Springboot

微服务 每一个部分都是一个简单的功能单元

Spring cloud 进行分布的设置 等spring的分布式架构

Springboot 的简单的配置 （父项目）

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.1.RELEASE</version>

</parent>

就是所有的 springboot的管理 （所有依赖版本）

如果缺少的话，自己来添加相应的版本号

场景启动器 web相应需要的模块 （start就是相应依赖的 启动器）

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

主程序类

@SpringBootApplication 其实是一个组合注解

标注在某个类上 ，说明是springboot 的主配置类

@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)

1 @SpringBootConfiguration 表示这是一个springboot的配置类

@Configuration 配置类 其实就是一个容器中的组件

2@EnableAutoConfiguration 开启自动配置类

@AutoConfigurationPackage **自动配置包**

@Import(AutoConfigurationImportSelector.class)

@Import(AutoConfigurationPackages.Registrar.class)

将主配置类 （ @SpringBootApplication 标注的类） 的所有子包里面的组件扫描到spring容器中

@Import(AutoConfigurationImportSelector.class) 会给容器导入需要场景的配置类

3 使用spring Initializer 快速创建spring boot 项目

选择需要的模块，联网自动创建springboot项目

1) 主程序已经写好只需要写逻辑部分

2) resource 文件的目录结构

Static 就是所有的静态资源

Templates （保存所有的模板页面） springboot 使用的是嵌入式的tomcat 不支持jsp 可以使用 freemarker ，thymeleaf

Application.properities 应用的配置文件 修改相应的配置相当方便 （比如配置tomcat的端口）

4 springboot的配置文件

A 使用的全局配置文件 （名称是固定的）

Application.properties

Application.yml

作用修改springboot 默认配置

Yml配置问价 ( YAML Ain’t Markup Language) 以数据为中心 比xml和json更适合配置

具体的配置实例

server:

address: 1234

和之前的xml完全不同

Yaml 的基本语法

1 空格缩进就是表示不同的层间关系

2 值的写法

A 字面量 普通的值 （数字，字符串，布尔）

字符串默认不加单引号或者双引号

双引号不会转义字符串，特殊字符会作为本身的意思

Name : “zhangsan \n lis” 输出 zhangsan 换行 lis

单引号 会转移特殊字符 ，特殊字符最终只是一个普通的字符串数据

Name : “zhangsan \n lis” 输出 zhangsan \n lis

B 对象 map(属性 键值对) 注意缩进

Friends :

lastName :zhangsan

age :20

行内写法

Frineds :{ lastname：zhangsan ,age:20}

C 数组 （List set）

Pets:

* Cat
* Dog

行内写法

Pets: [ cat,dog]

测试配置文件 （需要进行绑定参数）

@ConfigurationProperties(prefix="person")

可能需要导入相应的处理器

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

如果是传入相应的值的话 两种配置方式的数值传入都是可以的 （只是格式不一样）

@ConfigurationProperties 功能可以批量的进入注入 （支持松散绑定）

补充 SpEl 的表达式 一般是在配置文件中注入或者是在注解中使用就是value调用的一种方式

 @Value("#{systemProperties['user.language']}")

或者是

1. **<bean** id="MyMessage" class="cn.spy.spel.injection.MyMessage"**>**
2. **<property** name="message" value="#{systemProperties['user.language']}"**></property>**
3. **</bean>**

松散绑定 字母之间有-符号

JSR303 数据校检

|  |  |  |  |
| --- | --- | --- | --- |
|  | @ConfigurationProperties | @value |  |
| 功能 | 批量注入配置文件属性值 | 一个个指定 |  |
| 松散绑定 | 支持 | 不支持 |  |
| SPEL | 不支持 | 支持 |  |

|  |  |  |
| --- | --- | --- |
| JSR303 | 支持 | 不支持 |
| 复杂类型分装 | 支持 | 不支持 |

如果只是在某个业务下直接从配置文件中取出值 就使用@value

如果专门写一个 javabean 来和配置文件进行映射我们就直接使用@ConfigurationProperties

校检部分实现 （对Javabean实现校检）

@ConfigurationProperties(prefix="person")

@Validated

public class person {

@NotNull

private String lastname;

@ConfigurationProperties(prefix="person") 默认使用的全局配置

@PropertySource 来加载制定的配置文件 @PropertySource("classpath:config.properties")

通过@ImportResource实现xml配置的装载

Spring boot 没有 spring的配置文件 自己编写的也不能识别。想让spring的配置文件生效的话 加载进来@ImportResource标注一个配置文件

补充junit 测试容器中是否含有bean 的方法

首先编写 bean.xml 文件 然后再主方法上添加

在相应的test文件中添加

@Autowired

ApplicationContext ioc;

@Test

public void testHelloService() {

ioc.containBean(name:"helloService");

}

最后的添加是

@ImportResource(locations="classpath:beanx.xml")

Springboot 推荐添加组件的方式 推荐使用注解的方式

1 配置类 相当于spring的配置文件 并且不需要@ImportResource 添加配置

2 使用Bean给容器添加组件

@Configuration

public class MyAppConfig {

@Bean

public HelloService helloService() {

System.out.println("laalla");

return new HelloService();

}

配置文件的占位符

RandomValuePropertySource ： 配置文件中 可以使用随机数

${random.value} ${random.int}

${random.uuid}

配置文件占位符如果没有的话 可以是用 制定默认值

person.lastname=zhangsan ${random.uuid}

person.age=${random.int}

5 Profile

Profile 是Spring对不同的环境提供不同的配置功能的支持，可以通过激活指定参数的方式来快速激活切换环境

1 多profile方式

编写多个properties文件 并且在里面进配置

激活这个配置文件便会被使用

# spring.profiles.active4 = dev

2 使用yml配置文件

三个横线 区分文档快

Server:

Port :20

Spring:

Profiles:

Active: dev

---

Server:

Port :20

Spring:

Profiles: product

3 使用命令行的方式来激活

在主方法运行的时候 program arguments 配上 –spring.profiles.active=dev

4注解方式引用

@ActiveProfiles("dev") 注意

6 配置文件加载位置

1 file：/config/

2 File

3 classpath:/config

4 classpath

并且是按照优先级从高到低的顺序

还可以通过spring.config.location 来改变配置文件的位置

项目打包以后 我们可以使用命令行参数的形式， 启动项目时指定配置文件的位置 指定文件和默认加载的这些配置文件会共同起作用

--spring.config.location ：D:\test\lala.yml (打包后进行java –jar --)

加载外部配置文件的顺序

1 首先是命令行配置

Java –jar lala.jar –server.port=2020

多个配置用空格分开

2 来自java:compe/env 的NDI属性

3 java系统属性（System.getProperties）

4操作系统环境变量

5 RandomVaulePropertySource 配置的random.\* 属性值

都是由jar包外向jar包进行寻找

优先加载 再有profile

6 jar包外部的application-(profile).properties或者application.yml（带spring.profile配置文件）

7 jar包内部部的application-(profile).properties或者application.yml（带spring.profile配置文件）

再来加载不带profile

8jar包外部的application-(profile).properties或者application.yml（不带spring.profile配置文件）

9 jar包内部部的application-(profile).properties或者application.yml（不带spring.profile配置文

10 通过 @Configuration注解上的@PropertySource

8 自动配置 的原理

EnableAutoConfiguration注解

在创建Application时我们使用了SpringBootApplication注解，在spring boot实战(第九篇)Application创建源码分析中曾有所分析，再来看下其定义：

[html] view plain copy

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@Configuration

@EnableAutoConfiguration

@ComponentScan

public @interface SpringBootApplication {

Class<?>[] exclude() default {};

}

该注解上存在元注解@EnableAutoConfiguration，这就是Spring Boot自动配置实现的核心入口；其定义为：

[html] view plain copy

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@Import({ EnableAutoConfigurationImportSelector.class,

AutoConfigurationPackages.Registrar.class })

public @interface EnableAutoConfiguration {

/\*\*

\* Exclude specific auto-configuration classes such that they will never be applied.

\* @return the classes to exclude

\*/

Class<?>[] exclude() default {};

}

很显然能看出有一特殊的注解@Import，该注解在spring boot实战(第十篇)Spring boot Bean加载源码分析中有讲解到，加载bean时会解析Import注解，因此需要讲目光聚集在这段代码

[html] view plain copy

@Import({ EnableAutoConfigurationImportSelector.class,

AutoConfigurationPackages.Registrar.class })

EnableAutoConfigurationImportSelector

来看EnableAutoConfigurationImportSelector类

[html] view plain copy

public String[] selectImports(AnnotationMetadata metadata) {

try {

AnnotationAttributes attributes = AnnotationAttributes.fromMap(metadata

.getAnnotationAttributes(EnableAutoConfiguration.class.getName(),

true));

Assert.notNull(attributes, "No auto-configuration attributes found. Is "

+ metadata.getClassName()

+ " annotated with @EnableAutoConfiguration?");

// Find all possible auto configuration classes, filtering duplicates

List<String> factories = new ArrayList<String>(new LinkedHashSet<String>(

SpringFactoriesLoader.loadFactoryNames(EnableAutoConfiguration.class,

this.beanClassLoader)));

// Remove those specifically disabled

factories.removeAll(Arrays.asList(attributes.getStringArray("exclude")));

// Sort

factories = new AutoConfigurationSorter(this.resourceLoader)

.getInPriorityOrder(factories);

return factories.toArray(new String[factories.size()]);

}

catch (IOException ex) {

throw new IllegalStateException(ex);

}

}

看如下代码，获取类路径下spring.factories下key为EnableAutoConfiguration全限定名对应值

[html] view plain copy

SpringFactoriesLoader.loadFactoryNames(EnableAutoConfiguration.class,

this.beanClassLoader))

其结果为：

[html] view plain copy

# Auto Configure

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\

org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\

org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\

org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\

org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\

org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,\

org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfiguration,\

org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\

org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\

org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfiguration,\

org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration,\

org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration,\

org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\

org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\

org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\

org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\

org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\

org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\

org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\

org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\

org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\

org.springframework.boot.autoconfigure.jms.hornetq.HornetQAutoConfiguration,\

org.springframework.boot.autoconfigure.jta.JtaAutoConfiguration,\

org.springframework.boot.autoconfigure.elasticsearch.ElasticsearchAutoConfiguration,\

org.springframework.boot.autoconfigure.elasticsearch.ElasticsearchDataAutoConfiguration,\

org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\

org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\

org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\

org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\

org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\

org.springframework.boot.autoconfigure.mobile.DeviceResolverAutoConfiguration,\

org.springframework.boot.autoconfigure.mobile.DeviceDelegatingViewResolverAutoConfiguration,\

org.springframework.boot.autoconfigure.mobile.SitePreferenceAutoConfiguration,\

org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\

org.springframework.boot.autoconfigure.mongo.MongoDataAutoConfiguration,\

org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration,\

org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\

org.springframework.boot.autoconfigure.reactor.ReactorAutoConfiguration,\

org.springframework.boot.autoconfigure.redis.RedisAutoConfiguration,\

org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\

org.springframework.boot.autoconfigure.security.FallbackWebSecurityAutoConfiguration,\

org.springframework.boot.autoconfigure.social.SocialWebAutoConfiguration,\

org.springframework.boot.autoconfigure.social.FacebookAutoConfiguration,\

org.springframework.boot.autoconfigure.social.LinkedInAutoConfiguration,\

org.springframework.boot.autoconfigure.social.TwitterAutoConfiguration,\

org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\

org.springframework.boot.autoconfigure.velocity.VelocityAutoConfiguration,\

org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\

org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration,\

org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration,\

org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration,\

org.springframework.boot.autoconfigure.web.GzipFilterAutoConfiguration,\

org.springframework.boot.autoconfigure.web.HttpEncodingAutoConfiguration,\

org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration,\

org.springframework.boot.autoconfigure.web.MultipartAutoConfiguration,\

org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration,\

org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration,\

org.springframework.boot.autoconfigure.websocket.WebSocketAutoConfiguration

以上为Spring Boot中所有的自动配置相关类；在启动过程中会解析对应类配置信息，以RabbitMQ为例，则会去解析RabbitAutoConfiguration

RabbitAutoConfiguration

首先来看RabbitAutoConfiguration类上的注解：

[html] view plain copy

@Configuration

@ConditionalOnClass({ RabbitTemplate.class, Channel.class })

@EnableConfigurationProperties(RabbitProperties.class)

@Import(RabbitAnnotationDrivenConfiguration.class)

public class RabbitAutoConfiguration {

@Configuration: 应该不需要解释

@ConditionalOnClass：表示存在对应的Class文件时才会去解析RabbitAutoConfiguration，否则直接跳过不解析，这也是为什么在不导入RabbitMQ依赖Jar时工程能正常启动的原因

@EnableConfigurationProperties：表示对@ConfigurationProperties的内嵌支持，默认会将对应Class这是为bean，例如这里值为RabbitProperties.class，其定义为：

[html] view plain copy

@ConfigurationProperties(prefix = "spring.rabbitmq")

public class RabbitProperties {

/\*\*

\* RabbitMQ host.

\*/

private String host = "localhost";

/\*\*

\* RabbitMQ port.

\*/

private int port = 5672; .... //省略部分代码}

RabbitProperties提供对RabbitMQ的配置信息，其前缀为spring.rabbitmq，因此在上篇中配置的host、port等信息会配置到该类上，随后@EnableConfigurationProperties会将RabbitProperties注册为一个bean。

@Import为导入配置，RabbitAnnotationDrivenConfiguration具体实现如下：

[html] view plain copy

@Configuration

@ConditionalOnClass(EnableRabbit.class)

class RabbitAnnotationDrivenConfiguration {

@Autowired(required = false)

private PlatformTransactionManager transactionManager;

@Bean

@ConditionalOnMissingBean(name = "rabbitListenerContainerFactory")

public SimpleRabbitListenerContainerFactory rabbitListenerContainerFactory(

ConnectionFactory connectionFactory) {

SimpleRabbitListenerContainerFactory factory = new SimpleRabbitListenerContainerFactory();

factory.setConnectionFactory(connectionFactory);

if (this.transactionManager != null) {

factory.setTransactionManager(this.transactionManager);

}

return factory;

}

@EnableRabbit

@ConditionalOnMissingBean(name = RabbitListenerConfigUtils.RABBIT\_LISTENER\_ANNOTATION\_PROCESSOR\_BEAN\_NAME)

protected static class EnableRabbitConfiguration {

}

}

这里又涉及到一个重要的注解：@ConditionalOnMissingBean，其功能为如果存在指定name的bean，则该注解标注的bean不创建，

[html] view plain copy

@ConditionalOnMissingBean(name = "rabbitListenerContainerFactory")

表示的意思为：如果存在名称为rabbitListenerContainerFactory的bean，则该部分代码直接忽略，这是Spring Boot人性化体现之一，开发者申明的bean会放在第一位，实在是太懂礼貌了~~

本篇中涉及到的注解比较多，其具体实现原理在以后有时间再具体分析。

再回到RabbitAutoConfiguration类的具体实现

首先来看：

[html] view plain copy

@Configuration

@ConditionalOnMissingBean(ConnectionFactory.class)

protected static class RabbitConnectionFactoryCreator {

@Bean

public ConnectionFactory rabbitConnectionFactory(RabbitProperties config) {

CachingConnectionFactory factory = new CachingConnectionFactory();

String addresses = config.getAddresses();

factory.setAddresses(addresses);

if (config.getHost() != null) {

factory.setHost(config.getHost());

factory.setPort(config.getPort());

}

if (config.getUsername() != null) {

factory.setUsername(config.getUsername());

}

if (config.getPassword() != null) {

factory.setPassword(config.getPassword());

}

if (config.getVirtualHost() != null) {

factory.setVirtualHost(config.getVirtualHost());

}

return factory;

}

}

创建了默认的ConnectionFactory，需要注意的时，这里的ConnectionFactory无回调的设置(解答了上篇中的疑问)

[html] view plain copy

@Bean

@ConditionalOnMissingBean(RabbitTemplate.class)

public RabbitTemplate rabbitTemplate() {

return new RabbitTemplate(this.connectionFactory);

}

创建了默认的RabbitTemplate；下面创建的RabbitMessagingTemplate实现对RabbitTemplate的包装。

9 日志

9.1 日志框架

常见的日志框架 JUL ,JCL,jboss-logging,logback,log4j,

|  |  |
| --- | --- |
| 日志门面 （日志的抽象层） | 日志的实现 |
| JCL(Jakarta Common Logging) SLF4j jboss-logging | Log4j JUL()  Log4j2 Logback |

左边一般选用的是 日志门面 （抽象层） SLF4j

日志实现 Logback

而springboot 底层是spring 默认使用的是JCL

Springboot使用的也是 SLF4j 和 Logback

以后开发的时候 日志记录方法的调用 不应该直接调用日志的实现类 而是调用日志的抽象层里面的方法

使用是 首先导入slf4j 和logback 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");

}

}

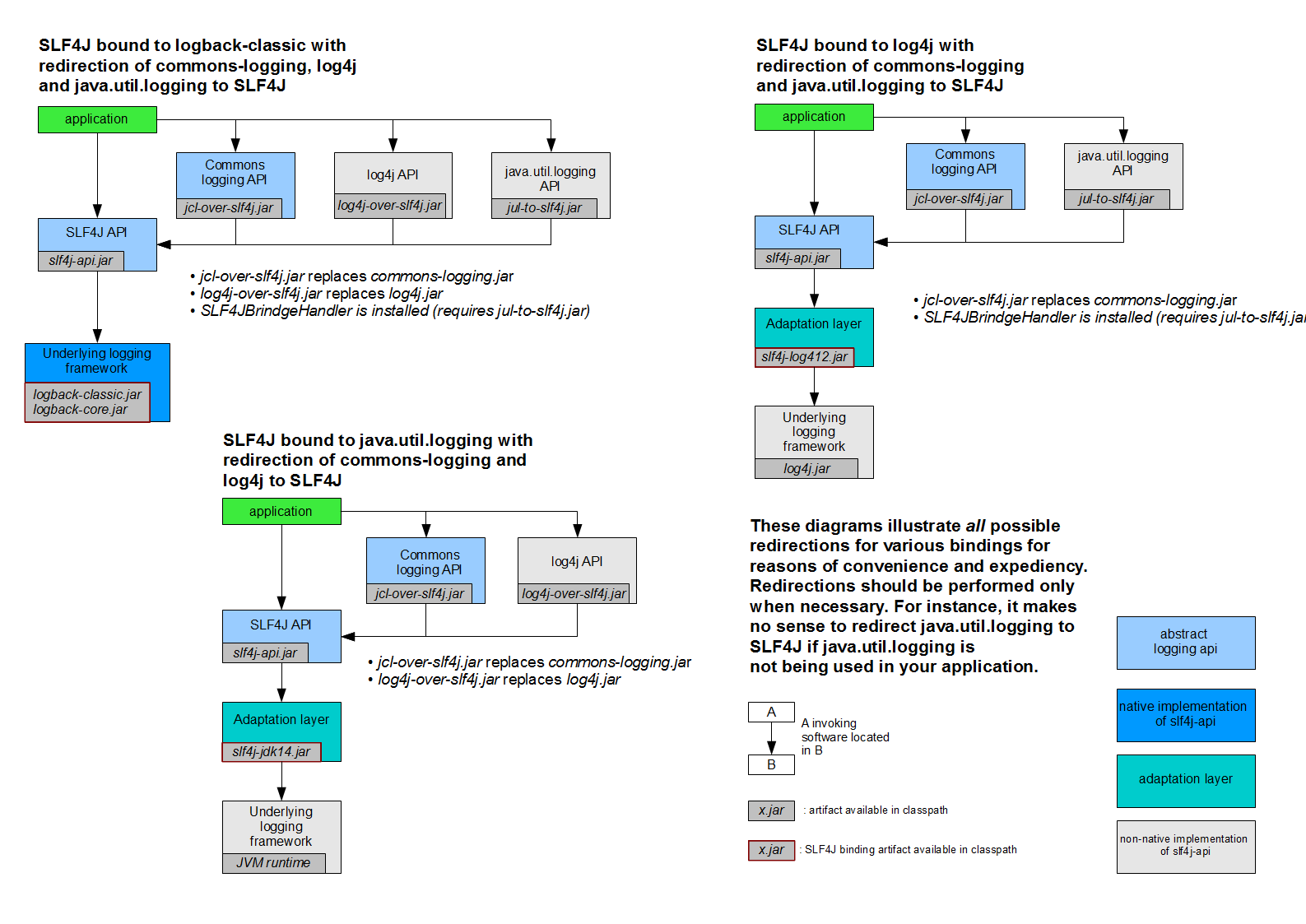
不过也能使用相其他的日志文件 需要一个适配 层用来调用

每一个日志文件都有自己的配置文件 使用slf4j以后 配置文件还是做成 日志文件 实现框架自己本身的日志文件

9.2 遗留问题

（slf4j +logback）: spring (commons-logging) Hiberante(jboss-logging) mybatis

同一日志文件 即便是逼人的框架也是利用上slf4j进行输出



如何让系统所有的日志 都统一到 slf4j

1 将其他的日志框架排除出去

2 用中间包来替换原有的日志框架

3 导入slf4j其他实现

具体的使用日志文件

@Test

public void contextLoads() {

Logger logger = LoggerFactory.getLogger(getClass());

// 日志的级别 trace <debug<info < warn < error

logger.trace("这是trace日志");

logger.debug("这是日志文件");

// springboot 打印的是info之后的 但是可以在配置文件中指定相应的配置

logger.info("这是info日志");

logger.warn("这是warn");

logger.error("这是error日志");

}

}

具体的配置的修改 并不是在

#logging.level.com.com.example.demo.Demo1Application=trace

# 不指定路径在当前目录下生成springboot

# 可以指定完整的路径

# 在当前磁盘的根路径下 生成spring 文件夹 和里面的log 文件夹 使用 spring.Log 作为默认文件

logging.path=/spring/log

指定配置

| **Logging System** | **Customization** |
| --- | --- |
| Logback | logback-spring.xml, logback-spring.groovy, logback.xml, or logback.groovy |
| Log4j2 | log4j2-spring.xml or log4j2.xml |
| JDK (Java Util Logging) | logging.properties |

1 如果是logback.xml 直接被日志框架识别

基本的配置

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

<configuration debug="false">

<!--定义日志文件的存储地址 勿在 LogBack 的配置中使用相对路径-->

<property name="LOG\_HOME" value="/home" />

<!-- 控制台输出 -->

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">

<!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符-->

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS}-------> [%thread] %-5level %logger{50} - %msg%n</pattern>

</encoder>

</appender>

<!-- 按照每天生成日志文件 -->

<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<!--日志文件输出的文件名-->

<FileNamePattern>${LOG\_HOME}/TestWeb.log.%d{yyyy-MM-dd}.log</FileNamePattern>

<!--日志文件保留天数-->

<MaxHistory>30</MaxHistory>

</rollingPolicy>

<encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">

<!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符-->

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n</pattern>

</encoder>

<!--日志文件最大的大小-->

<triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">

<MaxFileSize>10MB</MaxFileSize>

</triggeringPolicy>

</appender>

<!-- 日志输出级别 -->

<root level="INFO">

<appender-ref ref="STDOUT" />

</root>

</configuration>

2 但是loaback-spring.xml 就是springboot 识别 所有能够使用springfile 环境 可以指定某种

此处配置是在 <https://blog.csdn.net/qianyiyiding/article/details/76565810>

直接加在confiuration 中

<springProfile name="staging">

<!-- configuration to be enabled when the "staging" profile is active -->

指定某个特定的环境才能生效

</springProfile>

3 切换日志文件

首先需要 导入包支持

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<artifactId>logback-classic</artifactId>

<groupId>ch.qos.logback</groupId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId> slf4j-log4j12</artifactId>

</dependency>

然后 在配置文件中添加 Log4j.properties 的配置文件即可

如果是转换log4j2 的话

直接在

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<artifactId>spring-boot-starter-logging</artifactId>

<groupId>org.springframework.boot</groupId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<artifactId>spring-boot-starter-log4j2</artifactId>

<groupId>org.springframework.boot</groupId>

</dependency>

四 springboot 与web开发

1 使用springboot

1 创建应用 选在我们需要的模块

2 springboot已经为这些默认配置好了 只需要在配置文件中指定少量的配置

3 自己编写业务逻辑代码

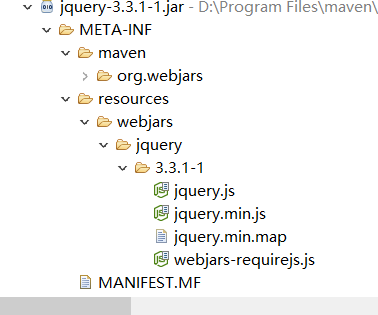
2 使用静态资源的映射原则

一webJars 的方式来引入静态资源 找好依赖在pom中添加

@ConfigurationProperities (prefix = “spring.properties”)

Public class ResourceProperties implements ResourceLoaderAware {

// 可以设置静态资源有关的参数 缓存时间



具体的访问路径是 localhost:8080/webjars/jquery/3.3.1-1/jquery.js

2 /\*\* 访问当前项目的任何资源 （ 静态资源的文件夹）

Classpath:/META-INF/resource/

Classpath: /resource/

Classpath:/static/

Classpath:/public/ 访问类似

3 欢迎页的：静态资源文件下 所有的index.html 被/\*\* 映射

4 所有的\*\*/favicon.ico 就是设置当前页面的图标（ 的名称）

但是可以在配置文件下进行静态资源的配置

#spring.resources.static-locations=classpath:/hell0,classpath:/wudeape

模板引擎 <https://blog.csdn.net/qq_32923745/article/details/78257686>

Springboot 推荐使用的Thymeleleaf 默认的版本是 thymeleaf 2和 layout 2

但是版本的可以在配置中进行版本的覆盖

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

布局功能的 适配的中间包版本的修改需要注意官网最新的相应配置是否支持

<thymeleaf.version>3.0.2 RELEASE</thymeleaf.version>

<thymeleaf-layout-dialect.version>2.1.1</thymeleaf-layout-dialect.version>

</properties>

开发时修改thymeleaf模板自动重新加载配置

Spring-boot使用thymeleaf时默认是有缓存的，即你把一个页面代码改了不会刷新页面的效果，你必须重新运行spring-boot的main()方法才能看到页面更改的效果。我们可以把thymeleaf的缓存关掉，用于支持页面修改后重新发布到spring-boot内嵌的tomcat中去。在application.properties配置文件中加入以下配置。

# Allow Thymeleaf templates to be reloaded at dev time

spring.thymeleaf.cache: false

server.tomcat.access\_log\_enabled: true

server.tomcat.basedir: target/tomcat

四、thymeleaf常用基础知识点

1、在html页面中引入thymeleaf命名空间，即<html xmlns:th=http://www.thymeleaf.org></html> ，此时在html模板文件中动态的属性使用th:命名空间修饰

2、引用静态资源文件，比如CSS和JS文件，语法格式为“@{}”，如@{/js/blog/blog.js}会引入/static目录下的/js/blog/blog.js文件

3、访问spring-mvc中model的属性，语法格式为“${}”，如${user.id}可以获取model里的user对象的id属性

4、循环，在html的标签中，加入th:each=“value:${list}”形式的属性，如<span th:each=”user:${users}”></span>可以迭代users的数据

5、判断，在html标签中，加入th:if=”表达式”可以根据条件显示html元素

?

1

2

3

<span th:if="${not #lists.isEmpty(blog.publishTime)}">

<span id="publishtime" th:text="${#dates.format(blog.publishTime, 'yyyy-MM-dd HH:mm:ss')}"></span>

</span>

以上代码表示若blog.publishTime时间不为空，则显示时间

6、时间的格式化，

?

1

${#dates.format(blog.publishTime,'yyyy-MM-dd HH:mm:ss')}

表示将时间格式化为”yyyy-MM-dd HH:mm:ss”格式化写法与Java格式化Date的写法是一致的。

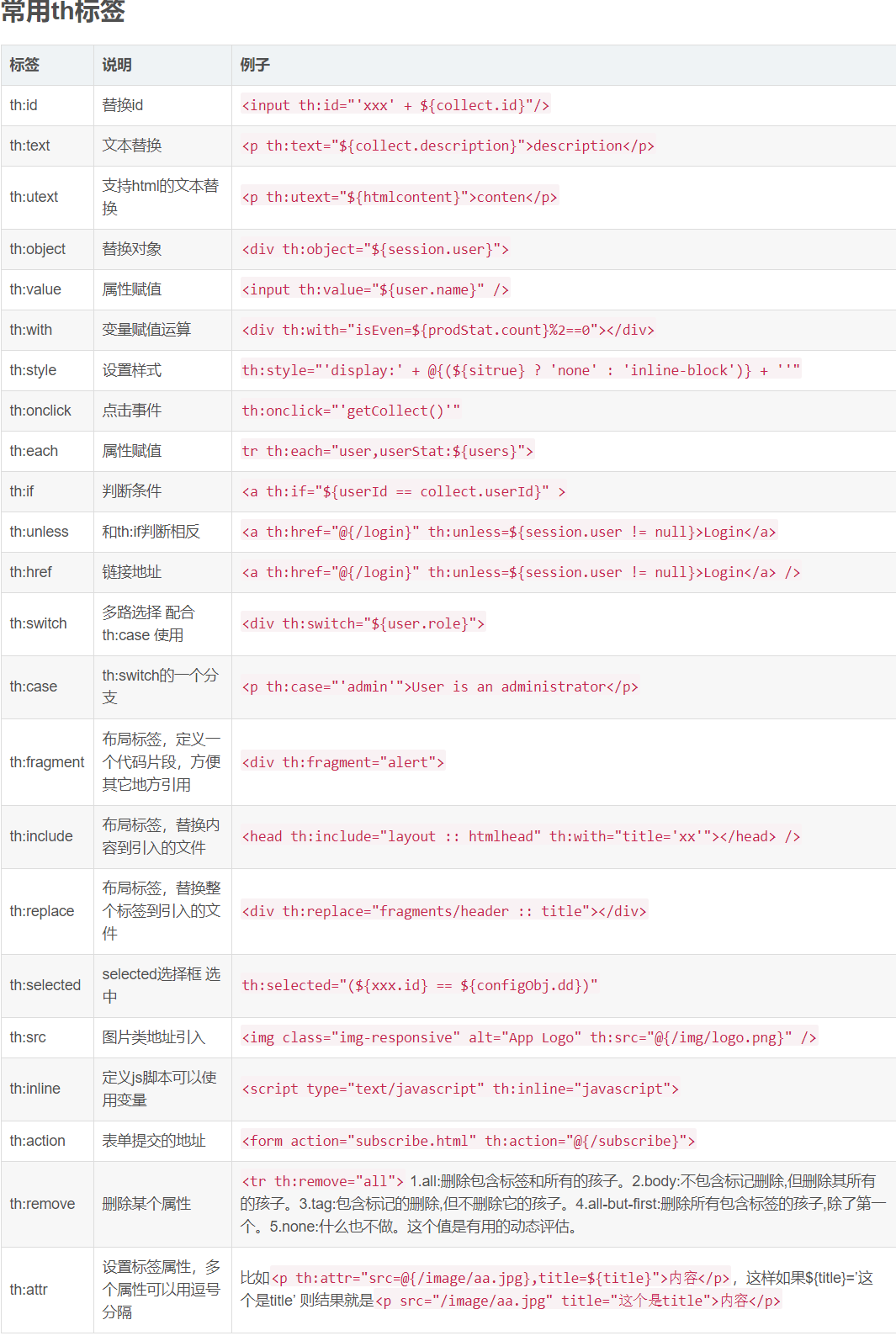
7、字符串拼接，有两种形式

比如拼接这样一个URL:/blog/delete/{blogId}

第一种：th:href="'/blog/delete/' + ${blog.id }" rel="external nofollow"

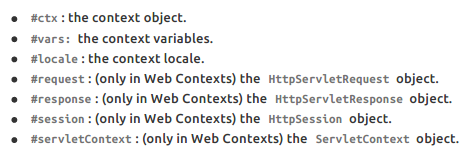
第二种：th:href="${'/blog/delete/' + blog.id }" rel="external nofollow"

常见的标签的使用

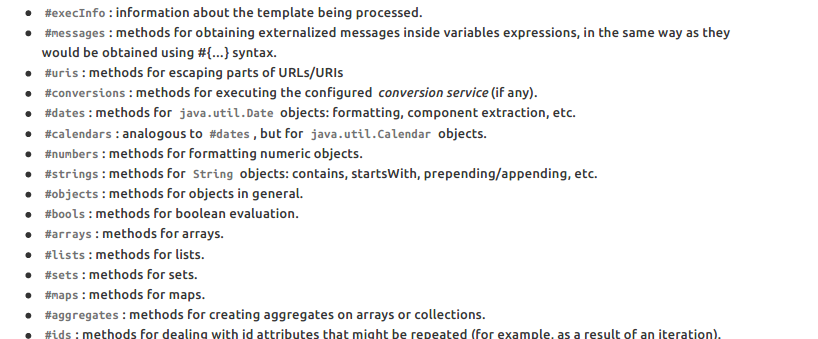


简单的表达式

1. th简单表达式：



还包括一些内置的工具类对象



1. 

　　　 ① ${...} 变量表达式：

<input type="text" name="userName" value="Beyrl" th:value="${user.name}" />

　　　　上述代码为引用user对象的name属性值。

　　　　② \*{...} 选择表达式:

<div th:object="${session.user}">

<p>Nationality: <span th:text="\*{nationality}">XXXX</span>.</p>

</div>

　　　　选择表达式一般跟在th:object后，直接选择object中的属性。

选择表达式 \* { } 和{ }的功能是一样的 以下部分是补充部分



**③ #{...} 消息文字表达式：获取国际化信息**

<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>

　　　　④ @{...} 链接url表达式：

<a href="details.html" th:href="@{/webPage/details(orderId=${o.id})}">view</a>

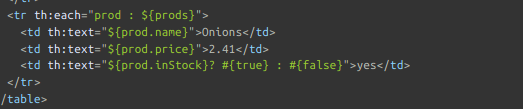
　　　　@{……}支持决定路径和相对路径。其中相对路径又支持跨上下文调用url和协议的引用（//code.jquery.com/jquery-2.0.3.min.js）。

　　　当URL为后台传出的参数时，代码如下：

<img src="../../webPage/food/images/pizza.jpg" th:src="@{${path}}" alt="披萨" />

Iteration 遍历

基本的使用方法 就是<th:each> ( uth:each 是可以解析标签的)



Inling行内加载 (两种使用方式 [[ ]] 和 [ ( ) ] )



Springboot 的web开发 <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#boot-features-spring-mvc>

The [Spring Web MVC framework](https://docs.spring.io/spring/docs/5.0.6.RELEASE/spring-framework-reference/web.html" \l "mvc" \t "https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/_top) (often referred to as simply “Spring MVC”) is a rich “model view controller” web framework. Spring MVC lets you create special@Controller or @RestController beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP by using @RequestMappingannotations.

The following code shows a typical @RestController that serves JSON data:

@RestController@RequestMapping(value="/users")

public class MyRestController {

@RequestMapping(value="/{user}", method=RequestMethod.GET)

public User getUser(@PathVariable Long user) {

// ...

}

@RequestMapping(value="/{user}/customers", method=RequestMethod.GET)

List<Customer> getUserCustomers(@PathVariable Long user) {

// ...

}

@RequestMapping(value="/{user}", method=RequestMethod.DELETE)

public User deleteUser(@PathVariable Long user) {

// ...

}

}

 Spring MVC Auto-configuration 自动配置

1 Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans.

自动配置了ViewResolver （视图解析器：根据方法的返回值得到视图对象（view） 然后决定如何渲染 转发 重定向 ）

ContentNegotiatingViewResolver 组合所有的视图解析器

2 Support for serving static resources, including support for WebJars (covered [later in this document](https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/" \l "boot-features-spring-mvc-static-content" \o "27.1.5 Static Content))). 静态资源的文件夹路径webjars

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

Converter 转化器 （类型）

Formatter  格式化器 把接受的一个时间类型变成 date类型

4 Support for HttpMessageConverters (covered [later in this document](https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/" \l "boot-features-spring-mvc-message-converters" \o "27.1.2 HttpMessageConverters)).

HttpMessageConverters  ：springmvc 的消息转换器

也可以自定义一个转换器 只需要加入容器中就可以了

@Configuration**public** **class** MyConfiguration {

@Bean

**public** HttpMessageConverters customConverters() {

HttpMessageConverter<?> additional = ...

HttpMessageConverter<?> another = ...

**return** **new** HttpMessageConverters(additional, another);

}

5 Automatic registration of MessageCodesResolver (covered [later in this document](https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/" \l "boot-features-spring-message-codes" \o "27.1.4 MessageCodesResolver)).

Static index.html support.

定义错误代码生成规则

6 Custom Favicon support (covered [later in this document](https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/" \l "boot-features-spring-mvc-favicon" \o "27.1.7 Custom Favicon)).、

Automatic use of a ConfigurableWebBindingInitializer bean (covered [later in this document](https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/" \l "boot-features-spring-mvc-web-binding-initializer" \o "27.1.9 ConfigurableWebBindingInitializer)).

如何修改springboot 的默认组件

1 使用组件时 扫描如果没有的话才会使用springboot的默认组件 如果有的话 会采用 组合的形式 来使用

2