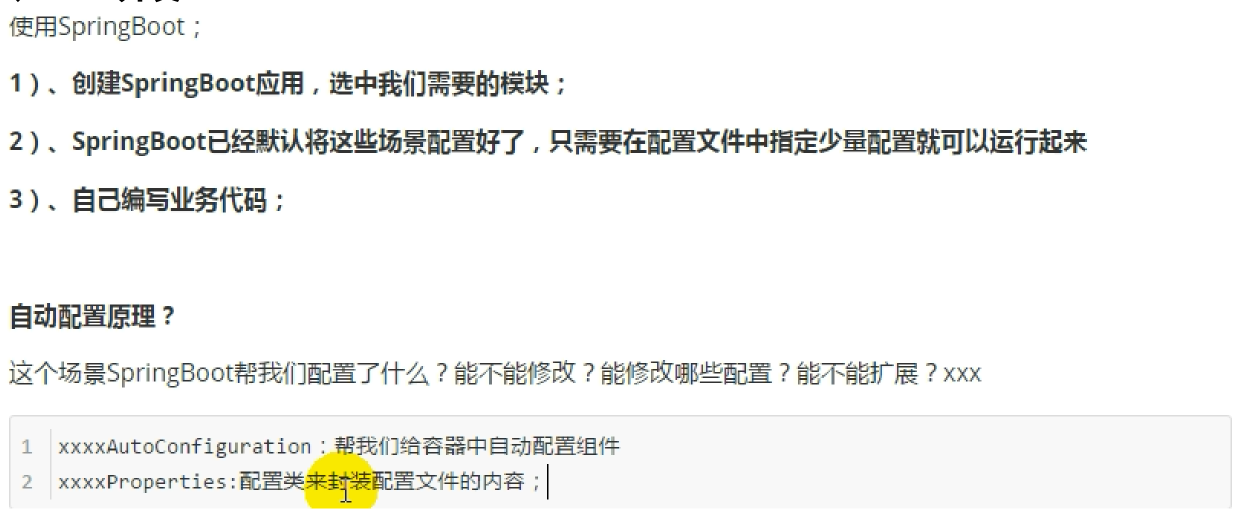
1、Web开发



使用SpringBoot；

1）、创建SpringBoot应用，选中我们需要的模块；

2）、SpringBoot已经默认将这些场景配置好了，只需要在配置文件中指定少量配置就可以运行起来

3）、自己编写业务代码；

自动配置原理？

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

```
xxxxAutoConfiguration：帮我们给容器中自动配置组件
xxxxProperties:配置类来封装配置文件的内容；
```

2、SpringBoot对静态资源的映射规则

WebMvcAutoConfiguration.java类中有如下方法

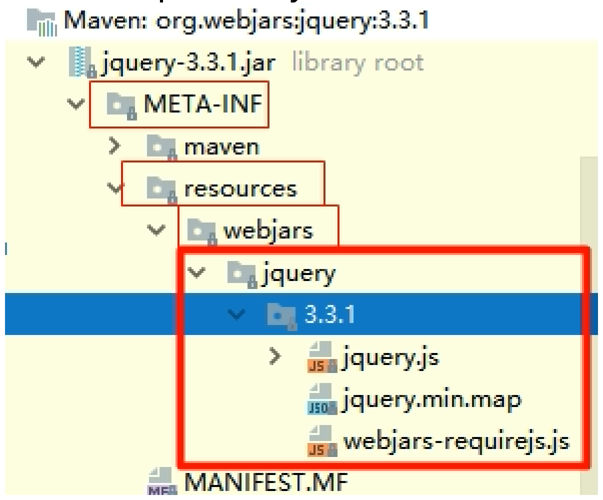
```java
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    if (!this.resourceProperties.isAddMappings()) {
        logger.debug("Default resource handling disabled");
    return;
    }
    Duration cachePeriod = this.resourceProperties.getCache().getPeriod();
    CacheControl cacheControl = this.resourceProperties.getCache()
        .getCachecontrol().toHttpCacheControl();
    if (!registry.hasMappingForPattern("/webjars/**")) {
        customizeResourceHandlerRegistration(registry
                .addResourceHandler("/webjars/**")
                .addResourceLocations("classpath:/META-INF/resources/webjars/")
                .setCachePeriod(getSeconds(cachePeriod))
                .setCacheControl(cacheControl));
    }
    String staticPathPattern = this.mvcProperties.getStaticPathPattern();
    if (!registry.hasMappingForPattern(staticPathPattern)) {
        customizeResourceHandlerRegistration(
                registry.addResourceHandler(staticPathPattern)
                        .addResourceLocations(getResourceLocations( // "/**"
                                this.resourceProperties.getStaticLocations()))
                        .setCachePeriod(getSeconds(cachePeriod))
                        .setCacheControl(cacheControl));
    }
}
```

1）所有/webjars/**路径下的资源，都去classpath:/META-INF/resources/webjars/目录找；

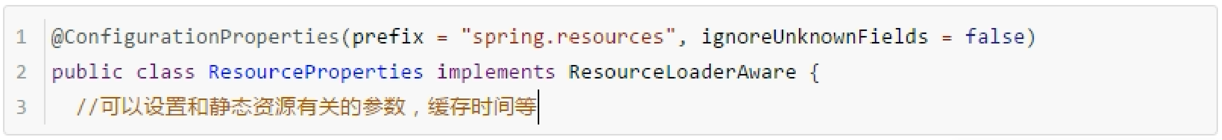
webjars：以jar包的方式引入静态资源

在pom文件中导入依赖

```
<!-- 引入jquery-webjar，以jar包的形式引入静态资源，访问的时候自需要写webjars下面
的资源名称即可-->
        <dependency>
            <groupId>org.webjars</groupId>
            <artifactId>jquery</artifactId>
            <version>3.3.1</version>
        </dependency>
```



启动项目访问jquery.js地址为：localhost:8080/webjars/jquery/3.3.1/jquery.js



```
@ConfigurationProperties(prefix = "spring.resources", ignoreUnknownFields = false)
public class ResourceProperties implements ResourceLoaderAware {
  //可以设置和静态资源有关的参数，缓存时间等
```

2）、”/**”访问当前项目的任何资源，（静态资源的文件夹）

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

  private static final String[] CLASSPATH_RESOURCE_LOCATIONS = {
      "classpath:/META-INF/resources/", "classpath:/resources/",
      "classpath:/static/", "classpath:/public/" };
```

静态资源存放的位置：

```
 "classpath:/META-INF/resources/"
  "classpath:/resources/"
  "classpath:/static/"
  "classpath:/public/"
  "/": 当前项目的根路径
```

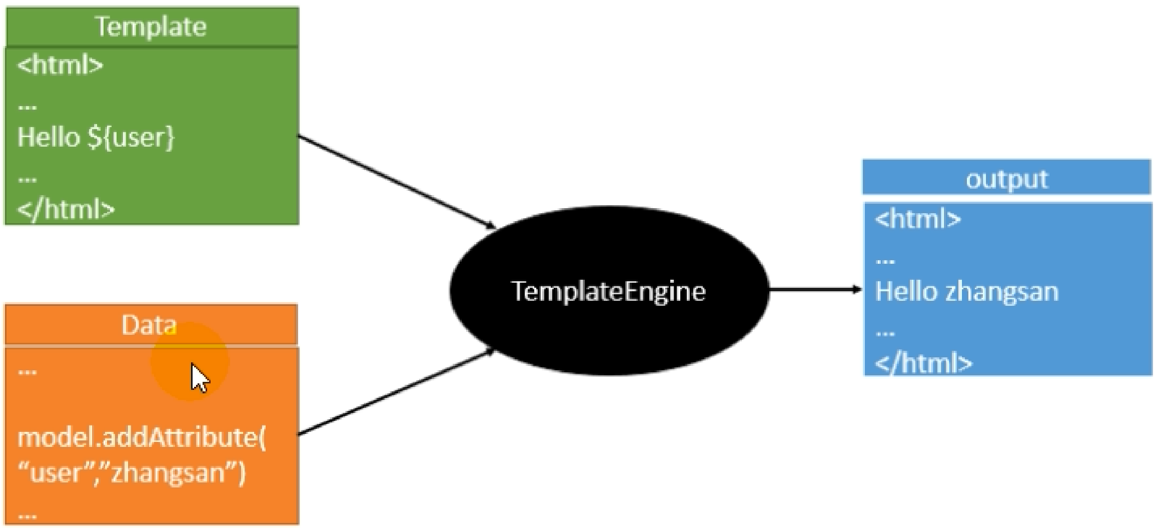
没有被处理的请求都会去静态资源目录查找对应的文件

3）、欢迎页：静态资源文件夹下的所有index.html页面都被”/**”映射，用户访问localhost:8080/ 会去找所有静态资源目录下的index.html页面。

4）、所有的 **/favicon.ico 都是在静态资源文件下找

3、模板引擎

JSP、Velocity、Freemarker、Thymeleaf



SpringBoot推荐的Thymeleaf，语法更简单，功能更强大。

1、引入Thymeleaf

```
<!-- thymeleaf-layout-dialect是布局功能支持程序，thymeleaf3主程序 需要搭配
layout2以上版本使用 -->
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

查看在spring-boot-dependencies配置的版本

```
<thymeleaf.version>3.0.12.RELEASE</thymeleaf.version>
<thymeleaf-extras-data-attribute.version>2.0.1</thymeleaf-extras-data-attribute.version>
<thymeleaf-extras-java8time.version>3.0.4.RELEASE</thymeleaf-extras-java8time.version>
<thymeleaf-extras-springsecurity.version>3.0.4.RELEASE</thymeleaf-extras-springsecurity.version>
<thymeleaf-layout-dialect.version>2.5.3</thymeleaf-layout-dialect.version>
```

2、Thymeleaf 的使用和语法

```
@ConfigurationProperties(prefix = "spring.thymeleaf")
public class ThymeleafProperties {

  private static final Charset DEFAULT_ENCODING = StandardCharsets.UTF_8;

  public static final String DEFAULT_PREFIX = "classpath:/templates/";
  public static final String DEFAULT_SUFFIX = ".html";
  // 只要我们把HTML页面放在classpath:/templates/, thymeleaf就能自动渲染
```

使用：

1）、在html页面中导入thymeleaf的命名空间

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

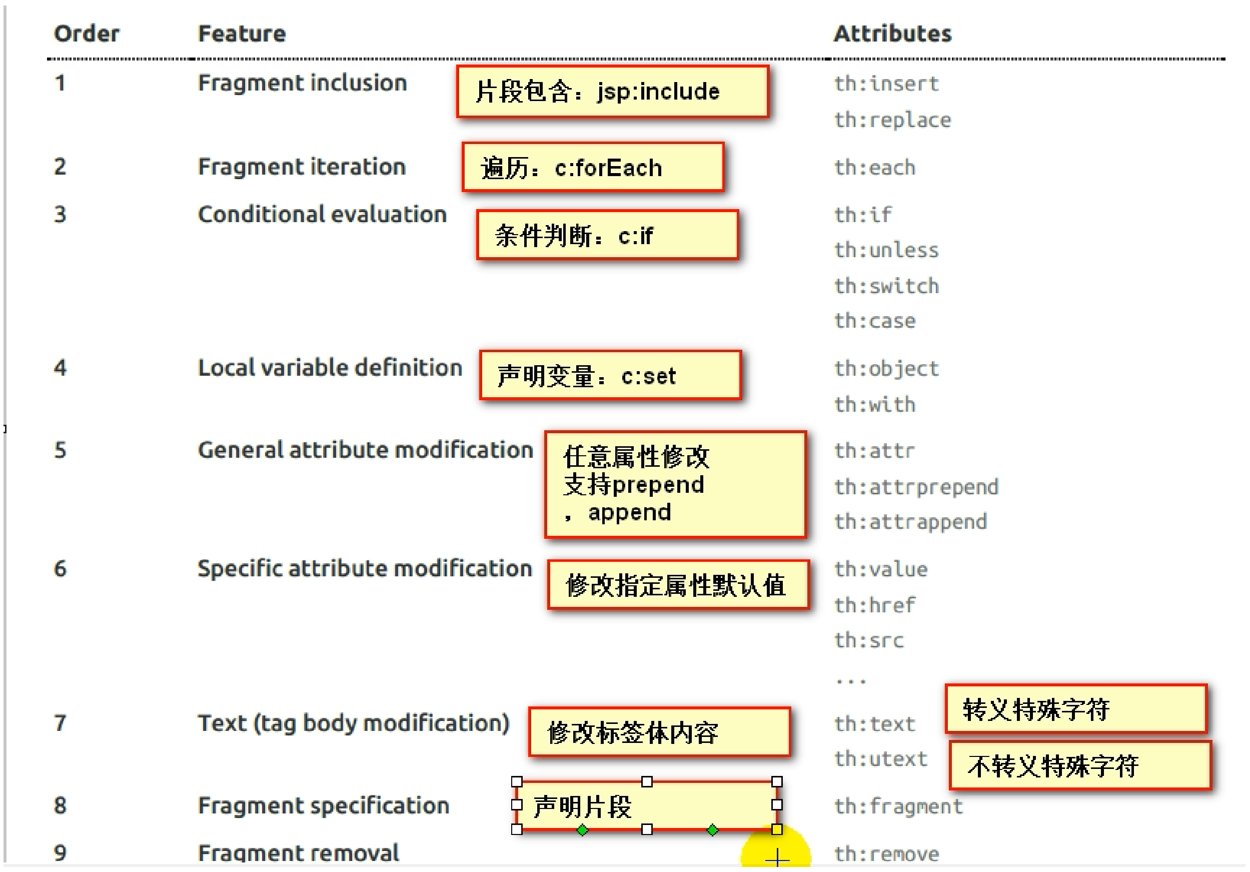
2）、使用thymeleaf语法

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <h2>Thymeleaf ViewResolver Success!</h2>
    <!--th:text 将div里面的文本内容设置为${hello}-->
    <div th:text="${hello}"></div>
</body>
</html>
```

3）、语法规则

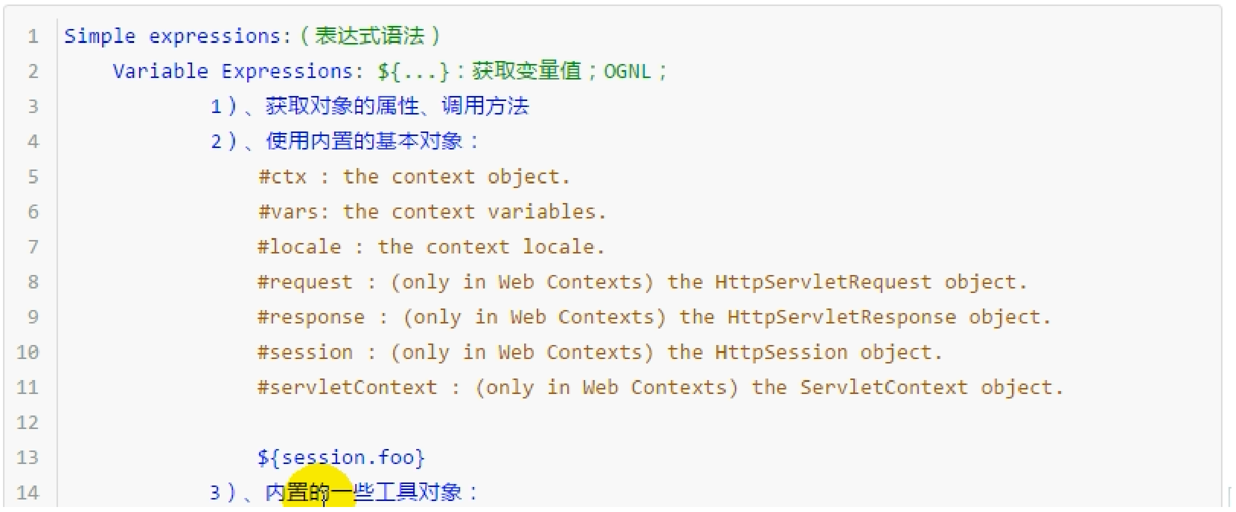
a、th:text，改变当前元素里面的文本内容，原样替换，不会渲染html标签，th:utext则会渲染html标签

th : 任意html属性，来替换原生属性值；

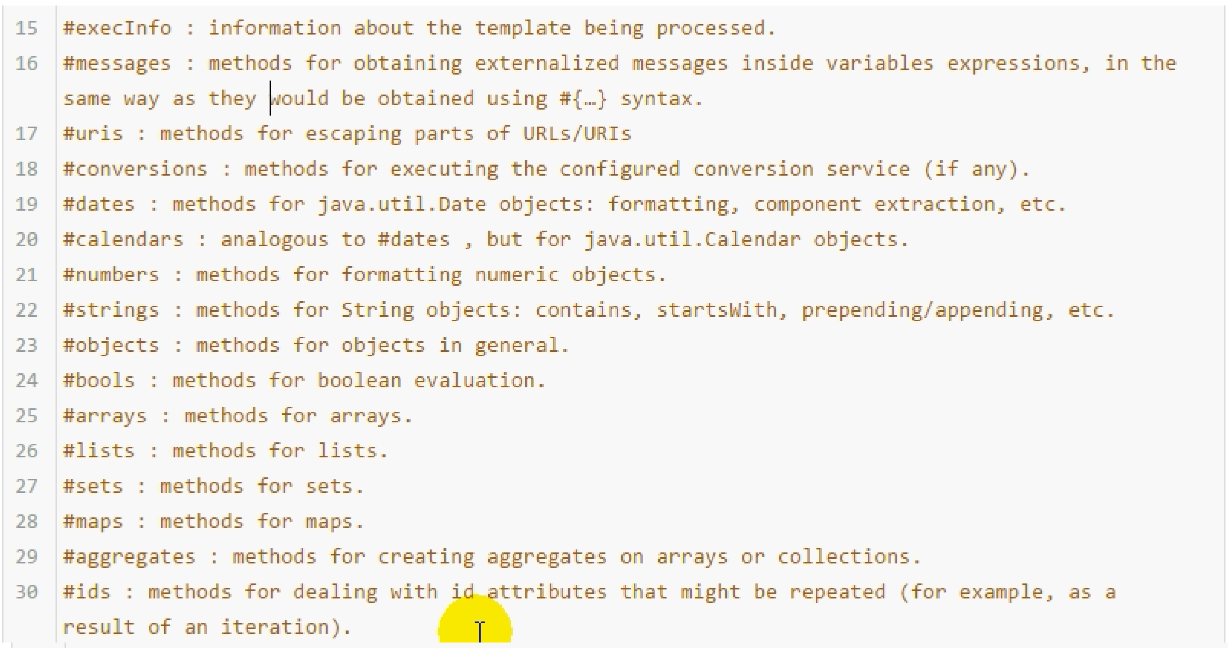


Order	Feature		Attributes	
1	Fragment inclusion	片段包含：jsp:include	th:insert th:replace	
2	Fragment iteration	遍历：c:forEach	th:each	
3	Conditional evaluation	条件判断：c:if	th:if th:unless th:switch th:case	
4	Local variable definition	声明变量：c:set	th:object th:with	
5	General attribute modification	任意属性修改 支持prepend ，append	th:attr th:attrprepend th:attrappend	
6	Specific attribute modification	修改指定属性默认值	th:value th:href th:src ...	
7	Text (tag body modification)	修改标签体内容	th:text th:utext	转义特殊字符 不转义特殊字符
8	Fragment specification	声明片段	th:fragment	
9	Fragment removal		th:remove	

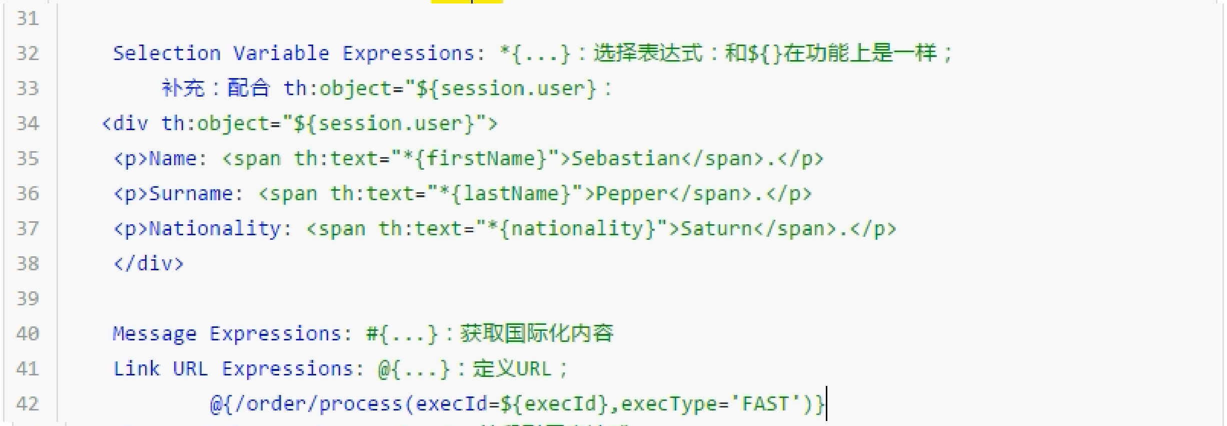
b、表达式



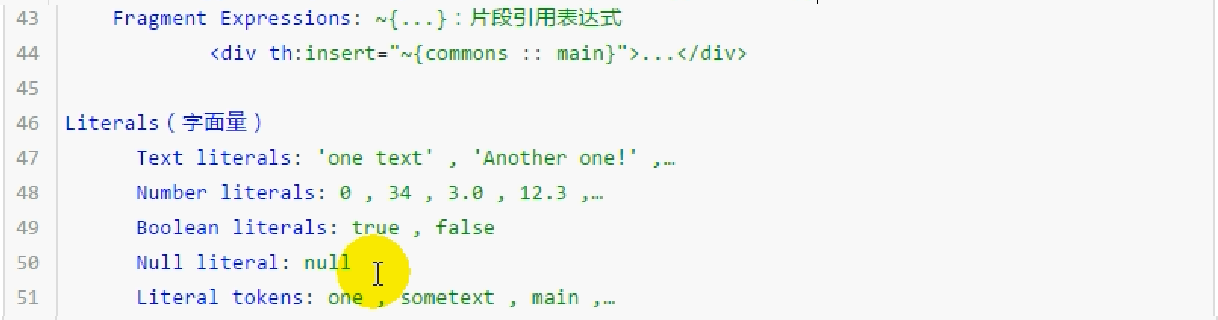
```
Simple expressions:（表达式语法）
    Variable Expressions: ${...}：获取变量值；OGNL；
            1）、获取对象的属性、调用方法
            2）、使用内置的基本对象：
                #ctx : the context object.
                #vars: the context variables.
                #locale : the context locale.
                #request : (only in Web Contexts) the HttpServletRequest object.
                #response : (only in Web Contexts) the HttpServletResponse object.
                #session : (only in Web Contexts) the HttpSession object.
                #servletContext : (only in Web Contexts) the ServletContext object.

                ${session.foo}
            3）、内置的一些工具对象：
```



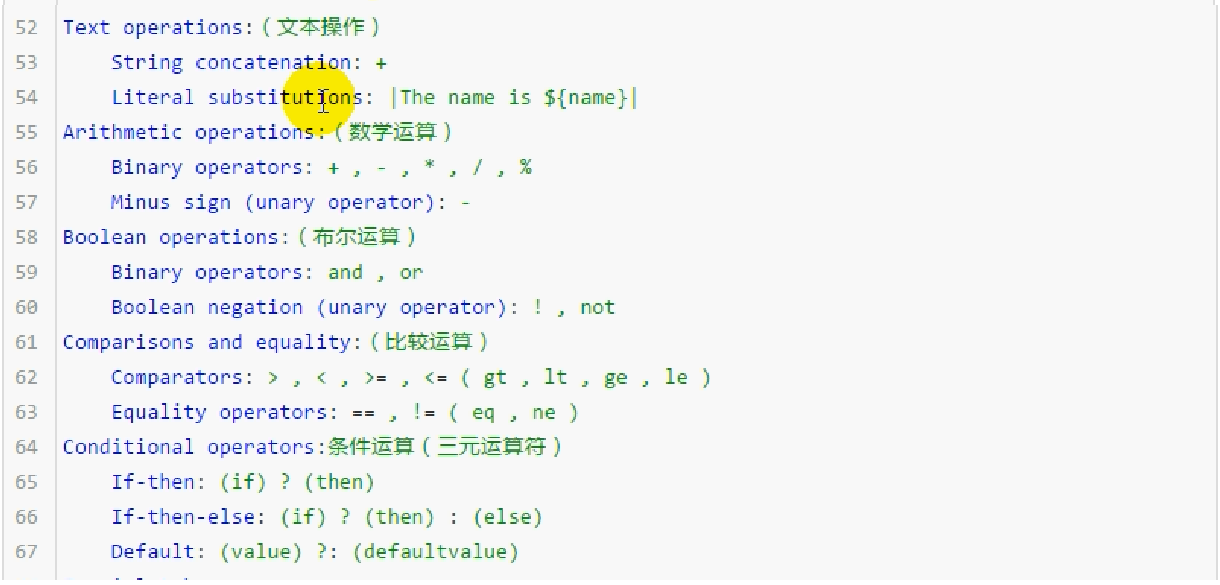
```
#execInfo : information about the template being processed.
#messages : methods for obtaining externalized messages inside variables expressions, in the
same way as they would be obtained using #{…} syntax.
#uris : methods for escaping parts of URLs/URIs
#conversions : methods for executing the configured conversion service (if any).
#dates : methods for java.util.Date objects: formatting, component extraction, etc.
#calendars : analogous to #dates , but for java.util.Calendar objects.
#numbers : methods for formatting numeric objects.
#strings : methods for String objects: contains, startsWith, prepending/appending, etc.
#objects : methods for objects in general.
#bools : methods for boolean evaluation.
#arrays : methods for arrays.
#lists : methods for lists.
#sets : methods for sets.
#maps : methods for maps.
#aggregates : methods for creating aggregates on arrays or collections.
#ids : methods for dealing with id attributes that might be repeated (for example, as a
result of an iteration).
```



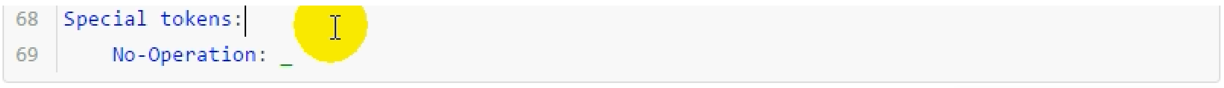
```

    Selection Variable Expressions: *{...}：选择表达式：和${}在功能上是一样；
        补充：配合 th:object="${session.user}：
   <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>

    Message Expressions: #{...}：获取国际化内容
    Link URL Expressions: @{...}：定义URL；
            @{/order/process(execId=${execId},execType='FAST')}
```



```
    Fragment Expressions: ~{...}：片段引用表达式
            <div th:insert="~{commons :: main}">...</div>

Literals（字面量）
      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 ,…
```



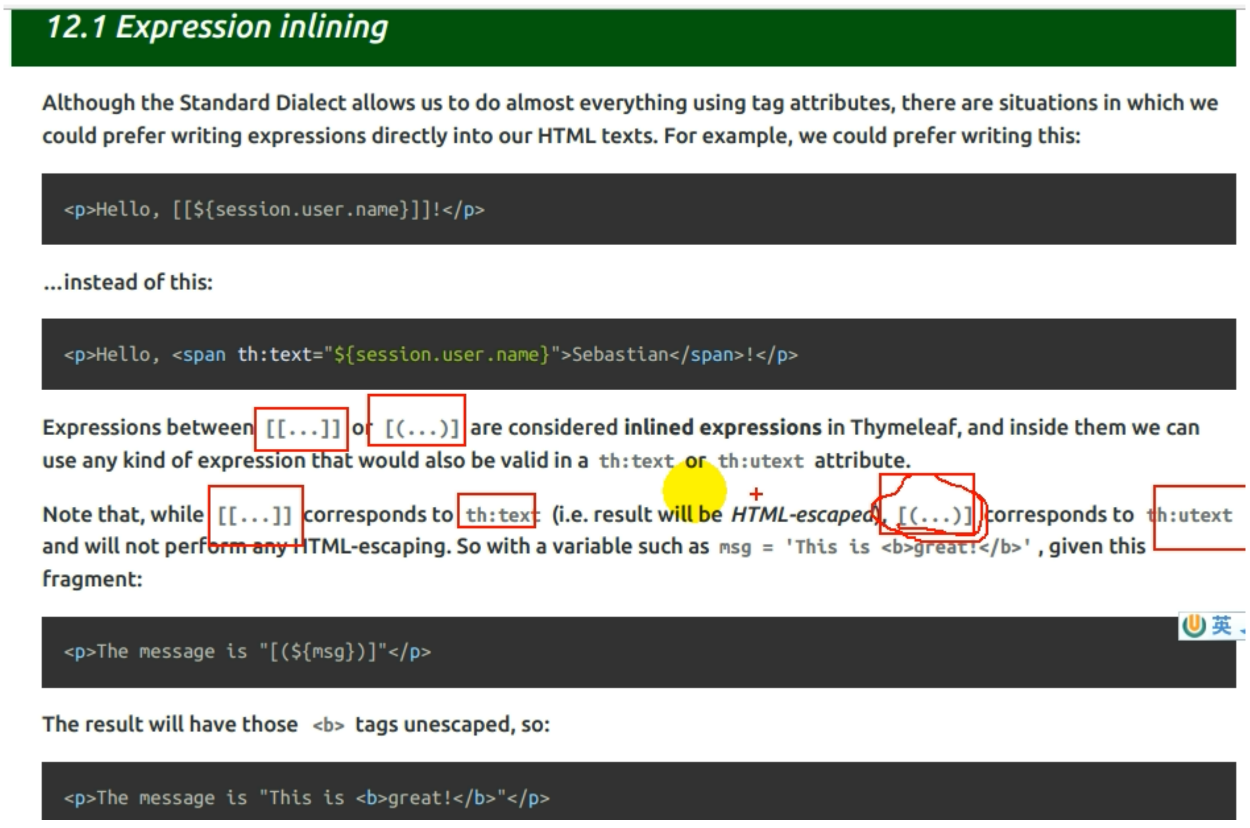
```
Text operations:（文本操作）
    String concatenation: +
    Literal substitutions: |The name is ${name}|
Arithmetic operations:（数学运算）
    Binary operators: + , - , * , / , %
    Minus sign (unary operator): -
Boolean operations:（布尔运算）
    Binary operators: and , or
    Boolean negation (unary operator): ! , not
Comparisons and equality:（比较运算）
    Comparators: > , < , >= , <= ( gt , lt , ge , le )
    Equality operators: == , != ( eq , ne )
Conditional operators:条件运算（三元运算符）
    If-then: (if) ? (then)
    If-then-else: (if) ? (then) : (else)
    Default: (value) ?: (defaultvalue)
```


```
Special tokens:
    No-Operation: _
```

常用th标签都有那些?

关键字	功能介绍	案例
th:id	替换id	<input th:id="'xxx' + ${collect.id}"/>
th:text	文本替换	<p th:text="${collect.description}">description</p>
th:utext	支持html的文本替换	<p th:utext="${htmlcontent}">conten</p>
th:object	替换对象	<div th:object="${session.user}">
th:value	属性赋值	<input th:value="${user.name}" />
th:with	变量赋值运算	<div th:with="isEven=${prodStat.count}%2==0"></div>
th:style	设置样式	th:style="'display:' + @{(${sitrue} ? 'none' : 'inline-block')} + ''"
th:onclick	点击事件	th:onclick="'getCollect()'"
th:each	属性赋值	tr th:each="user,userStat:${users}">
th:if	判断条件	<a th:if="${userId == collect.userId}" >
th:unless	和th:if判断相反	<a th:href="@{/login}" th:unless=${session.user != null}>Login</a>
th:href	链接地址	<a th:href="@{/login}" th:unless=${session.user != null}>Login</a> />
th:switch	多路选择 配合th:case 使用	<div th:switch="${user.role}">
th:case	th:switch的一个分支	<p th:case="'admin'">User is an administrator</p>
th:fragment	布局标签，定义一个代码片段，方便其它地方引用	<div th:fragment="alert">
th:include	布局标签，替换内容到引入的文件	<head th:include="layout :: htmlhead" th:with="title='xx'"></head> />
th:replace	布局标签，替换整个标签到引入的文件	<div th:replace="fragments/header :: title"></div>
th:selected	selected选择框 选中	th:selected="(${xxx.id} == ${configObj.dd})"
th:src	图片类地址引入	<img class="img-responsive" alt="App Logo" th:src="@{/img/logo.png}" />
th:inline	定义js脚本可以使用变量	<script type="text/javascript" th:inline="javascript">
th:action	表单提交的地址	<form action="subscribe.html" th:action="@{/subscribe}">
th:remove	删除某个属性	<tr th:remove="all"> 1.all:删除包含标签和所有的孩子。 2.body:不包含标记删除,但删除其所有的孩子。 3.tag:包含标记的删除,但不删除它的孩子。 4.all-but-first:删除所有包含标签的孩子,除了第一个。 5.none:什么也不做。这个值是有用的动态评

估。
th:attr　设置标签属性，多个属性可以用逗号分隔　比如 th:attr="src=@{/image/aa.jpg},title=#{logo}"，此标签不太优雅，一般用的比较少。



12.1 Expression inlining

Although the Standard Dialect allows us to do almost everything using tag attributes, there are situations in which we could prefer writing expressions directly into our HTML texts. For example, we could prefer writing this:

```
<p>Hello, [[${session.user.name}]]!</p>
```

...instead of this:

```
<p>Hello, <span th:text="${session.user.name}">Sebastian</span>!</p>
```

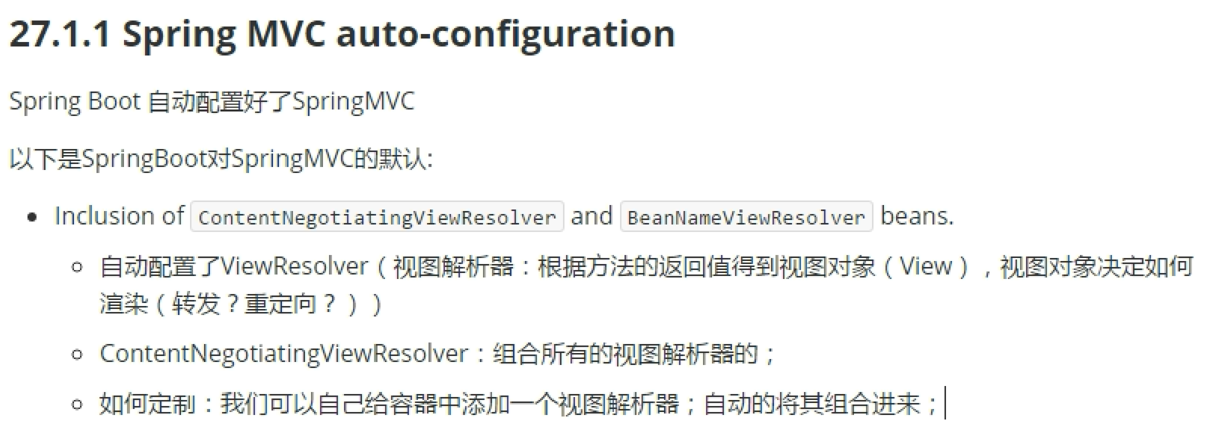
Expressions between `[[...]]` or `[(...)]` are considered **inlined expressions** in Thymeleaf, and inside them we can use any kind of expression that would also be valid in a `th:text` or `th:utext` attribute.

Note that, while `[[...]]` corresponds to `th:text` (i.e. result will be *HTML-escaped*), `[(...)]` corresponds to `th:utext` and will not perform any HTML-escaping. So with a variable such as `msg = 'This is <b>great!</b>'`, given this fragment:

```
<p>The message is "[(${msg})]"</p>
```

The result will have those `<b>` tags unescaped, so:

```
<p>The message is "This is <b>great!</b>"</p>
```



27.1.1 Spring MVC auto-configuration

Spring Boot 自动配置好了SpringMVC

以下是SpringBoot对SpringMVC的默认:

- 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），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的；
  - 如何定制：我们可以自己给容器中添加一个视图解析器；自动的将其组合进来；

```
@SpringBootApplication
public class SpringBoot04RestfulcrudApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBoot04RestfulcrudApplication.class, args);
    }

    public ViewResolver myViewResolver(){
        return new MyViewResolver();
    }
    public static class MyViewResolver implements ViewResolver{

        @Override
        public View resolveViewName(String viewName, Locale locale) throws
Exception {
            return null;
        }
    }

}
```

- Support for serving static resources, including support for WebJars (see below).静态资源文件夹路径,webjars
- Static `index.html` support. 静态首页访问
- Custom `Favicon` support (see below). favicon.ico

- 自动注册了 of `Converter` , `GenericConverter` , `Formatter` beans.
  - Converter：转换器； public String hello(User user)：类型转换使用Converter
  - `Formatter` 格式化器； 2017.12.17===Date；

```
        @Bean
        @ConditionalOnProperty(prefix = "spring.mvc", name = "date-format")//在文件中配置日期格式化的规则
        public Formatter<Date> dateFormatter() {
            return new DateFormatter(this.mvcProperties.getDateFormat());//日期格式化组件
        }
```

自己添加的格式化器转换器，我们只需要放在容器中即可

- Support for `HttpMessageConverters` (see below).
  - HttpMessageConverter：SpringMVC用来转换Http请求和响应的；User---Json；
  - `HttpMessageConverters` 是从容器中确定；获取所有的HttpMessageConverter；

    自己给容器中添加HttpMessageConverter，只需要将自己的组件注册容器中（@Bean,@Component）

- Automatic registration of `MessageCodesResolver` (see below).定义错误代码生成规则
- Automatic use of a `ConfigurableWebBindingInitializer` bean (see below).

  我们可以配置一个ConfigurableWebBindingInitializer来替换默认的；（添加到容器）

```
初始化WebDataBinder；
请求数据=====JavaBean；
```

org.springframework.boot.autoconfigure.web：web的所有自动场景；

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

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

2、扩展SpringMVC

```xml
    <mvc:view-controller path="/hello" view-name="success"/>
    <mvc:interceptors>
        <mvc:interceptor>
            <mvc:mapping path="/hello"/>
            <bean></bean>
        </mvc:interceptor>
    </mvc:interceptors>
```

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

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

既保留了所有的自动配置，也能用我们扩展的配置；

```java
//使用WebMvcConfigurerAdapter可以来扩展SpringMVC的功能
@Configuration
public class MyMvcConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
       // super.addViewControllers(registry);
        //浏览器发送 /atguigu 请求来到 success
        registry.addViewController("/atguigu").setViewName("success");
    }
}
```

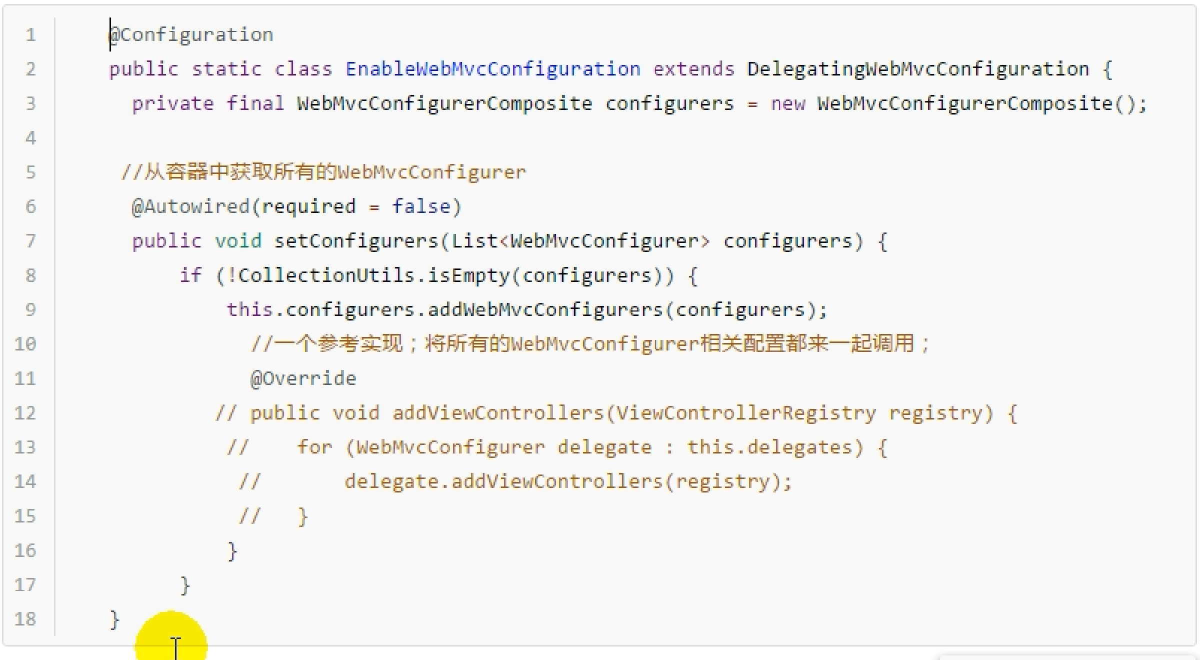
// 使用WebMvcConfigurerAdapter可以扩展SpringMVC的功能, Spring 5.0后，WebMvcConfigurerAdapter被废弃，取代的方法有两种：
/*
①implements WebMvcConfigurer（官方推荐）
②extends WebMvcConfigurationSupport
使用第一种方法是实现了一个接口，可以任意实现里面的方法，不会影响到Spring Boot自身的@EnableAutoConfiguration，
使用第二种方法相当于覆盖了@EnableAutoConfiguration里的所有方法，每个方法都需要重写，比如，若不实现方法addResourceHandlers()，则会导致静态资源无法访问
 */
@Configuration

```
public class MyMvcConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
       // super.addViewControllers(registry);
       // 浏览器发送/hello请求，来到success页面
       registry.addViewController("/hello").setViewName("success");
    }
}
```

原理：

1）、WebMvcAutoConfiguration是SpringMVC的自动配置类

2）、在做其他自动配置时会导入；@Import(**EnableWebMvcConfiguration**.class)



```
    @Configuration
    public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration {
      private final WebMvcConfigurerComposite configurers = new WebMvcConfigurerComposite();

     //从容器中获取所有的WebMvcConfigurer
      @Autowired(required = false)
      public void setConfigurers(List<WebMvcConfigurer> configurers) {
          if (!CollectionUtils.isEmpty(configurers)) {
              this.configurers.addWebMvcConfigurers(configurers);
                //一个参考实现；将所有的WebMvcConfigurer相关配置都来一起调用；
                @Override
             // public void addViewControllers(ViewControllerRegistry registry) {
              //    for (WebMvcConfigurer delegate : this.delegates) {
               //       delegate.addViewControllers(registry);
               //   }
              }
          }
    }
```

3）、容器中所有的WebMvcConfigurer都会一起起作用；

4）、我们的配置类也会被调用；

效果：SpringMVC的自动配置和我们的扩展配置都会起作用；

3、全面接管SpringMVC；

SpringBoot对SpringMVC的自动配置不需要了，所有都是我们自己配置；所有的SpringMVC的自动配置都失效了

我们需要在配置类中添加@EnableWebMvc即可；

```java
//使用WebMvcConfigurerAdapter可以来扩展SpringMVC的功能
@EnableWebMvc
@Configuration
public class MyMvcConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
       // super.addViewControllers(registry);
        //浏览器发送 /atguigu 请求来到 success
        registry.addViewController("/atguigu").setViewName("success");
    }
}
```

原理：

为什么@EnableWebMvc自动配置就失效了；

1）@EnableWebMvc的核心

```java
@Import(DelegatingWebMvcConfiguration.class)
public @interface EnableWebMvc {
```

2）、

```java
@Configuration
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {
```

3）、

```java
@Configuration
@ConditionalOnWebApplication
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class,
        WebMvcConfigurerAdapter.class })
//容器中没有这个组件的时候，这个自动配置类才生效
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class,
        ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {
```

4）、@EnableWebMvc将WebMvcConfigurationSupport组件导入进来；

5）、导入的WebMvcConfigurationSupport只是SpringMVC最基本的功能；

5、如何修改SpringBoot的默认配置

模式：

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

2）、在SpringBoot中会有非常多的xxxConfigurer帮助我们进行扩展配置

3）、登陆

开发期间模板引擎页面修改以后，要实时生效

1）、禁用模板引擎的缓存

2）、页面修改完成以后ctrl+f9：重新编译；

1）、RestfulCRUD：CRUD满足Rest风格；

URI： /资源名称/资源标识　　HTTP请求方式区分对资源CRUD操作

	普通CRUD（uri来区分操作）	RestfulCRUD
查询	getEmp	emp---GET
添加	addEmp?xxx	emp---POST
修改	updateEmp?id=xxx&xxx=xx	emp/{id}---PUT
删除	deleteEmp?id=1	emp/{id}---DELETE

如何定制错误页面

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 放在模板引起文件夹里面的 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）；

页面能获取的信息；

timestamp：时间戳

status：状态码

error：错误提示

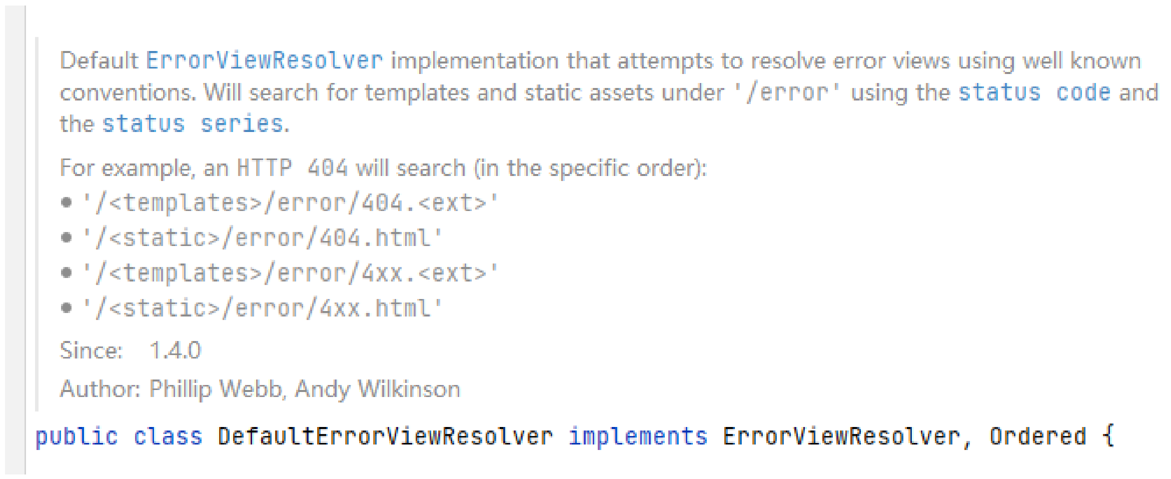
exception：异常对象

message：异常消息

errors：JSR303数据校验的错误都在这里

2）、没有模板引擎（模板引擎找不到这个错误页面），静态资源文件夹下找；

3）、以上都没有错误页面，就是默认来到SpringBoot默认的错误提示页面；



Default `ErrorViewResolver` implementation that attempts to resolve error views using well known conventions. Will search for templates and static assets under `'/error'` using the `status code` and the `status series`.

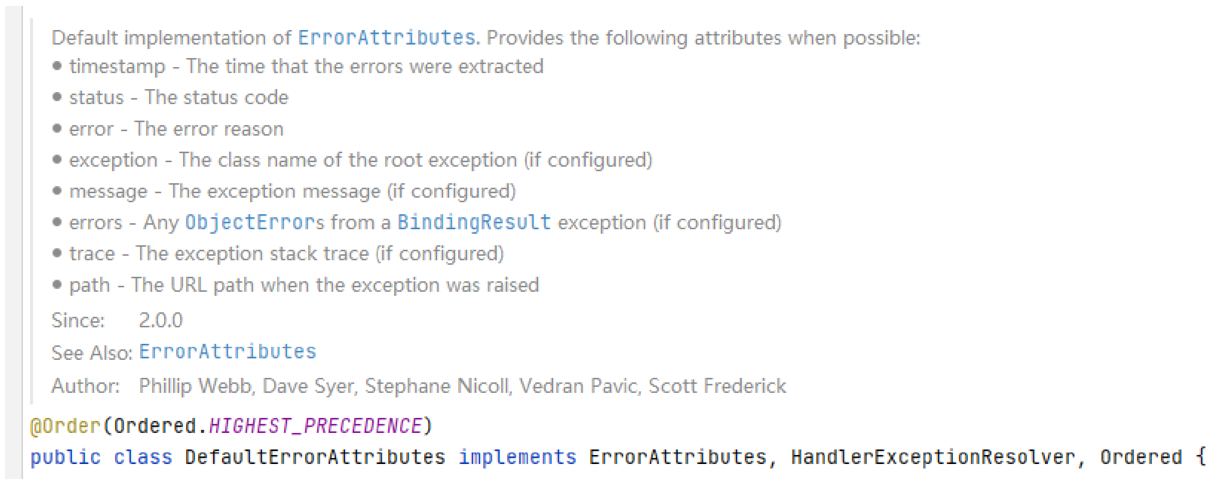
For example, an `HTTP 404` will search (in the specific order):

- `'/<templates>/error/404.<ext>'`
- `'/<static>/error/404.html'`
- `'/<templates>/error/4xx.<ext>'`
- `'/<static>/error/4xx.html'`

Since: 1.4.0

Author: Phillip Webb, Andy Wilkinson

```java
public class DefaultErrorViewResolver implements ErrorViewResolver, Ordered {
```



Default implementation of `ErrorAttributes`. Provides the following attributes when possible:

- timestamp - The time that the errors were extracted
- status - The status code
- error - The error reason
- exception - The class name of the root exception (if configured)
- message - The exception message (if configured)
- errors - Any `ObjectError`s from a `BindingResult` exception (if configured)
- trace - The exception stack trace (if configured)
- path - The URL path when the exception was raised

Since: 2.0.0

See Also: `ErrorAttributes`

Author: Phillip Webb, Dave Syer, Stephane Nicoll, Vedran Pavic, Scott Frederick

```java
@Order(Ordered.HIGHEST_PRECEDENCE)
public class DefaultErrorAttributes implements ErrorAttributes, HandlerExceptionResolver, Ordered {
```