

一、Spring Boot入门

1.1、Spring Boot简介

1.1.1、简介

spring boot 简化了spring应用开发，约定大于配置，去繁从简，just run 就能创建一个独立的，产品级别的应用

1.1.2、背景

J2EE笨重的开发、繁多的配置、底下的开发效率、复杂的部署流程、第三方继承难度太大

1.1.3、解决方案

Spring 全家桶时代

Spring Boot：J2EE一站式解决方案

Spring Cloud：分布式整体解决方案

1.1.4、Spring Boot 的优点

- 1.快速创建独立运行的Spring项目以及与主流框架集成
- 2.使用嵌入式Servlet容器，应用无需打成WAR包，直接可以打成jar包用java命令就可以运行，无需servlet容器
- 3.有很多starters负责自动依赖管理和版本控制
- 4.大量的自动配置，简化开发，也可以修改默认值
- 5.无需配置XML，无代码生成，开箱即用
- 6.准生产环境的运行时应用监控
- 7.与云计算的天然集成

1.1.5、缺点

入门容易，精通难

Spring Boot 是基于Spring 等框架的再封装，所以要深入理解了spring等框架才能更好地理解SpringBoot的原理

1.2、微服务简介

微服务是一种架构风格

一个应用应该由多个小型服务组成，服务之间使用http进行轻量级通信

1.2.1、单体应用

与微服务对应的，以前采用的是单体应用架构，一个应用中所有的代码都在一个项目下，统一打成war包部署到应用服务器上。

优点：

开发测试简单，不涉及到各个模块间的互联互通，所有的模块都在一个项目下

部署简单，只需要将整个应用打成war包部署就可以了，后期运维也不复杂

拓展简单，只需要将相同的应用部署到多个服务器上通过负载均衡访问即可

缺点：

1.修改一个模块就需要整个项目重新部署

2.项目维护和扩展复杂，且不利于分工合作开发

1.2.2、微服务

将每个功能元素分成一个独立的服务，后期的扩展维护只需要修改某一块

并发量高的模块可以多部署一些，可以按需分配资源

将服务细化后，每一个模块都是一个可替换的、可独立升级维护的软件单元

详细参照微服务文档

1.3holle world

1.3.1创建一个maven工程

1.3.2导入依赖spring boot 的相关依赖

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.8.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

1.3.3编写一个主程序，用于启动spring boot 应用

```

/*
 * @SpringBootApplication
 * 用来标注一个主程序类，说明这是一个spring boot 应用
 */
@SpringBootApplication
public class HolleWorldMainApplication {

    public static void main(String[] args) {
        // 让spring boot 应用启动起来
        SpringApplication.run(HolleWorldMainApplication.class, args);
    }
}

```

1.3.4编写相关的Controller、service等等

1.3.5直接运行主程序进行测试

1.3.6简化部署

导入maven插件

```

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

```

可以将这个应用打成可执行的jar包，用java -jar命令执行

1.4HolleWorld探究

1.4.1POM文件

1.4.1.1父项目

```

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.8.RELEASE</version>
</parent>

```

他的父项目是：

```

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>

    <version>1.5.8.RELEASE</version>

```

```
<relativePath>../../spring-boot-dependencies</relativePath>
</parent>
```

spring-boot-dependencies这个父项目来真正管理Spring Boot应用的所有依赖版本。在他的<properties>标签下。也可以称之为Spring Boot的版本仲裁中心，以后导入依赖默认是不需要写版本的。但是，没有在dependencies里面管理的依赖需要声明版本号

1.4.1.2导入的依赖

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

spring-boot-starter-web

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了，做成了一个starters（启动器）。

只需要在项目中引用这些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启动器。

1.4.2主程序类，主入口类

```
/*
 * @SpringBootApplication
 * 用来标注一个主程序类，说明这是一个spring boot 应用
 */
@SpringBootApplication
public class HolleWorldMainApplication {

    public static void main(String[] args) {
        // 让spring boot 应用启动起来
        SpringApplication.run(HolleWorldMainApplication.class, args);
    }
}
```

@SpringBootApplication：标注在Spring Boot 应用的某个类上，说明这个类是Spring Boot的主程序类。Spring Boot就应该运行这个类的主方法来启动Spring Boot应用。

@SpringBootApplication注解类部分源码：

```

@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 {
    //代码
}

```

@SpringBootConfiguration : Spring Boot的配置类，这个注解标注在某个类上，表示这是一个Spring Boot的配置类。

@SpringBootConfiguration源码：

```

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

}

```

@Configuration : 配置类上来标注这个注解

配置类 相当于 配置文件

@Configuration源码

```

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

}

```

配置类也是容器中的一个组件，用@Component注解标注

@EnableAutoConfiguration : 开启自动配置功能。

以前需要我们自己配置的东西，Spring Boot帮我们自动配置。@EnableAutoConfiguration告诉Spring Boot开启自动配置功能，这样自动配置才能生效。

@EnableAutoConfiguration注解类：

```

@SuppressWarnings("deprecation")
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
}

```

@AutoConfigurationPackage：自动配置包

@AutoConfigurationPackage：

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {
}

```

@Import(AutoConfigurationPackages.Registrar.class)：

Spring的底层注解@Import，给容器中导入一个组件。

将主程序类（@SpringBootApplication标注的类）所在的包及下面所有的子包里面所有的组件扫描到Spring容器。

@Import(EnableAutoConfigurationImportSelector.class)

EnableAutoConfigurationImportSelector：导入哪些组件的选择器。

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（xxxxAutoConfiguration），就是给容器中导入这个场景的所有组件，并配置好这些组件。

有了自动配置类，免去了我们手动编写配置，注入功能组件等工作。

SpringFactoriesLoader.loadFactoryNames(EnableAutoConfiguration.class,classLoader)；

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，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；

J2EE的整体整合解决方案和自动配置都在spring-boot-autoconfigure-1.5.9.RELEASE.jar；

1.5快速创建Spring Boot项目

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

二、配置文件

2.1 配置文件

Spring Boot默认支持两种全局配置文件(配置文件名是固定的)：

application.properties

application.yml

配置文件的作用：SpringBoot在底层都已经自动配置好了，配置文件就是用来修改SpringBoot的自动配置的默认值。

yml文件用的是YAML语言 (YAML Ain't Markup Language)

YAML A Markup Language：是一个标记语言

YAML isn't Markup Language：不是一个标记语言

标记语言：

以前我们的配置文件，大多都是XML文件

YAML:以数据为中心，比json、xml等更适合做配置文件

配置例子：

YAML:

```
Server:
  Port: 8081
```

XML

```
<SERVER>
  <port>8081</port>
</SERVER>
```

2.2YAML语法

2.2.1基本语法

K：(空格) V 表示一个键值对 (value前面必须有一个空格)

以空格缩进来控制层级关系。只要是左对齐的一列数据，都是同一层级的。

属性和值都是大小写敏感的

2.2.2值的写法

2.2.2.1字面量：普通的值（数字、字符串、布尔变量等）

字面量直接写：K: V

字符串默认不用加上单引号和双引号。

双引号：不会转义字符串里面的特殊字符，特殊字符作为本身想表示的意思

单引号：会转义特殊字符，特殊字符最终只作为普通的字符输出。

例如：

"zhangsan /n lisi" 输出：

```
zhangsan
lisi
```

'zhangsan /n lisi' 输出：

```
zhangsan /n lisi
```

2.2.2.2对象、Map（键值对）

还是以k: v表示，在下一行来写对象的属性和值的关系，注意缩进

如：friend对象

```
Friend:
  lastName: zhangsan
  age: 20
```

行内写法：

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

2.2.2.3数组或集合（List、Set）

用- 值表示数组的一个元素

```
Pets:
- cat
- dog
- pig
```

行内写法

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

2.3配置文件值的注入

2.3.1配置文件

配置文件：


```
Person:
  lastName: zhangsan
  age: 18
  boss: false
  birth: 2017/12/13
  maps: {k1: v1,k2: v2}
  list:
    - lisi
    - wangwu
  dog:
    name: 小狗
    age: 2
```

2.3.2 实体类

JavaBean :

```
/**
 * 将配置文件中配置的每一个属性值，映射到这个组件中
 * @ConfigurationProperties: 告诉springboot将本类中所有的属性和配置文件中相关的配置进行绑定
 * prefix = "person" : 与配置文件中哪个下面的所有属性一一对应
 *
 * 只有这个组件是容器中的组件时，才能使用容器提供的功能，所以要加上@Component注解
 */
@Component
@ConfigurationProperties(prefix = "person")
public class Person {

    private String lastName;
    private Integer age;
    private Boolean boss;
    private Date birth;

    private Map<String,Object> maps;
    private List<Object> lists;
    private Dog dog;

    @Override
    public String toString() {
        return "Person{" +
            "lastName='" + lastName + '\'' +
            ", age=" + age +
            ", boss=" + boss +
            ", birth=" + birth +
            ", maps=" + maps +
            ", lists=" + lists +
            ", dog=" + dog +
            '}';
    }

    public String getLastName() {
        return lastName;
    }
}
```

```
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public Integer getAge() {
    return age;
}

public void setAge(Integer age) {
    this.age = age;
}

public Boolean getBoss() {
    return boss;
}

public void setBoss(Boolean boss) {
    this.boss = boss;
}

public Date getBirth() {
    return birth;
}

public void setBirth(Date birth) {
    this.birth = birth;
}

public Map<String, Object> getMaps() {
    return maps;
}

public void setMaps(Map<String, Object> maps) {
    this.maps = maps;
}

public List<Object> getLists() {
    return lists;
}

public void setLists(List<Object> lists) {
    this.lists = lists;
}

public Dog getDog() {
    return dog;
}

public void setDog(Dog dog) {
    this.dog = dog;
}

}
```

```
}
```

2.3.3配置文件处理器

可以导入配置文件处理器，以后编写配置文件就有javabean对应的提示了

在pom.xml文件中添加依赖：

```
<!--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>
```

2.3.4单元测试

使用springboot的test 进行单元测试

```
/**
 * SpringBoot单元测试
 * @SpringBootTest：标注这是一个springBoot测试类
 *
 * @RunWith(SpringRunner.class)：标注这个单元测试用SpringRunner这个驱动器来跑，而不是原来的junit
 *
 * 可以再测试期间很方便的类似编码一样进行自动注入到容器的功能
 *
 */
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBoot001ApplicationTests {

    @Autowired
    private Person person;

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

2.3.5@Value注解读取配置文件

```
/**
 * @Value 注解：Spring的注解，对应xml配置文件的<bean>标签
 * <bean class="Person">
 *     <property name="lastName" value="?"></property>
 * </bean>
 *
 */
```

```

* value的值可以是：
*     1.字面量
*     2.${key}  从环境变量、配置文件中获取值
*     3.#{SpEL} SpEL表达式
*
*/

//从配置文件取值
@Value("${person.last-name}")
private String lastName;

//SpEL表达式
@Value("#{11*2}")
private Integer age;

//字面量
@Value("true")
private Boolean boss;

```

2.3.6@Value和@ConfigurationProperties的区别

	@ConfigurationProperties	@Value
功能	批量注入配置文件中的属性	需要一个个指定
松散绑定（松散语法）	如果实体类的字段为lastName。 配置文件中可以写成： person.lastName Person.last-name Person.last_name PERSON_LAST_NAME(系统属性推荐使用，不区分大小写)	必须一一对应，且大小写敏感
SpEL	不支持	支持
JSR303数据校验	支持	不支持
复杂类型封装（map、对象等）	支持	不支持

配置文件yml还是properties文件他们都能取到值

如果我们只是在某个业务逻辑中获取配置文件中某一项的值，使用@value

如果我们专门编写一个javaBean映射配置文件，我们就使用@ConfigurationProperties

2.3.7JSR303数据校验

```

@Component
@ConfigurationProperties(prefix = "person")
@Validated
public class Person {
    // lastName必须是邮箱格式
    @Email
    private String lastName;
}

```

2.3.8@PropertySource与@ImportResource

2.3.8.1@PropertySource

@PropertySource : 加载指定的配置文件

```

/**
 * @ConfigurationProperties : 默认读取全局配置文件application.properties下的配置
 * 要读取其他配置文件中的配置，需要用到@PropertySource注解指定配置文件
 */
@PropertySource(value = {"classpath:person.properties"})
@Component
@ConfigurationProperties(prefix = "person")
//@Validated
public class Person {

    private String lastName;

    private Integer age;

    private Boolean boss;
    private Date birth;

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

```

2.3.8.2@ImportResource

@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，我们自己创建的配置也不能自动识别。

想让Spring的配置文件生效，加载进来。把@ImportResource注解标注到一个配置类上

HolleService

```

public class HolleService {
}

```

配置文件 : beans.xml 配置holleService组件

```
<?xml version="1.0" encoding="UTF-8"?>
<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.xsd">

    <bean id="holleService" class="com.example.springboot001.service.HolleService"></bean>
</beans>
```

主类：用@ImportResource将配置文件导入

```
@ImportResource(locations = {"classpath:beans.xml"})
@SpringBootApplication
public class SpringBoot001Application {

    public static void main(String[] args) {
        SpringApplication.run(SpringBoot001Application.class, args);
    }
}
```

测试类：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBoot001ApplicationTests {

    @Autowired
    private ApplicationContext ioc;

    @Test
    public void testHolleService(){
        boolean b = ioc.containsBean("holleService");
        System.out.print(b);
    }
}
```

2.3.8.3 SpringBoot推荐给容器添加组件的方式

SpringBoot不推荐使用配置文件，而是使用全注解的方式

推荐使用配置类，相当于Spring的配置类，使用@Configuration标注配置类

使用@Bean给容器添加组件

```
/**
 * @Configuration: 指明当前类是一个配置类，用于替代之前的Spring配置文件
 * 配置文件中<bean>标签来添加组件的
 *
 */
@Configuration
public class MyAppConfig {

    /**
```

```

* @Bean: 对应配置文件中的<bean>标签
* 将方法的返回值添加到容器中，容器中这个组件的默认id就是方法名
*
* @return
*/
@Bean
public HolleService holleService(){
    System.out.print("配置类给容器添加组件：holleService");
    return new HolleService();
}
}

```

2.3.9 配置文件占位符

2.3.9.1 随机数

RandomValuePropertySource:

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

2.3.9.2 占位符

通过占位符获取之前配置的值，如果没有值可以使用：指定默认值

```

app.name=MyApp
app.description=${app.name:MyApp} is a Spring Boot application

```

2.4 Profile

Profile是Spring提供多环境支持的功能，对不同环境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环境

2.4.1多profile文件

多profile文件方式实现

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

如：`application-dev.properties`、`application-prod.properties`分别为开发环境和生产环境的配置文件。

默认使用`application.properties`的配置文件

2.4.2激活指定的profile

1. 在`application.properties`文件中指定：

`spring.profiles.active=dev` 指定激活开发环境的profile

2. 命令行方式：

`--spring.profiles.active=dev`

如`java -jar springboot.jar --spring.profiles.active=dev`

指定使用项目的dev环境

3. 虚拟机参数

-Dspring.profiles.active=dev

2.4.3 yml支持多文档块方式

```
spring:
  profile:
    active: dev

server:
  port: 8080

---
server:
  port: 8090

spring:
  profiles: dev
---
server:
  port: 8082

spring:
  profiles: prod
```

yaml可以用--- 划分文档块，每个文档块用spring.profiles指定profile

然后在第一个文档块用spring.profile.active指定要激活的profile

2.5 配置文件加载位置

SpringBoot启动时会扫描以下位置的application.properties或application.yml配置文件作为主配置文件：

file:./config/ 项目根目录的config文件夹下

file:./ 项目的根目录下

Classpath:./config/ 类路径下的config文件夹下

Classpath:/ 类路径的根目录下

优先级由高到低，所有位置上的配置文件都会被加载，如果出现重复的内容，高优先级的配置文件会覆盖低优先级的配置文件

项目打包完成后，可以通过命令行参数的形式，使用spring.config.location指定新的配置文件的位置，这个新的配置文件会和项目下的配置文件形成互补配置

2.6 SpringBoot外部配置加载顺序

Spring Boot除了从配置文件加载配置，还可以从以下位置加载配置。高优先级配置会覆盖低优先级配置。

1. [Devtools](#) 主目录上的[全局设置属性](#)（~/spring-boot-devtools.properties当devtools处于活动状态时）。
2. @TestPropertySource 测试上的注释。

3. @SpringBootTest#properties 测试中的注释属性。
4. 命令行参数。
5. 来自SPRING_APPLICATION_JSON (嵌入在环境变量或系统属性中的内联JSON) 的属性。
6. ServletConfig init参数。
7. ServletContext init参数。
8. JNDI属性来自java:comp/env。
9. Java系统属性 (System.getProperties()) 。
10. OS环境变量。
11. 一RandomValuePropertySource , 只有在拥有性能random.*。
12. 特定于配置文件的应用程序属性在打包的jar (application-{profile}.properties和YAML变体) 之外。
13. 打包在jar中的特定于配置文件的应用程序属性 (application-{profile}.properties 以及YAML变体) 。
14. 应用程序属性在打包的jar之外 (application.properties和YAML变体) 。
15. 打包在jar中的应用程序属性 (application.properties和YAML变体) 。
16. @PropertySource 你的@Configuration课上的注释。
17. 默认属性 (由设置指定SpringApplication.setDefaultProperties) 。

2.7 SpringBoot自动配置原理

2.7.1 自动配置原理

可配置的内容参考官方文档附录中的Common application properties

自动配置原理：

- 1) SpringBoot启动的时候加载了主配置类，开启了自动配置功能@EnableAutoConfiguration
- 2) @EnableAutoConfiguration作用：

a. 利用AutoConfigurationImportSelector给容器导入组件。

b. 可以查看selectImports()方法的相关内容。

c. List<String> configurations = this.getCandidateConfigurations(annotationMetadata, attributes);获取候选的配置

a. SpringFactoriesLoader.loadFactoryNames()

b. 扫描所有jar包类路径下 META/spring.factories

c. 把扫描到的这些文件的内容包装成properties对象

d. 从properties中获取到的@EnableAutoConfiguration.class对应的值，把他们添加到容器中

一句话来说，就是@EnableAutoConfiguration就是把所有jar包类路径下 META/spring.factories里面配置的所有@EnableAutoConfiguration的值加入到容器中。

如：

```
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,  
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,  
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,  
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,  
.....等等，参考AutoConfiguration jar包下的 META/spring.factories
```

每一个xxxxAutoConfiguration这样的类都是容器的一个组件，加入到容器中。容器用他们来做自动配置。

3) 每一个自动配置类分别进行自动配置功能

4) 以HttpEncodingAutoConfiguration为例 解释自动配置原理

```
@Configuration  
//表示这是一个配置类，相当于以前的配置文件，给容器添加组件  
@EnableConfigurationProperties({HttpEncodingProperties.class})  
//启动指定类的ConfigurationProperties功能，将配置文件的内容与HttpEncodingProperties绑定起来，并把  
HttpEncodingProperties添加到ioc容器中  
@ConditionalOnWebApplication(  
    type = Type.SERVLET  
)  
//spring底层的@Conditional注解，根据不条件判断配置类是否生效  
//判断当前项目是不是web应用，是配置类才生效  
@ConditionalOnClass({CharacterEncodingFilter.class})  
//判断有没有CharacterEncodingFilter这个类，CharacterEncodingFilter是springmvc中解决乱码问题的过滤器  
@ConditionalOnProperty(  
    prefix = "spring.http.encoding",  
    value = {"enabled"},  
    matchIfMissing = true  
)  
//判断配置文件中是否存在spring.http.encoding.enabled=true这个配置  
//matchIfMissing ：如果不存在，默认为true  
public class HttpEncodingAutoConfiguration {  
  
    //已经通过@ConfigurationProperties注解和SpringBoot配置文件映射了  
    private final HttpEncodingProperties properties;  
  
    //只有一个有参构造方法，参数会从容器中去取  
    public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {  
        this.properties = properties;  
    }  
  
    @Bean  
    //给容器添加一个组件，这个组件的某些值需要从properties里面获取  
    @ConditionalOnMissingBean  
    public CharacterEncodingFilter characterEncodingFilter() {  
        CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();  
        filter.setEncoding(this.properties.getCharset().name());  
    }  
}
```

```

    filter.setForceRequestEncoding(this.properties.shouldForce(org.springframework.boot.autoconfigure.http.HttpEncodingProperties.Type.REQUEST));

    filter.setForceResponseEncoding(this.properties.shouldForce(org.springframework.boot.autoconfigure.http.HttpEncodingProperties.Type.RESPONSE));
    return filter;
}

```

所有能配置的属性都在xxxxxProperties类中，这些都是实体类，用@ConfigurationProperties注解将配置文件的信息和实体类的属性进行绑定，所有能在配置文件中配置的属性都在这些类中

HttpEncodingProperties 类：

```

@ConfigurationProperties(
    prefix = "spring.http.encoding"
)
//从配置文件中获取指定的值和bean的属性进行绑定
public class HttpEncodingProperties {
}

```

2.7.2 SpringBoot的精髓

SpringBoot的精髓就是自动配置。

1. SpringBoot启动的时候会加载大量的自动配置类
2. 我们先看我们需要的功能有没有SpringBoot已有的自动配置类
3. 再看这个自动配置类到底配置了哪些组件（只要有我们要用的组件，我们就不用再配置了）
4. 给容器中自动配置类添加组件的时候，会从properties类中获取某些属性，我们就可以再配置文件中指定这些属性的值。

XxxxxxAutoConfiguration：自动配置类，给容器中添加组件

XxxxxxProperties：封装配置文件中相关的属性

2.7.3 @Conditional派生注解

作用：必须是@Conditional指定的条件成立，注解下的代码才生效

@Conditional扩展注解	作用（判断是否满足当前指定条件）
@ConditionalOnJava	系统的java版本是否符合要求
@ConditionalOnBean	容器中存在指定Bean；
@ConditionalOnMissingBean	容器中不存在指定Bean
@ConditionalOnExpressio	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

可以通过启用debug=true属性，来让控制台打印自动配置报告。

2.8 修改pom文件默认版本

在pom.xml的properties标签下加入版本信息即可覆盖默认版本配置

如<thymeleaf.version>3.0.9.RELEASE</thymeleaf.version> 将版本修改为3.0.9

三、日志

3.1 日志框架

常用日志框架：

JUL、JCL、Jboss-logging、logback、log4j、log4j2、slf4j....

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

用的时候左边选一个门面，右边选一个实现

SpringBoot底层使用的是Spring，Spring默认用的是JCL

SpringBoot选用slf4j和logback

3.2 slf4j的使用

3.2.1 如何在系统中使用slf4j

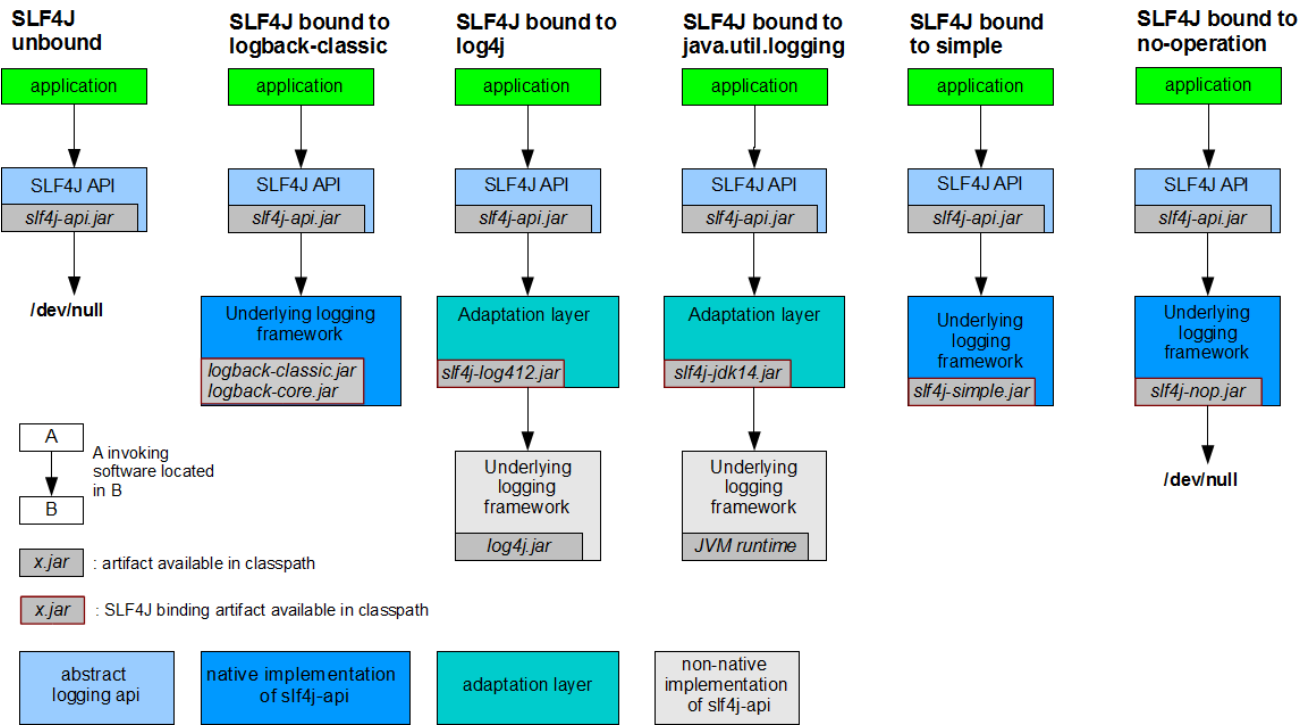
以后在开发的时候，日志调用的方法，不应该直接调用实现类，而是调用日志抽象层里面的方法。

给项目导入slf4j的jar和logback的实现jar

```
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");
    }
}
```

使用不同的日志实现，需要依赖的jar包：



各个日志实现框架的配置文件保持不变，使用slf4j只是提供一个抽象层的接口，配置文件还是写实现框架的配置文件

3.2.2 遗留问题

项目中用的是：slf4j+logback

而项目中还要集成一些其他的框架，这些框架内部都有自己的日志模块

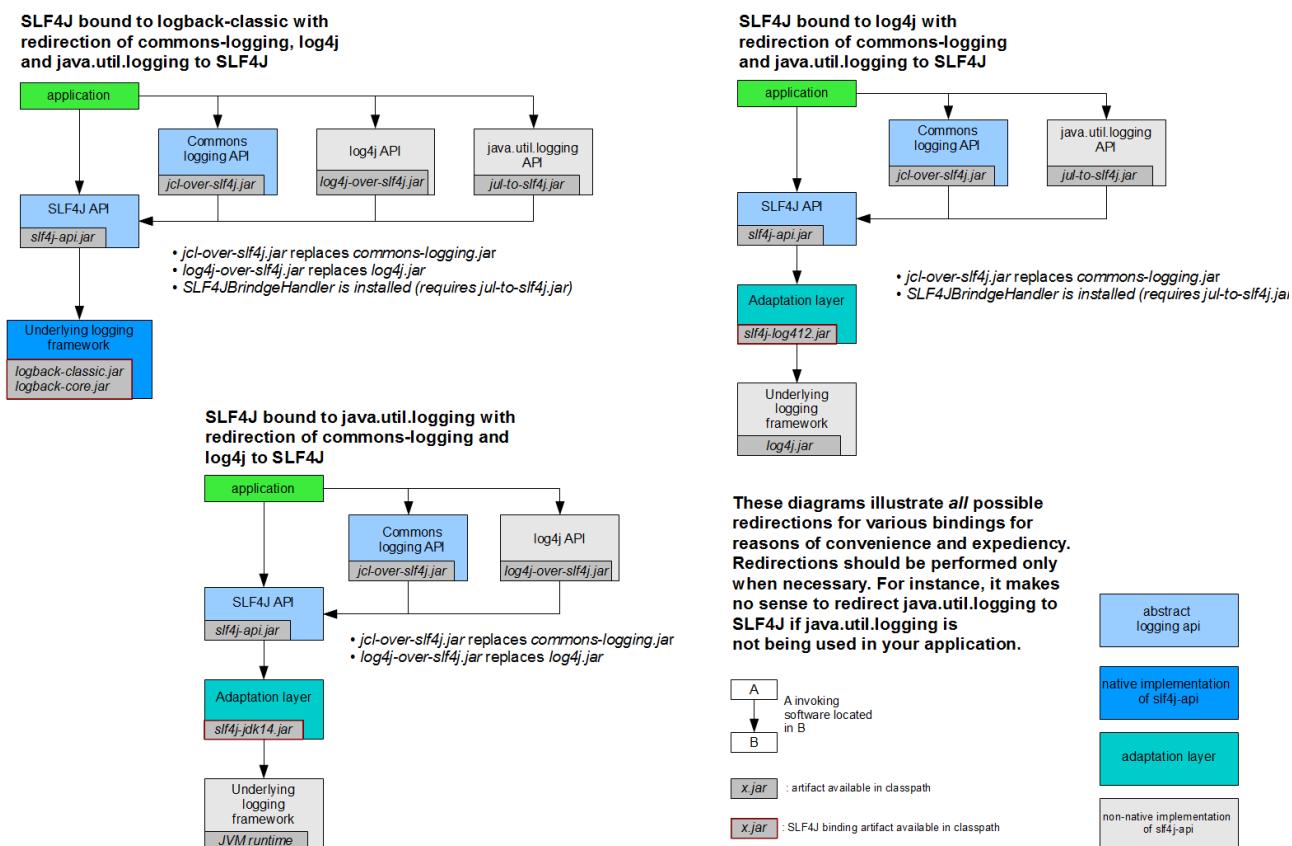
如：

Spring : commons-logging

Hibernate : jboss-logging

等等

所以要统一日志记录，即使是别的框架也要用slf4j+logback

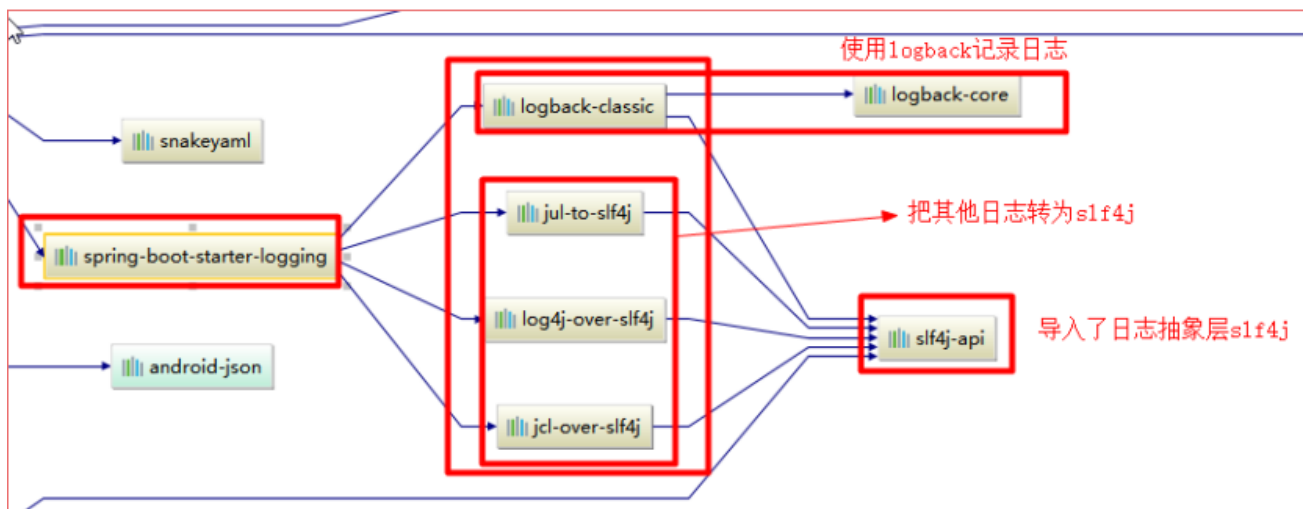


要统一日志记录，可以用对应的中间jar包替换掉原来的依赖包

3.2.3 springBoot的日志依赖关系

SpringBoot依赖他来做日志功能

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-logging</artifactId>
  <version>2.0.3.RELEASE</version>
  <scope>compile</scope>
</dependency>
```



总结：

1. SpringBoot底层也是使用Slf4j+logback的方式进行日志记录
2. SpringBoot也把其他的日志记录方式替换成了slf4j
3. 替换的中间包下的类名和被替换的实现包一样，只是内部实现代码中用的是slf4j的api调用系统统一的日志记录实现
4. 如果我们要引入的其他框架有默认的日志依赖，一定要移除掉
如,spring框架的commons-logging被springboot移除掉

```
<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slf4j+logback的方式记录日志，我们唯一需要做的是引入其他框架的时候，把这个框架依赖的日志框架排除掉

3.2.4 日志的使用

3.2.4.1 默认配置

SpringBoot默认帮我们配置好了日志

```
//日志记录器
Logger logger = LoggerFactory.getLogger(getClass());

@Test
public void contextLoads() {
```

```

//日志的级别，由低到高 trace<debug<info<warn<error
//可以调整需要输出的日志级别，只打印指定级别及更高级别的日志
//跟踪信息
logger.trace("这是trace日志...");

//调试信息
logger.debug("这是debug日志...");

//springboot默认的日志输出级别为info
//自定义的一些输出信息
logger.info("这是info日志...");

//警告日志
logger.warn("这是warn日志...");

//错误日志
logger.error("这是error日志...");

}

```

修改日志默认配置的常用配置

```

#指定com.example.springboot002包下的日志输出级别为trace
#如果没有指定那么就使用springboot的默认级别，info
logging.level.com.example.springboot002=trace

#不指定路径，只有文件名，在当前项目的根目录下生成日志文件springboot.log
#logging.file=springboot.log

#指定路径下生成一个日志文件D:/springboot.log
#logging.file=D:/springboot.log

#指定日志文件目录，在当前磁盘下生成一个spring/log文件夹，日志文件名为springboot默认的spring.log
#logging.path=/spring/log

#logging.file和logging.path都不指定，日志只在控制台打印
#logging.file和logging.path同时指定，只有logging.file生效

#指定在控制台输出的日志格式
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss.SSS} [ %thread ] - [ %-5level ] [ %logger{50} ] :
%line ] - %msg%n

#%d指定日期
#%thread 线程名
#%-5level -5: 从左边开始最多显示5个字符 %level 日志级别
#%logger{50} logger所在的全类名，最多50个字符
#%msg 消息
#%n 换行

#指定在文件中输出的日志格式
logging.pattern.file=

```


3.2.4.2 指定配置

Springboot要使用自己的日志文件，只需要把指定的配置文件名放到类路径下

Logback : logback-spring.xml logback-spring.groovy logback.xml logback.groovy

log4j2 : log4j2-spring.xml log4j2.xml

JDK(Java Util Logging) logging.properties

推荐使用logback-spring.xml

logback.xml 直接被日志框架识别了，不能使用一些特殊的配置

Logback-spring.xml 日志框架识别不了，由springBoot来解析日志的配置文件，就可以使用springProfile标签

```
<springProfile name="dev">
    //里面的配置只在指定的环境中生效
</springProfile>
```

3.2.5 切换日志框架

可以按照slf4j日志适配图进行相关的切换

3.2.5.1 自己进行依赖切换

slf4j+log4j

1. 去除logback的依赖
2. 去除适配log4j的依赖-----将log4j转为slf4j
3. 添加使用log4j的适配包

3.2.5.2 使用starter进行切换

Springboot提供了spring-boot-starter-logging和spring-boot-starter-log4j2两个启动器

spring-boot-starter-logging : slf4j+logback

spring-boot-starter-log4j2 : slf4j+log4j2

四、Web开发

使用springboot步骤

1. 创建springboot应用，选择我们需要的模块
2. Springboot已经默认配置好了这些场景，我们只需要进行少量的配置就可以运行起来
3. 自己编写业务代码

4.1 springboot对静态资源的映射规则

WebMvcAutoConfiguration类部分源码 静态资源映射

```
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    if (!this.resourceProperties.isAddMappings()) {
```

```

        logger.debug("Default resource handling disabled");
    } else {
        Duration cachePeriod = this.resourceProperties.getCache().getPeriod();
        CacheControl cacheControl =
this.resourceProperties.getCache().getCachecontrol().toHttpCacheControl();
        if (!registry.hasMappingForPattern("/webjars/**")) {
            this.customizeResourceHandlerRegistration(registry.addHandler(new String[]
{"/webjars/**"}).addResourceLocations(new String[]{"classpath:/META-
INF/resources/webjars/"}).setCachePeriod(this.getSeconds(cachePeriod)).setCacheControl(cacheCont
rol));
        }
        String staticPathPattern = this.mvcProperties.getStaticPathPattern();
        if (!registry.hasMappingForPattern(staticPathPattern)) {
            this.customizeResourceHandlerRegistration(registry.addHandler(new String[]
{staticPathPattern}).addResourceLocations(getResourceLocations(this.resourceProperties.getStatic
Locations())).setCachePeriod(this.getSeconds(cachePeriod)).setCacheControl(cacheControl));
        }
    }
}

```

1. 所有/webjars/**, 都去classpath:/META-INF/resources/webjars/下找资源

Webjars : 以jar包的方式引入静态资源

例如要引入jquery, 就在pom文件中添加jquery的webjar依赖

可以再webjar官网找到对应的maven依赖

2. /** 访问当前项目的任何资源

```

"classpath:/META-INF/resources/"
"classpath:/resources/"
"classpath:/static/"
"classpath:/public/"
"/" 根路径

```

默认在以上路径下寻找静态资源, 以上路径称为静态资源文件夹

3. WebMvcAutoConfiguration类部分源码 欢迎页映射

```

@Bean
public WelcomePageHandlerMapping welcomePageHandlerMapping(ApplicationContext
applicationContext) {
    return new WelcomePageHandlerMapping(new
TemplateAvailabilityProviders(applicationContext), applicationContext,
this.getWelcomePage(), this.mvcProperties.getStaticPathPattern());
}

```

静态资源文件夹下的index.html文件 被"/**"映射

如果没有欢迎页, 会出现404页面

4) WebMvcAutoConfiguration类部分源码 配置标签图标

```

@Configuration

```

```

@ConditionalOnProperty(
    value = {"spring.mvc.favicon.enabled"},
    matchIfMissing = true
)
public static class FaviconConfiguration implements ResourceLoaderAware {
    private final ResourceProperties resourceProperties;
    private ResourceLoader resourceLoader;

    public FaviconConfiguration(ResourceProperties resourceProperties) {
        this.resourceProperties = resourceProperties;
    }

    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader = resourceLoader;
    }

    @Bean
    public SimpleUrlHandlerMapping faviconHandlerMapping() {
        SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
        mapping.setOrder(-2147483647);
        mapping.setUrlMap(Collections.singletonMap("**/favicon.ico",
this.faviconRequestHandler()));
        return mapping;
    }

    @Bean
    public ResourceHttpRequestHandler faviconRequestHandler() {
        ResourceHttpRequestHandler requestHandler = new ResourceHttpRequestHandler();
        requestHandler.setLocations(this.resolveFaviconLocations());
        return requestHandler;
    }

    private List<Resource> resolveFaviconLocations() {
        String[] staticLocations =
WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter.getResourceLocations(this.resourcePropert
ies.getStaticLocations());
        List<Resource> locations = new ArrayList(staticLocations.length + 1);
        Stream var10000 = Arrays.stream(staticLocations);
        ResourceLoader var10001 = this.resourceLoader;
        this.resourceLoader.getClass();
        var10000.map(var10001::getResource).forEach(locations::add);
        locations.add(new ClassPathResource("/"));
        return Collections.unmodifiableList(locations);
    }
}

```

所有的****/favicon.ico** 都是在静态资源文件夹下寻找

所有的静态映射配置都在ResourceProperties类中

```
@ConfigurationProperties(  
    prefix = "spring.resources",  
    ignoreUnknownFields = false  
)  
public class ResourceProperties {  
}
```

所以可以再配置文件中修改spring.resources修改相关的配置

4.2 模板引擎

数据和页面通过模板引擎绑定起来，生成一个我们想要展示的页面

常用的有：JSP、Velocity、Freemarker、Thymeleaf等

SpringBoot默认不支持JSP，推荐使用Thymeleaf

语法更简单，功能更强大

4.2.1 引入Thymeleaf

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

4.2.2 Thymeleaf使用&语法

ThymeleafProperties类 部分代码：

```
public class ThymeleafProperties {  
    private static final Charset DEFAULT_ENCODING;  
    public static final String DEFAULT_PREFIX = "classpath:/templates/";  
    public static final String DEFAULT_SUFFIX = ".html";  
}
```

只要将html页面放到classpath:/templates/下面，thymeleaf就会自动渲染

1. 导入thymeleaf的名称空间

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

2. 使用thymeleaf

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
  <h1>成功！</h1>

  <!-- th:text 将div的文本内容设置为指定内容 -->
  <div th:text="${name}"></div>
</body>
</html>

```

4.2.3 Thymeleaf语法规则

4.2.3.1 th: + html任意属性

th: + 任意html属性：替换原来属性的值

如：

th:id="id" 替换元素id

th:class="class" 替换元素class

th:text="张三" 替换元素的文本

1. Fragment inclusion 片段包含

th:insert th:replace 相当于jsp的jsp:include

2. fragment iteration 遍历

th:each 相当于jsp的c:forEach

3. Conditional evaluation 条件判断

th:if th:unless th:switch th:case 相当于jsp的c:if

4. Local variable definition 声明变量

th:object th:with 相当于c:set

5. General attribute modification 任意属性修改 支持prepend、append添加属性

th:attr th:attrprepend th:attrappend

6. Specific attribute modification 修改指定属性

th:value th:src th:href th:id th:class 等等

7. Text(tag body modification) 修改标签体文本内容的

th:text--转义特殊字符，当普通字符串处理 th:utext--不转义特殊字符

8. Fragment specification 声明片段

th:fragment

9. Fragment removal 移除片段

th:remove

4.2.3.2 表达式

1. \${...} 获取变量值 就是OGNL表达式

1. 获取对象、集合的值，调用对象的方法
2. 使用内置的基本对象 \${#ctx.....} 通过#获取内置基本对象

1. ctx 当前的上下文对象
2. vars 当前上下文的变量值
3. locale 区域信息
4. request ---HttpServletRequest
5. response ---HttpServletResponse
6. Session ----HttpSession
7. ServletContext ----ServletContext

3. 使用内置的一些工具对象

如String number 等等一些工具类，详细参考thymeleaf文档

2. *{...} 变量选择表达式，与\${}功能上是一样的

特殊用法，可以配合th:object使用

```
<div th:object="${session.user}">
  <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="*{lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
</div>
```

3. #{...} 获取国际化内容

4. @{...}定义url链接

@{/order/process(execId=\${execId},execType='FAST')}

直接用/就代表当前路径下，所有参数直接用小括号括起来，多个参数用逗号分隔，参数可以用\${...}获取变量传参

5. ~{...} 片段引用表达式

以上是thymeleaf的五种表达式

在表达式中支持：

1.Literals 字面量：字符串、数字、布尔值、null、多个值用逗号分隔等

Text literals: 'one text', 'Another one!', ...

Number literals: 0, 34, 3.0, 12.3, ...

Boolean literals: true, false

Null literal: null

Literal tokens: one, sometext, main, ...

Text operations: 文本操作 +号连接符、用\${...}选择变量

String concatenation: +

Literal substitutions: |The name is \${name}|

Arithmetic operations: 数学运算

Binary operators: + , - , * , / , %

Minus sign (unary operator): -

Boolean operations: 布尔运算

Binary operators: and , or

Boolean negation (unary operator): ! , not

Comparisons and equality: 比较运算

Comparators: > , < , >= , <= (gt , lt , ge , le)

Equality operators: == , != (eq , ne)

Conditional operators: 条件运算 支持三元运算符

If-then: (if) ? (then)

If-then-else: (if) ? (then) : (else)

Default: (value) ?: (defaultvalue)

Special tokens: 特殊标志 用_表示没有操作

No-Operation: _

[\$ {...}] 相当于th:text

[\$ {...}] 相当于th:utext

4.3 SpringMVC 自动配置

4.3.1 自动配置

springBoot 自动配置好了springMVC使用的默认配置

以下是springBoot对springMVC的配置：

- Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans.

自动配置了ViewResolver（视图解析器）根据方法的返回值的到视图对象（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，

通过getCandidateViews方法将容器中所有的视图解析器都获取到，

然后通过getBestView选择最适合的一个视图解析器

我们要自定义一个视图解析器，只需要将我们的ViewResolver添加到容器中即可

- Support for serving static resources, including support for WebJars (covered [later in this document](#)).
- Static index.html support.
- Custom Favicon support (covered [later in this document](#)).

支持静态资源的配置，静态资源文件夹、webjars、首页及图标favicon.ico

- Automatic registration of Converter, GenericConverter, and Formatter beans.

自动注册了Converter：转换器，自动进行类型转换

Formatter：格式化器，如格式化日期等

- Support for `HttpMessageConverters` (covered [later in this document](#)).
`HttpMessageConverters` : SpringMVC用来转换http请求和响应的转换器
- Automatic registration of `MessageCodesResolver` (covered [later in this document](#)).
`MessageCodesResolver` : 定义错误代码生成规则
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered [later in this document](#)).

`ConfigurableWebBindingInitializer` : 初始化WebDataBinder

`WebDataBinder` : 用来将请求参数绑定到javabean的

4.3.2 扩展SpringMVC

If you want to keep Spring Boot MVC features and you want to add additional MVC configuration (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but without `@EnableWebMvc`. If you wish to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, you can declare a `WebMvcRegistrationsAdapter` instance to provide such components.

扩展方式：

编写一个配置类（用`@configuration`注解标注），是`WebMvcConfigurer`类型。不能标注`@EnableWebMvc`注解

例如：

```
//通过实现WebMvcConfigurer接口来实现springMVC扩展
//注意：WebMvcConfigurerAdapter抽象类已过时
// WebMvcConfigurationSupport类一旦存在，对springMVC的默认配置全部失效
//所以只需要实现WebMvcConfigurer接口即可，该接口中的方法都使用了JDK1.8的default关键字
//不需要实现所有方法，需要用哪个方法就实现哪个方法
@Configuration
public class MyMvcConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry){

        //super.addViewControllers(registry);

        //浏览器发送/demo请求，回到success页面
        registry.addViewController("/demo").setViewName("success");
    }
}
```

SpringBoot既保留了默认配置，也能使用我们的扩展配置

原理：

1. `WebMvcAutoConfiguration` SpringMVC的自动配置类
2. 在做其他自动配置的时候会导入 `EnableWebMvcConfiguration`
3. `EnableWebMvcConfiguration` 的父类`DelegatingWebMvcConfiguration`


```

public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {
    private final WebMvcConfigurerComposite configurers = new WebMvcConfigurerComposite();

    public DelegatingWebMvcConfiguration() {
    }

    @Autowired(
        required = false
    )
    //从容器中获取所有的WebMvcConfigurer
    public void setConfigurers(List<WebMvcConfigurer> configurers) {
        if (!CollectionUtils.isEmpty(configurers)) {
            this.configurers.addWebMvcConfigurers(configurers);
        }
    }
}

```

通过一个WebMvcConfigurerComposite configurers 调用他的方法

```

public void addInterceptors(InterceptorRegistry registry) {
    Iterator var2 = this.delegates.iterator();

    while(var2.hasNext()) {
        WebMvcConfigurer delegate = (WebMvcConfigurer)var2.next();
        delegate.addInterceptors(registry);
    }
}

```

在WebMvcConfigurerComposite 中是遍历所有的WebMvcConfigurer，每一个都调用一次该方法，把扩展组件全部加载进来

容器中所有的WebMvcConfigurer都会起作用，包括我们自定义的WebMvcConfigurer

4.3.3 全面接管SpringMVC

If you want to take complete control of Spring MVC, you can add your own @Configuration annotated with @EnableWebMvc.

在我们自己的配置类上加上@EnableWebMvc注解，那么所有的SpringMvc自动配置都不会生效了，所有的配置都需要我们自己设置

4.4 如何修改SpringBoot的默认配置

1. SpringBoot在自动配置很多组件的时候，都会先看容器中是否有用户自定义的组件，如果有，就用用户自定义的。如果没有在自动配置。有些组件可以有多个，如ViewResolver，SpringBoot会将用户自定义的和默认配置的组合起来。用户通过@Bean、@component注解给容器添加组件。
2. 在SpringBoot会有非常多的xxxxConfigurer，帮助我们进行扩展配置，如WebMvcConfigurer
3. 还会有非常多的xxxxCustomizer，帮助我们定制配置，如WebServerFactoryCustomizer用来定制嵌入式servlet容器的相关配置

4.5 RestfulCRUD

4.5.1 设置默认首页

```
//自己向容器添加一个WebMvcConfigurer，将“/”请求映射到index.html上
@Bean
public WebMvcConfigurer myWebMvcConfigurer(){
    WebMvcConfigurer config = new WebMvcConfigurer() {

        @Override
        public void addViewControllers(ViewControllerRegistry registry){
            registry.addViewController("/").setViewName("index");
            registry.addViewController("/index.html").setViewName("index");
        }
    };
    return config;
}
```

4.5.2 添加前端框架

如果要添加bootstrap、jquery等资源，通过webjar的方式用maven依赖导入

然后通过默认的webjar映射路径/webjar/**路径映射到对应的资源

4.5.3 设置项目名

修改application.properties文件

```
#设置端口号
server.port=8080
#设置项目名
server.servlet.context-path=/demo
```

4.5.4 国际化

原来的步骤：

1. 编写国际化配置文件
2. 使用ResourceBundleMessageSource来管理国际化资源文件
3. 在JSP页面使用fmt:message去除国际化内容

springBoot使用：

1. 编写国际化配置文件，抽取页面需要国际化的内容
 1. 创建index.html页面的国际化配置文件index.properties
 2. 中文和英文的index_zh_CN.properties index_en_US.properties
 3. 这时idea会自动识别在进行国际化配置，可进入ResourceBundle视图进行国际化配置
2. SpringBoot自动配置好了ResourceBundleMessageSource

在MessageSourceAutoConfiguration类中的部分源码

```
@Bean
@ConfigurationProperties(
```

```

        prefix = "spring.messages"
    )
    public MessageSourceProperties messageSourceProperties() {
        return new MessageSourceProperties();
    }

    @Bean
    public MessageSource messageSource() {
        MessageSourceProperties properties = this.messageSourceProperties();
        ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
        if (StringUtils.hasText(properties.getBasename())) {

            messageSource.setBasenames(StringUtils.commaDelimitedListToStringArray(StringUtils.trimAllWhites
            pace(properties.getBasename())));
        }

        if (properties.getEncoding() != null) {
            messageSource.setDefaultEncoding(properties.getEncoding().name());
        }

        messageSource.setFallbackToSystemLocale(properties.isFallbackToSystemLocale());
        Duration cacheDuration = properties.getCacheDuration();
        if (cacheDuration != null) {
            messageSource.setCacheMillis(cacheDuration.toMillis());
        }

        messageSource.setAlwaysUseMessageFormat(properties.isAlwaysUseMessageFormat());
        messageSource.setUseCodeAsDefaultMessage(properties.isUseCodeAsDefaultMessage());
        return messageSource;
    }

```

通过setBasenames设置国际化资源文件的基础名（去掉语言国家代码后的名字）

在MessageSourceProperties有默认的基础名：

private String basename = "\"messages\"";

也就是说如果我们将国际化配置在类路径下的messages.properties文件中，将不需要做任何配置直接使用

如果基础名不是messages，那么就需要在application.properties文件中进行配置

```

#配置资源文件的基础名 在类路径下的i18n文件夹下的index.properties
spring.messages.basename=i18n.index

```

3. 去页面获取国际化的值

在thymeleaf中有#{...}表达式专门用来处理国际化的

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <p>[[#{login.tip}]]</p>
</body>
</html>

```

这样就会根据浏览器语言信息显示不同的文本内容

4. 实现按钮切换语言

原理：

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

WebMvcAutoConfiguration中关于LocaleResolver的自动配置源码

```

@Bean
@ConditionalOnMissingBean
@ConditionalOnProperty(
    prefix = "spring.mvc",
    name = {"locale"}
)
public LocaleResolver localeResolver() {
    if (this.mvcProperties.getLocaleResolver() ==
org.springframework.boot.autoconfigure.web.servlet.WebMvcProperties.LocaleResolver.FIXED) {
        return new FixedLocaleResolver(this.mvcProperties.getLocale());
    } else {
        AcceptHeaderLocaleResolver localeResolver = new AcceptHeaderLocaleResolver();
        localeResolver.setDefaultLocale(this.mvcProperties.getLocale());
        return localeResolver;
    }
}

```

如果设置成了FIXED那么将会固定区域化信息，否则将会从请求头中获取

可以自己写一个区域信息解析器不从请求头中获取，而从请求连接中获取

```

public class MyLocaleResolver implements LocaleResolver {
    @Override
    public Locale resolveLocale(HttpServletRequest httpServletRequest) {
        String l = httpServletRequest.getParameter("l");
        Locale locale = Locale.getDefault();
        if (!StringUtils.isEmpty(l)) {
            String[] split = l.split("_");
            locale = new Locale(split[0], split[1]);
        }
        return locale;
    }
}

```

```

@Override
public void setLocale(HttpServletRequest httpRequest, HttpServletResponse
httpServletResponse, Locale locale) {
    }
}

```

将自己的LocaleResolver 通过@Bean注解添加到容器中取代自动配置的LocaleResolver 即可

4.5.5 登录

开发期间模板引擎页面修改后，要实时生效：

1. 禁用thymeleaf模板引擎缓存

```

#禁用thymeleaf缓存
spring.thymeleaf.cache=false

```

2. 修改完成后重新编译，就能够刷新页面了

4.5.6 用拦截器进行登录检查

1. 通过实现HandlerInterceptor接口定义一个登录拦截器

```

/**
 * 通过实现HandlerInterceptor来定义一个拦截器，进行登录检查
 */
public class LoginHandlerInterceptor implements HandlerInterceptor {

    //在目标方法前执行
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {

        Object user = request.getSession().getAttribute(request.getParameter("sessionId"));
        if(user == null){
            //未登录
            return false;
        }else{
            //已登录
            return true;
        }
    }
}

```

2. 在自己的WebMvcConfigurer中重写addInterceptors方法注册拦截器

```
//注册拦截器
@Override
public void addInterceptors(InterceptorRegistry registry) {
    //addPathPatterns添加要拦截的映射路径 "/"**"表示拦截所有请求
    //excludePathPatterns用来排除部分请求
    //静态资源不用排除，因为springBoot已经做好了资源映射
    registry.addInterceptor(new
LoginHandlerInterceptor()).addPathPatterns("/").excludePathPatterns("/index.html");
}
```

4.5.7 Restful与普通请求的区别

Restful请求通过http请求方法来区分增删改查，而普通请求通过url区分增删改查

	Restful	普通请求
查找	emp/{id}---GET	getEmp?id=xxx
修改	emp---PUT	updateEmp?id=xxx&xxx=xxx
添加	emp---POST	addEmp?id=xxx&xxx=xxx
删除	emp/{id}---DELETE	deleteEmp?id=xxx

4.6 错误处理机制

4.6.1 SpringBoot默认错误处理机制

默认效果：

1. 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.

Mon Feb 26 17:33:50 GMT+08:00 2018

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

No message available

2. 如果不是浏览器而是其他客户端，返回一个默认的json数据

```
{
  "timestamp": 1519637719324,
  "status": 404,
  "error": "Not Found",
  "message": "No message available",
  "path": "/crud/aaa"
}
```

原理：可以参照ErrorMvcAutoConfiguration；错误处理的自动配置

在这个自动配置类中，主要给容器添加了以下组件：

1. DefaultErrorAttributes：

默认的共享的错误响应内容

```
public Map<String, Object> getErrorAttributes(WebRequest webRequest, boolean includeStackTrace)
{
    Map<String, Object> errorAttributes = new LinkedHashMap();
    errorAttributes.put("timestamp", new Date());
    this.addStatus(errorAttributes, webRequest);
    this.addErrorDetails(errorAttributes, webRequest, includeStackTrace);
    this.addPath(errorAttributes, webRequest);
    return errorAttributes;
}

//timestamp 时间戳
//status 状态码
//error 错误信息
//exception 异常对象
//message 异常消息
//errors JSR303数据校验的错误都在这里
```

2. BasicErrorController：//就是一个controller，如果配置文件没配置默认处理/error请求

```
@Controller
@RequestMapping("${server.error.path:${error.path:/error}}")
public class BasicErrorController extends AbstractErrorController {

    //1.产生html页面数据
    @RequestMapping(
        produces = {"text/html"}
    )
    public ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse response) {
        HttpStatus status = this.getStatus(request);
        Map<String, Object> model = Collections.unmodifiableMap(this.getErrorAttributes(request,
            this.isIncludeStackTrace(request, MediaType.TEXT_HTML)));
        response.setStatus(status.value());
        //去哪个页面作为错误页面，是通过resolveErrorView获得的modelAndView，包含页面地址和页面内容
        ModelAndView modelAndView = this.resolveErrorView(request, response, status, model);
        return modelAndView != null ? modelAndView : new ModelAndView("error", model);
    }

    //产生json数据
    @RequestMapping
    @ResponseBody
    public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
        Map<String, Object> body = this.getErrorAttributes(request,
            this.isIncludeStackTrace(request, MediaType.ALL));
        HttpStatus status = this.getStatus(request);
        return new ResponseEntity(body, status);
    }
}
```

两个方法返回错误数据，是根据请求头的accept来选择的

浏览器请求优先接收text/html数据：

✓

▼ 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: en-US,zh-CN;q=0.8,zh;q=0.6,en;q=0.4

Cache-Control: no-cache

Connection: keep-alive

而其他客户端没有指明优先接收什么数据

👉 Request Headers:

cache-control: "no-cache"

postman-token: "b34bebc4-07a5-4c20-8f3f-952f3daec38f"

user-agent: "PostmanRuntime/7.1.1"

accept: "*/"

host: "localhost:8080"

cookie: "JSESSIONID=DDB37833549894367D63323D1F21957C; JSESSIONID=1BBFE9718FD60"

accept-encoding: "gzip, deflate"

3. ErrorPageCustomizer :

```
@Value("${error.path:/error}")
private String path = "/error";
//系统出现错误以后来到error请求进行处理；（相当于web.xml注册的错误页面规则）
//如果配置文件没有配置默认为/error
```

4. DefaultErrorViewResolver：容器默认的错误视图解析器

```
public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status, Map<String,
Object> model) {
    ModelAndView modelAndView = this.resolve(String.valueOf(status), model);
    if (modelAndView == null && SERIES_VIEWS.containsKey(status.series())) {
        modelAndView = this.resolve((String)SERIES_VIEWS.get(status.series()), model);
    }

    return modelAndView;
}

private ModelAndView resolve(String viewName, Map<String, Object> model) {
    //默认会去找"error/" + 状态码 页面
    String errorViewName = "error/" + viewName;
    //如果有模板引擎，就用模板引擎来解析这个映射路径
    TemplateAvailabilityProvider provider =
this.templateAvailabilityProviders.getProvider(errorViewName, this.applicationContext);
    //如果没有模板引擎，就在静态资源文件夹下寻找对应的资源
    return provider != null ? new ModelAndView(errorViewName, model) :
this.resolveResource(errorViewName, model);
}
```


步骤：

一旦系统出现4xx或者5xx之类的错误；ErrorPageCustomizer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；

BasicErrorController根据accept请求头判断是返回html还是json数据

1. Html

通过resolveErrorView方法获取ModelAndView 对象，包含响应的页面和页面数据

resolveErrorView代码：

```
protected ModelAndView resolveErrorView(HttpServletRequest request, HttpServletResponse response, HttpStatus status, Map<String, Object> model) {  
    //通过迭代所有的ErrorViewResolver得到modelAndView，如果没有就返回null  
    //容器默认注册了一个DefaultErrorViewResolver，如果没有自定义的ErrorViewResolver  
    //那么就是用这个默认的错误视图解析器解析得到具体返回哪个错误页面  
    Iterator var5 = this.errorViewResolvers.iterator();  
    ModelAndView modelAndView;  
    do {  
        if (!var5.hasNext()) {  
            return null;  
        }  
        ErrorViewResolver resolver = (ErrorViewResolver)var5.next();  
        modelAndView = resolver.resolveErrorView(request, status, model);  
    } while(modelAndView == null);  
    return modelAndView;  
}
```

4.6.2 定制错误响应

4.6.2.1 定制错误页面

- 1、有模板引擎的情况下；error/状态码；【将错误页面命名为 错误状态码.html 放在模板引擎文件夹里面的error文件夹下】，发生此状态码的错误就会来到 对应的页面；

我们可以使用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（优先寻找精确的状态码.html）；

- 2、没有模板引擎的情况（模板引擎找不到这个错误页面），静态资源文件夹下找；
3. 以上都没有，就来到SpringBoot默认的错误页面

4.6.2.2 定制错误数据

1. 处理自定义异常&返回自定义json数据

```

@ControllerAdvice
public class MyExceptionHandler {
    @ResponseBody
    @ExceptionHandler(UserNotExistException.class)
    public Map<String, Object> handleException(Exception e){
        Map<String, Object> map = new HashMap<>();
        map.put("code", "user.notexist");
        map.put("message", e.getMessage());
        return map;
    }
}
//没有自适应效果，浏览器和客户端都返回的是json数据

```

2. 转发到/error进行自适应响应效果处理

```

@ExceptionHandler(UserNotExistException.class)
public String handleException(Exception e, HttpServletRequest request){
    Map<String, Object> map = new HashMap<>();
    //传入我们自己的错误状态码 4xx 5xx，否则就不会进入定制错误页面的解析流程
    /**
     * Integer statusCode = (Integer) request
     * .getAttribute("javax.servlet.error.status_code");
     */
    request.setAttribute("javax.servlet.error.status_code", 500);
    map.put("code", "user.notexist");
    map.put("message", e.getMessage());
    //转发到/error
    return "forward:/error";
}
//有自适应效果但是不能将我们自定义的数据携带出去

```

3. 将我们的定制数据携带出去；

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）；

所以有两种解决办法：

1. 完全来编写一个ErrorController的实现类【或者是编写AbstractErrorController的子类】，放在容器中；
2. 自定义ErrorAttributes

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；
容器中DefaultErrorAttributes.getErrorAttributes()；默认进行数据处理的；

```
//给容器中加入我们自己定义的ErrorAttributes
@Component
public class MyErrorAttributes extends DefaultErrorAttributes {
    @Override
    public Map<String, Object> getErrorAttributes(RequestAttributes requestAttributes,
boolean includeStackTrace) {
        Map<String, Object> map = super.getErrorAttributes(requestAttributes,
includeStackTrace);
        map.put("company", "atguigu");
        return map;
    }
}
```

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

4.5 配置嵌入式Servlet容器

SpringBoot默认使用的是tomcat作为嵌入式的servlet容器

4.5.1 定制和修改servlet容器的相关配置

4.5.1.1 application.properties

在application.properties中修改和server有关的配置

如：

```
server.port=8081
server.context-path=/crud
server.tomcat.uri-encoding=UTF-8
#通用的Servlet容器设置
server.xxx
#Tomcat的设置
server.tomcat.xxx
```

4.5.1.2 EmbeddedServletContainerCustomizer

在springBoot 1.x版本中可以编写一个EmbeddedServletContainerCustomizer：嵌入式的Servlet容器的定制器，来修改servlet容器的相关配置

```
@Bean //一定要将这个定制器加入到容器中
public EmbeddedServletContainerCustomizer embeddedServletContainerCustomizer(){
    return new EmbeddedServletContainerCustomizer() {
        //定制嵌入式的Servlet容器相关的规则
        @Override
        public void customize(ConfigurableEmbeddedServletContainer container) {
            container.setPort(8083);
        }
    };
}
```

4.5.1.3 WebServerFactoryCustomizer

在springBoot 2.x版本中使用WebServerFactoryCustomizer接口来定制tomcat配置

```
@Bean
public WebServerFactoryCustomizer webServerFactoryCustomizer(){
    return new WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>(){
        @Override
        public void customize(ConfigurableServletWebServerFactory server) {
            server.setPort(5000);
        }
    };
}
```

4.5.2 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容器来启动SpringBoot的web应用，没有web.xml文件。

4.5.2.1 springBoot的Registration

springBoot提供了ServletRegistrationBean、FilterRegistrationBean、ServletListenerRegistrationBean 分别用于注册三大组件

注册servlet ServletRegistrationBean

```
@Bean
public ServletRegistrationBean myServlet(){
    ServletRegistrationBean servletRegistrationBean = new ServletRegistrationBean(new
    MyServlet(), "/myServlet");
    return servletRegistrationBean;
}
```

注册Listener ServletListenerRegistrationBean

```
@Bean
public ServletListenerRegistrationBean myListener(){
    ServletListenerRegistrationBean<MyListener> registrationBean = new
    ServletListenerRegistrationBean<>(new MyListener());
    return registrationBean;
}
```

注册filter FilterRegistrationBean

```
@Bean
public FilterRegistrationBean myFilter(){
    FilterRegistrationBean registrationBean = new FilterRegistrationBean();
    registrationBean.setFilter(new MyFilter());
    registrationBean.setUrlPatterns(Arrays.asList("/hello", "/myServlet"));
    return registrationBean;
}
```

4.5.2.2 使用servlet3.0的注解

@WebServlet、@WebListener、@WebFilter

Servlet

```
@WebServlet(name = "IndexServlet",urlPatterns = "/hello")
public class IndexServlet extends HttpServlet {
    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        resp.getWriter().print("hello word");
        resp.getWriter().flush();
        resp.getWriter().close();
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        this.doGet(req, resp);
    }
}
```

Listener

```
@WebListener
public class IndexListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent) {
        System.out.println("IndexListener contextInitialized");
    }

    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {

    }
}
```

Filter

```
@WebFilter(urlPatterns = "/*", filterName = "indexFilter")
public class IndexFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("init IndexFilter");
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
```

```

    FilterChain filterChain) throws IOException, ServletException {
        System.out.println("doFilter IndexFilter");
        filterChain.doFilter(servletRequest, servletResponse);

    }

    @Override
    public void destroy() {

    }

}

```

需要配置一个核心的注解@ServletComponentScan,具体配置项如下，可以配置扫描的路径，将该注解添加到主程序即可

```

@SpringBootApplication
@ServletComponentScan("com.example.springboot002.servlet")
public class SpringBoot002Application {

}

```

4.5.2.3 自动配置springMVC的DispatcherServlet

SpringBoot帮我们自动配置SpringMVC时，自动注册了DispatcherServlet

DispatcherServletAutoConfiguration中注册DispatcherServlet部分源码：

```

@Bean(
    name = {"dispatcherServletRegistration"}
)
@ConditionalOnBean(
    value = {DispatcherServlet.class},
    name = {"dispatcherServlet"}
)
public ServletRegistrationBean<DispatcherServlet>
dispatcherServletRegistration(DispatcherServlet dispatcherServlet) {
    ServletRegistrationBean<DispatcherServlet> registration = new
    ServletRegistrationBean(dispatcherServlet, new String[]
    {this.serverProperties.getServlet().getServletMapping()});
    //在 getServletMapping()方法中可以看到默认的拦截路径为"/"
    //拦截所有请求，包括静态资源，但是不包括JSP    "/"会拦截jsp
    //在serverProperties中可以看到可以通过server.Servlet.path修改拦截路径
    registration.setName("dispatcherServlet");
    registration.setLoadOnStartup(this.webMvcProperties.getServlet().getLoadOnStartup());
    if (this.multipartConfig != null) {
        registration.setMultipartConfig(this.multipartConfig);
    }

    return registration;
}

```

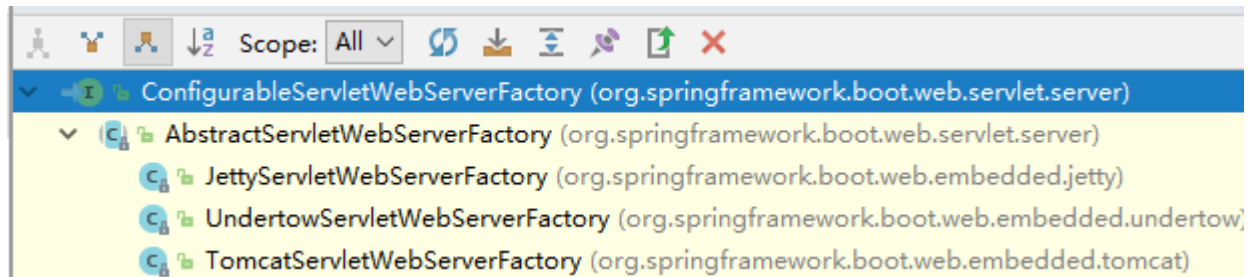
4.5.3 使用其他嵌入式Servlet容器

springBoot除了tomcat之外，还支持Jetty和Undertow两种servlet容器

Undertow不支持jsp,是一个高性能的非阻塞的servlet容器，并发性能很好

Jetty更适合开发长连接应用

从SpringBoot的容器配置定制器的继承树就可以看出springBoot支持的嵌入式servlet容器



使用其他容器的步骤

1. 在pom.xml的web启动器模块去掉默认tomcat的依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <artifactId>spring-boot-starter-tomcat</artifactId>
      <groupId>org.springframework.boot</groupId>
    </exclusion>
  </exclusions>
</dependency>
```

2. 添加其他容器的依赖

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

4.5.4 嵌入式Servlet容器的自动配置原理

嵌入式Servlet容器的自动配置类：

```
@Configuration
@EnableConfigurationProperties(ServerProperties.class)
public class EmbeddedWebServerFactoryCustomizerAutoConfiguration {

    //ConditionalOnClass注解，判断项目中是否包含Tomcat和UpgradeProtocol两个类
    //也就是说如果引入了Tomcat依赖，那么就会加载TomcatWebServerFactoryCustomizer组件
    @ConditionalOnClass({ Tomcat.class, UpgradeProtocol.class })
    public static class TomcatWebServerFactoryCustomizerConfiguration {

        @Bean
        public TomcatWebServerFactoryCustomizer tomcatWebServerFactoryCustomizer(
```

```

        Environment environment, ServerProperties serverProperties) {
            return new TomcatWebServerFactoryCustomizer(environment, serverProperties);
        }
    }

    //判断是否有Jetty依赖
    @Configuration
    @ConditionalOnClass({ Server.class, Loader.class, WebApplicationContext.class })
    public static class JettyWebServerFactoryCustomizerConfiguration {

        @Bean
        public JettyWebServerFactoryCustomizer jettyWebServerFactoryCustomizer(
            Environment environment, ServerProperties serverProperties) {
            return new JettyWebServerFactoryCustomizer(environment, serverProperties);
        }

    }

    //判断是否有Undertow依赖
    @Configuration
    @ConditionalOnClass({ Undertow.class, SslClientAuthMode.class })
    public static class UndertowWebServerFactoryCustomizerConfiguration {

        @Bean
        public UndertowWebServerFactoryCustomizer undertowWebServerFactoryCustomizer(
            Environment environment, ServerProperties serverProperties) {
            return new UndertowWebServerFactoryCustomizer(environment, serverProperties);
        }

    }

}

```

TomcatWebServerFactoryCustomizer中重写了WebServerFactoryCustomizer的customize方法：

```

@Override
public void customize(ConfigurableTomcatWebServerFactory factory) {
    ServerProperties properties = this.serverProperties;

    //获取一个tomcat实例
    ServerProperties.Tomcat tomcatProperties = properties.getTomcat();

    //下面是进行相关的配置
    PropertyMapper propertyMapper = PropertyMapper.get();
    propertyMapper.from(tomcatProperties::getBasedir).whenNonNull()
        .to(factory::setBaseDirectory);
    propertyMapper.from(tomcatProperties::getBackgroundProcessorDelay).whenNonNull()
        .as(Duration::getSeconds).as(Long::intValue)
        .to(factory::setBackgroundProcessorDelay);
    customizeRemoteIpValve(factory);

    propertyMapper.from(tomcatProperties::getMaxThreads).when(this::isPositive)

```



```

        .to((maxThreads) -> customizeMaxThreads(factory,
            tomcatProperties.getMaxThreads()));
    propertyMapper.from(tomcatProperties::getMinSpareThreads).when(this::isPositive)
        .to((minSpareThreads) -> customizeMinThreads(factory, minSpareThreads));
    propertyMapper.from(() -> determineMaxHttpHeaderSize()).when(this::isPositive)
        .to((maxHttpHeaderSize) -> customizeMaxHttpHeaderSize(factory,
            maxHttpHeaderSize));
    propertyMapper.from(tomcatProperties::getMaxHttpPostSize)
        .when((maxHttpPostSize) -> maxHttpPostSize != 0)
        .to((maxHttpPostSize) -> customizeMaxHttpPostSize(factory,
            maxHttpPostSize));
    propertyMapper.from(tomcatProperties::getAccesslog)
        .when(ServerProperties.Tomcat.Accesslog::isEnabled)
        .to((enabled) -> customizeAccessLog(factory));
    propertyMapper.from(tomcatProperties::getUriEncoding).whenNonNull()
        .to(factory::setUriEncoding);
    propertyMapper.from(properties::getConnectionTimeout).whenNonNull()
        .to((connectionTimeout) -> customizeConnectionTimeout(factory,
            connectionTimeout));
    propertyMapper.from(tomcatProperties::getMaxConnections).when(this::isPositive)
        .to((maxConnections) -> customizeMaxConnections(factory, maxConnections));
    propertyMapper.from(tomcatProperties::getAcceptCount).when(this::isPositive)
        .to((acceptCount) -> customizeAcceptCount(factory, acceptCount));
    customizeStaticResources(factory);
    customizeErrorReportValve(properties.getError(), factory);
}

```

4.6 使用外置的Servlet容器

4.6.1 嵌入式Servlet容器的优缺点

嵌入式Servlet容器：

优点：简单、便携

缺点：不支持JSP、优化定制比较复杂（使用定制器---ServerProperties或自定义WebServerFactoryCustomizer，也可以编写自己的Servlet容器的创建工厂）

4.6.2 使用外置Servlet容器的步骤

外置的Servlet容器：在外面安装一个Servlet容器，将应用打成war包在Servlet容器中运行

步骤：

1. 必须创建一个war项目（创建好项目的目录结构）
2. 将嵌入式的Tomcat指定为provided

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
  <scope>provided</scope>
</dependency>

```

3.必须编写一个SpringBootServletInitializer的子类，并重写configure方法

```
public class ServletInitializer extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        //传入SpringBoot应用的主程序
        return application.sources(SpringBoot003Application.class);
    }

}
```

4.用服务器启动应用即可

注：建议使用Tomcat8.0以上版本，否则会出现el-api版本问题导致启动失败

在pom.xml文件中添加在Tomcat8.0能正常启动

```
<dependency>
    <groupId>javax.el</groupId>
    <artifactId>javax.el-api</artifactId>
    <version>3.0.0</version>
    <scope>provided</scope>
</dependency>
```

4.6.3 外置Servlet容器启动原理

4.6.3.1 jar包启动和war包启动的区别

Jar包：直接执行主程序的main方法，启动ioc容器，会创建嵌入式Servlet容器

war包：启动服务器，**服务器启动SpringBoot应用【SpringBootServletInitializer】**，然后启动ioc容器

4.6.3.2 war包启动原理及流程

在Servlet3.0规范中定义的规则

- 1.服务器启动（Web应用启动）会创建当前web应用里面每一个jar包里面ServletContainerInitializer的实例
- 2.ServletContainerInitializer的实现必须放在jar包的META-INF/services路径下，必须有一个名为javax.servlet.ServletContainerInitializer的文件，内容就是ServletContainerInitializer实现类的全类名
- 3.还可以使用@HandlesTypes，在应用启动的时候加载我们感兴趣的类

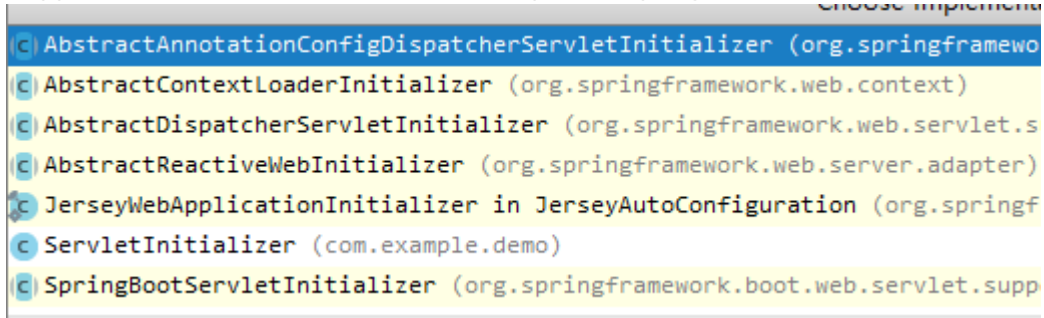
基于Servlet3.0规范定义的规则，启动流程：

- 1.启动Tomcat
 - 2.在spring-web.jar里面的META-INF/services有一个javax.servlet.ServletContainerInitializer文件
- 内容是：

```
org.springframework.web.SpringServletContainerInitializer
```

3.SpringServletContainerInitializer将所有@HandlesTypes(WebApplicationInitializer.class)注解标注的WebApplicationInitializer类传入onStartup方法，onStartup为所有不是接口和抽象类的类创建实例

4.每一个WebApplicationInitializer都调用自己的onStartup方法，SpringBoot项目中有以下实现类：



The screenshot shows a list of classes implementing the WebApplicationInitializer interface. The classes are: AbstractAnnotationConfigDispatcherServletInitializer (org.springframework), AbstractContextLoaderInitializer (org.springframework.web.context), AbstractDispatcherServletInitializer (org.springframework.web.servlet.s), AbstractReactiveWebInitializer (org.springframework.web.server.adapter), JerseyWebApplicationInitializer in JerseyAutoConfiguration (org.springf), ServletInitializer (com.example.demo), and SpringBootServletInitializer (org.springframework.boot.web.servlet.sup).

5.也就是我们自己编写的SpringBootServletInitializer的子类也是一个WebApplicationInitializer类，会在启动的时候实例化，并执行onStartup方法

6.SpringBootServletInitializer实例执行onStartup方法的时候，会调用createRootApplicationContext方法创建容器

```
protected WebApplicationContext createRootApplicationContext(  
    ServletContext servletContext) {  
    //1.创建SpringApplicationBuilder构建器  
    SpringApplicationBuilder builder = createSpringApplicationBuilder();  
    builder.main(getClass());  
    ApplicationContext parent = getExistingRootWebApplicationContext(servletContext);  
    if (parent != null) {  
        this.logger.info("Root context already created (using as parent).");  
        servletContext.setAttribute(  
            WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, null);  
        builder.initializers(new ParentContextApplicationContextInitializer(parent));  
    }  
    builder.initializers(  
        new ServletContextApplicationContextInitializer(servletContext));  
    builder.contextClass(AnnotationConfigServletWebServerApplicationContext.class);  
  
    //调用configure方法，子类重写了configure方法，将我们的SpringBoot主程序类传入进来  
    builder = configure(builder);  
    builder.listeners(new WebEnvironmentPropertySourceInitializer(servletContext));  
  
    //使用builder创建一个spring应用  
    SpringApplication application = builder.build();  
    if (application.getAllSources().isEmpty() && AnnotationUtils.  
        findAnnotation(getClass(), Configuration.class) != null) {  
        application.addPrimarySources(Collections.singleton(getClass()));  
    }  
    Assert.state(!application.getAllSources().isEmpty(),  
        "No SpringApplication sources have been defined. Either override the "  
        + "configure method or add an @Configuration annotation");  
    // Ensure error pages are registered  
    if (this.registerErrorPageFilter) {  
        application.addPrimarySources(  
            Collections.singleton(ErrorPageFilterConfiguration.class));  
    }  
}
```

```
    }  
  
    //启动应用  
    return run(application);  
}
```

7.Spring的应用就启动并且创建IOC容器

五、Docker

5.1 简介

Docker是一个开源的应用容器引擎，类似于虚拟机，但是不是虚拟机，而是一个轻量级容器技术，实现了虚拟机技术的资源隔离，且性能远远高于虚拟机。

基于Go语言并遵从Apache2.0协议开源，Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。

容器是完全使用沙箱机制，相互之间不会有任何接口,更重要的是容器性能开销极低。

Docker支持将软件编译成一个镜像；然后在镜像中各种软件做好配置，将镜像发布出去，其他使用者可以直接使用这个镜像。运行中的这个镜像称为容器，容器启动是非常快速的。类似windows里面的ghost操作系统，安装好后什么都有了；

5.2 Docker核心概念

docker主机(Host)：一个物理或者虚拟的机器用于执行Docker 守护进程和容器。也就是说一个安装了Docker程序的机器（Docker是直接安装在操作系统上的，操作系统可以是Linux、Windows、Mac等等）

docker客户端(Client)：客户端通过命令行或者其他工具使用DockerAPI与 Docker 的守护进程通信。

DockerAPI地址：https://docs.docker.com/reference/api/docker_remote_api

docker仓库(Registry)：Docker 仓库用来保存镜像，可以理解为代码控制中的代码仓库。

公共docker仓库DockerHub地址：<https://hub.docker.com>

docker镜像(Images)：打包好的软件镜像，直接通过docker命令运行docker镜像可以运行一个对应的docker容器

docker容器(Container)：容器是独立运行的一个或一组应用。就是docker镜像的运行实例。

使用Docker的步骤：

- 1.安装Docker
- 2.去Docker仓库找到这个软件对应的镜像
- 3.使用Docker运行这个镜像，就会生成一个Docker容器

4.对容器的启动停止就是对软件的启动停止

5.3 Linux下Docker的安装

```
1. 检查内核版本是否符合安装Docker的要求
uname -r
2. 安装Docker
yum install docker
3. 启动Docker
在centOs7中：systemctl start docker
4. 查看docker版本号：
docker -v 这个命令可以执行说明docker启动完成
5. 开机启动docker
systemctl enable docker
6. 停止docker
systemctl stop docker
```

5.4 常用操作

1.docker search 关键字 从镜像仓库搜索镜像

如docker search mysql、docker search redis等等

2.docker pull 镜像名 从镜像仓库拉取镜像 如docker pull mysql

默认下载latest版本，如果要选择需要添加tag标签 如 docker pull mysql:5.5 则会下载5.5版本的mysql镜像文件

3.docker images 查看当前所有的镜像列表

4.docker rmi imageId 删除指定ID的镜像 镜像ID通过docker images命令查看

5.docker ps 查看当前运行的docker容器

注：各个镜像的安装及其他操作可参考docker官方文档操作

六、数据访问

6.1 SpringBoot对数据访问的支持

对于数据访问，无论是关系型数据库还是NoSql数据库，SpringBoot都是默认采用Spring-Data来进行统一处理的。在SpringBoot中含有大量的自动配置，引入各种xxxTemplate、xxxRepository来简化对数据访问的操作。

SpringBoot提供大量的Starter来支持数据访问：

<code>spring-boot-starter-data-cassandra</code>	Starter for using Cassandra distributed database and Spring Data Cassandra	Pom
<code>spring-boot-starter-data-cassandra-reactive</code>	Starter for using Cassandra distributed database and Spring Data Cassandra Reactive	Pom
<code>spring-boot-starter-data-couchbase</code>	Starter for using Couchbase document-oriented database and Spring Data Couchbase	Pom
<code>spring-boot-starter-data-couchbase-reactive</code>	Starter for using Couchbase document-oriented database and Spring Data Couchbase Reactive	Pom
<code>spring-boot-starter-data-elasticsearch</code>	Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch	Pom
<code>spring-boot-starter-data-jpa</code>	Starter for using Spring Data JPA with Hibernate	Pom
<code>spring-boot-starter-data-ldap</code>	Starter for using Spring Data LDAP	Pom
<code>spring-boot-starter-data-mongodb</code>	Starter for using MongoDB document-oriented database and Spring Data MongoDB	Pom
<code>spring-boot-starter-data-mongodb-reactive</code>	Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive	Pom
<code>spring-boot-starter-data-neo4j</code>	Starter for using Neo4j graph database and Spring Data Neo4j	Pom
<code>spring-boot-starter-data-redis</code>	Starter for using Redis key-value data store with Spring Data Redis and the Lettuce client	Pom
<code>spring-boot-starter-data-redis-reactive</code>	Starter for using Redis key-value data store with Spring Data Redis reactive and the Lettuce client	Pom
<code>spring-boot-starter-data-rest</code>	Starter for exposing Spring Data repositories over REST using Spring Data REST	Pom
<code>spring-boot-starter-data-solr</code>	Starter for using the Apache Solr search platform with Spring Data Solr	Pom

6.2 JDBC

6.2.1 使用JDBC

要使用jdbc要在pom文件中引入jdbc的starter以及数据库驱动：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

然后在配置文件中进行相应的配置：

```
spring:
  datasource:
    username: rootss
    password: 123456
    url: jdbc:mysql://localhost:3306/db
    driver-class-name: com.mysql.jdbc.Driver
```

在配置完成后就可以使用jdbc对数据库进行访问了。

默认使用的是：`org.apache.tomcat.jdbc.pool.DataSource`作为数据源

所有跟数据源有关的配置都在DataSourceProperties中

6.2.2 自动配置

1.DataSourceConfiguration类中是关于数据源的配置，默认使用Tomcat连接池。可以使用spring.datasource.type在配置文件中对数据源进行配置

2.默认可以支持：

```
org.apache.tomcat.jdbc.pool.DataSource
com.zaxxer.hikari.HikariDataSource
org.apache.commons.dbcp2.BasicDataSource
```

3.除了默认的数据源外，也可以支持其他的数据源：

```
@ConditionalOnMissingBean(DataSource.class)
@ConditionalOnProperty(name = "spring.datasource.type")
static class Generic {
    @Bean
    public DataSource dataSource(DataSourceProperties properties) {
        //使用DataSourceBuilder通过反射来创建指定的数据源，并绑定相关属性
        return properties.initializeDataSourceBuilder().build();
    }
}
```

4.DataSourceInitializer

作用：

initSchema（）和runScripts（）两个方法，可以执行初始化建表语句和数据初始化文件

只需要在配置文件中指定好sql文件的位置

```
spring.datasource.schema    ----建表语句
spring.datasource.data      ----数据初始化语句
```

5.操作数据库：自动配置了JdbcTemplate操作数据库

```
@Controller
public class JdbcController {

    //自动配置了JdbcTemplate，可以直接自动注入
    @Autowired
    JdbcTemplate jdbcTemplate;

    @ResponseBody
    @GetMapping("/jdbcQuery")
    public Map<String, Object> map(){
        List<Map<String, Object>> result = jdbcTemplate.queryForList("SELECT * FROM PERSON");
        return result.get(0);
    }
}
```

6.2.3 引入druid数据源

1.首先在pom文件中添加对应的依赖

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.10</version>
</dependency>
```

2.在配置文件中配置数据源类型spring.datasource.type

```
spring:
  datasource:
    username: rootss
    password: 123456
    url: jdbc:mysql://localhost:3306/db
    driver-class-name: com.mysql.jdbc.Driver
    #数据源类型
    type: com.alibaba.druid.pool.DruidDataSource

    # 数据源其他配置
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true
    # 配置监控统计拦截的filters, 去掉后监控界面sql无法统计, 'wall'用于防火墙
    filters: stat,wall,log4j
    maxPoolPreparedStatementPerConnectionSize: 20
    useGlobalDataSourceStat: true
    connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
```

3.对druid进行配置

要对除默认支持的数据源之外的其他数据源进行配置的时候,需要自己将配置信息绑定到数据源上


```

@Configuration
public class DruidConfig {

    //将spring.datasource下的配置绑定到DruidDataSource上，然后返回给ioc容器
    @ConfigurationProperties(prefix = "spring.datasource")
    @Bean
    public DataSource dataSource(){
        return new DruidDataSource();
    }
}

```

4.druid的数据源监控配置

```

//配置druid监控
//1.配置管理后台的Servlet,在springboot中通过ServletRegistrationBean给容器添加servlet
@Bean
public ServletRegistrationBean statViewServlet(){
    ServletRegistrationBean bean = new ServletRegistrationBean(new
    StatViewServlet(),"/druid/*");

    //对StatViewServlet一些初始化参数进行配置
    Map<String,String> initParams = new HashMap<String, String>();
    initParams.put("loginUsername","admin");//登录名
    initParams.put("loginPassword","123456");//密码
    //initParams.put("allow","localhost");//只允许localhost登录
    initParams.put("allow","");//默认是允许所有登录
    initParams.put("deny","192.168.0.5");//拒绝某个IP访问

    //通过ServletRegistrationBean的setInitParameters方法可以将servlet的初始化参数传入
    bean.setInitParameters(initParams);
    return bean;
}

//2.配置web监控的filter,springboot通过FilterRegistrationBean注册filter
@Bean
public FilterRegistrationBean webStatFilter(){
    FilterRegistrationBean filter = new FilterRegistrationBean();
    filter.setFilter(new WebStatFilter());//将WebStatFilter注册
    filter.setUrlPatterns(Arrays.asList("/*"));//设置过滤所有请求

    Map<String,String> initParams = new HashMap<String, String>();
    initParams.put("exclusions","*.js,*.css,/druid/*");//排除部分请求不拦截

    //通过FilterRegistrationBean的setInitParameters方法设置初始化参数
    filter.setInitParameters(initParams);
    return filter;
}

```

6.3 MyBatis

6.3.1 引入依赖

1.首先在pom文件中引入要使用MyBatis所需的数据库驱动、mybatis相关的依赖

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
</dependency>
```

注：mybatis-spring-boot-starter依赖于spring-boot-starter-jdbc，所以不需要手动添加jdbc的依赖，引入mybatis依赖的时候就已经添加了jdbc的依赖

2.MyBatis可以通过注解版和配置版对数据库进行操作

6.3.2 注解版mapper设置

注解版中的mapper类有三种方式进行扫描设置：

1.在主程序上通过@MapperScan注解进行标注：

```
@SpringBootApplication
@MapperScan("com.neo.mapper")
public class Application {

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

2.通过MapperScannerConfigurer进行配置

```
@Configuration
public class MyBatisMapperScannerConfig {
    @Bean
    public MapperScannerConfigurer mapperScannerConfigurer() {
        MapperScannerConfigurer mapperScannerConfigurer = new MapperScannerConfigurer();
        mapperScannerConfigurer.setSqlSessionFactoryBeanName("sqlSessionFactory");
        mapperScannerConfigurer.setBasePackage("com.platfrom.*.*.model");
        Properties properties = new Properties();
        properties.setProperty("mappers", "com.platfrom.util.base.BaseDao");
        properties.setProperty("notEmpty", "false");
        mapperScannerConfigurer.setProperties(properties);
        return mapperScannerConfigurer;
    }
}
```

3.在每个mapper类上添加@Mapper注解

6.3.3 mybatis具体操作

1.写mapper文件

```
@Select("select a.org_no as orgid,a.org_Name as orgrname,a.ywid as ywid,a.ywname as ywname,b.count,b.status from t_business_info a left join (select orgid,orgName,ywid,ywname,count(1) as count,status from t_queue_serial where updatetime >= #{startDate} and updatetime <= #{endDate} and status = #{status} group by orgid,orgName,ywid,ywname,status) b on a.org_no = b.orgid and a.ywid = b.ywid where org_no = #{orgId} order by ywid")
public List<BusinessInfo> queryByType(@Param("orgId") String orgId, @Param("startDate") Date startDate, @Param("endDate") Date endDate, @Param("status") String status);

@Select("select ywid,ywname from t_business_info where org_no = #{orgId} order by ywid")
public List<Map<String, String>> getBusinessTypes(@Param("orgId") String orgId);

@SelectProvider(type = BusinessSqlProvider.class, method = "queryUseTime")
public List<Map<String, Object>> queryUseTime(@Param("startDate") Date startDate, @Param("endDate") Date endDate, @Param("orgId") String orgId, @Param("ywid") String ywid, @Param("salesmanid") String salesmanid);
```

@Select、@Update、@Insert、@Delete 四个注解依次对应增删改查。sql语句可以通过四个注解直接写在抽象方法上。

动态sql通过@SelectProvider添加，该注解有两个属性：

type：sqlProvider对应的类

method：sqlProvider类中对应的方法

在sqlProvider中的方法返回String作为sql语句，动态sql可以通过字符串拼接或SQL对象

```
public String queryUseTime(@Param("startDate") Date startDate, @Param("endDate") Date endDate, @Param("orgId") String orgId, @Param("ywid") String ywid, @Param("salesmanid") String salesmanid) {
    SQL sql = new SQL() {
        {
            SELECT("ywname,ywid,salesmanid,salesmannname,count(1) as count,sum(timestampdiff(2,char(ENDTIME-SLTIME))) as sumtime");
            FROM("t_queue_serial");
            WHERE("STATUS = '3'");
            WHERE(" updatetime >= #{startDate}");
            WHERE("updatetime <= #{endDate}");
            WHERE("orgid = #{orgId}");
            if (!"".equals(ywid) && ywid != null) {
                WHERE("ywid = #{ywid}");
            }
            if (!"".equals(salesmanid) && salesmanid != null) {
                WHERE("salesmanid = #{salesmanid}");
            }
            GROUP_BY("ywname");
            GROUP_BY("ywid");
        }
    };
    return sql.toString();
}
```

```

        GROUP_BY("salesmanid");
        GROUP_BY("salesmanname");
        ORDER_BY("ywid");
        ORDER_BY("salesmanid");
    }
};

return sql.toString();
}

```

2.在service中直接通过@Autowired注解将mapper自动注入即可

6.4. Spring Data

6.4.1 简介

Spring Data项目的目的是为了简化构建基于Spring框架应用的数据访问技术，包括非关系型数据库、Map-Reduce 框架、云数据服务等，另外也包含对关系型数据库的访问支持。

Spring Data 包含多个子项目：

- Spring Data Commons
- Spring Data JPA
- Spring Data KeyValue
- Spring Data LDAP
- Spring Data MongoDB
- Spring Data Gemfire
- Spring Data REST
- Spring Data Redis
- Spring Data for Apache Cassandra
- Spring Data for Apache Solr
- Spring Data Couchbase (community module)
- Spring Data Elasticsearch (community module)
- Spring Data Neo4j (community module)

等等

6.4.2 特点

SpringData为我们提供使用统一的API来对数据访问层进行操作；这主要是Spring Data Commons项目来实现的。Spring Data Commons让我们在使用关系型或者非关系型数据访问技术时都基于Spring提供的统一标准，标准包含了CRUD（创建、获取、更新、删除）、查询、排序和分页的相关操作。

6.4.3 统一的Repository接口

Spring Data提供统一的Repository接口来规范数据操作：

Repository：统一接口，仅仅是一个标识，表明任何继承它的均为仓库接口类 RevisionRepository>：基于乐观锁机制 CrudRepository：继承 Repository，实现了一组 CRUD 相关的方法 PagingAndSortingRepository：继承 CrudRepository，实现了一组分页排序相关的方法

JpaRepository：继承 PagingAndSortingRepository，实现一组 JPA 规范相关的方法

自定义的 XxxRepository 需要继承 JpaRepository，这样的 XxxRepository 接口就具备了通用的数据访问控制层的能力。

JpaSpecificationExecutor：不属于Repository体系，实现一组 JPA Criteria 查询相关的方法。

6.4.4 数据访问模板类

Spring Data给我们提供了很多数据访问模板类 xxxTemplate。

如：MongoTemplate、RedisTemplate等

6.4.5 JPA与Hibernate

JPA是对数据访问的一种规范，他有很多种实现，如Hibernate、Toplink、OpenJPA等等。

有了Spring Data JPA，我们只需要按照JPA规范对数据库进行操作就行了，不需要关心具体实现方式。

具体JPA规范参考JPA官方文档

6.4.6 使用Repository

6.4.6.1 Repository接口简介

Repository接口是一个标记型接口，它不包含任何方法。Spring Data提供该接口的多个子接口，如 CrudRepository、PagingAndSortingRepository、JpaRepository等提供了一些常用的增删改查、排序、翻页等方法。

我们的持久层Dao接口只需要继承Repository或其子接口，该接口使用了泛型，需要为其提供两个类型：第一个为该接口处理的域对象类型，第二个为该域对象的主键类型。然后框架会根据我们定义的方法名完成具体的业务逻辑实现。

6.4.6.2 自定义方法命名规则

解析方法名的规则：

关键字	含义	示例
And	等价于 SQL 中的 and 关键字	findByUsernameAndPassword(String user, String pwd)
Or	等价于 SQL 中的 or 关键字	findByUsernameOrAddress(String user, String addr)
Between	等价于 SQL 中的 between 关键字	findBySalaryBetween(int max, int min)
LessThan	等价于 SQL 中的 "<"	findBySalaryLessThan(int max)
GreaterThan	等价于 SQL 中的 ">"	findBySalaryGreaterThan(int min)
IsNull	等价于 SQL 中的 "is null"	findByUsernameIsNull()
IsNotNull	等价于 SQL 中的 "is not null"	findByUsernameIsNotNull()
NotNull	与 IsNotNull 等价	
Like	等价于 SQL 中的 "like"	findByUsernameLike(String user)
NotLike	等价于 SQL 中的 "not like"	findByUsernameNotLike(String user)
OrderBy	等价于 SQL 中的 "order by"	findByUsernameOrderBySalaryAsc(String user)
Not	等价于 SQL 中的 "!="	findByUsernameNot(String user)
In	等价于 SQL 中的 "in"，方法的参数可以是 Collection 类型，也可以是数组或者不定长参数	findByUsernameIn(Collection userList)
NotIn	等价于 SQL 中的 "not in" 方法的参数可以是 Collection 类型，也可以是数组或者不定长参数	findByUsernameNotIn(Collection userList)

Keyword	Sample	JPQL snippet
And	<code>findByLastnameAndFirstname</code>	<code>... where x.lastname = ?1 and x.firstname = ?2</code>
Or	<code>findByLastnameOrFirstname</code>	<code>... where x.lastname = ?1 or x.firstname = ?2</code>
Between	<code>findByStartDateBetween</code>	<code>... where x.startDate between 1? and ?2</code>
LessThan	<code>findByAgeLessThan</code>	<code>... where x.age < ?1</code>
GreaterThan	<code>findByAgeGreaterThan</code>	<code>... where x.age > ?1</code>
After	<code>findByStartDateAfter</code>	<code>... where x.startDate > ?1</code>
Before	<code>findByStartDateBefore</code>	<code>... where x.startDate < ?1</code>
IsNull	<code>findByAgeIsNull</code>	<code>... where x.age is null</code>

Keyword	Sample	JPQL snippet
IsNotNull,NotNull	<code>findByAge(Is)NotNull</code>	<code>... where x.age not null</code>
Like	<code>findByFirstnameLike</code>	<code>... where x.firstname like ?1</code>
NotLike	<code>findByFirstnameNotLike</code>	<code>... where x.firstname not like ?1</code>
StartingWith	<code>findByFirstnameStartingWith</code>	<code>... where x.firstname like ?1 (parameter bound with appended %)</code>
EndingWith	<code>findByFirstnameEndingWith</code>	<code>... where x.firstname like ?1 (parameter bound with prepended %)</code>
Containing	<code>findByFirstnameContaining</code>	<code>... where x.firstname like ?1 (parameter bound wrapped in %)</code>
OrderBy	<code>findByAgeOrderByLastnameDesc</code>	<code>... where x.age = ?1 order by x.lastname desc</code>
Not	<code>findByLastnameNot</code>	<code>... where x.lastname <> ?1</code>
In	<code>findByAgeIn(Collection<Age>ages)</code>	<code>... where x.age in ?1</code>
NotIn	<code>findByAgeNotIn(Collection<Age>age)</code>	<code>... where x.age not in ?1</code>
True	<code>findByActiveTrue()</code>	<code>... where x.active = true</code>
False	<code>findByActiveFalse()</code>	<code>... where x.active = false</code>

6.4.6.3 使用@Query注解自定义sql

1.使用@Query对接口方法简单标注为jpql查询

```
@Query("select u from User u where u.sex=:sex")
public List<User> getUsersBySex(@Param("sex") String sex);
@Query("select u from User u where u.sex=?1")
public List<User> getUsersBySex(String sex);
```

2.如需实现分页，需要Repository继承PagingAndSortingRepository，并将方法定义为：

```
@Query("select u from User u where u.sex=:sex")
public Page<User> getUsersBySex(@Param("sex") String sex, Pageable pageable);
```

3.自定义删改sql

删除和修改需要事务支持，只使用简单的@Query将报错，还需要标注@Modifying和@Transactional

4.自定义原生sql语句

```
@Query(value="select user.id from user where user.age > 18", nativeQuery = true)
```

6.4.6.4 @NamedQuery注解

1.在实体类上使用@NamedQuery，示例如下：

```
@NamedQuery(name = "UserModel.findByAge", query = "select o from UserModel o where o.age >= ?1")
```

2.在自己实现的DAO的Repository接口里面定义一个同名的方法，示例如下：

```
public List findByAge(int age);
```

3.然后就可以使用了，Spring会先找是否有同名的NamedQuery，如果有，那么就不会按照接口定义的方法来解析

6.4.5 使用EntityManager

