Web开发探究

> 简介

好的，同学们，那么接下来呢，我们开始学习SpringBoot与Web开发，从这一章往后，就属于我们实战部分的内容了；

其实SpringBoot的东西用起来非常简单，因为SpringBoot最大的特点就是自动装配。

使用SpringBoot的步骤：

1、创建一个SpringBoot应用，选择我们需要的模块，SpringBoot就会默认将我们的需要的模块自动配置好。

2、手动在配置文件中配置部分配置项目就可以运行起来了。

3、专注编写业务代码，不需要考虑以前那样一大堆的配置了。

要熟悉掌握开发，之前学习的自动配置的原理一定要搞明白！

比如SpringBoot到底帮我们配置了什么？我们能不能修改？我们能修改哪些配置？我们能不能扩展？

- 向容器中自动配置组件： *** Autoconfiguration
- 自动配置类，封装配置文件的内容：***Properties

没事就找找类，看看自动装配原理！

我们之后来进行一个单体项目的小项目测试，让大家能够快速上手开发！

静态资源处理

> 静态资源映射规则

首先，我们搭建一个普通的SpringBoot项目，回顾一下HelloWorld程序！【演示】

写请求非常简单，那我们要引入我们前端资源，我们项目中有许多的静态资源，比如css，js等文件，这个SpringBoot怎么处理呢？

如果我们是一个web应用，我们的main下会有一个webapp，我们以前都是将所有的页面导在这里面的，对吧！但是我们现在的pom呢，打包方式是为jar的方式，那么这种方式SpringBoot能不能来给我们写页面呢？当然是可以的，但是SpringBoot对于静态资源放置的位置，是有规定的！

我们先来聊聊这个静态资源映射规则：

SpringBoot中，SpringMVC的web配置都在 WebMvcAutoConfiguration 这个配置类里面；

我们可以去看看 WebMvcAutoConfigurationAdapter 中有很多配置方法；

有一个方法：`addResourceHandlers` 添加资源处理

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

```
            // 已禁用默认资源处理
            logger.debug("Default resource handling disabled");
            return;
        }
        // 缓存控制
        Duration cachePeriod = this.resourceProperties.getCache().getPeriod();
        CacheControl cacheControl =
this.resourceProperties.getCache().getCachecontrol().toHttpCacheControl();
        // webjars 配置
        if (!registry.hasMappingForPattern("/webjars/**")) {

 customizeResourceHandlerRegistration(registry.addResourceHandler("/webjars/
**")

.addResourceLocations("classpath:/META-INF/resources/webjars/")

.setCachePeriod(getSeconds(cachePeriod)).setCacheControl(cacheControl));
        }
        // 静态资源配置
        String staticPathPattern = this.mvcProperties.getStaticPathPattern();
        if (!registry.hasMappingForPattern(staticPathPattern)) {

 customizeResourceHandlerRegistration(registry.addResourceHandler(staticPath
Pattern)

.addResourceLocations(getResourceLocations(this.resourceProperties.getStatic
Locations()))

.setCachePeriod(getSeconds(cachePeriod)).setCacheControl(cacheControl));
        }
    }
```

读一下源代码：比如所有的 /webjars/** ， 都需要去 classpath:/META-INF/resources/webjars/ 找对应的资源；

> 那什么是webjars呢?

Webjars本质就是以jar包的方式引入我们的静态资源 ， 我们以前要导入一个静态资源文件，直接导入即可。

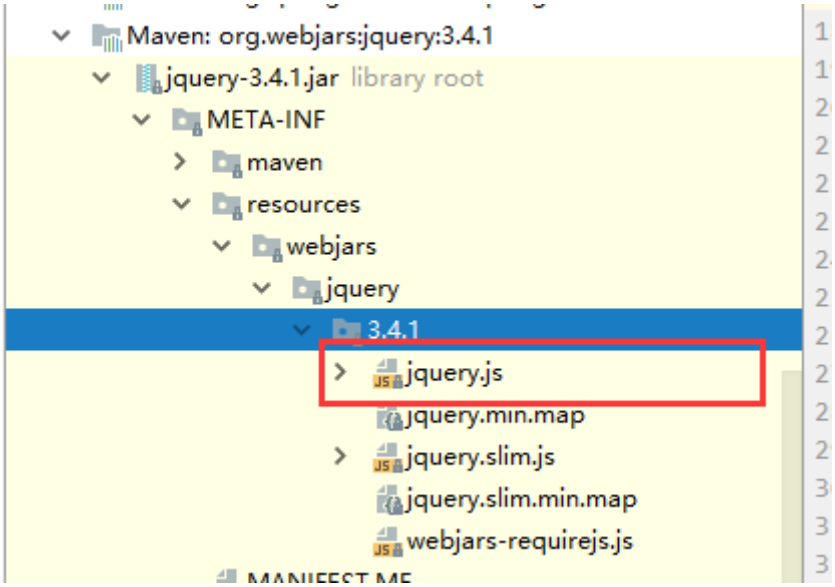
使用SpringBoot需要使用Webjars，我们可以去搜索一下：

网站：https://www.webjars.org 【网站带看，并引入jQuery测试】

要使用jQuery，我们只要要引入jQuery对应版本的pom依赖即可！

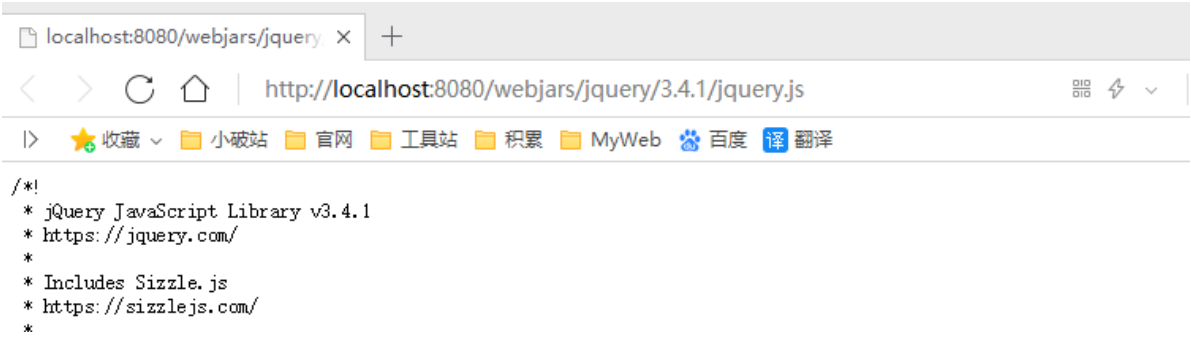
```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>3.4.1</version>
</dependency>
```

导入完毕，查看webjars目录结构，并访问Jquery.js文件！



访问：只要是静态资源，SpringBoot就会去对应的路径寻找资源，我们这里访问：

http://localhost:8080/webjars/jquery/3.4.1/jquery.js



> 第二种静态资源映射规则

那我们项目中要是使用自己的静态资源该怎么导入呢？我们看下一行代码；

我们去找staticPathPattern发现第二种映射规则： /** , 访问当前的项目任意资源，它会去找resourceProperties 这个类，我们可以点进去看一下分析：

```
// 进入方法
public String[] getStaticLocations() {
    return this.staticLocations;
}
// 找到对应的值
private String[] staticLocations = CLASSPATH_RESOURCE_LOCATIONS;
// 找到路径
private static final String[] CLASSPATH_RESOURCE_LOCATIONS = {
    "classpath:/META-INF/resources/",
    "classpath:/resources/",
    "classpath:/static/",
    "classpath:/public/"
};
```

ResourceProperties 可以设置和我们静态资源有关的参数；这里面指向了它会去寻找资源的文件夹，即上面数组的内容。

所以得出结论，以下四个目录存放的静态资源可以被我们识别：

```
"classpath:/META-INF/resources/"
"classpath:/resources/"
"classpath:/static/"
"classpath:/public/"
```

我们可以在resources根目录下新建对应的文件夹，都可以存放我们的静态文件；

比如我们访问 http://localhost:8080/1.js , 他就会去这些文件夹中寻找对应的静态资源文件；

> 自定义静态资源路径

我们也可以自己通过配置文件来指定一下，哪些文件夹是需要我们放静态资源文件的，在application.properties中配置；

```
spring.resources.static-locations=classpath:/coding/,classpath:/kuang/
```

一旦自己定义了静态文件夹的路径，原来的自动配置就都会失效了！

首页处理

静态资源文件夹说完后，我们继续向下看源码！可以看到一个欢迎页的映射，就是我们的首页！

```
@Bean
public WelcomePageHandlerMapping
welcomePageHandlerMapping(ApplicationContext applicationContext,

FormattingConversionService mvcConversionService,

ResourceUrlProvider mvcResourceUrlProvider) {
    WelcomePageHandlerMapping welcomePageHandlerMapping = new
WelcomePageHandlerMapping(
        new TemplateAvailabilityProviders(applicationContext),
applicationContext, getWelcomePage(), // getWelcomePage 获得欢迎页
        this.mvcProperties.getStaticPathPattern());

 welcomePageHandlerMapping.setInterceptors(getInterceptors(mvcConversionServ
ice, mvcResourceUrlProvider));
    return welcomePageHandlerMapping;
}
```

点进去继续看

```java
private Optional<Resource> getWelcomePage() {
    String[] locations =
getResourceLocations(this.resourceProperties.getStaticLocations());
    // ::是java8 中新引入的运算符
    // Class::function的时候function是属于Class的，应该是静态方法。
    // this::function的funtion是属于这个对象的。
    // 简而言之，就是一种语法糖而已，是一种简写
    return
Arrays.stream(locations).map(this::getIndexHtml).filter(this::isReadable).fi
ndFirst();
}
// 欢迎页就是一个location下的的 index.html 而已
private Resource getIndexHtml(String location) {
    return this.resourceLoader.getResource(location + "index.html");
}
```

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

比如我访问 http://localhost:8080/ ，就会找静态资源文件夹下的 index.html 【可以测试一下】

新建一个 index.html ，在我们上面的3个目录中任意一个；然后访问测试 http://localhost:8080/ 看结果！

关于网站图标说明：

Welcome Page

Spring Boot supports both static and templated welcome pages. It first looks for an `index.html` file in the configured static content locations. If one is not found, it then looks for an `index` template. If either is found, it is automatically used as the welcome page of the application.

Custom Favicon

As with other static resources, Spring Boot looks for a `favicon.ico` in the configured static content locations. If such a file is present, it is automatically used as the favicon of the application.

与其他静态资源一样，Spring Boot在配置的静态内容位置中查找 favicon.ico。如果存在这样的文件，它将自动用作应用程序的favicon。

1、关闭SpringBoot默认图标

```properties
#关闭默认图标
spring.mvc.favicon.enabled=false
```

2、自己放一个图标在静态资源目录下，我放在 public 目录下

3、清除浏览器缓存！刷新网页，发现图标已经变成自己的了！

Thymeleaf

模板引擎

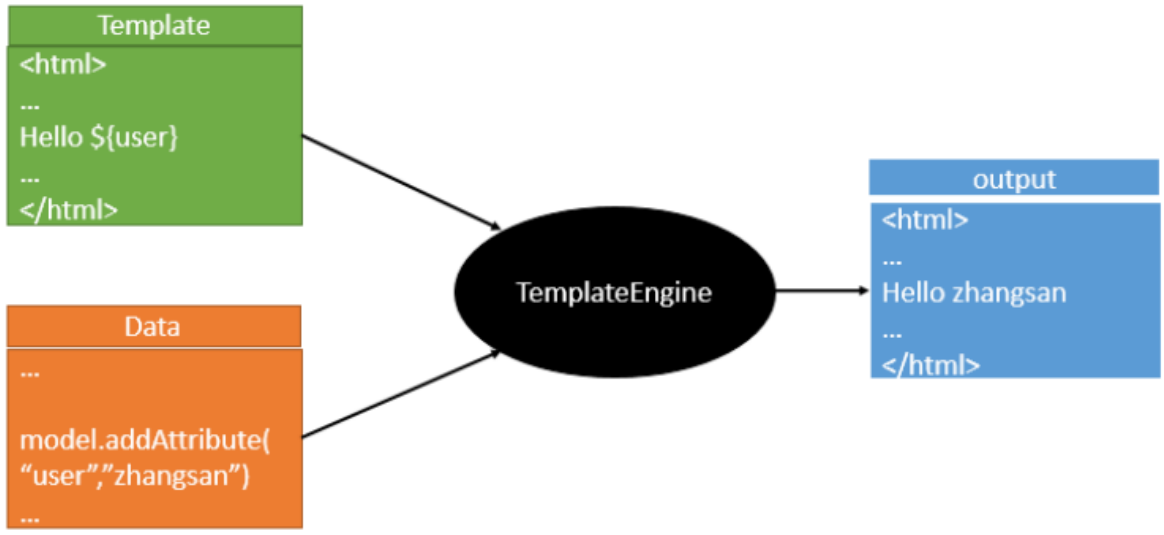
前端交给我们的页面，是html页面。如果是我们以前开发，我们需要把他们转成jsp页面，jsp好处就是当我们查出一些数据转发到JSP页面以后，我们可以用jsp轻松实现数据的显示，及交互等。

jsp支持非常强大的功能，包括能写Java代码，但是呢，我们现在的这种情况，SpringBoot这个项目首先是以jar的方式，不是war，像第二，我们用的还是嵌入式的Tomcat，所以呢，**他现在默认是不支持jsp的**。

那不支持jsp，如果我们直接用纯静态页面的方式，那给我们开发会带来非常大的麻烦，那怎么办呢?

SpringBoot推荐你可以来使用模板引擎:

模板引擎，我们其实大家听到很多，其实jsp就是一个模板引擎，还有以用的比较多的freemarker，包括SpringBoot给我们推荐的Thymeleaf，模板引擎有非常多，但再多的模板引擎，他们的思想都是一样的，什么样一个思想呢我们来看一下这张图:



模板引擎的作用就是我们来写一个页面模板，比如有些值呢，是动态的，我们写一些表达式。而这些值，从哪来呢，就是我们在后台封装一些数据。然后把这个模板和这个数据交给我们模板引擎，模板引擎按照我们这个数据帮你把这表达式解析、填充到我们指定的位置，然后把这个数据最终生成一个我们想要的内容给我们写出去，这就是我们这个模板引擎，不管是jsp还是其他模板引擎，都是这个思想。只不过呢，就是说不同模板引擎之间，他们可能这个语法有点不一样。其他的我就不介绍了，我主要来介绍一下SpringBoot给我们推荐的Thymeleaf模板引擎，这模板引擎呢，是一个高级语言的模板引擎，他的这个语法更简单。而且呢，功能更强大。

我们呢，就来看一下这个模板引擎，那既然要看这个模板引擎。首先，我们来看SpringBoot里边怎么用。

引入Thymeleaf

怎么引入呢，对于springboot来说，什么事情不都是一个start的事情嘛，我们去在项目中引入一下。给大家三个网址:

Thymeleaf 官网：https://www.thymeleaf.org/

Thymeleaf 在Github 的主页：https://github.com/thymeleaf/thymeleaf

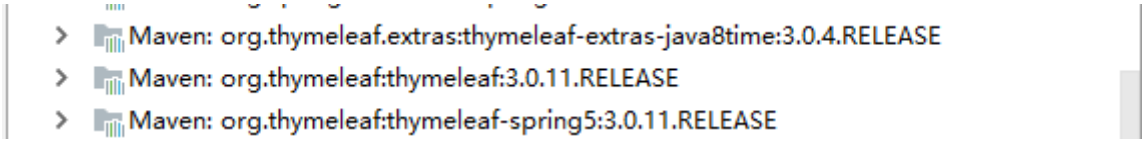
Spring官方文档： 找到我们对应的版本

https://docs.spring.io/spring-boot/docs/2.2.5.RELEASE/reference/htmlsingle/#using-boot-starter

找到对应的pom依赖：可以适当点进源码看下本来的包！

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

Maven会自动下载jar包，我们可以去看下下载的东西；



> thymeleaf 分析

前面呢，我们已经引入了Thymeleaf，那这个要怎么使用呢?

我们首先得按照SpringBoot的自动配置原理看一下我们这个Thymeleaf的自动配置规则，在按照那个规则，我们进行使用。

我们去找一下Thymeleaf的自动配置类：ThymeleafProperties

```
@ConfigurationProperties(
    prefix = "spring.thymeleaf"
)
public class ThymeleafProperties {
    private static final Charset DEFAULT_ENCODING;
    public static final String DEFAULT_PREFIX = "classpath:/templates/";
    public static final String DEFAULT_SUFFIX = ".html";
    private boolean checkTemplate = true;
    private boolean checkTemplateLocation = true;
    private String prefix = "classpath:/templates/";
    private String suffix = ".html";
    private String mode = "HTML";
    private Charset encoding;
}
```

我们可以在其中看到默认的前缀和后缀！

我们只需要把我们的html页面放在类路径下的templates下，thymeleaf就可以帮我们自动渲染了。

使用thymeleaf什么都不需要配置，只需要将他放在指定的文件夹下即可！

测试：

1、编写一个TestController

```java
@Controller
public class TestController {

    @RequestMapping("/t1")
    public String test1(){
        //classpath:/templates/test.html
        return "test";
    }

}
```

2、编写一个测试页面 test.html 放在 templates 目录下

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1>测试页面</h1>

</body>
</html>
```

3、启动项目请求测试

> Thymeleaf 语法学习

要学习语法，还是参考官网文档最为准确，我们找到对应的版本看一下；

Thymeleaf 官网：https://www.thymeleaf.org/ ， 简单看一下官网！我们去下载Thymeleaf的官方文档！

我们做个最简单的练习： 我们需要查出一些数据，在页面中展示

1、修改测试请求，增加数据传输；

```java
@RequestMapping("/t1")
public String test1(Model model){
    //存入数据
    model.addAttribute("msg","Hello,Thymeleaf");
    //classpath:/templates/test.html
    return "test";
}
```

2、我们要使用thymeleaf，需要在html文件中导入命名空间的约束，方便提示。

我们可以去官方文档的#3中看一下命名空间拿来过来：

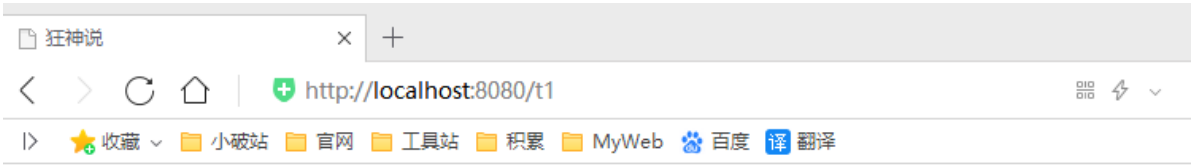
```
 xmlns:th="http://www.thymeleaf.org"
```

3、我们去编写下前端页面

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>狂神说</title>
</head>
<body>
<h1>测试页面</h1>

<!--th:text就是将div中的内容设置为它指定的值，和之前学习的Vue一样-->
<div th:text="${msg}"></div>
</body>
</html>
```

4、启动测试！



测试页面

Hello,Thymeleaf

OK，入门搞定，我们来认真研习一下Thymeleaf的使用语法！

1、我们可以使用任意的 th:attr 来替换Html中原生属性的值！参考官网文档#10； th语法

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 转义特殊字符 th:utext 不转义特殊字符
8	Fragment specification	声明片段	th:fragment
9	Fragment removal		th:remove

2、我们能写那些表达式呢？我们可以看到官方文档 #4

```
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：  #18
         #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.

    3）、内置的一些工具对象：
          #execInfo : information about the template being processed.
          #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.
================================================================================

    Selection Variable Expressions: *{...}：选择表达式：和${}在功能上是一样；
    Message Expressions: #{...}：获取国际化内容
    Link URL Expressions: @{...}：定义URL；
    Fragment Expressions: ~{...}：片段引用表达式

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: _
```

练习测试：

1、 我们编写一个Controller，放一些数据

```
@RequestMapping("/t2")
public String test2(Map<String,Object> map){
    //存入数据
    map.put("msg","<h1>Hello</h1>");
    map.put("users", Arrays.asList("qinjiang","kuangshen"));
    //classpath:/templates/test.html
    return "test";
}
```

2、测试页面取出数据

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>狂神说</title>
</head>
<body>
<h1>测试页面</h1>

<div th:text="${msg}"></div>
<!--不转义-->
<div th:utext="${msg}"></div>

<!--遍历数据-->
<!--th:each每次遍历都会生成当前这个标签：官网#9-->
<h4 th:each="user :${users}" th:text="${user}"></h4>

<h4>
    <!--行内写法：官网#12-->
    <span th:each="user:${users}">[[${user}]]</span>
</h4>

</body>
</html>
```

3、启动项目测试！

我们看完语法，很多样式，我们即使现在学习了，也会忘记，所以我们在学习过程中，需要使用什么，根据官方文档来查询，才是最重要的，要熟练使用官方文档！

MVC自动配置原理

> 官网阅读

在进行项目编写前，我们还需要知道一个东西，就是SpringBoot对我们的SpringMVC还做了哪些配置，包括如何扩展，如何定制。

只有把这些都搞清楚了，我们在之后使用才会更加得心应手。 途径一：源码分析，途径二：官方文档!

地址：https://docs.spring.io/spring-boot/docs/2.2.5.RELEASE/reference/htmlsingle/#boot-features-spring-mvc-auto-configuration

```
Spring MVC Auto-configuration
// Spring Boot为Spring MVC提供了自动配置，它可以很好地与大多数应用程序一起工作。
Spring Boot provides auto-configuration for Spring MVC that works well with most applications.
// 自动配置在Spring默认设置的基础上添加了以下功能：
The auto-configuration adds the following features on top of Spring's defaults:
// 包含视图解析器
Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans.
// 支持静态资源文件夹的路径，以及webjars
Support for serving static resources, including support for WebJars
// 自动注册了Converter：
// 转换器，这就是我们网页提交数据到后台自动封装成为对象的东西，比如把"1"字符串自动转换为int类型
// Formatter：【格式化器，比如页面给我们了一个2019-8-10，它会给我们自动格式化为Date对象】
Automatic registration of Converter, GenericConverter, and Formatter beans.
// HttpMessageConverters
// SpringMVC用来转换Http请求和响应的的，比如我们要把一个User对象转换为JSON字符串，可以去看官网文档解释；
Support for HttpMessageConverters (covered later in this document).
// 定义错误代码生成规则的
Automatic registration of MessageCodesResolver (covered later in this document).
// 首页定制
Static index.html support.
// 图标定制
Custom Favicon support (covered later in this document).
// 初始化数据绑定器：帮我们把请求数据绑定到JavaBean中！
Automatic use of a ConfigurableWebBindingInitializer bean (covered later in this document).

/*
如果您希望保留Spring Boot MVC功能，并且希望添加其他MVC配置（拦截器、格式化程序、视图控制器和其他功能），则可以添加自己
的@configuration类，类型为webmvcconfiguer，但不添加@EnableWebMvc。如果希望提供RequestMappingHandlerMapping、RequestMappingHandlerAdapter或ExceptionHandlerExceptionResolver的自定义
实例，则可以声明WebMVCregistrationAdapter实例来提供此类组件。
*/
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.

// 如果您想完全控制Spring MVC，可以添加自己的@Configuration，并用@EnableWebMvc进行
注释。
If you want to take complete control of Spring MVC, you can add your own
@Configuration annotated with @EnableWebMvc.
```

我们来仔细对照，看一下它怎么实现的，它告诉我们SpringBoot已经帮我们自动配置好了SpringMVC，然后自动配置了哪些东西呢？

> **ContentNegotiatingViewResolver 内容协商视图解析器**

自动配置了ViewResolver，就是我们之前学习的SpringMVC的视图解析器；

即根据方法的返回值取得视图对象（View），然后由视图对象决定如何渲染（转发，重定向）。

我们去看看这里的源码：我们找到 WebMvcAutoConfiguration， 然后搜索 ContentNegotiatingViewResolver。找到如下方法！

```
@Bean
@ConditionalOnBean(ViewResolver.class)
@ConditionalOnMissingBean(name = "viewResolver", value =
ContentNegotiatingViewResolver.class)
public ContentNegotiatingViewResolver viewResolver(BeanFactory beanFactory)
{
    ContentNegotiatingViewResolver resolver = new
ContentNegotiatingViewResolver();

 resolver.setContentNegotiationManager(beanFactory.getBean(ContentNegotiatio
nManager.class));
    // ContentNegotiatingViewResolver使用所有其他视图解析器来定位视图，因此它应该具
有较高的优先级
    resolver.setOrder(Ordered.HIGHEST_PRECEDENCE);
    return resolver;
}
```

我们可以点进这类看看！找到对应的解析视图的代码；

```
@Nullable // 注解说明：@Nullable 即参数可为null
public View resolveViewName(String viewName, Locale locale) throws Exception
{
    RequestAttributes attrs = RequestContextHolder.getRequestAttributes();
    Assert.state(attrs instanceof ServletRequestAttributes, "No current
ServletRequestAttributes");
    List<MediaType> requestedMediaTypes =
this.getMediaTypes(((ServletRequestAttributes)attrs).getRequest());
    if (requestedMediaTypes != null) {
        // 获取候选的视图对象
        List<View> candidateViews = this.getCandidateViews(viewName, locale,
requestedMediaTypes);
        // 选择一个最适合的视图对象，然后把这个对象返回
```

```
        View bestView = this.getBestView(candidateViews,
requestedMediaTypes, attrs);
        if (bestView != null) {
            return bestView;
        }
    }
    // .....
}
```

我们继续点进去看，他是怎么获得候选的视图的呢?

getCandidateViews中看到他是把所有的视图解析器拿来，进行while循环，挨个解析!

```
Iterator var5 = this.viewResolvers.iterator();
```

所以得出结论：**ContentNegotiatingViewResolver 这个视图解析器就是用来组合所有的视图解析器的**

我们再去研究下他的组合逻辑，看到有个属性viewResolvers，看看它是在哪里进行赋值的!

```
protected void initServletContext(ServletContext servletContext) {
    // 这里它是从beanFactory工具中获取容器中的所有视图解析器
    // ViewRescolver.class 把所有的视图解析器来组合的
    Collection<ViewResolver> matchingBeans =
BeanFactoryUtils.beansOfTypeIncludingAncestors(this.obtainApplicationContext
(), ViewResolver.class).values();
    ViewResolver viewResolver;
    if (this.viewResolvers == null) {
        this.viewResolvers = new ArrayList(matchingBeans.size());
    }
    // ..............
}
```

既然它是在容器中去找视图解析器，我们是否可以猜想，我们就可以去实现一个视图解析器了呢?

我们可以自己给容器中去添加一个视图解析器；这个类就会帮我们自动的将它组合进来；**我们去实现一下**

1、我们在我们的主程序中去写一个视图解析器来试试；

```
@Bean //放到bean中
public ViewResolver myViewResolver(){
    return new MyViewResolver();
}

//我们写一个静态内部类，视图解析器就需要实现ViewResolver接口
private static class MyViewResolver implements ViewResolver{
    @Override
    public View resolveViewName(String s, Locale locale) throws Exception {
        return null;
    }
}
```

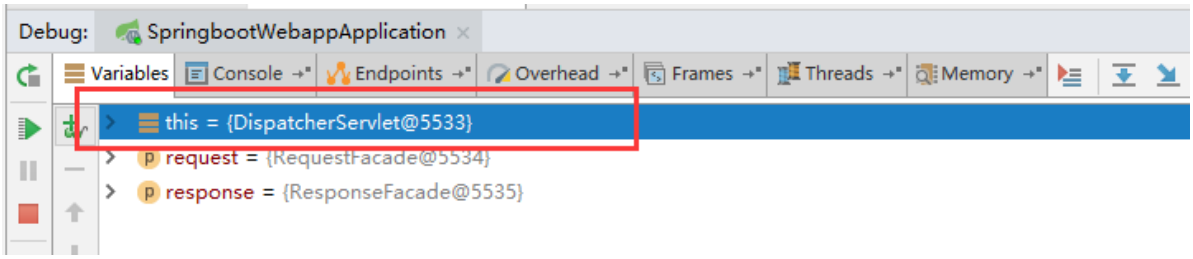
2、怎么看我们自己写的视图解析器有没有起作用呢?

我们给 DispatcherServlet 中的 doDispatch方法 加个断点进行调试一下，因为所有的请求都会走到这个方法中 //启动项目后随意发起一个请求,然后项目会进入debug运行中。

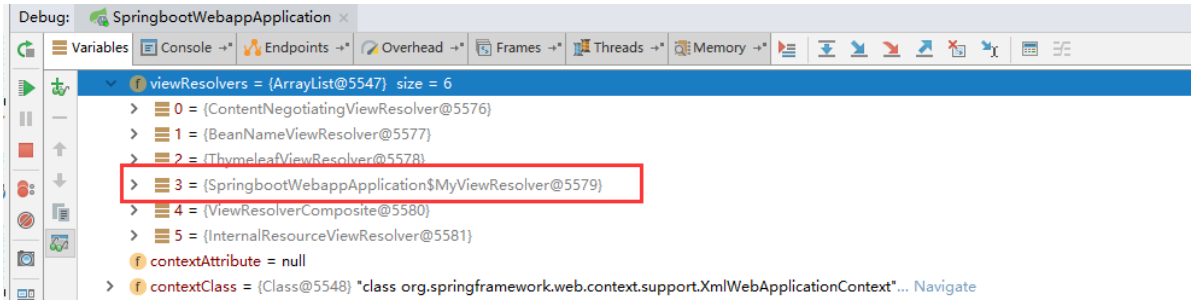
```
289
290     private void doDispatch(ServletRequest request, ServletResponse response) throws ServletException, IOException {
291         ApplicationDispatcher.State state = new ApplicationDispatcher.State(request, response, including: false);
292         this.wrapResponse(state);
```

3、我们启动我们的项目，然后随便访问一个页面，看一下Debug信息；

找到this



找到视图解析器，我们看到我们自己定义的就在这里了；



所以说，我们如果想要使用自己定制化的东西，我们只需要给容器中添加这个组件就好了！剩下的事情SpringBoot就会帮我们做了

> 转换器和格式化器

找到格式化转换器：

```java
@Bean
@Override
public FormattingConversionService mvcConversionService() {
    // 拿到配置文件中的格式化规则
    WebConversionService conversionService =
        new WebConversionService(this.mvcProperties.getDateFormat());
    addFormatters(conversionService);
    return conversionService;
}
```

点击去：

```java
public String getDateFormat() {
    return this.dateFormat;
}

/**
* Date format to use. For instance, `dd/MM/yyyy`. 默认的
 */
private String dateFormat;
```

可以看到在我们的Properties文件中，我们可以进行自动配置它！

如果配置了自己的格式化方式，就会注册到Bean中生效，我们可以在配置文件中配置日期格式化的规则：

```
spring.mvc.date-format=
```

```
public void setDateFormat(String dateFormat) {
    this.dateFormat = dateFormat;
}
```

其余的就不一一举例了，大家可以下去多研究探讨即可！

> 修改SpringBoot的默认配置

这么多的自动配置，原理都是一样的，通过这个WebMVC的自动配置原理分析，我们要学会一种学习方式，通过源码探究，得出结论；这个结论一定是属于自己的，而且一通百通。

SpringBoot的底层，大量用到了这些设计细节思想，所以，没事需要多阅读源码！得出结论；

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（如果用户自己配置@bean），如果有就用用户配置的，如果没有就用自动配置的；

如果有些组件可以存在多个，比如我们的视图解析器，就将用户配置的和自己默认的组合起来！

扩展使用SpringMVC 官方文档如下：

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.

我们要做的就是编写一个@Configuration注解类，并且类型要为WebMvcConfigurer，还不能标注@EnableWebMvc注解；我们去自己写一个；我们新建一个包叫config，写一个类MyMvcConfig；

```
//应为类型要求为WebMvcConfigurer，所以我们实现其接口
//可以使用自定义类扩展MVC的功能
@Configuration
public class MyMvcConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        // 浏览器发送/test ，  就会跳转到test页面；
        registry.addViewController("/test").setViewName("test");
    }
}
```

我们去浏览器访问一下：

测试页面

确实也跳转过来了！所以说，我们要扩展SpringMVC，官方就推荐我们这么去使用，既保SpringBoot留所有的自动配置，也能用我们扩展的配置！

我们可以去分析一下原理：

1、WebMvcAutoConfiguration 是 SpringMVC的自动配置类，里面有一个类 WebMvcAutoConfigurationAdapter

2、这个类上有一个注解，在做其他自动配置时会导入：

`@Import(EnableWebMvcConfiguration.class)`

3、我们点进EnableWebMvcConfiguration这个类看一下，它继承了一个父类：DelegatingWebMvcConfiguration

这个父类中有这样一段代码：

```
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {
    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);
        }
    }
}
```

4、我们可以在这个类中去寻找一个我们刚才设置的viewController当做参考，发现它调用了一个

```
protected void addViewControllers(ViewControllerRegistry registry) {
    this.configurers.addViewControllers(registry);
}
```

5、我们点进去看一下

```
public void addViewControllers(ViewControllerRegistry registry) {
    Iterator var2 = this.delegates.iterator();

    while(var2.hasNext()) {
        // 将所有的WebMvcConfigurer相关配置来一起调用！包括我们自己配置的和Spring给我们配置的
        WebMvcConfigurer delegate = (WebMvcConfigurer)var2.next();
        delegate.addViewControllers(registry);
    }

}
```

所以得出结论：所有的WebMvcConfiguration都会被作用，不止Spring自己的配置类，我们自己的配置类当然也会被调用；

> 全面接管SpringMVC

官方文档：

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

全面接管即：SpringBoot对SpringMVC的自动配置不需要了，所有都是我们自己去配置！

只需在我们的配置类中要加一个@EnableWebMvc。

我们看下如果我们全面接管了SpringMVC了，我们之前SpringBoot给我们配置的静态资源映射一定会无效，我们可以去测试一下；

不加注解之前，访问首页：

给配置类加上注解：`@EnableWebMvc`

我们发现所有的SpringMVC自动配置都失效了！回归到了最初的样子；

当然，我们开发中，不推荐使用全面接管SpringMVC

思考问题？为什么加了一个注解，自动配置就失效了！我们看下源码：

1、这里发现它是导入了一个类，我们可以继续进去看

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

2、它继承了一个父类 WebMvcConfigurationSupport

```
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport
{
    // ......
}
```

3、我们来回顾一下Webmvc自动配置类

```java
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class,
WebMvcConfigurer.class })
// 这个注解的意思就是：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,
TaskExecutionAutoConfiguration.class,
        ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {

}
```

总结一句话：@EnableWebMvc将WebMvcConfigurationSupport组件导入进来了；

而导入的WebMvcConfigurationSupport只是SpringMVC最基本的功能！

在SpringBoot中会有非常多的扩展配置，只要看见了这个，我们就应该多留心注意~

配置项目环境及首页

> 把昨天的 mybatis 整合代码拿过来

1、导入依赖

```xml
<!-- lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
<!-- 数据层 -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.1.1</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

2、导入实体类

```java
@Data
public class Department {

    private Integer id;
    private String departmentName;

}
```

```java
@Data
public class Employee {

    private Integer id;
    private String lastName;
    private String email;
    //1 male, 0 female
    private Integer gender;
    private Integer department;
    private Date birth;

    private Department eDepartment;
}
```

3、导入mapp接口以及对应的配置文件

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.kuang.mapper.DepartmentMapper">

    <select id="getDepartments" resultType="Department">
       select * from department;
    </select>

    <select id="getDepartment" resultType="Department" parameterType="int">
       select * from department where id = #{id};
    </select>

</mapper>
```

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.kuang.mapper.EmployeeMapper">

    <resultMap id="EmployeeMap" type="Employee">
        <id property="id" column="eid"/>
        <result property="lastName" column="last_name"/>
        <result property="email" column="email"/>
        <result property="gender" column="gender"/>
        <result property="birth" column="birth"/>
        <association property="eDepartment"  javaType="Department">
            <id property="id" column="did"/>
            <result property="departmentName" column="dname"/>
        </association>
    </resultMap>

    <select id="getEmployees" resultMap="EmployeeMap">
        select e.id as eid,last_name,email,gender,birth,d.id as
did,d.department_name as dname
        from department d,employee e
        where d.id = e.department
```

```
    </select>

    <insert id="save" parameterType="Employee">
        insert into employee (last_name,email,gender,department,birth)
        values (#{lastName},#{email},#{gender},#{department},#{birth});
    </insert>

    <select id="get" resultType="Employee">
        select * from employee where id = #{id}
    </select>

    <delete id="delete" parameterType="int">
        delete from employee where id = #{id}
    </delete>

</mapper>
```

4、注意Maven资源导出问题!

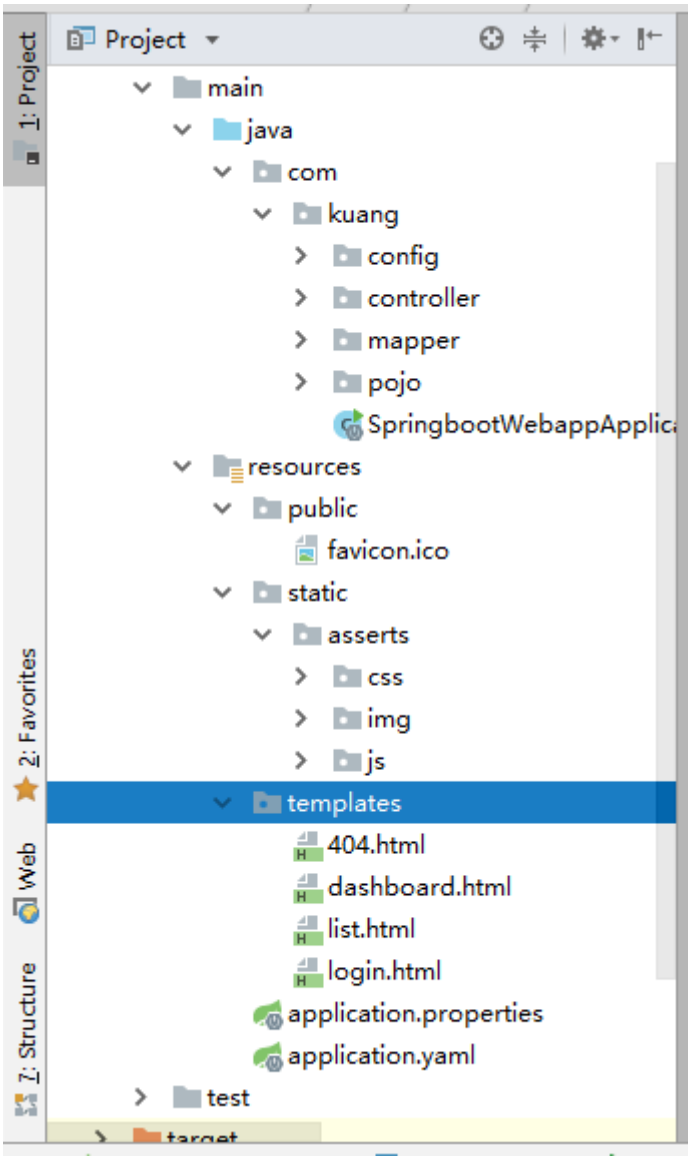
```
<resources>
    <resource>
        <directory>src/main/java</directory>
        <includes>
            <include>**/*.xml</include>
        </includes>
        <filtering>true</filtering>
    </resource>
</resources>
```

> 导入静态资源

1、css，js等放在static文件夹下

2、html 放在 templates文件夹下

最终结构如下:



> 首页实现

方式一：写一个controller实现！

```
//会解析到templates目录下的index.html页面
@RequestMapping({"/","/index.html"})
public String index(){
    return "index";
}
```

方式二：自己编写MVC的扩展配置

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

解决了首页问题，我们还需要解决一个资源导入的问题；

为了保证资源导入稳定，我们建议在所有资源导入时候使用 th:去替换原有的资源路径！这也是模板规范

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<link th:href="@{/asserts/css/bootstrap.min.css}" rel="stylesheet">
```

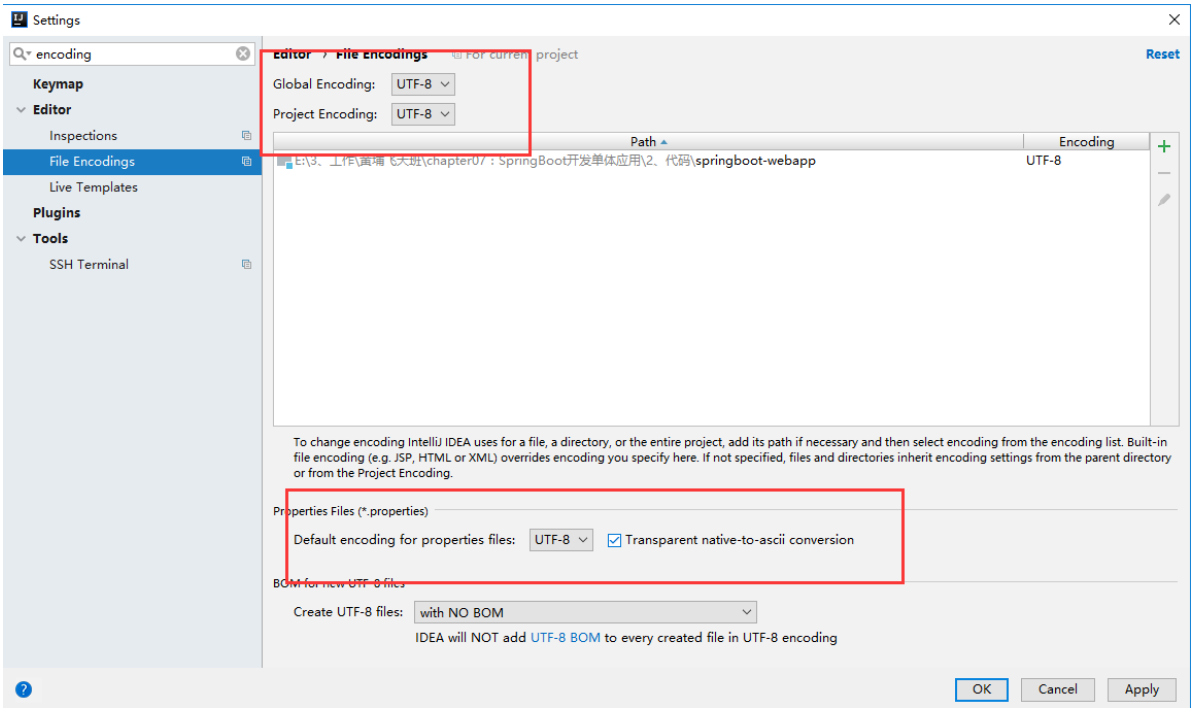
ok，如果都替换完成了，我们的准备工作也就全部结束了！

页面国际化

有的时候，我们的网站会去涉及中英文甚至多语言的切换，这时候我们就需要学习国际化了！

> 准备工作

先在IDEA中统一设置properties的编码问题！

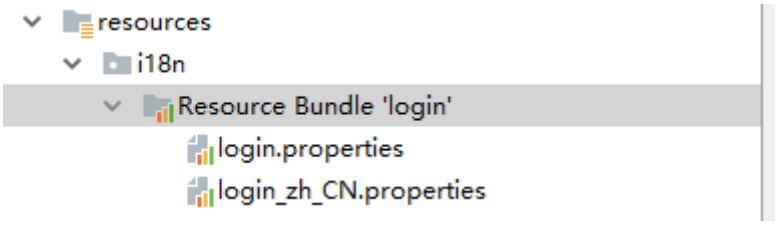


编写国际化配置文件，抽取页面需要显示的国际化页面消息。我们可以去登录页面查看一下，哪些内容我们需要编写国际化的配置！

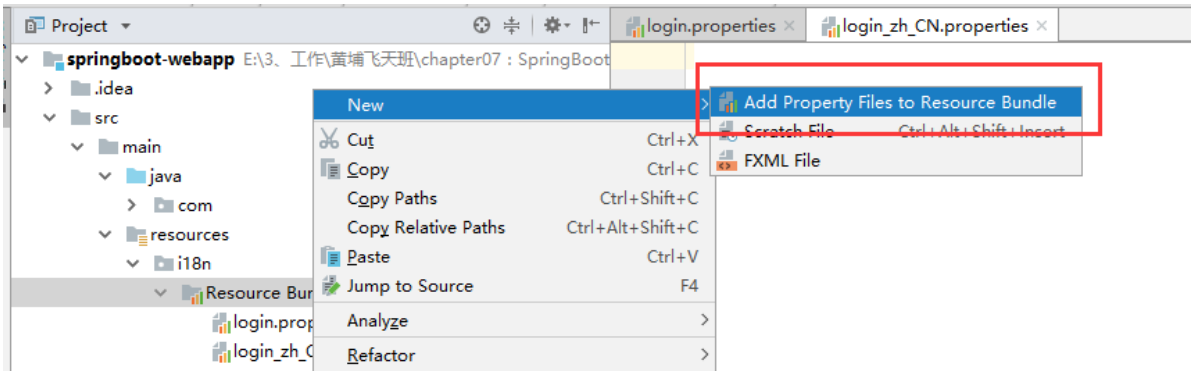
> 配置文件编写

1、我们在resources资源文件下新建一个i18n目录，存放国际化配置文件

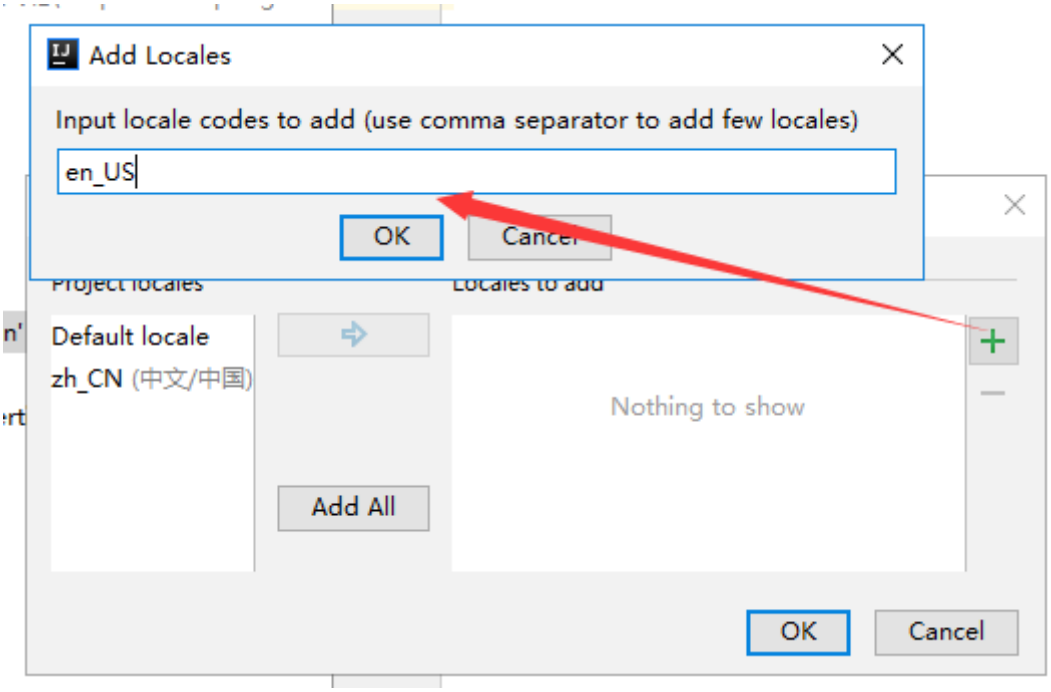
2、建立一个login.properties文件，还有一个login_zh_CN.properties；发现IDEA自动识别了我们要做国际化操作；文件夹变了！



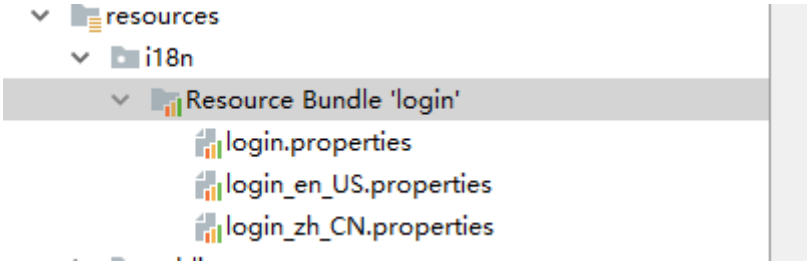
3、我们可以在这上面去新建一个文件；



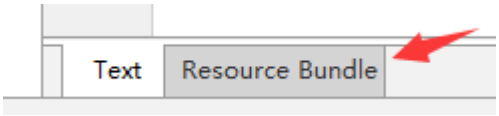
弹出如下页面：我们再添加一个英文的；



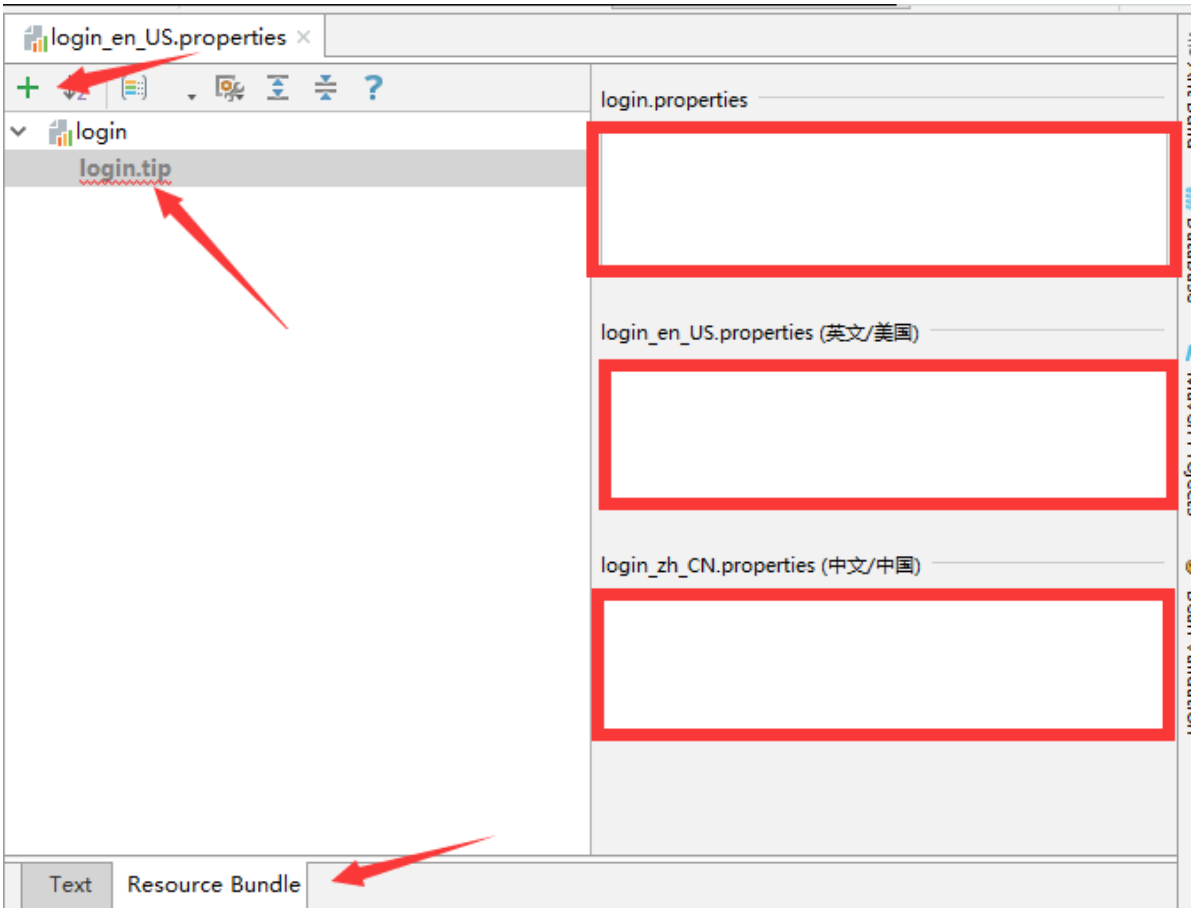
这样就快捷多了！



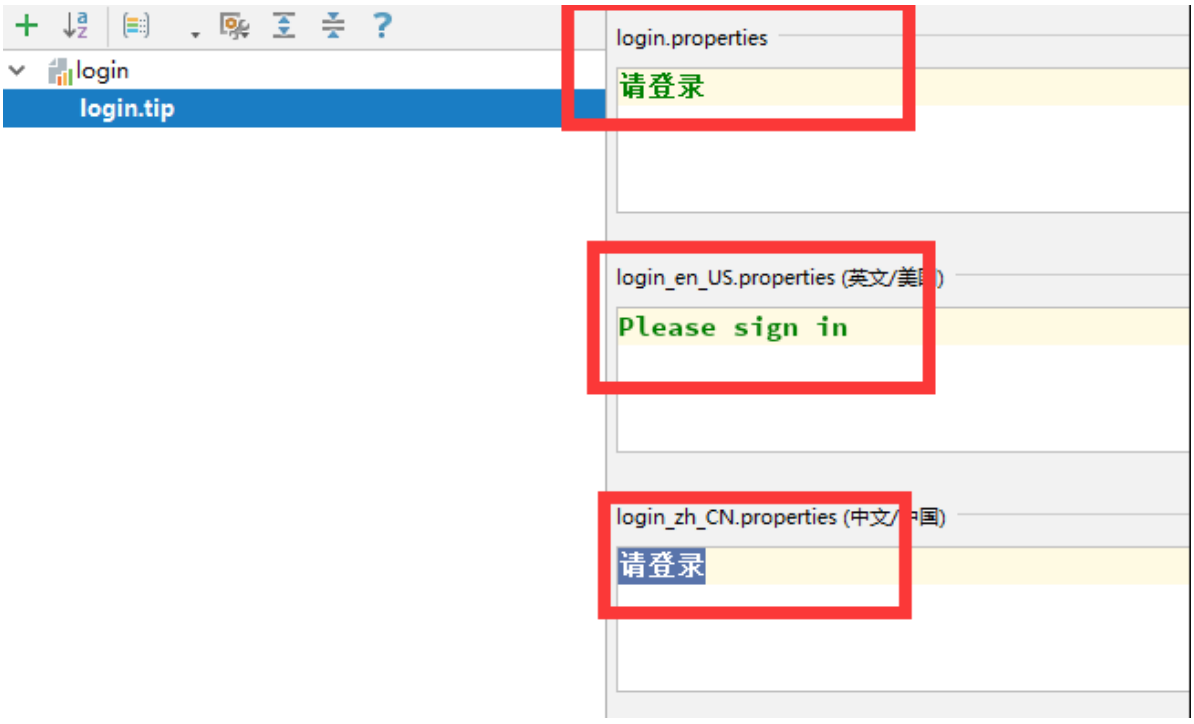
4、接下来，我们就来编写配置，我们可以看到idea下面有另外一个视图；



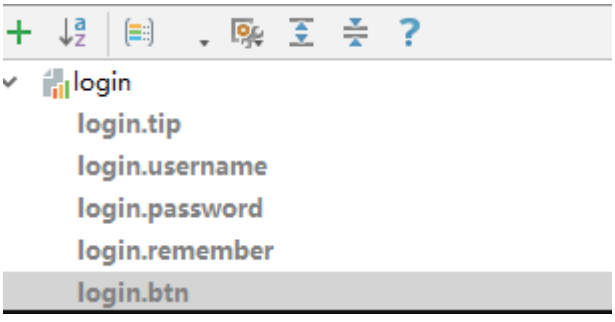
这个视图我们点击 + 号就可以直接添加属性了；我们新建一个login.tip，可以看到边上有三个文件框可以输入



我们添加一下首页的内容!



然后依次添加其他页面内容即可!



然后去查看我们的配置文件；

login.properties： 默认

```
login.btn=登录
login.password=密码
login.remember=记住我
login.tip=请登录
login.username=用户名
```

英文：

```
login.btn=Sign in
login.password=Password
login.remember=Remember me
login.tip=Please sign in
login.username=Username
```

中文：

```
login.btn=登录
login.password=密码
login.remember=记住我
login.tip=请登录
login.username=用户名
```

OK，配置文件步骤搞定！

> 配置文件生效探究

我们去看一下SpringBoot对国际化的自动配置！这里又涉及到一个类：
`MessageSourceAutoConfiguration`

里面有一个方法，这里发现SpringBoot已经自动配置好了管理我们国际化资源文件的组件ResourceBundleMessageSource；

```
// 获取 properties 传递过来的值进行判断
@Bean
public MessageSource messageSource(MessageSourceProperties properties) {
    ResourceBundleMessageSource messageSource = new
ResourceBundleMessageSource();
    if (StringUtils.hasText(properties.getBasename())) {
        // 设置国际化文件的基础名（去掉语言国家代码的）
        messageSource.setBasenames(
            StringUtils.commaDelimitedListToStringArray(

StringUtils.trimAllWhitespace(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.isUseCodeAsDefaultMessa
ge());
    return messageSource;
}
```

我们真实 的情况是放在了i18n目录下，所以我们要去配置这个messages的路径；

```
spring.messages.basename=i18n.login
```

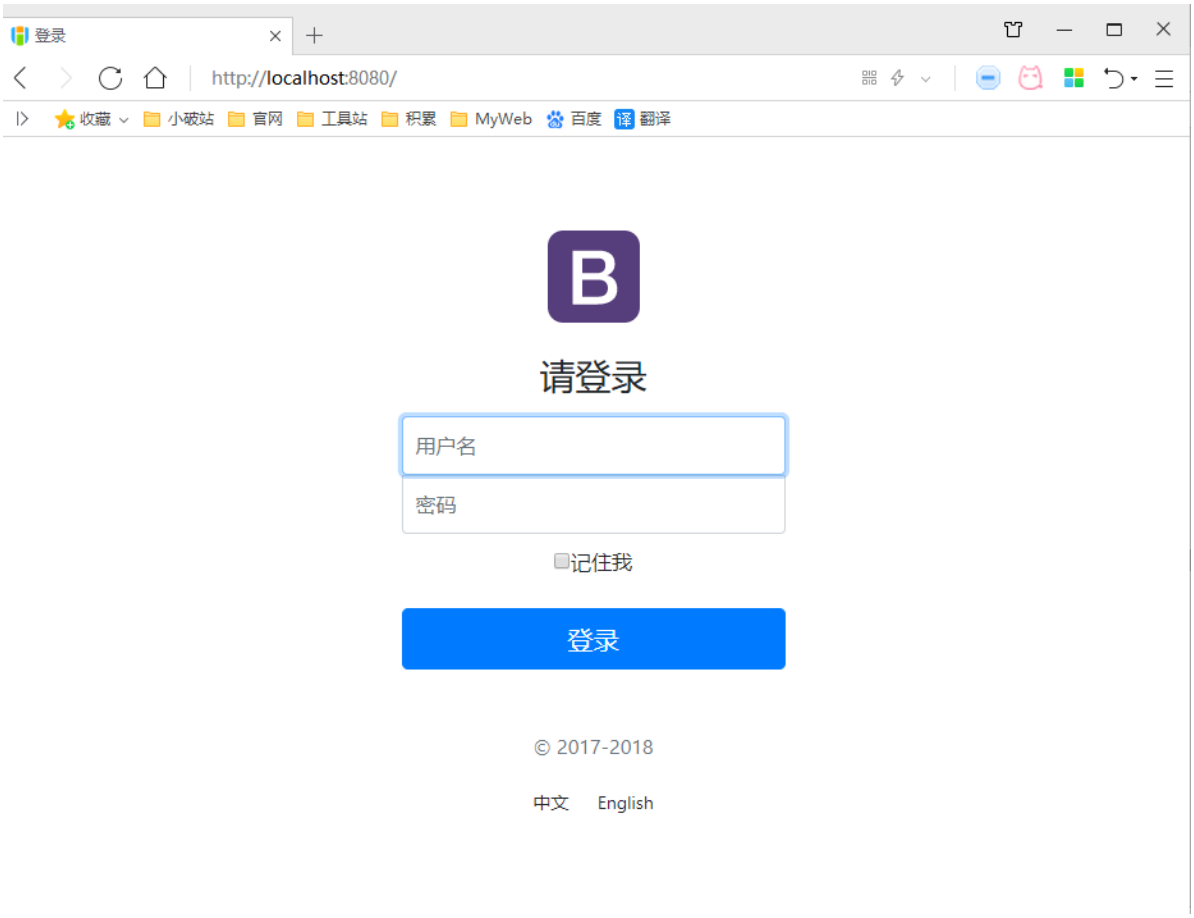
> 配置页面国际化值

去页面获取国际化的值，查看Thymeleaf的文档，找到message取值操作为： #{...}。我们去页面测试下：

IDEA还有提示，非常智能的！

```
<body class="text-center">
    <form class="form-signin" action="dashboard.html">
        <img class="mb-4"  th:src="@{/asserts/img/bootstrap-solid.svg}" alt="" width="72" height="72">
        <h1 class="h3 mb-3 font-weight-normal" th:text="#{login.tip}">Please sign in</h1>
        <label class="sr-only">Username</label>
        <input type="text" class="form-control" th:placeholder="#{login.username}" required="" autofocus="">
        <label class="sr-only">Password</label>
        <input type="password" class="form-control" th:placeholder="#{login.password}" required="">
        <div class="checkbox mb-3">
            <label>
      <input type="checkbox" value="remember-me" th:text="#{login.remember}">
    </label>
        </div>
        <button class="btn btn-lg btn-primary btn-block" type="submit" th:text="#{login.btn}">Sign in</button>
        <p class="mt-5 mb-3 text-muted">© 2017-2018</p>
        <a class="btn btn-sm">中文</a>
        <a class="btn btn-sm">English</a>
```

我们可以去启动项目，访问一下，发现已经自动识别为中文的了！



但是我们想要更好！可以根据按钮自动切换中文英文！

> 配置国际化解析

在Spring中有一个国际化的Locale （区域信息对象）；里面有一个叫做LocaleResolver （获取区域信息对象）的解析器！

我们去我们webmvc自动配置文件，寻找一下！看到SpringBoot默认配置：

```
@Bean
@ConditionalOnMissingBean
@ConditionalOnProperty(prefix = "spring.mvc", name = "locale")
public LocaleResolver localeResolver() {
    // 容器中没有就自己配，有的话就用用户配置的
    if (this.mvcProperties.getLocaleResolver() ==
WebMvcProperties.LocaleResolver.FIXED) {
        return new FixedLocaleResolver(this.mvcProperties.getLocale());
    }
    // 接收头国际化分解
    AcceptHeaderLocaleResolver localeResolver = new
AcceptHeaderLocaleResolver();
    localeResolver.setDefaultLocale(this.mvcProperties.getLocale());
    return localeResolver;
}
```

AcceptHeaderLocaleResolver 这个类中有一个方法

```
public Locale resolveLocale(HttpServletRequest request) {
    Locale defaultLocale = this.getDefaultLocale();
    // 默认的就是根据请求头带来的区域信息获取Locale进行国际化
```

```
    if (defaultLocale != null && request.getHeader("Accept-Language") == null) {
        return defaultLocale;
    } else {
        Locale requestLocale = request.getLocale();
        List<Locale> supportedLocales = this.getSupportedLocales();
        if (!supportedLocales.isEmpty() && !supportedLocales.contains(requestLocale)) {
            Locale supportedLocale = this.findSupportedLocale(request, supportedLocales);
            if (supportedLocale != null) {
                return supportedLocale;
            } else {
                return defaultLocale != null ? defaultLocale : requestLocale;
            }
        } else {
            return requestLocale;
        }
    }
}
```

那假如我们现在想点击链接让我们的国际化资源生效，就需要让我们自己的Locale生效！

我们去自己写一个自己的LocaleResolver，可以在链接上携带区域信息！

修改一下前端页面的跳转连接：

```
<!-- 这里传入参数不需要使用 ？  使用 （key=value）-->
<a class="btn btn-sm" th:href="@{/index.html(l='zh_CN')}">中文</a>
<a class="btn btn-sm" th:href="@{/index.html(l='en_US')}">English</a>
```

我们去写一个处理的组件类！

```
package com.kuang.component;

import org.springframework.util.StringUtils;
import org.springframework.web.servlet.LocaleResolver;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Locale;

//可以在链接上携带区域信息
public class MyLocaleResolver implements LocaleResolver {

    //解析请求
    @Override
    public Locale resolveLocale(HttpServletRequest request) {

        String language = request.getParameter("l");
        Locale locale = Locale.getDefault(); // 如果没有获取到就使用系统默认的
        //如果请求链接不为空
        if (!StringUtils.isEmpty(language)){
            //分割请求参数
            String[] split = language.split("_");
            //国家，地区
            locale = new Locale(split[0],split[1]);
```

```
        }
        return locale;
    }

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

    }
}
```

为了让我们的区域化信息能够生效，我们需要再配置一下这个组件！在我们自己的MvcConofig下添加bean；

```
@Bean
public LocaleResolver localeResolver(){
    return new MyLocaleResolver();
}
```

我们重启项目，来访问一下，发现点击按钮可以实现成功切换！搞定收工！

登录+拦截器

> 禁用模板缓存

说明：页面存在缓存，所以我们需要禁用模板引擎的缓存

```
#禁用模板缓存
spring.thymeleaf.cache=false
```

模板引擎修改后，想要实时生效！页面修改完毕后，IDEA小技巧： Ctrl + F9 重新编译！即可生效！

> 登录

我们这里就先不连接数据库了，输入任意用户名都可以登录成功！

1、我们把登录页面的表单提交地址写一个controller！

```
<form class="form-signin" th:action="@{/user/login}" method="post">
//这里面的所有表单标签都需要加上一个name属性

</form>
```

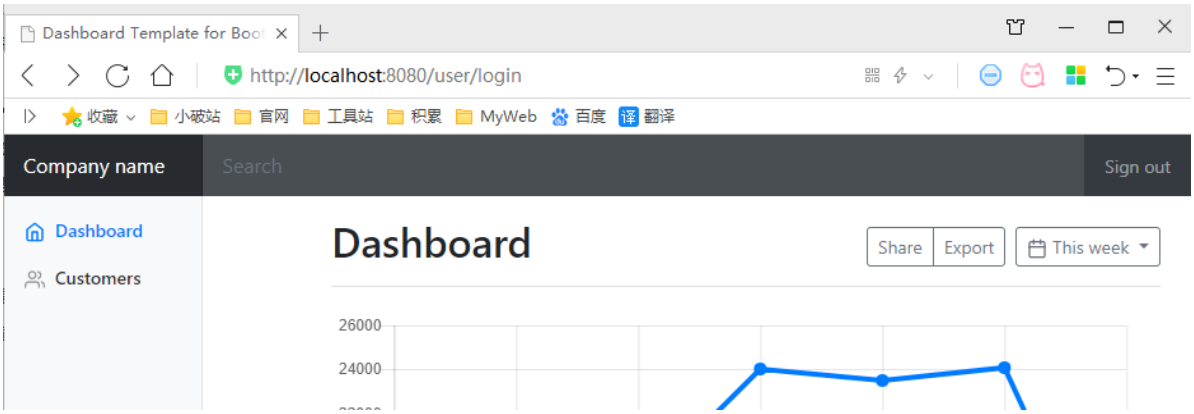
2、去编写对应的controller

```
@Controller
public class LoginController {

    //@RequestMapping(value = "/user/login",method = RequestMethod.POST)
    @PostMapping("/user/login") //相当于使用Post方法的requestmapping注解
    public String login(@RequestParam("username") String username,
                        @RequestParam("password") String password,
                        Model model, HttpSession session){
```

```
9
10          if (!StringUtils.isEmpty(username) && "123456".equals(password)){
11              //登录成功！将用户信息放入session
12              session.setAttribute("loginUser",username);
13              return "dashboard"; // 跳转到首页
14          }else {
15              //登录失败！存放错误信息
16              model.addAttribute("msg","用户名密码错误");
17              return "index";
18          }
19
20      }
21
22  }
```

OK，测试登录成功！



3、登录失败的话，我们需要将后台信息输出到前台，可以在首页标题下面加上判断！

```
1  <!--判断是否显示，使用if，${}可以使用工具类，可以看thymeleaf的中文文档-->
2  <p style="color: red" th:text="${msg}" th:if="${not #strings.isEmpty(msg)}">
   </p>
```

重启登录失败测试：

优化，登录成功后，由于是转发，链接不变，我们可以重定向到首页！

4、我们再添加一个视图控制映射，在我们的自己的MyMvcConfig中：

```
1  registry.addViewController("/main.html").setViewName("dashboard");
```

5、将 Controller 的代码改为重定向；

```
//登录成功！防止表单重复提交，我们重定向
return "redirect:/main.html";
```

重启测试，重定向成功！后台主页正常显示！

> 登录拦截器

但是又发现新的问题，我们可以直接登录到后台主页，不用登录也可以实现！怎么处理这个问题呢？我们可以使用拦截器机制，实现登录检查！

1、我们先自定义一个拦截器：

```
public class LoginHandlerInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        // 获取 loginUser 信息进行判断
        Object user = request.getSession().getAttribute("loginUser");
        if (user == null){ // 未登录，返回登录页面
            request.setAttribute("msg","没有权限，请先登录");
            request.getRequestDispatcher("/index.html").forward(request,response);
            return false;
        }else {
            // 登录，放行
            return true;
        }
    }
}
```

2、然后将拦截器注册到我们的SpringMVC配置类当中！

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    // 注册拦截器，及拦截请求和要剔除哪些请求！
    // 我们还需要过滤静态资源文件，否则样式显示不出来
    registry.addInterceptor(new LoginHandlerInterceptor())
        .addPathPatterns("/**")
        .excludePathPatterns("/index.html","/","/user/login","/asserts/**");
}
```

3、我们然后在后台主页，获取用户登录的信息

```
<!--后台主页显示登录用户的信息-->
[[${session.loginUser}]]
```

然后我们登录测试拦截！完美！

员工列表实现

> RestFul 风格

要求： 我们需要使用 Restful风格实现我们的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

看看一些具体的要求，就是我们小实验的架构；

实验功能	请求URI	请求方式
查询所有员工	emps	GET
查询某个员工(来到修改页面)	emp/1	GET
来到添加页面	emp	GET
添加员工	emp	POST
来到修改页面（查出员工进行信息回显）	emp/1	GET
修改员工	emp	PUT
删除员工	emp/1	DELETE

我们根据这些要求，来完成第一个功能，就是我们的员工列表功能！

> 员工列表页面跳转

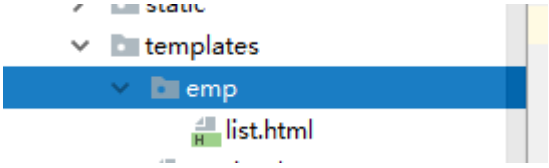
我们在主页点击Customers，就显示列表页面；我们去修改下

1、将首页的侧边栏Customers改为员工管理

2、a链接添加请求

```
<a class="nav-link" th:href="@{/emps}">员工管理</a>
```

3、将list放在emp文件夹下



4、编写处理请求的controller

```
@Controller
public class EmployeeController {

    @Autowired
    EmployeeMapper employeeMapper;

    //查询所有员工，返回列表页面
    @GetMapping("/emps")
```

```
    public String list(Model model){
        List<Employee> employees = employeeMapper.getEmployees();
        //将结果放在请求中
        model.addAttribute("emps",employees);
        return "emp/list";
    }

}
```

我们启动项目，测试一下看是否能够跳转，测试OK！我们只需要将数据渲染进去即可！

但是发现了一个问题，侧边栏和顶部都相同，我们是不是应该将它抽取出来呢？

> **Thymeleaf 公共页面元素抽取**　　//在A页面抽取公共片段,在B页面引入公共片段。

步骤：

1、抽取公共片段 th:fragment 定义模板名

2、引入公共片段 th:insert 插入模板名

实现：

1、我们来抽取一下，使用list列表做演示！我们要抽取头部nav标签，我们在dashboard中将nav部分定义一个模板名；

```
<!-- 定义th:fragment="topbar" -->
<nav th:fragment="topbar" class="navbar navbar-dark sticky-top bg-dark ">
    <!--后台主页显示登录用户的信息-->
    <a class="navbar-brand col-sm-3 col-md-2 mr-0" href="#">
        [[${session.loginUser}]]
    </a>
    <input class="form-control form-control-dark w-100" type="text"
placeholder="Search" aria-label="Search">
    <ul class="navbar-nav px-3">
        <li class="nav-item text-nowrap">
            <a class="nav-link" href="#">Sign out</a>
        </li>
    </ul>
</nav>
```

3、默认效果：
insert的公共片段在div标签中
如果使用th:insert等属性进行引入，可以不用写~{}：
行内写法可以加上：[[~{}]];[(~{})]；

2、然后我们在list页面中去引入，可以删掉原来的nav

```
<!--引入抽取的topbar-->
<!--模板名 ：  会使用thymeleaf的前后缀配置规则进行解析
使用~{模板::标签名}-->
<div th:insert="~{dashboard::topbar}"></div>
```

//将topbar引入到dashboard页面中。

3、启动再次测试，可以看到已经成功加载过来了！

说明：

除了使用insert插入，还可以使用replace替换，或者include包含，三种方式会有一些小区别，可以见名知义；

我们使用replace替换，可以解决div多余的问题，可以查看thymeleaf的文档学习

侧边栏也是同理，当做练手，可以也同步一下！

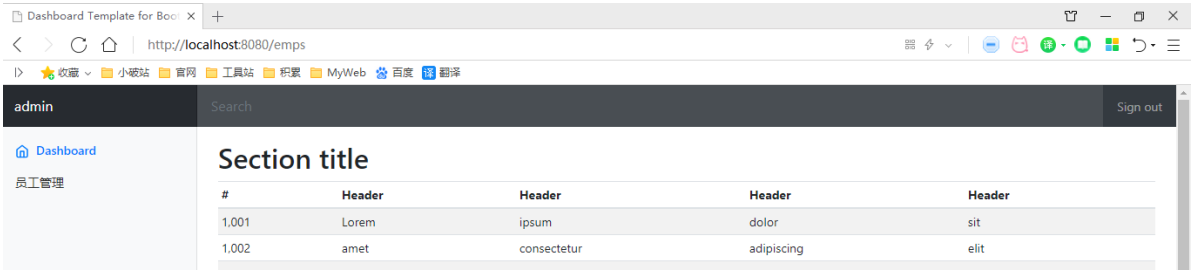
定义模板：

```
<nav th:fragment="sitebar" class="col-md-2 d-none d-md-block bg-light
sidebar">
```

然后我们在list页面中去引入：

```
<div th:insert="~{dashboard::sitebar}"></div>
```

启动再试试，看效果！

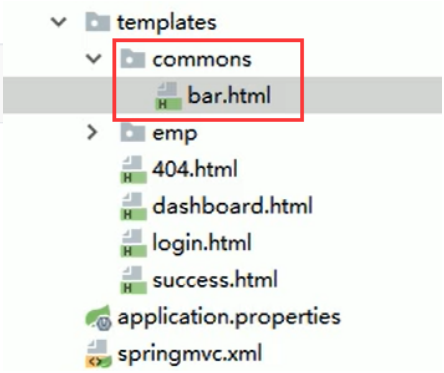


我们发现一个小问题，侧边栏激活的问题，它总是激活第一个；按理来说，这应该是动态的才对！

为了重用更清晰，我们建立一个commons文件夹，专门存放公共页面；

我们去页面中引入一下

```
<div th:replace="~{commons/bar::topbar}"></div>
<div th:replace="~{commons/bar::sitebar}"></div>
```



我们先测试一下，保证所有的页面没有出问题！ok！

侧边栏激活问题：

1、将首页的超链接地址改到项目中

2、我们在a标签中加一个判断，使用class改变标签的值；

```
<a class="nav-link active"
   th:class="${activeUrl=='main.html'?'nav-link active':'nav-link'}"
   th:href="@{/main.html}">
   首页
</a>

<a class="nav-link"
   th:class="${activeUrl=='emps'?'nav-link active':'nav-link'}"
   th:href="@{/emps}">员工管理
</a>
```

3、修改请求链接

```
<div th:replace="~{commons/bar::sitebar(activeUrl='main.html')}"></div>
<div th:replace="~{commons/bar::sitebar(activeUrl='emps')}"></div>
```

4、我们刷新页面，去测试一下，OK，动态激活搞定！

> 员工信息页面展示

现在我们来遍历我们的员工信息！顺便美化一些页面，增加添加，修改，删除的按钮！

```
<main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-4">
    <!--添加员工按钮-->
    <h2> <button class="btn btn-sm btn-success">添加员工</button></h2>
    <div class="table-responsive">
        <table class="table table-striped table-sm">
            <thead>
                <tr>
                    <th>id</th>
                    <th>lastName</th>
                    <th>email</th>
                    <th>gender</th>
                    <th>department</th>
                    <th>birth</th>
                    <!--我们还可以在显示的时候带一些操作按钮-->
                    <th>操作</th>
                </tr>
            </thead>
            <tbody>
                <tr th:each="emp:${emps}">
                    <td th:text="${emp.id}"></td>
                    <td>[[${emp.lastName}]]</td>
                    <td th:text="${emp.email}"></td>
                    <td th:text="${emp.gender==0?'女':'男'}"></td>
                    <td th:text="${emp.EDepartment.departmentName}"></td>
                    <!--<td th:text="${emp.birth}"></td>-->
                    <!--使用时间格式化工具-->
                    <td th:text="${#dates.format(emp.birth,'yyyy-MM-dd
HH:mm')}"></td>

                    <!--操作-->
                    <td>
                        <button class="btn btn-sm btn-primary">编辑</button>
                        <button class="btn btn-sm btn-danger">删除</button>
                    </td>
                </tr>
            </tbody>
        </table>
    </div>
</main>
```

OK，显示全部员工OK！

添加员工实现

> 表单及细节优化

1、将添加员工信息改为超链接

```
<!--添加员工按钮-->
<h2>
    <a class="btn btn-sm btn-success" href="emp" th:href="@{/emp}">添加员工</a>
</h2>
```

2、编写对应的controller

```
//to员工添加页面
@GetMapping("/emp")
public String toAddPage(){
    return "emp/add";
}
```

3、添加前端页面；复制list页面，修改即可

bootstrap官网文档： https://v4.bootcss.com/docs/4.0/components/forms/

我们去可以里面找自己喜欢的样式！我这里给大家提供了编辑好的：

```
<form>
    <div class="form-group">
        <label>LastName</label>
        <input type="text" class="form-control" placeholder="kuangshen">
    </div>
    <div class="form-group">
        <label>Email</label>
        <input type="email" class="form-control" placeholder="24736743@qq.com">
    </div>
    <div class="form-group">
        <label>Gender</label><br/>
        <div class="form-check form-check-inline">
            <input class="form-check-input" type="radio" name="gender" value="1">
            <label class="form-check-label">男</label>
        </div>
        <div class="form-check form-check-inline">
```

```
            <input class="form-check-input" type="radio" name="gender"
 value="0">
            <label class="form-check-label">女</label>
        </div>
    </div>
    <div class="form-group">
        <label>department</label>
        <select class="form-control">
            <option>1</option>
            <option>2</option>
            <option>3</option>
            <option>4</option>
            <option>5</option>
        </select>
    </div>
    <div class="form-group">
        <label>Birth</label>
        <input type="text" class="form-control" placeholder="kuangstudy">
    </div>
    <button type="submit" class="btn btn-primary">添加</button>
</form>
```

4、部门信息下拉框应该选择的是我们提供的数据，所以我们要修改一下前端和后端

Controller

```
@Autowired
DepartmentMapper departmentMapper;

//to员工添加页面
@GetMapping("/emp")
public String toAddPage(Model model){
    //查出所有的部门，提供选择
    List<Department> departments = departmentMapper.getDepartments();
    model.addAttribute("departments",departments);
    return "emp/add";
}
```

前端

```
<div class="form-group">
    <label>department</label>
    <!--提交的是部门的ID-->
    <select class="form-control">
        <option th:each="dept:${departments}"
th:text="${dept.departmentName}" th:value="${dept.id}">1</option>
    </select>
</div>
```

OK，修改了controller，重启项目测试！

> 完整增加员工功能，我们来具体实现添加功能；

1、修改add页面form表单提交地址和方式

```
<form th:action="@{/emp}" method="post">
```

2、编写controller；

```
//员工添加功能，使用post接收
@PostMapping("/emp")
public String addEmp(){      //方法的形参可能需要传入参数：Employee emp。

    // 回到员工列表页面，可以使用redirect或者forward，就不会被视图解析器解析
    return "redirect:/emps";
}
```

回忆：重定向和转发 以及 /的问题？

原理探究： ThymeleafViewResolver

```
public static final String REDIRECT_URL_PREFIX = "redirect:";
public static final String FORWARD_URL_PREFIX = "forward:";

protected View createView(String viewName, Locale locale) throws Exception {
    // 简单分析下源码
}
```

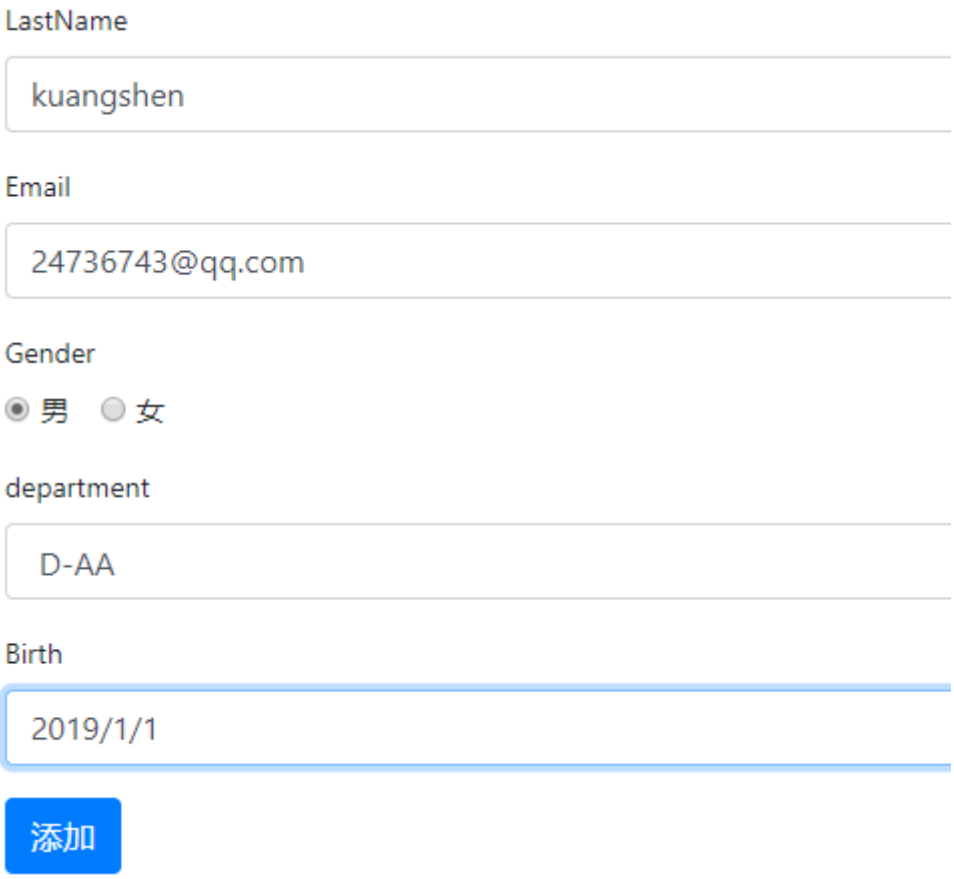
OK，看完源码，我们继续编写代码！

3、我们要接收前端传过来的属性，将它封装成为对象！首先需要将前端页面空间的name属性编写完毕！【操作】

4、编写controller接收调试打印【操作】

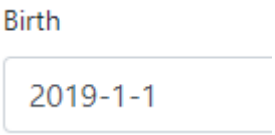
```
//员工添加功能
//接收前端传递的参数，自动封装成为对象[要求前端传递的参数名，和属性名一致]
@PostMapping("/emp")
public String addEmp(Employee employee){
    System.out.println(employee);
    employeeDao.save(employee); //保存员工信息
    //回到员工列表页面，可以使用redirect或者forward
    return "redirect:/emps";
}
```

> 启动测试，前端填写数据，注意时间问题：

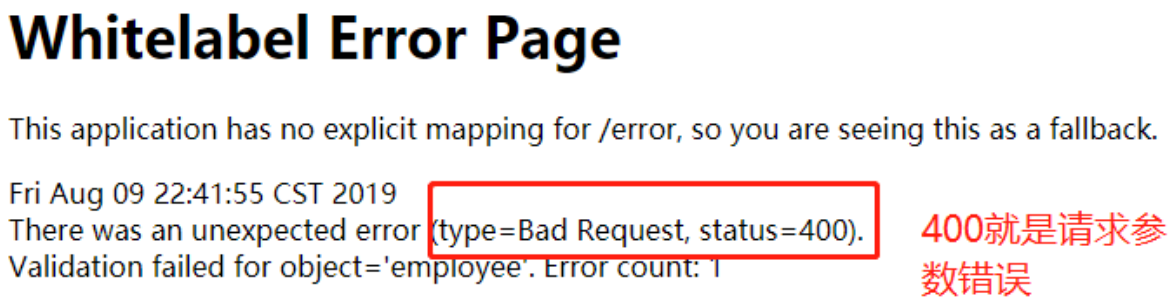


点击提交，后台输出正常！页面跳转及数据显示正常！OK！

那我们将时间换一个格式提交



提交发现页面出现了400错误！



生日我们提交的是一个日期， 我们第一次使用的 / 正常提交成功了，后面使用 - 就错误了，所以这里面应该存在一个日期格式化的问题；

SpringMVC会将页面提交的值转换为指定的类型，默认日期是按照 / 的方式提交； 比如将2019/01/01转换为一个date对象。

那思考一个问题？我们能不能修改这个默认的格式呢？

我们去看webmvc的自动配置文件；找到一个日期格式化的方法，我们可以看一下

```
@Bean
public FormattingConversionService mvcConversionService() {
    WebConversionService conversionService = new
WebConversionService(this.mvcProperties.getDateFormat());
    this.addFormatters(conversionService);
    return conversionService;
}
```

调用了 getDateFormat 方法;

```
public String getDateFormat() {
    return this.dateFormat;
}
```

这个在配置类中，所以我们可以自定义的去修改这个时间格式化问题，我们在我们的配置文件中修改一下;

```
spring.mvc.date-format=yyyy-MM-dd
```

这样的话，我们现在就支持 - 的格式了，但是又不支持 / 了， 2333吧

测试OK!

员工信息修改

注意： 尚硅谷的讲解的是"修改添加二合一表单",而KS的只是简单的添加了一个新页面update。

逻辑分析:

我们要实现员工修改功能，需要实现两步;

1、点击修改按钮，去到编辑页面，我们可以直接使用添加员工的页面实现

2、显示原数据，修改完毕后跳回列表页面!

> 实现

1、我们去实现一下，首先修改跳转链接的位置;

```
<a class="btn btn-sm btn-primary" th:href="@{/emp/}+${emp.id}">编辑</a>
```

2、编写对应的controller

```
//to员工修改页面
@GetMapping("/emp/{id}")
public String toUpdateEmp(@PathVariable("id") Integer id,Model model){
    //根据id查出来员工
    Employee employee = employeeMapper.get(id);
    System.out.println(employee);
    //将员工信息返回页面
    model.addAttribute("emp",employee);
    //查出所有的部门，提供修改选择
    List<Department> departments = departmentMapper.getDepartments();
    model.addAttribute("departments",departments);

    return "emp/update";
}
```

3、我们需要在这里将add页面复制一份，改为update页面；需要修改页面，将我们后台查询数据回显

```
<form th:action="@{/emp}" method="post">
    <div class="form-group">
        <label>LastName</label>
        <input name="lastName" type="text" class="form-control"
th:value="${emp.lastName}">
    </div>
    <div class="form-group">
        <label>Email</label>
        <input name="email" type="email" class="form-control"
th:value="${emp.email}">
    </div>
    <div class="form-group">
        <label>Gender</label><br/>
        <div class="form-check form-check-inline">
            <input class="form-check-input" type="radio" name="gender"
value="1"
                   th:checked="${emp.gender==1}">
            <label class="form-check-label">男</label>
        </div>
        <div class="form-check form-check-inline">
            <input class="form-check-input" type="radio" name="gender"
value="0"
                   th:checked="${emp.gender==0}">
            <label class="form-check-label">女</label>
        </div>
    </div>
    <div class="form-group">
        <label>department</label>
        <!--提交的是部门的ID-->
        <select class="form-control" name="department">
            <option th:selected="${dept.id == emp.department}"
th:each="dept:${departments}"
                    th:text="${dept.departmentName}" th:value="${dept.id}">1
            </option>
        </select>
    </div>
    <div class="form-group">
        <label>Birth</label>
        <input name="birth" type="text" class="form-control"
th:value="${emp.birth}">
    </div>
    <button type="submit" class="btn btn-primary">修改</button>
</form>
```

测试OK！

发现我们的日期显示不完美，可以使用日期工具，进行日期的格式化！

```
<input name="birth" type="text" class="form-control"
th:value="${#dates.format(emp.birth,'yyyy-MM-dd HH:mm')}">
```

数据回显OK，我们继续完成数据修改问题！

4、修改表单提交的地址：

```
<form th:action="@{/updateEmp}" method="post">
```

5、编写对应的controller

```
@PostMapping("/updateEmp")
public String updateEmp(Employee employee){
    employeeMapper.update(employee);
    //回到员工列表页面
    return "redirect:/emps";
}
```

6、编写Mapper接口及对应的 xml 文件

```
// 修改员工信息
int update(Employee employee);
```

```
<update id="update" parameterType="Employee">
    update employee
    set last_name = #{lastName},email=#{email},gender=#{gender},department=#{department},birth=#{birth}
    where id = #{id} ;
</update>
```

编写完毕后，启动测试！

问题：发现页面提交的没有id；我们在前端加一个隐藏域，提交id；

```
<input name="id" type="hidden" class="form-control" th:value="${emp.id}">
```

重启，修改信息测试OK！

删除员工实现

1、list页面，编写提交地址

```
<a class="btn btn-sm btn-danger" th:href="@{/delEmp/}+${emp.id}">删除</a>
```

//这里是2个不同的表达式,因此需要连接符"+"。

2、编写Controller

```
@GetMapping("/delEmp/{id}")
public String delEmp(@PathVariable("id") Integer id){
    employeeMapper.delete(id);
    return "redirect:/emps";
}
```

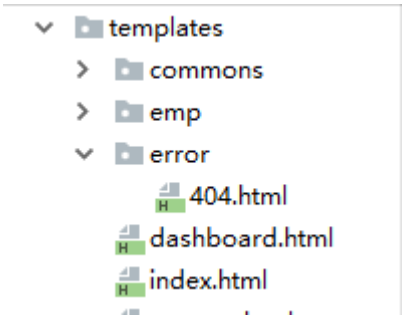
测试OK！

404及注销

> 404

我们只需要在模板目录下添加一个error文件夹，文件夹中存放我们相应的错误页面；

比如404.html 或者 4xx.html 等等，SpringBoot就会帮我们自动使用了！



测试使用！

> 注销

1、注销请求

```
<a class="nav-link" href="#" th:href="@{/user/loginOut}">Sign out</a>
```

2、对应的controller

```
@GetMapping("/user/loginOut")
public String loginOut(HttpSession session){
    session.invalidate();
    return "redirect:/index.html";
}
```

相信大家学到这里，SpringBoot的单体应用开发基本就没有问题了！

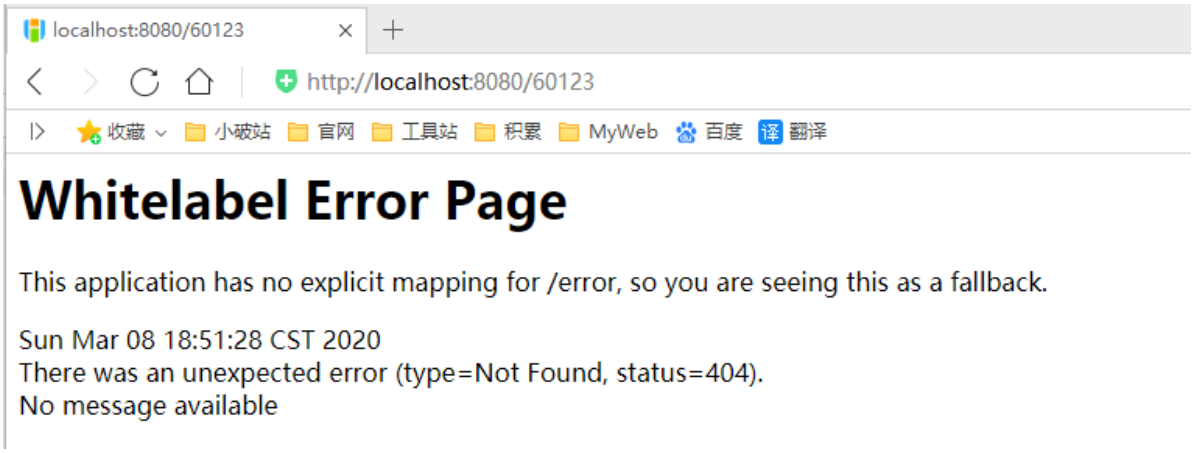
但是还是那句话，学会技术只是浅层次的东西，要消化理解我的思想方法，这才是最有价值的点！

定制错误数据

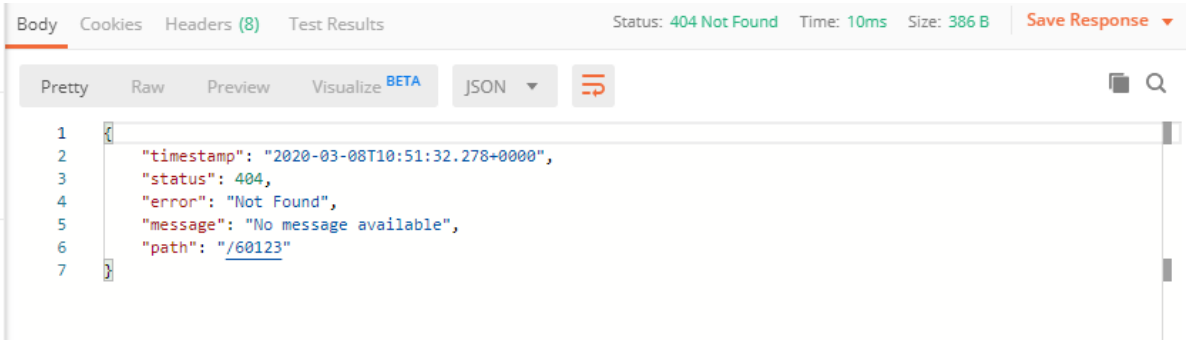
//这里KS也简化了,可以看看尚硅谷的：7、错误处理机制

> SpringBoot 默认的错误处理机制

1、浏览器访问的默认的错误处理效果：



2、如果是其他客户端，默认响应一个 json 数据；



> 错误处理原理分析

我们看到自动配置类：ErrorMvcAutoConfiguration 错误处理的自动配置类；

这里面注入了几个很重要的 bean；

1、DefaultErrorAttributes

2、BasicErrorController

3、ErrorPageCustomizer

4、DefaultErrorViewResolver

错误处理步骤：

一旦系统出现了 4xx 或者 5xx 之类的错误，ErrorPageCustomizer 就会生效（定制错误的响应规则）

```
@Bean
public ErrorPageCustomizer errorPageCustomizer(DispatcherServletPath
dispatcherServletPath) {
    // 点进这个类
    return new ErrorPageCustomizer(this.serverProperties,
dispatcherServletPath);
}
```

发现一个方法 registerErrorPages 注册错误页面

```
@Override
public void registerErrorPages(ErrorPageRegistry errorPageRegistry) {
    ErrorPage errorPage = new ErrorPage(
        // 这里有个 getPath() 路径，我们点进去

 this.dispatcherServletPath.getRelativePath(this.properties.getError().getPa
th()));
    errorPageRegistry.addErrorPages(errorPage);
}

// getPath
public String getPath() {
    return this.path;
}

// this.path;
@Value("${error.path:/error}")
private String path = "/error";
```

系统一旦出现错误之后就会来到 /error 请求进行处理；这个请求会被 BasicErrorController 处理：

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

}
```

这个类有两个方法：

```
// 产生html类型的数据，浏览器发送的请求会被这个方法处理
@RequestMapping(produces = MediaType.TEXT_HTML_VALUE)
public ModelAndView errorHtml(HttpServletRequest request,
HttpServletResponse response) {
    HttpStatus status = getStatus(request);
    Map<String, Object> model = Collections
        .unmodifiableMap(getErrorAttributes(request,
isIncludeStackTrace(request, MediaType.TEXT_HTML)));
    response.setStatus(status.value());
    // 去哪个页面拿错误页面呢？resolveErrorView 方法
    ModelAndView modelAndView = resolveErrorView(request, response, status,
model);
    return (modelAndView != null) ? modelAndView : new ModelAndView("error",
model);
}

// 返回 json 类型的数据，其他的客户端请求会被这个方法处理
@RequestMapping
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request)
{
    HttpStatus status = getStatus(request);
    if (status == HttpStatus.NO_CONTENT) {
        return new ResponseEntity<>(status);
    }
    Map<String, Object> body = getErrorAttributes(request,
isIncludeStackTrace(request, MediaType.ALL));
    return new ResponseEntity<>(body, status);
}
```

我们来看看resolveErrorView 这个方法：

```
protected ModelAndView resolveErrorView(HttpServletRequest request,
                                        HttpServletResponse response,
                                        HttpStatus status,
                                        Map<String, Object> model) {
    // 拿到所有的 errorViewResolvers 错误视图解析器
    for (ErrorViewResolver resolver : this.errorViewResolvers) {
        ModelAndView modelAndView = resolver.resolveErrorView(request,
status, model);
        if (modelAndView != null) {
            return modelAndView;
        }
    }
    return null;
}
```

我们在之前看到有这样一个bean DefaultErrorViewResolver 默认的错误视图解析器：

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

    private static final Map<Series, String> SERIES_VIEWS;

    static {
        Map<Series, String> views = new EnumMap<>(Series.class);
        views.put(Series.CLIENT_ERROR, "4xx"); // 客户端错误
        views.put(Series.SERVER_ERROR, "5xx"); // 服务端错误
        SERIES_VIEWS = Collections.unmodifiableMap(views);
    }
    // .....

    @Override // HttpStatus 状态码
    public ModelAndView resolveErrorView(HttpServletRequest request,
HttpStatus status, Map<String, Object> model) {
        ModelAndView modelAndView = resolve(String.valueOf(status.value()),
model);
        if (modelAndView == null &&
SERIES_VIEWS.containsKey(status.series())) {
            // 通过状态码解析视图
            modelAndView = resolve(SERIES_VIEWS.get(status.series()),
model);
        }
        return modelAndView;
    }

    // 去 error 路径下解析视图
    private ModelAndView resolve(String viewName, Map<String, Object> model)
{
        // 比如 error/404     error/500
        String errorViewName = "error/" + viewName;
        TemplateAvailabilityProvider provider =
this.templateAvailabilityProviders.getProvider(errorViewName,
                this.applicationContext);
        if (provider != null) {
            return new ModelAndView(errorViewName, model);
        }
        return resolveResource(errorViewName, model);
    }

}
```

所以说：定制错误页面，我们可以建立一个 error 目录，然后放入对应的错误码html文件！

比如：404.html 500.html 4xx.html 5xx.html

这些页面的信息数据在哪里呢？我们找到 DefaultErrorAttributes 这个bean对象；

里面有很多的 addxx 方法，就是添加不同的信息；

```java
// addStatus
// addErrorDetails
// addErrorMessage
// addStackTrace
// addPath

// 这里面存了一些错误的信息，我们可以在错误页面直接取出来
```

到了这里，我们的SpringBoot开发一个简单的单体应用对我们来说就没什么太大的问题了！

//KS没有讲解嵌入式servlet和外部servlet,可能是现在不用这些知识了。