/index.html(language='zh_CN')}">[[#{login.chinese}]]
3 </a>
4 <a class="btn btn-sm"
5     th:href="@{/index.html(language='en_US')}">[[#{login.english}]]
6 </a>

```

```

1 /**
2 * 可以在链接上携带区域信息
3 *
4 * @author colg
5 */
6 public class MyLocaleResolver implements LocaleResolver {
7
8 /**
9 * 区域信息解析器 <br>
10 * 通过给定的请求解析当前的语言环境
11 *
12 * @param request
13 * @return
14 */
15 @Override
16 public Locale resolveLocale(HttpServletRequest request) {

```

```

17     String language = request.getParameter("language");
18     Locale locale = Locale.getDefault();
19     if (StrUtil.isNotEmpty(language)) {
20         // 获取 语言_国家
21         String[] split = language.split("_");
22         locale = new Locale(split[0], split[1]);
23     }
24     return locale;
25 }
26
27 @Override
28 public void setLocale(HttpServletRequest request,
29 HttpServletResponse response, Locale locale) {}
30 }
```

```

1 /**
2 * <pre>
3 * SpringMVC 扩展
4 * `WebMvcConfigurerAdapter`：扩展SpringMVC的功能
5 * </pre>
6 *
7 * @author colg
8 */
9 // @EnableWebMvc 不要接管SpringMVC
10 @Configuration
11 public class MyMvcConfig extends WebMvcConfigurerAdapter {
12
13     /**
14      * 区域解析
15      *
16      * @return
17      * @author colg
18      */
19     @Bean
20     public LocaleResolver localeResolver() {
21         return new MyLocaleResolver();
22     }
23 }
```

4.4.5.4. 登录

- 开发期间模版引擎页面修改以后，要实时生效

```
1 | spring:  
2 |   thymeleaf:  
3 |     cache: false          # 禁用模版引擎缓存
```

- 登录错误消息的显示

```
1 | @PostMapping("/user/login")  
2 | public String login(String username, String password, Model  
model) {  
3 |     if (StrUtil.isNotBlank(username) &&  
"123456".equals(password)) {  
4 |         // 登录成功  
5 |         // return "dashboard";  
6 |         // 防止表单重复提交，使用重定向到主页  
7 |         return "redirect:/main.html";  
8 |     } else {  
9 |         // 登录失败  
10 |         model.addAttribute("msg", "用户名或密码错误");  
11 |         return "login";  
12 |     }  
13 | }
```

```
1 | <!--/* 条件判断 */-->  
2 | <p style="color: red;" th:if="${!#strings.isEmpty(msg)}">  
[[${msg}]]</p>
```

```
1 | // 跳转: classpath:/templates/dashboard.html  
2 | registry.addViewController("/main.html").setViewName("dashboard");
```

- 拦截器进行登录检查

- 登录成功，用户信息放入session

```
1 | @PostMapping("/user/login")  
2 | public String login(String username, String password, Model  
model, HttpSession session) {  
3 |     if (StrUtil.isNotBlank(username) &&  
"123456".equals(password)) {  
4 |         // 登录成功  
5 |         session.setAttribute("loginUser", username);  
6 |  
7 |         // return "dashboard";  
8 |         // 防止表单重复提交，使用重定向到主页
```

```
9         return "redirect:/main.html";
10    } else {
11        // 登录失败
12        model.addAttribute("msg", "用户名或密码错误");
13        return "login";
14    }
15 }
```

- 配置拦截器

```
1 public class LoginHandlerInterceptor implements HandlerInterceptor {
2
3     @Override
4     public boolean preHandle(HttpServletRequest request,
5                             HttpServletResponse response, Object handler) throws Exception {
6         Object loginUser =
7             request.getSession().getAttribute("loginUser");
8         if (loginUser == null) {
9             // 未登录，返回登录页面
10            request.setAttribute("msg", "没有权限，请先登录");
11
12            request.getRequestDispatcher("/index.html").forward(request,
13                                response);
14            return false;
15        } else {
16            // 已登录，放行
17            return true;
18        }
19    }
20 }
```

```
1 @Override
2 public void addInterceptors(InterceptorRegistry registry) {
3     // SpringBoot已经做好了静态资源映射，不需要再处理
4     registry.addInterceptor(new LoginHandlerInterceptor())
5         // 拦截的请求
6         .addPathPatterns("/**")
7         // 不拦截的请求
8         .excludePathPatterns("/", "/index", "/index.html",
9         "/user/login");
10 }
```

4.4.5.5. CRUD-员工列表

4.4.5.5.1. 要求

- CRUD满足Restful风格
 - URI : /资源名称/资源标识 HTTP请求区分对资源CRUD操作

	普通CRUD (uri来区分操作)	RestfulCR
查询	getEmp	emp---GET
添加	addEmp?xxx	emp---POST
修改	updateEmp?id=xxx&xxx=xxx	emp/{id}---PUT
删除	deleteEmp?id=xxx	emp/{id}---DELETE

- 请求架构

	请求URI	请求方式
查询所有员工	emps	GET
查询某个员工	emp/{id}	GET
跳转添加页面	emp	GET
添加员工	emp	POST
跳转修改页面 (查出员工信息回显)	emp/{id}	GET
修改员工	emp	PUT
删除员工	emp/{id}	DELETE

4.4.5.5.1. 实现

- 员工列表
 - thymeleaf公共页面元素抽取，引用

```

1 <!-- 1. 抽取公共片段 -->
2 <div th:fragment="copy">
3   &copy; 2011 The Good Thymes Virtual Grocery
4 </div>
5
6 <!-- 2. thymeleaf引用公共页面 -->
7 <div th:replace="~{footer :: copy}"></div>
8 <!--
9   ~{templatename::selector} 模板名::选择器
10  ~{templatename::fragmentname} 模板名::片段名

```

```
11 -->
12
13 <!-- 3. 默认效果
14     insert的功能片段写在div标签里
15     如果使用th:insert等属性进行引入，可以不用写~{}
16     行内写法需要加上~{} : [[~{}]], [(~{})]
17 -->
```

- 页面抽取，引用三种方式

```
1 <!-- 抽取 -->
2 <footer th:fragment="copy">
3     &copy; 2011 The Good Thymes Virtual Grocery
4 </footer>
5
6 <!-- 引用 -->
7 <div th:insert="footer :: copy"></div>
8 <div th:replace="footer :: copy"></div>
9 <div th:include="footer :: copy"></div>
10
11 <!-- 效果 -->
12 <div>
13     <footer>
14         &copy; 2011 The Good Thymes Virtual Grocery
15     </footer>
16 </div>
17
18 <footer>
19     &copy; 2011 The Good Thymes Virtual Grocery
20 </footer>
21
22 <div>
23     &copy; 2011 The Good Thymes Virtual Grocery
24 </div>
```

- 引入片段的时候传入参数

```
1 <nav th:replace="commons/bar::#sidebar(activeUri = 'main.html')">
2     </nav>
3 <nav th:replace="commons/bar::#sidebar(activeUri = 'emps')"></nav>
```

- 员工添加

- SpringBoot日期的格式化（页面提交数据）
 - 默认日期格式

```
1  /**
2   * Date format to use (e.g. dd/MM/yyyy).
3   */
4  private String dateFormat;
```

- 修改默认格式

```
1 mvc:
2     date-format: yyyy-MM-dd HH:mm:ss
```

- 员工修改

- 回显radio、select

```
1 th:value="#dates.format(emp.birth, 'yyyy-MM-dd HH:mm:ss')"
2 th:checked="${emp.gender == 1}"
3 th:selected="${emp.department.id == dept.id}"
```

- 员工删除

```
1 <!--/* 设置自定义属性 */-->
2 <button class="btn btn-sm btn-danger deleteBtn"
3     th:attr="del_uri=@{/emp/{empId}}(empId=${emp.id})">删除</button>
4
5 <!--/* 删除 , delete请求 */-->
6 <form id="deleteEmpForm" method="post" style="display: inline-
7 block;">
8     <input type="hidden" name="_method" value="delete" />
9 </form>
10
11    $(".deleteBtn").click(function() {
12        // 删除当前员工
13        let deluri = $(this).attr("del_uri")
14        // 修改删除表单的action地址
15        $("#deleteEmpForm").attr("action", deluri).submit()
16        return false
17    })
```

4.4.6. SpringBoot错误处理

4.4.6.1. SpringBoot默认的错误处理

- 默认效果

- 返回一个默认的错误页面

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Sep 30 15:41:28 CST 2018

There was an unexpected error (type=Not Found, status=404).

No message available

▼ Request Headers [view source](#)

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7
Cache-Control: max-age=0
Connection: keep-alive
```

- 如果是其他客户端，默认响应一个json数据

The screenshot shows a Postman request to 'localhost:8080/crud/aaaaaa'. The 'Body' tab is selected, displaying a JSON response:

```
1 {  
2   "timestamp": 1538296572395,  
3   "status": 404,  
4   "error": "Not Found",  
5   "message": "No message available",  
6   "path": "/crud/aaaa"  
7 }
```

Below the body, the 'Request Headers' section is expanded, showing the following headers:

```
cache-control: "no-cache"  
postman-token: "c9dbeb4f-bb16-4bd7-9c2a-8cec0d1746b2"  
user-agent: "PostmanRuntime/7.3.0"  
accept: "*/*"  
host: "localhost:8080"  
cookie: "JSESSIONID=BAF4AFAA3C8580984518263F66A0256F; SESSION=7bf8e071-0ed6-43a4-9f55-f26793950907"  
accept-encoding: "gzip, deflate"  
content-type: "multipart/form-data; boundary=-----161150634499194480120835"  
content-length: 281
```

- 原理：参照 `ErrorMvcAutoConfiguration`；错误处理的自动配置
- 给容器中添加了以下组件
 - `DefaultErrorAttributes`
 - `BasicErrorController`：处理默认/error请求

```
1 @Controller  
2 @RequestMapping("${server.error.path:${error.path:/error}}")
```

```
3 public class BasicErrorController extends AbstractErrorController {
4
5     // 产生html类型的数据；浏览器发送的请求来到这个方法处理
6     @RequestMapping(produces = "text/html")
7     public ModelAndView errorHtml(HttpServletRequest request,
8         HttpServletResponse response) {
9         HttpStatus status = getStatus(request);
10        Map<String, Object> model =
11            Collections.unmodifiableMap(getErrorAttributes(
12                request, isIncludeStackTrace(request,
13                MediaType.TEXT_HTML)));
14        response.setStatus(status.value());
15        // 去哪个页面作为错误页面；包含页面地址和页面内容
16        ModelAndView modelAndView = resolveErrorView(request,
17            response, status, model);
18        return (modelAndView != null) ? modelAndView : new
19            ModelAndView("error", model);
20    }
21
22    // 产生json数据；其他客户端来到这个方法处理
23    @RequestMapping
24    @ResponseBody
25    public ResponseEntity<Map<String, Object>>
26    error(HttpServletRequest request) {
27        Map<String, Object> body = getErrorAttributes(request,
28            isIncludeStackTrace(request, MediaType.ALL));
29        HttpStatus status = getStatus(request);
30        return new ResponseEntity<Map<String, Object>>(body,
31            status);
32    }
33}
```

- o `ErrorPageCustomizer`

```
1 @Value("${error.path:/error}")
2 private String path = "/error"; // 系统出现错误以来到error请求进
3 行处理；web.xml
```

- o `DefaultErrorViewResolver`

```
1     @Override
2     public ModelAndView resolveErrorView(HttpServletRequest
3         request, HttpStatus status,
4             Map<String, Object> model) {
```

```

4         ModelAndView modelAndView =
5             resolve(String.valueOf(status), model);
6             if (modelAndView == null &&
7                 SERIES_VIEWS.containsKey(status.series())) {
8                 modelAndView =
9                     resolve(SERIES_VIEWS.get(status.series()), model);
10            }
11
12     private ModelAndView resolve(String viewName, Map<String,
13                                     Object> model) {
14         // 默认springBoot可以去找到一个页面 ? error/404
15         String errorViewName = "error/" + viewName;
16         // 模版引擎可以解析这个页面地址就用模版引擎解析
17         TemplateAvailabilityProvider provider =
18             this.templateAvailabilityProviders
19                 .getProvider(errorViewName,
20                  this.applicationContext);
21         if (provider != null) {
22             // 模版引擎可以用的情况下返回到errorViewName指定的视图地址
23             return new ModelAndView(errorViewName, model);
24         }
25         // 模版引擎不可用，就在静态资源文件夹下找errorViewName对应的页面
26         // error/404.html
27         return resolveResource(errorViewName, model);
28     }

```

- 步骤：

- 一旦系统出现4xx或者5xx之类的错误；`ErrorResponseCustomizer` 就会生效（定制错误的响应规则）；就会来到/error请求；就会被 `BasicErrorController` 处理；
- 响应页面；去哪个页面是由 `DefaultErrorViewResolver` 解析得到

```

1     protected ModelAndView resolveErrorView(HttpServletRequest
request,
2             HttpServletResponse response, HttpStatus status,
3             Map<String, Object> model) {
4                 // 所有的ErrorViewResolver得到ModelAndView
5                 for (ErrorViewResolver resolver : this.errorViewResolvers) {
6                     ModelAndView modelAndView =
7                         resolver.resolveErrorView(request, status, model);
8                     if (modelAndView != null) {
9                         return modelAndView;
10                    }
11                }
12            }

```

- 如何定制错误响应：

- 如何定制错误页面
 - 有模版引擎：error/状态码.html 【将错误页面明为：错误状态码.html，放在模版引擎文件夹里面的error文件夹下】，发生此状态码的错误就会来到对应的页面；可以使用4xx和5xx作为错误页面的文件名来匹配这种类型的所有错误，精确优先；
 - 页面能获取的信息

```

1 timestamp:时间戳
2 status:状态码
3 error:错误提示
4 exception:异常对象
5 message:异常消息
6 path:请求路径
7 errors:JSR303数据校验的错误都在这里

```

- 没有模版引擎（模版引擎找不到这个错误页面）：静态资源文件夹下找，不推荐；不能使用模版引擎的语法，也不能获取status等页面的信息
- 以上都没有错误页面，就是默认来到SpringBoot默认的错误提示页面

- 如何定制错误数据
 - 自定义异常处理&返回定制json数据

```

1 @RestControllerAdvice
2 public class MyExceptionHandler {
3
4     /**
5      * 1. 没有自适应效果，浏览器和客户端返回的都是json

```

```

6   *
7   * @param e
8   * @return
9   * @author colg
10  */
11 @ExceptionHandler(UserNotExistException.class)
12 public Dict handleException(Exception e) {
13     Dict dict = Dict.create()
14         .set("code", "user.notexist")
15         .set("message", e.getMessage());
16     return dict;
17 }
18 }
```

- 自定义异常处理&转发到error进行自适应响应效果处理

```

1 /**
2  * 异常处理器2
3  *
4  * @author colg
5  */
6 @ControllerAdvice
7 public class MyExceptionHandler2 {
8
9 /**
10  * 2. 转发到/error进行自定义响应效果处理
11  *
12  * @param e
13  * @return
14  * @author colg
15  */
16 @ExceptionHandler(UserNotExistException.class)
17 public String handleException(Exception e,
HttpServletRequest request) {
18     Dict dict = Dict.create()
19         .set("code", "user.notexist")
20         .set("message", e.getMessage());
21     // 传入自己定义的错误状态码 4xx 5xx , 否则就不会进入定制错误页
22     // 面的解析流程
23
24     request.setAttribute("javax.servlet.error.status_code",
HttpStatus.BAD_REQUEST.value());
25     // 转发到/error
26     return "forward:/error";
27 }
```

- 自定义异常处理&将定制数据携带出去

- 出现错误以后，会来到/error请求，会被 `BasicErrorController` 处理，响应出去可以获取的数据是由 `getErrorAttributes` 得到的（是 `AbstractErrorController`，`ErrorController` 规定的方法）；
- 编写一个 `ErrorController` 的实现类，或者编写 `AbstractErrorController` 的子类，放在容器中
- 页面上能用的数据，或者是json返回能用的数据都是通过 `errorAttributes.getErrorAttributes` 得到
 - 自定义 `ErrorAttributes`

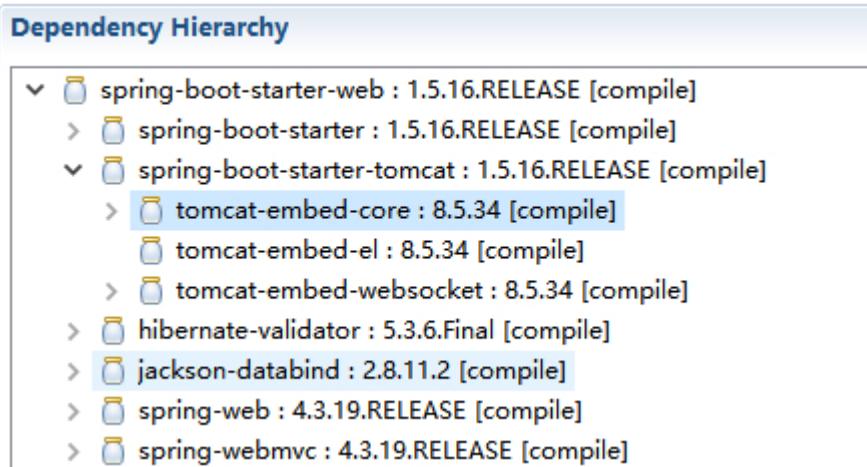
```

1  /**
2  * 给容器中加入自己定义的ErrorAttributes
3  *
4  * @author colg
5  */
6 @Component
7 public class MyErrorAttributes extends
DefaultErrorAttributes{
8
9     @Override
10    public Map<String, Object>
11        getErrorAttributes(RequestAttributes
12        requestAttributes, boolean includeStackTrace) {
13            Map<String, Object> map =
14                super.getErrorAttributes(requestAttributes,
15                includeStackTrace);
15            map.put("company", "colg");
16            return map;
17        }
18    }
19 }
```

- 最终的效果：响应是自适应的，可以通过定制 `ErrorAttributes` 改变需要返回的内容

4.4.7. 配置嵌入式Servlet容器

SpringBoot默认使用Tomcat作为嵌入式的Servlet；



4.4.7.1. 定制和修改Servlet容器的相关配置

- 配置方式：修改和server有关的配置 `ServerProperties`，也是 `EmbeddedServletContainerCustomizer`

```

1 # 通用的servlet容器设置 server: xxx
2 server:
3   port: 8081
4   context-path: /servlet          # 项目访问路径
5 # Tomcat的设置
6   tomcat:
7     uri-encoding: UTF-8

```

- 编码方式：编写一个 `EmbeddedServletContainerCustomizer`，嵌入式的Servlet容器的定制器；来修改Servlet容器的配置

```

1 /**
2  * 修改Servlet配置
3  *
4  * @author colg
5 */
6 @Configuration
7 public class MyServerConfig extends WebMvcConfigurerAdapter{
8
9 /**
10  * 配置嵌入式的Servlet
11  *
12  * @return
13  * @author colg
14  */
15 @Bean
16 public EmbeddedServletContainerCustomizer
embeddedServletContainerCustomizer() {

```

```
17     return new EmbeddedServletContainerCustomizer() {
18
19         /**
20          * 定制嵌入式的servlet容器相关的规则
21          *
22          * @param container
23          */
24         @Override
25         public void
26         customize(ConfigurableEmbeddedServletContainer container) {
27             container.setPort(8083);
28         }
29     };
30 }
```

4.4.7.2. 注册Servlet三大组件 (Servlet、Filter、Listener)

由于SpringBoot默认是以jar包的方式启动嵌入式的Servlet容器，没有web.xml文件。

注册三大组件用以下下方式：

- Servlet

```
1 /**
2  * 标准Servlet
3  *
4  * @author colg
5  */
6 public class MyServlet extends HttpServlet{
7
8     private static final long serialVersionUID = 1L;
9
10    /**
11     * 处理get请求
12     *
13     * @param req
14     * @param resp
15     * @throws ServletException
16     * @throws IOException
17     */
18    @Override
19    protected void doGet(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
20        doPost(req, resp);
21    }
```

```
22
23     /**
24      * 处理post请求
25      *
26      * @param req
27      * @param resp
28      * @throws ServletException
29      * @throws IOException
30      */
31     @Override
32     protected void doPost(HttpServletRequest req,
33                           HttpServletResponse resp) throws ServletException, IOException {
34         resp.getWriter().println("Hello MyServlet");
35     }
}
```

```
1 /**
2  * 注册Servlet
3  *
4  * @return
5  * @author colg
6  */
7 @Bean
8 public ServletRegistrationBean servletRegistrationBean() {
9     ServletRegistrationBean servletRegistrationBean = new
ServletRegistrationBean(new MyServlet(), "/myServlet");
10    return servletRegistrationBean;
11 }
```

- Filter

```
1 /**
2  * 标准Filter
3  *
4  * @author colg
5  */
6 @Slf4j
7 public class MyFilter implements Filter {
8
9     @Override
10    public void init(FilterConfig filterConfig) throws
ServletException {}
11
12    @Override
```

```
13     public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
14         log.info("MyFilter 执行");
15         // 放行
16         chain.doFilter(request, response);
17     }
18
19     @Override
20     public void destroy() {}
21
22 }
```

```
1 /**
2  * 注册Filter
3  *
4  * @return
5  * @author colg
6  */
7 @Bean
8 public FilterRegistrationBean filterRegistrationBean() {
9     FilterRegistrationBean filterRegistrationBean = new
FilterRegistrationBean();
10    filterRegistrationBean.setFilter(new MyFilter());
11
12    filterRegistrationBean.setUrlPatterns(Arrays.asList("/hello",
"/myServlet"));
13    return filterRegistrationBean;
14 }
```

- Listener

```
1 /**
2  * 标准Listener
3  *
4  * @author colg
5  */
6 @Slf4j
7 public class MyListener implements ServletContextListener {
8
9     @Override
10    public void contextInitialized(ServletContextEvent sce) {
11        log.info("contextInitialized... web应用启动");
12    }
13
14     @Override
```

```
15     public void contextDestroyed(ServletContextEvent sce) {  
16         log.info("contextDestroyed... web项目销毁");  
17     }  
18  
19 }
```

```
1  /**  
2  * 注册Listener  
3  *  
4  * @return  
5  * @author colg  
6  */  
7  @Bean  
8  public ServletListenerRegistrationBean<MyListener>  
servetListenerRegistrationBean() {  
    ServletListenerRegistrationBean<MyListener>  
servetListenerRegistrationBean = new  
ServletListenerRegistrationBean<>(new MyListener());  
10    return servetListenerRegistrationBean;  
11 }
```

4.4.7.3. 替换为其他嵌入式Servlet容器

Type hierarchy of 'org.springframework.boot.context.embedded.ConfigurableEmbeddedServletContainer': ▾

- ▼ ⓘ ConfigurableEmbeddedServletContainer - org.springframework.boot.context.embedded
 - ▼ ⓘ AbstractConfigurableEmbeddedServletContainer - org.springframework.boot.context.embedded
 - ▼ ⓘ AbstractEmbeddedServletContainerFactory - org.springframework.boot.context.embedded
 - ⓘ JettyEmbeddedServletContainerFactory - org.springframework.boot.context.embedded.jetty
 - ⓘ TomcatEmbeddedServletContainerFactory - org.springframework.boot.context.embedded.tomcat
 - ⓘ UndertowEmbeddedServletContainerFactory - org.springframework.boot.context.embedded.undertow

默认支持：

- Tomcat (默认使用)

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4   <!-- 排除Tomcat容器 -->
5   <exclusions>
6     <exclusion>
7       <artifactId>spring-boot-starter-
tomcat</artifactId>
8         <groupId>org.springframework.boot</groupId>
9       </exclusion>
10    </exclusions>
11  </dependency>
```

- Jetty

```
1 <!-- 引入其他的Servlet容器 : jetty -->
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-starter-jetty</artifactId>
5 </dependency>
```

- Undertow

```
1 <!-- 引入其他的Servlet容器 : Undertow -->
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-starter-undertow</artifactId>
5 </dependency>
```

4.4.7.4. 嵌入式Servlet容器自动配置原理

EmbeddedServletContainerAutoConfiguration : 嵌入式的Servlet容器自动配置

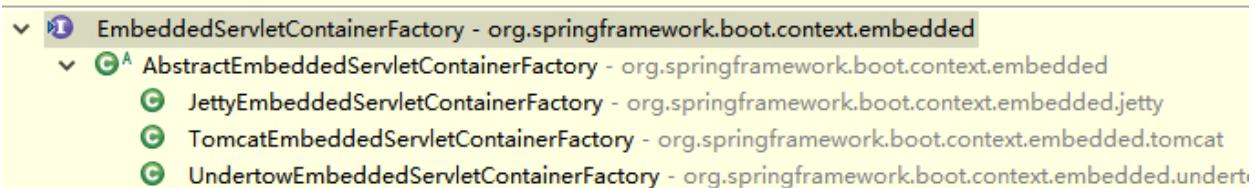
```
1 @AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
2 @Configuration
3 @ConditionalOnWebApplication
4 @Import(BeanPostProcessorsRegistrar.class)
5 public class EmbeddedServletContainerAutoConfiguration {
6   /**
7    * Nested configuration if Tomcat is being used.
8    */
9   @Configuration
10  // 判断当前是否引入了Tomcat依赖
11  @ConditionalOnClass({ Servlet.class, Tomcat.class })
```

```
12 // 判断当前容器有没有用户自定义EmbeddedServletContainerFactory：嵌入式的
13 // Servlet容器工厂；作用：创建嵌入式的Servlet容器
14 @ConditionalOnMissingBean(value =
15 EmbeddedServletContainerFactory.class, search = SearchStrategy.CURRENT)
16 public static class EmbeddedTomcat {
17
18     @Bean
19     public TomcatEmbeddedServletContainerFactory
20 tomcatEmbeddedServletContainerFactory() {
21         return new TomcatEmbeddedServletContainerFactory();
22     }
23 }
24 /**
25 * Nested configuration if Jetty is being used.
26 */
27 @Configuration
28 // 判断当前是否引入了Jetty依赖
29 @ConditionalOnClass({ Servlet.class, Server.class, Loader.class,
30                     WebApplicationContext.class })
31 @ConditionalOnMissingBean(value =
32 EmbeddedServletContainerFactory.class, search = SearchStrategy.CURRENT)
33 public static class EmbeddedJetty {
34
35     @Bean
36     public JettyEmbeddedServletContainerFactory
37 jettyEmbeddedServletContainerFactory() {
38         return new JettyEmbeddedServletContainerFactory();
39     }
40 }
41 /**
42 * Nested configuration if Undertow is being used.
43 */
44 @Configuration
45 // 判断当前是否引入了Undertow依赖
46 @ConditionalOnClass({ Servlet.class, Undertow.class,
47                     SslClientAuthMode.class })
48 @ConditionalOnMissingBean(value =
49 EmbeddedServletContainerFactory.class, search = SearchStrategy.CURRENT)
50 public static class EmbeddedUndertow {
51
52     @Bean
```

```
50     public UndertowEmbeddedServletContainerFactory  
undertowEmbeddedServletContainerFactory() {  
51         return new UndertowEmbeddedServletContainerFactory();  
52     }  
53 }  
54 }
```

- `EmbeddedServletContainerFactory` : 嵌入式Servlet容器工厂

```
1 public interface EmbeddedServletContainerFactory {  
2  
3     // 获取嵌入式的servlet容器  
4     EmbeddedServletContainer getEmbeddedServletContainer(  
5             ServletContextInitializer... initializers);  
6  
7 }
```



- `EmbeddedServletContainer`` : 嵌入式的Servlet容器



- 以 `TomcatEmbeddedServletContainerFactory` 为例

```
1     @Override  
2     public EmbeddedServletContainer getEmbeddedServletContainer(  
3             ServletContextInitializer... initializers) {  
4         // 创建一个Tomcat  
5         Tomcat tomcat = new Tomcat();  
6         // 配置Tomcat的基本环节  
7         File baseDir = (this.baseDirectory != null) ?  
this.baseDirectory  
                : createTempDir("tomcat");  
8         tomcat.setBaseDir(baseDir.getAbsolutePath());  
9         Connector connector = new Connector(this.protocol);  
10        tomcat.getService().addConnector(connector);  
11        customizeConnector(connector);  
12        tomcat.setConnector(connector);  
13        tomcat.getHost().setAutoDeploy(false);  
14 }
```

```

15     configureEngine(tomcat.getEngine());
16     for (Connector additionalConnector :
17         this.additionalTomcatConnectors) {
18         tomcat.getService().addConnector(additionalConnector);
19     }
20     prepareContext(tomcat.getHost(), initializers);
// 将配置好的Tomcat传入进入，返回一个嵌入式的Servlet容器，并且启动
21     Tomcat容器
22     return getTomcatEmbeddedServletContainer(tomcat);
}

```

4.4.7.5. 嵌入式Servlet容器启动原理

- SpringBoot应用启动run方法
- refreshContext(context) : SpringBoot刷新IOC容器（创建IOC容器对象，并初始化容器，创建容器中的每一个组件）
- refresh(context) : 刷新刚才创建好的IOC容器
- onRefresh() : web容器重写了onRefresh方法
- webioc容器会创建嵌入式的Servlet容器
-

4.4.8. 配置外部servlet容器

- 嵌入式Servlet容器：应用打成可执行的jar
 - 优点：简单，便捷
 - 缺点：默认不支持jsp，优化定制比较复杂（使用定制器【ServerProperties、自定义EmbeddedServletContainerCustomizer】，自己编写嵌入式Servlet容器的创建工作工厂）
- 外置的Servlet容器：应用打成war包
 - 必须创建一个war项目(注意目录结构)
 - 将嵌入式的Tomcat指定为 `provided`，不打包该依赖

```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-tomcat</artifactId>
4   <scope>provided</scope>
5 </dependency>

```

- 必须编写一个 `SpringBootServletInitializer` 的子类，并调用configure方法

```

1 /**
2 *
3 *

```

```
4 * @author colg
5 */
6 public class ServletInitializer extends
SpringBootServletInitializer {
7
8     @Override
9     protected SpringApplicationBuilder
configure(SpringApplicationBuilder application) {
10         // 传入SpringBoot应用的主程序
11         return
12         application.sources(SpringBoot09WebJspApplication.class);
13     }
14 }
```

- 启动服务器就可以使用

- 原理

- jar包：执行SpringBoot主类的main方法，启动ioc容器，创建嵌入式的Servlet容器
- war包：启动服务器，服务器启动SpringBoot应用 `SpringBootServletInitializer`，启动ioc容器

5. Docker

6. SpringBoot与数据访问

7. 自定义starter

8. 更多SpringBoot整合示例