

五、模板引擎FreeMarker

5.1 FreeMarker模板引擎引入

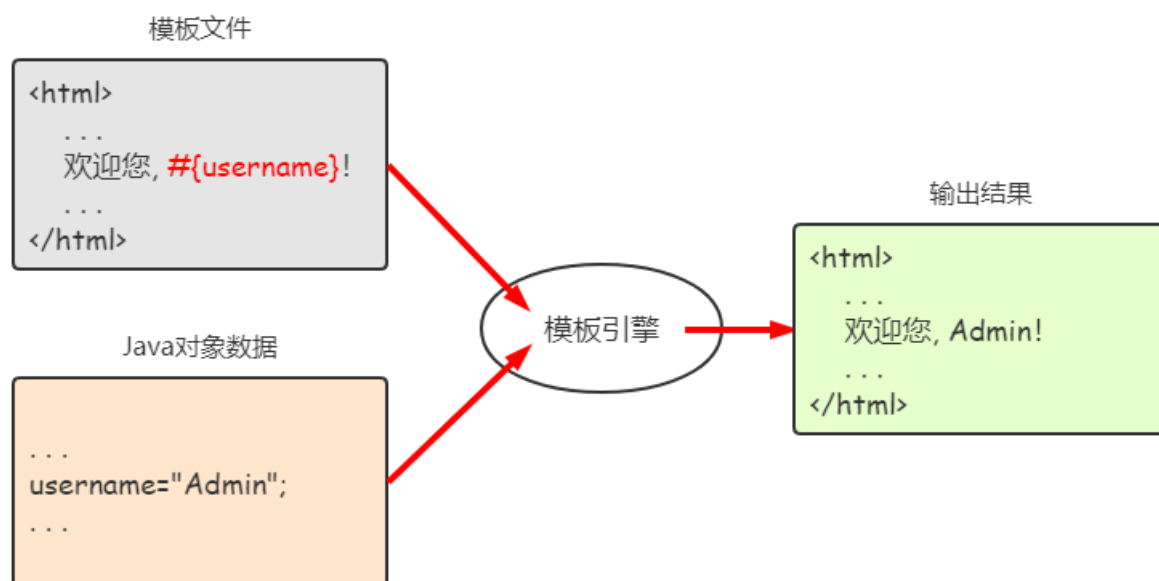
5.1.1 概念

FreeMarker是SpringBoot官方推荐的另一款模板引擎，它的历史很悠久，从1999年诞生第一个版本，经过多年的迭代，目前最新版本为2.3.31（发布于2021-02-16），并且其版权归属也在2015年被授予给了Apache软件基金会，基于Apache许可证2.0版本发布，是免费的。

官方网站（<https://freemarker.apache.org/>），中文参考手册（<http://freemarker.foofun.cn/>）

“FreeMarker 是一款 模板引擎：即一种基于模板和要改变的数据，并用来生成输出文本(HTML网页，电子邮件，配置文件，源代码等)的通用工具。它不是面向最终用户的，而是一个Java类库，是一款程序员可以嵌入他们所开发产品的组件。”

FreeMarker最初的设计，是被用来在MVC模式的Web开发框架中生成HTML页面的，将视图从业务逻辑中抽取出来，业务逻辑不再负责视图的展示，而是将视图交给FreeMarker来输出。在程序的业务代码中准备好显示的数据，然后在模板中专注于如何展现数据，由模板引擎生成页面显示出准备好的数据：



官方网站给出了定义：

模板 + 数据模型 = 输出

模板和静态HTML是相同的，只是它会包含一些 FreeMarker 将它们变成动态内容的指令，为模板准备的数据整体被称为数据模型。

FreeMarker不是一个Web应用框架，而适合作为Web应用框架一个组件，FreeMarker与容器无关，它没有被绑定到 Servlet或HTML或任意Web相关的东西上，因此也可以用于非Web应用环境中。

5.1.2 特性

- 强大的模板语言：条件语句、遍历迭代、创建和修改变量、字符串及算术操作、格式化、宏和函数，可以包含其他模板，等等
- 多用途和轻量级：零依赖，可以输出各种格式（例如HTML、XML、RTF、Java源码等），插件式模板载入器（可以从任何源载入模板，包括本地文件、数据库等），众多配置选项

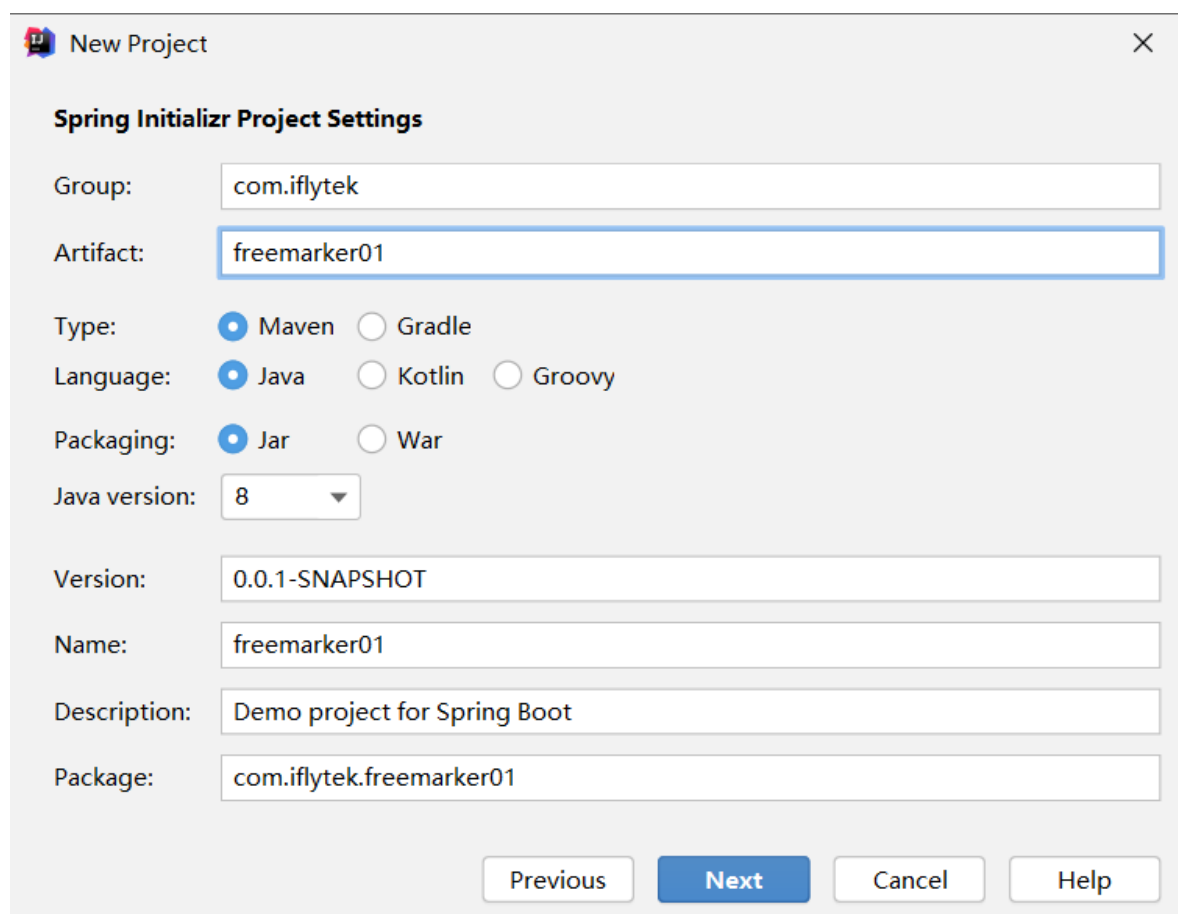
- 国际化/本地化：地区敏感的数字和日期/时间格式，多种不同语言的相同模板。
- XML 处理功能：将 XML 的 DOM 模型放入数据模型直观的遍历，还可以声明的方式处理它们
- 通用数据模型：FreeMarker 不是直接反射到 Java 对象，Java 对象通过插件式对象封装，以变量方式在模板中显示，可以使用抽象（接口）方式表示对象（JavaBean、XML 文档、SQL 查询结果集等等），告诉模板开发者使用方法，使其不受技术细节的打扰

5.1.3 环境搭建

5.1.3.1 创建项目

创建项目的两种方式：

- 1) 创建 Maven 项目，选择使用 webapp 原型创建项目，并添加 FreeMarker 的依赖
- 2) 创建 Spring Initializr 项目，选择 FreeMarker 的起步依赖



New Project

Spring Initializr Project Settings

Group:

Artifact:

Type: ☒ Maven ☐ Gradle

Language: ☒ Java ☐ Kotlin ☐ Groovy

Packaging: ☒ Jar ☐ War

Java version:

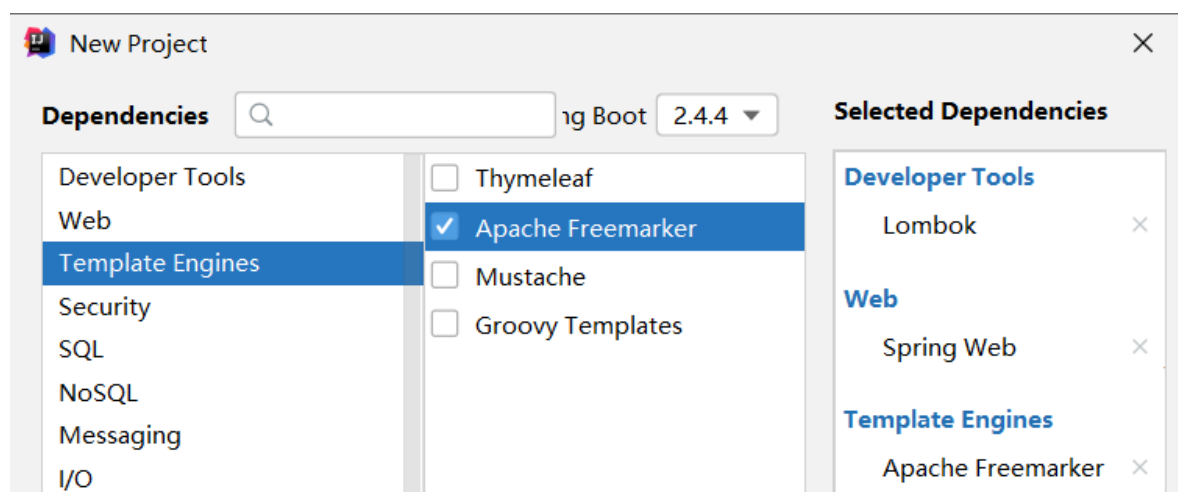
Version:

Name:

Description:

Package:

Previous Next Cancel Help



New Project

Dependencies

Developer Tools

Web

Template Engines

Security

SQL

NoSQL

Messaging

I/O

☐ Thymeleaf

☒ Apache Freemarker

☐ Mustache

☐ Groovy Templates

Selected Dependencies

Developer Tools

Lombok ×

Web

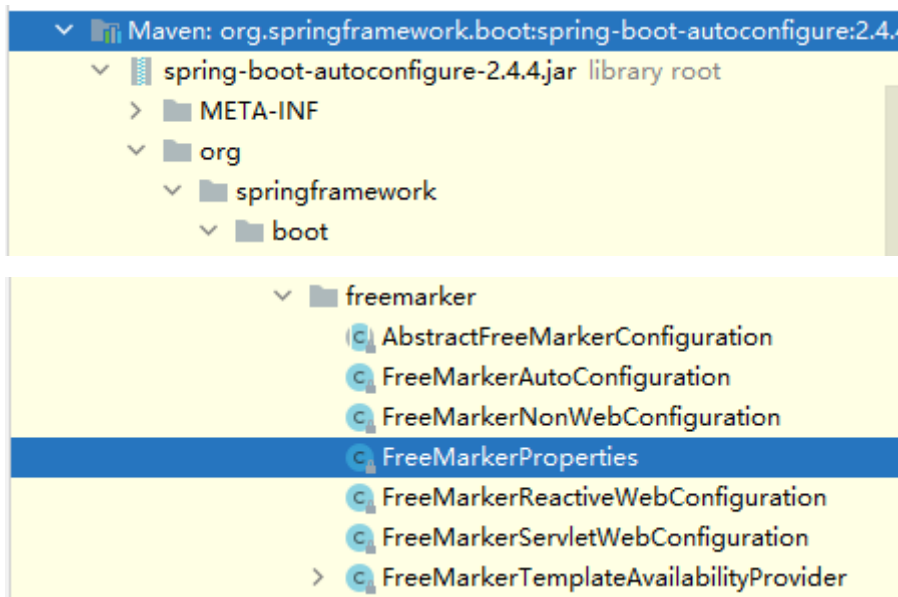
Spring Web ×

Template Engines

Apache Freemarker ×

5.1.3.2 自动配置

SpringBoot官方对FreeMarker提供了自动配置的支持，可以在 `spring-boot-autoconfigure` 包中找到：



其中的 `FreeMarkerProperties` 类中定义了自动配置属性，可以看到其中包括了默认的模板加载路径、前缀和后缀：

```
@ConfigurationProperties(prefix = "spring.freemarker")
public class FreeMarkerProperties extends AbstractTemplateViewResolverProperties {

    public static final String DEFAULT_TEMPLATE_LOADER_PATH = "classpath:/templates/";

    public static final String DEFAULT_PREFIX = "";

    public static final String DEFAULT_SUFFIX = ".ftlh";
```

需要注意这里的默认后缀为 `ftlh`，这是FreeMarker的模板文件后缀名，而很多的教程资料中对FreeMarker的模板文件的后缀名都使用 `ftl`，因此可以通过配置修改后缀名：

```
spring:
  freemarker:
    suffix: .ftl # 注意：初学者最容易犯的错是漏了这里后缀名前的“点”，千万不要漏了
    cache: false # 开发环境下配置为false，线上生产环境则为true
```

5.1.4 FreeMarker程序类型

5.1.4.1 非Web应用

根据对FreeMarker的介绍，其本身没有被绑定到Servlet或HTML或任意Web相关的东西上，因此可以用于非Web应用环境中，这种情况下需要编写代码创建FreeMarker的配置实例，然后进行模板加载以及数据模型的处理，之后生成输出，并且可以输出为文件或者直接进行打印输出：

```
@SpringBootApplication
public class Freemarker01Application {

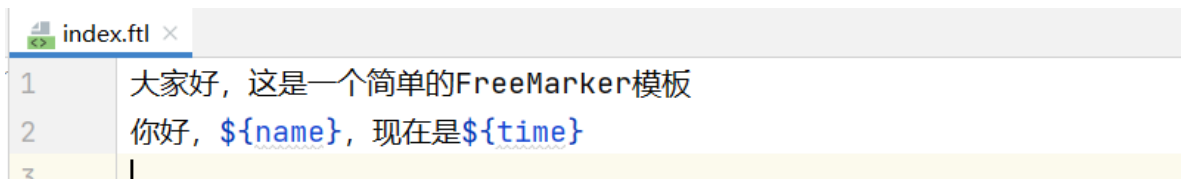
    public static void main(String[] args) throws IOException, TemplateException {
        //SpringApplication.run(Freemarker01Application.class, args);
        System.out.println("现在是一个普通的Java程序...");

        // step1.创建FreeMarker配置实例
        Configuration config = new Configuration(Configuration.VERSION_2_3_31);
        // step2.设置模板加载器
```

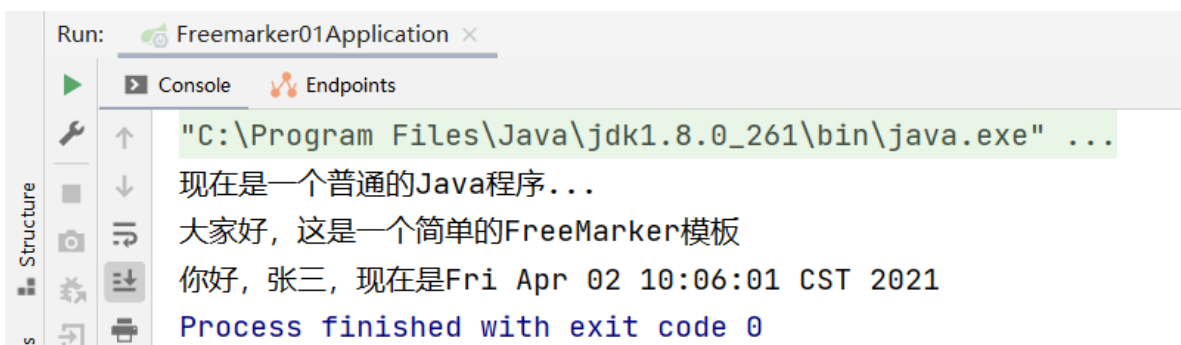
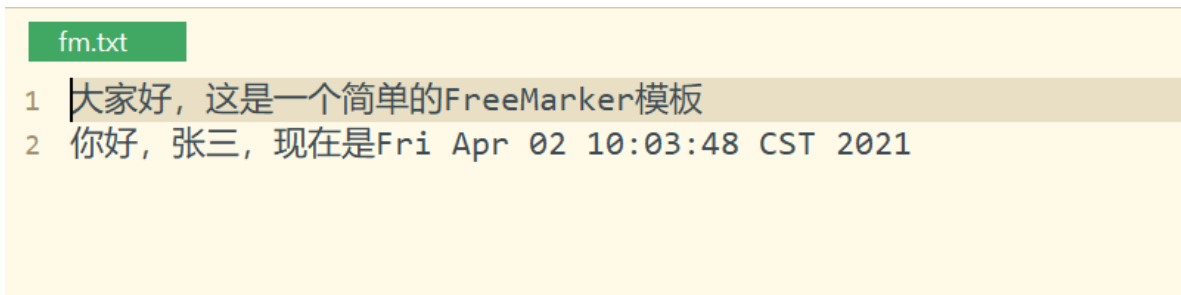
```

// 参考官方文档，可以使用内建的三种模板加载器，或者使用多加载器
//config.setDirectoryForTemplateLoading(new
File("src/main/resources/templates"));
config.setClassForTemplateLoading(this.getClass(), "/templates"); // 用Java的
ClassLoader来加载类，在实际运行的环境中， 类加载机制是首选用来加载模板的方法
// step3.加载指定模板
Template template = config.getTemplate("index.ftl");
// step4.构造数据模型
Map<String, Object> dataModel = new HashMap<>(); // 这里也可以创建一个学生类对象
dataModel.put("name", "张三");
dataModel.put("time", new Date().toString());
// step5.生成输出
// 根据参数传递的输出类型，可以是输出到文件，也可以直接输出到打印流
//template.process(dataModel, new FileWriter("E://fm.txt"));
template.process(dataModel, new PrintWriter(System.out));
}
}

```



输出结果：



5.1.4.2 Web应用

对于JavaEE后端开发来说，很显然FreeMarker将会被应用到Web项目中，因此，对于非Web环境下的FreeMarker的开发过程仅作简单了解即可。需要关注的是Web应用中如何使用，以及FreeMaker本身的模板开发语言技术。

5.2 入门案例

按照对模板引擎技术的理解，在SpringMVC框架中的FreeMaker的运行过程应当如下：

- 1、模板文件存放在web服务器上，就像通常存放静态HTML页面那样
- 2、浏览器访问某个控制器方法时，在Java代码中进行数据的查询，并添加到域对象中
- 3、模板引擎负责解析和查找指定的模板文件，并将模型数据内容替换模板中的特殊标签，形成最后渲染结果发送到访问者的浏览器中

下面以实际步骤来完成这些操作。

5.2.1 编写控制器

```
@GetMapping("/fm1")
public ModelAndView getFM1(){
    Map<String, Object> map = new HashMap<>(); // 这里也可以创建一个学生类对象
    map.put("name", "张三");
    map.put("age", 20);
    map.put("gender", "男");

    return new ModelAndView("stuinfo", "stu", map); // 会以视图名sutinfo结合默认加载路径和
    前缀、后缀名去查找
}
```

识点回顾：SpringMVC的数据共享方式

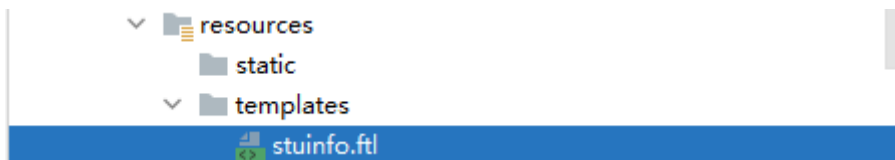
- 通过创建ModelAndView时传入视图名称以及属性数据（案例中的方式）
- 控制器方法中设置Model参数，SpringMVC自动绑定参数，直接对该参数设置属性值，控制器方法则直接返回视图名称
- 控制器方法中自动绑定request、session参数，进行数据传递

5.2.2 编写模板文件

FreeMarker中为编写模板设计的非常简单的编程语言称为 **FTL(Freemarker Template Language)**，模板文件的后缀名即为 **ftl**，在SpringBoot2.x版本中其后缀名默认为 **ftlh**。

最简单的模板通常是普通的**HTML文件**（或者是其他任何文本文件。FreeMarker本身不属于HTML，而目前我们主要的应用场景则是Java的Web应用中，所以为了页面的展示，通常还是使用HTML文件进行编写。）

由SpringBoot的自动配置属性文件可知，模板文件的默认加载路径在 **classpath:/templates/** 目录下，因此在该目录下创建一个名为stuinfo的HTML文件，并改名为 **stuinfo.ftl**：



其中编写显示学生信息的代码，为了能应用模板引擎读入与对象中的数据，在HTML中放置能被FreeMarker所解析的特殊代码片段：

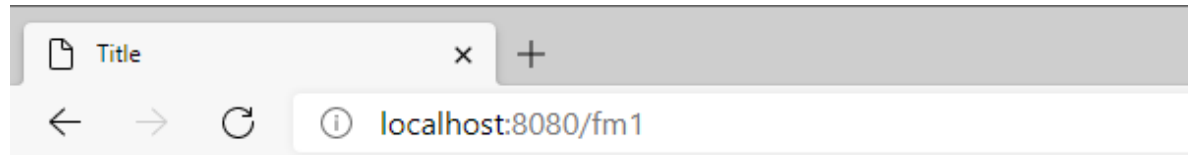
```

<body>
  <h1>FreeMarker模板引擎</h1>
  <div>
    <p>姓名: ${stu.name}</p> <!--stu为域对象中存储的对象名-->
    <p>年龄: ${stu.age}</p> <!--若域对象中存储了单独的属性数据，则可以直接通过属性名进行解析-->
    <p>性别: ${stu.gender}</p>
  </div>
</body>

```

这里的 `${*...}`，FreeMarker将会输出真实的值来替换大括号内的表达式，被称为 **interpolation插值**

5.2.3 运行测试



FreeMarker模板引擎

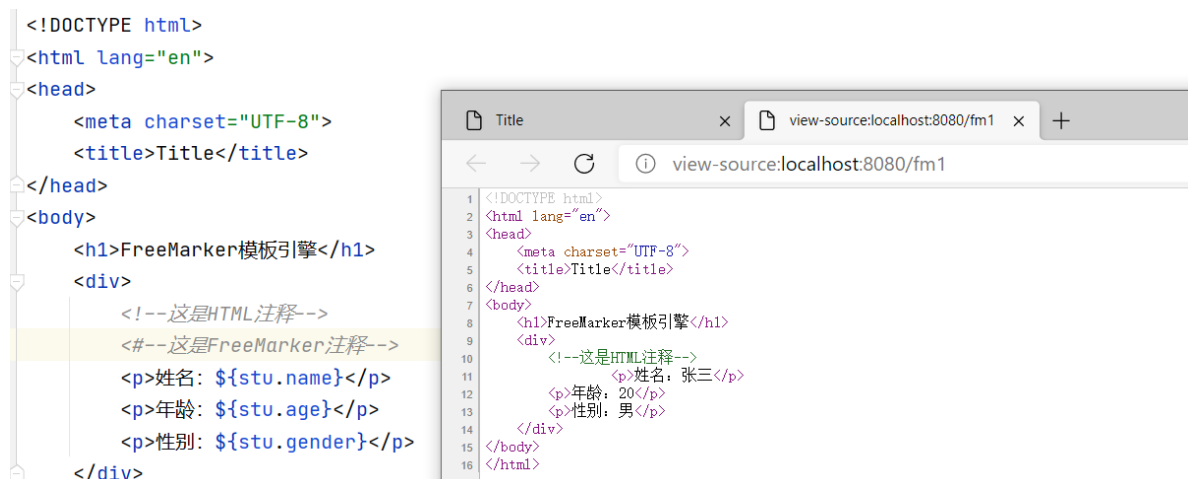
姓名：张三

年龄：20

性别：男

5.2.4 FTL与HTML

在FTL模板文件中，可以使用HTML注释也可以使用FreeMarker的注释，区别在于HTML注释在最后生成的输出文件中依然可见，而FreeMarker的注释则在生成输出时被忽略：



另外，在FTL模板文件中，HTML所有的标签都适用，所以在操作时可以直接创建HTML文件后更改后缀名为ftl。

并且JS与CSS也可以使用，和在HTML中的使用是一致的：



5.3 FreeMarker数据类型

官网文档里描述了FreeMarker支持的类型有：

- 标量（标量是最基本，最简单的数值类型）：
 - 字符串
 - 数字
 - 布尔值
 - 日期/时间(日期，时间或日期时间)
- 容器（这些值存在的目的是为了包含其他变量；它们只是容器）：
 - 哈希表
 - 序列
 - 集合
- 子程序：
 - 方法和函数（一个值是方法或函数的时候，那么它就可以计算其他值，结果取决于传递给它的参数）
 - 用户自定义指令（用户自定义指令是一种子程序，一种可以复用的模板代码段）
- 其它/很少使用：
 - 结点

下面介绍其中的几种比较常用的类型及FTL中的内建函数操作

5.3.1 字符串类型

在FTL中字符串类型相当于Java的字符串，可以直接输出，也可以通过多个内建函数进行处理，字符串内建函数可参见官方文档（http://freemarker.foofun.cn/ref_builtins_string.html），下面列举常见的字符串操作。

FmController.java:

```

@GetMapping("/fm2")
public String getFM2(Model model){
    /*字符串类型数据*/
    model.addAttribute("msg", "Hello World! spring boot freemarker.");

    return "string";
}

```

string.ftl:

```

<!--1.字符串输出，或者使用内建函数?string-->
消息: ${msg} <br>
消息: ${msg?string}<br>
${"传递的数据: ${msg}!"} <br> <!--在引号内可以继续使用${}进行插值-->
${"传递的数据: " + msg + "!"}<br> <!--插值中可以拼接字符串-->
<hr>
<!--2.字符串长度-->
字符串长度: ${msg?length} <br>
<hr>
<!--3.字符串截取，早期版本使用substring函数，新版本使用字符串切分替代-->
字符串截取: ${msg?substring(3)} <br> <!--从3开始截取到结尾-->
字符串截取: ${msg?substring(3, 10)} <br> <!--从3开始截取到9结束-->
获取指定位置的字符: ${msg[2]} <br><!--获取第2个字符-->
字符串切分2..8: ${msg[2..8]} <br><!--从2截取到8结束-->
字符串切分2..&lt;8: ${msg[2..<8]} <br><!--从2截取到7结束-->
字符串切分2..*5: ${msg[2..*5]} <br><!--从2截取，截5个，如果不足则截取到结尾-->
字符串切分2..: ${msg[2..]} <br><!--从2截取到结尾-->
<hr>
<!--4.大小写转换-->
首字母小写输出: ${msg?uncap_first} <br>
首字母大写输出: ${msg?cap_first} <br>
转小写: ${msg?lower_case} <br>
转大写: ${msg?upper_case} <br>
<hr>
<!--5.字符串检测-->
是否以指定字符串开头: ${msg?starts_with("hello"?string)} <br> <!--starts_with得到一个布尔值，不能直接输出-->
是否以指定字符串结尾: ${msg?ends_with(".")?string} <br> <!--ends_with得到一个布尔值，不能直接输出-->
查找指定字符串位置: ${msg?index_of("spring")} <br>
<hr>
<!--6.字符串编辑-->
去除两端空白: ${msg?trim} <br>
替换指定字符串: ${msg?replace("World", "FreeMarker")}<br>

```

5.3.2 数字类型

相当于Java中的int、float、double等数值类型，能够以数值形式、百分比形式、货币形式进行显示

```

@GetMapping("/fm3")
public String getFM3(Model model){
    model.addAttribute("age", 20);
    model.addAttribute("interestrate", 0.0435);
    model.addAttribute("price", 75.2);
    return "number";
}

```



```
直接输出-年龄: ${age} <br>
直接输出-价格: ${price} <br>
转为普通字符串输出-基准利率: ${interestrate?c} <br> <!--c是一个内建函数，用来将“计算机语言”的数字
转为字符串-->
转为百分比格式-基准利率: ${interestrate?string.percent} <br>
转为货币格式-价格: ${price?string.currency} <br>
保留指定小数位输出: ${interestrate?string["0.###"]}
```

5.3.3 布尔值

类似于Java中的布尔类型，但是要注意不能直接输出，会报错，需要转换为字符串进行输出。

```
@GetMapping("/fm4")
public String getFM4(Model model){
    model.addAttribute("flag", true);
    return "bool";
}
```

```
输出flag: ${flag?c} <br>
输出flag: ${flag?string("真", "假")} <br> <!--新版本中已经弃用，使用then函数替代-->
输出flag: ${flag?then("yes", "no")} <br>
```

5.3.4 日期

类似于Java中的java.util.Date类型，注意不能直接输出，需要转换为字符串进行输出。

另外需要注意，由于Java平台的限制，在FreeMarker中使用Date类型时，不能决定是否只有日期、只有时间或者是日期-时间都有，因此不要直接转字符串，需要使用指定的格式进行转换，或者使用内建函数单独获取日期时间部分。

```
@GetMapping("/fm5")
public String getFM5(Model model){
    model.addAttribute("now", new Date());
    return "datetime";
}
```

```
当前日期: ${now?date} <br>
当前时间: ${now?time} <br>
当前日期时间: ${now?datetime} <br>
格式化输出: ${now?string("yyyy年MM月dd日 HH:mm:ss")} <br>
```

5.3.5 空值判断和处理

FreeMarker中进行插值时如果未找到值则会抛出异常，且对于FreeMarker来说空引用 `null` 也是一样的，都当作不存在，需要使用运算符进行处理：

```
@GetMapping("/fm6")
public String getFM6(Model model){
    model.addAttribute("emptyObj", null);
    model.addAttribute("emptyStr", "");
    return "empty";
}
```

```

<!--测试空引用和不存在的值-->
<!--输出空引用: ${emptyObj} <br>-->
<!--输出不存在的值: ${ttt} <br>-->

<!--空值处理-->
<!--1. 使用!运算符指定缺失变量的默认值-->
输出空引用: ${emptyObj!} <br> <!--不加参数, 指定默认值为空字符串-->
输出不存在的值: ${ttt! "该值不存在"} <br>
输出空字符串: ${emptyStr! "字符串为空"} <br> <!--空串是有效内容, 不会被默认值替换-->
<!--2. 使用??运算符判断变量是否存在, 返回结果为布尔类型, 可以进行运算或转为字符串输出-->
emptyObj是否存在: ${ (emptyObj??) ?c } <br>

```

5.4 运算符

FreeMarker中可以进行算术运算、比较运算、逻辑运算

```

@GetMapping("/fm7")
public String getFM7(Model model) {
    model.addAttribute("num1", 10);
    model.addAttribute("num2", 20);
    return "op";
}

```

```

<!--算术运算: "+", "-", "*", "/", "%"-->
"+" : ${num1 + num2} <br>
 "-" : ${num1 - num2} <br>
 "*" : ${num1 * num2} <br>
 "/" : ${num1 / num2} <br>
 "%" : ${num1 % num2} <br>
<hr>
<!--比较运算: >(gt)、<(lt)、>=(gte)、<=(lte)、==(==)、!=(!)-->
"gt" : ${ (num1 gt num2) ?c } <br>
"lt" : ${ (num1 lt num2) ?c } <br>
"==" : ${ (num1 == num2) ?c } <br>
<hr>
<!--逻辑运算: 支持"&&"、"||"、"!"运算符-->
"&&" : ${ (num1 > 10 && num2 < 20) ?c } <br>
"||" : ${ (num1 gte 10 || num2 gte 20) ?c } <br>

```

5.5 FreeMarker常用指令

5.5.1 定义变量指令

在Web程序中通常都是在Java代码中获取数据变量等传输到页面里, FreeMarker也可以在模板中定义变量进行处理。使用的是 `assign` 指令。

指令常用语法: `<#assign name1=value1 name2=value2 ...>`

```

<!--assign指令-->
<#assign stuname="Jack"> <!--定义一个变量-->
学生姓名: ${stuname} <br>
<#assign age= 20 names=["张三", "李四", "王五"]> <!--定义多个变量，定义序列-->
年龄: ${age} <br>
姓名串: ${names?join("-")} <br>
<#assign data={"name":"Jack","gender":"male","age":20}> <!--定义map数据-->
${data.name} <br>

```

5.5.2 逻辑判断指令

语法：

```

<#if condition>
...
<#elseif condition2>
...
<#elseif condition3>
...
<#else>
...
</#if>

```

- `condition` , `condition2` , 等：将被计算成布尔值的表达式。
- `elseif` 和 `else` 是可选的。
- 最后以 `</#if>` 闭合，中间过程中的 `<#else>`、`<#elseif>` 没有闭合标签

```

<!--逻辑判断指令-->
<#assign score=75>
考试成绩：
<#if score lt 60>
    不及格
<#elseif score lt 70>
    及格
<#elseif score lt 80>
    中等
<#elseif score lt 90>
    良好
<#else>
    优秀
</#if>

```

5.5.3 遍历指令

在FTL中，可以使用 `list` 指令对序列或者集合进行遍历，同时还有一些用来支持遍历的指令：`else`、`items`、`sep`、`break`。

语法形式：

```

<#list 序列或集合表达式 as 循环变量名称>
    遍历操作，可以是任意的FTL（包括继续嵌套list指令）
<#else>
    当序列或集合为空，即有0项时执行的操作
</#list>

```

2.3.23版本中引入了新的语法形式：

```

<#list 序列或集合>
    如果序列或集合超过一项，此部分将执行一次
    <#items as 循环变量名称>
        对每一项进行变量操作
    </#items>
    如果序列或集合超过一项，此部分将执行一次
<#else>
    当序列或集合为空，即有0项时执行的操作
</#list>

```

对序列进行遍历（序列相当于Java中的数组、List、Set等集合类型）：

```

<#--list指令-->
<#assign names=["张三", "李四", "王五"]> <#--定义一个序列变量-->
<#if names??> <#--使用??判断对象是否存在-->
    <#list names as name> <#--对names进行遍历-->
        ${name},
    <#else> <#--当序列为空时-->
        姓名为空！
    </#list>
<#else>
    数据对象不存在！
</#if>

```

对Map进行遍历：

```

<#assign student={"name":"张三","age":20,"gender":"男", "major":"Java"}>
学生的信息：<br>
<#list student?keys as prop> <#--keys内建函数获取Map集合的key的序列-->
${prop} ---> ${student[prop]} <br> <#--使用map[key]获取key对应的value值-->
</#list>

```

5.5.4 自定义指令（了解）

自定义指令可以使用 `macro` 指令来定义，`macro`（宏）是有一个变量名的模板片段。可以在模板中使用宏作为自定义指令，这样就能进行重复性的工作。

`macro` 指令自身不输出任何内容，它只是用来创建宏变量，所以就会有一个名为 `greet` 的变量。在 `<#macro greet>` 和 `</macro>` 之间的内容（称为 **宏定义体**）将会在使用该变量作为指令时执行。可以在FTL标记中通过 `@` 代替 `#` 来使用自定义指令。使用变量名作为指令名。而且，自定义指令的结束标记也是需要的，可以写为自闭和形式。

```

<#--自定义指令-->
<#macro sayHi> <#--定义一个无参宏-->
    <p style="font-size: 20px; color: rebeccapurple;">Hello everyBody</p>
</#macro>
<#--使用宏-->
<@sayHi /><@sayHi />
<hr>

<#macro sum a b c> <#--定义一个有参宏-->
    ${a + b + c}
</#macro>
使用带参宏指令求和运算：<@sum a=1 b=2 c=4/>
<hr>

<#macro max a b c> <#--定义一个有参宏，宏定义体中继续使用其他指令-->

```

```
<#if a gt b> <#if a gt c>${a} <#else>${c}</#if> <#else><#if b gt c>${b}
<#else>${c}</#if> </#if>
</#macro>
求最大值: <@max a=3 b=5 c=4/>
<hr>
```

5.5.5 导入和包含指令（了解）

`import` 指令可以引入一个库，语法：

```
<#import 模板的路径 as 命名空间名称>
```

创建一个新的命名空间，然后在那个命名空间中执行给定的路径参数中的模板。例如有一个模板文件，其中包含了一些公共自定义指令：

`mylib.ftl`：

```
<#macro doAdd a b>
    ${a+b}
</#macro>
```

在其他模板文件中引入 `directive.ftl`：

```
<#import "mylib.ftl" as my>
执行导入模板中的自定义指令: <@my.doAdd a=10 b=5/>
```

`include` 指令可以使用它在你的模板中插入另外一个 FreeMarker 模板文件，被包含的文件内容就出现在 `include` 指令标签出现的位置，且被包含的文件和包含的文件的模板中的变量共享，就相当于将被包含的模板文件内容赋值粘贴在包含文件处：

```
<!-- 包含其他模板文件-->
<#include "stuinfo.ftl">
```

5.6 FreeMarker 页面静态化

当访问淘宝、网易等大型网站时，网站的首页、商品详情以及新闻详情页面是经常被访问的，通常这些页面的数据内容来自于数据库，每一次的访问都需要从数据库进行一次查询，在大并发的访问下将会造成数据库的高负载，而实际上这些页面中的很多数据其实是没有变化的，那么就需要将这些页面进行静态化处理。

例如在访问新闻、活动、商品、详情页面的时候，路径可以是 `xx/[id].html`，服务器端根据请求id，动态生成html网页，下次访问数据时，无需再查下数据，直接将html静态页面返回，可以减少对数据库的交互，提高访问的性能。

5.6.1 页面静态化实现过程

页面静态化的实现其实就是利用FreeMarker的 `Template` 类的 `process` 方法，将模板文件结合数据模型输出到本地文件中。

假设在服务器的html目录中存放了若干以id作为名称存储的html静态文件，其中展示了最近的新闻资讯，具体的数据存储于数据库中（包括新闻的标题、来源、发布时间、内容），新闻页面的模板为 `news.ftl`。

静态化的过程为：

- 1、访问页面xxxxx/news/1.html时，检测1.html文件是否存在
- 2、若不存在，则通过数据库查询id为1的新闻数据，并通过模板引擎生成静态页面
- 3、之后将页面进行响应返回

5.6.2 实现示例

控制器：

```
@GetMapping("/news/{id}")
public String getNews(@PathVariable("id") int id) throws IOException,
TemplateException {
    // 获取项目根目录（涉及中文目录，需要转码），并设置存放新闻页面的目录
    String resourcesPath
    =java.net.URLDecoder.decode(ResourceUtils.getURL("").getPath(),"utf-8");
    // 为了能够直接返回html页面，因此设置保存目录为static静态资源目录下，可以被SpringBoot直接解析
    File newsFilePath = new File(resourcesPath +
    "src/main/resources/static/newsFile/");
    if (!newsFilePath.exists()) {
        // 目录不存在则创建目录
        newsFilePath.mkdirs();
    }
    // 检测要访问的文件是否存在
    File newsFile = new File(newsFilePath, id + ".html");
    if (!newsFile.exists()){
        // 不存在，则进行页面静态化处理
        // 模拟查询出新闻数据
        String title = "可爱的中国，今天为你缅怀";
        String source = "江西新闻客户端";
        String publishTime = "2021-04-03 02:08:29";
        String content = "到那时，到处都是活跃的创造，到处都是日新月异的进步，这时，我们的民族
        就可以无愧色地立在人类的面前。" + "\n" +
        "\n在《可爱的中国》文章中，方志敏字字泣血，唤醒着中国亿万同胞的爱国之情。这些用忠诚和热
        血书写下来的文字，让许许多多的优秀青年走上救国道路，激励着一代代共产党人，为共产主义事业奋斗终身。";
        Map<String, Object> dataModel = new HashMap<>();
        dataModel.put("title", title);
        dataModel.put("source", source);
        dataModel.put("publishTime", publishTime);
        dataModel.put("content", content);
        // 创建FreeMarker配置实例-->加载模板-->生成输出
        Configuration config = new Configuration(Configuration.VERSION_2_3_31);
        config.setClassForTemplateLoading(this.getClass(), "/templates");
        Template template = config.getTemplate("news.ftl");
        template.process(dataModel, new FileWriter(newsFile));
    }
    // 返回静态页面文件
    return "/newsFile/" + id + ".html";
}
```

模板文件 `news.ftl`：

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>${title}</title>
  </head>
  <body>
    <h2>${title}</h2>
    <a href="#">${source}</a> <span>${publishTime}</span>
    <p>${content}</p>
  </body>
</html>

```

5.6.3 问题与改进

- 问题1: 将存储静态页面的文件夹的名字设置成和控制器方法的映射路径相同, 导致在控制器中转发页面地址时总是被SpringMVC给拦截, 再次映射到控制器方法, 结果出错 (把x.html自动绑定到整数参数id上类型转换失败)
 - 解决办法: 访问路径中为 `news`, 存储静态页面的文件夹命名 `newsFile`
- 问题2: 由于SpringBoot项目使用的是 `SpringInitializr` 向导创建的, 默认打包方式是 `jar` 包, 而为了方便控制器方法直接返回资源, FreeMarker输出的html文件存放在静态资源目录, 此时将会发生一个问题, 当前新生成的文件无法访问, 必须要重启项目后才能再次访问, 这是由于本身打的jar包, 当前项目运行环境下生成的文件并不在包中
 - 解决方法1: 将存储静态页面的文件夹配置进静态文件路径中:

- 修改系统配置文件:

```

spring:
  freemarker:
    suffix: .ftl # 注意: 初学者最容易犯的错是漏了这里后缀名前的“点”, 千万不要漏了
    cache: false # 开发环境下配置为false, 线上生产环境则为true
  web:
    resources:
      static-locations: classpath:/META-INF/resources/,classpath:/resources/,classpath:/static/,classpath:/public/,file:classpath:/static/newsFile/

```

- 在静态资源路径最后补充配置: `file:classpath:/static/newsFile/`
- 注意, 此时使用静态资源路径作为加载路径了, 可以直接以转发的形式直接返回静态页面, 修改控制器代码:

```

@GetMapping("/news/{id}")
public String getNews(@PathVariable("id") int id, HttpServletRequest req) throws IOException, TemplateException {
    // 获取项目webapp根目录 (涉及中文目录, 需要转码), 并设置存放新闻页面的目录
    String path = req.getServletContext().getRealPath("/");
    String resourcesPath = java.net.URLDecoder.decode(path, "utf-8");
    File newsFilePath = new File(resourcesPath + "newsFile/");
    // . . . 中间代码相同, 略

    // 返回静态页面文件, 注意返回形式不同
    return "forward:/newsFile/" + id + ".html" ;
}

```

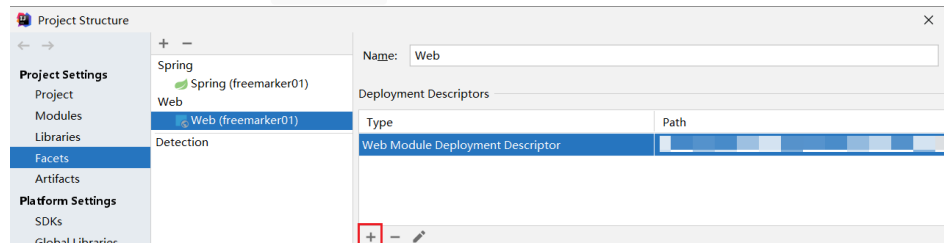
- 解决办法2（比较繁琐，推荐第一种）：修改打包方式为 `war`，并且将生成的文件存放在 `webapp` 目录下。详细修改步骤如下

- 1、修改打包方式

- 在 `pom.xml` 文件中项目坐标之后添加一句配置
`<packaging>war</packaging>`，指定打包方式为 `war` 包

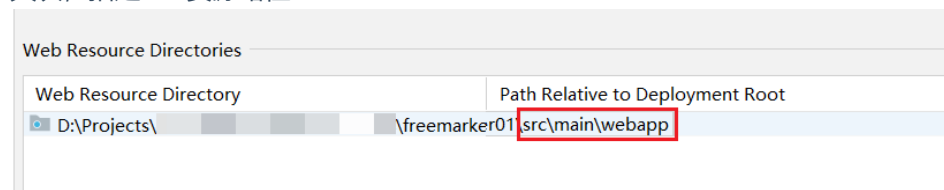
- 2、修改项目web配置，在项目结构Project Structure中

- 首先，添加部署描述符 `web.xml`

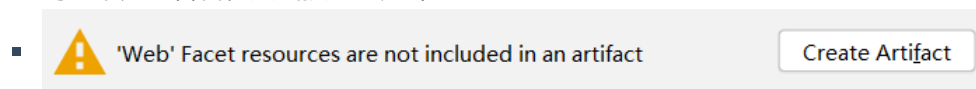


- 其中Path指定为项目目录下 `src/main/webapp`

- 其次，指定web资源路径



- 对于下方的警告，点击按钮创建即可



六、SpringBoot日志集成、异常处理、热部署

6.1 日志集成

6.1.1 日志框架的选择

市面上有很多的日志框架：

- JCL (Jakarta Commons Logging)
- JUL (java.util.logging)
- SLF4J (simpler logging facade for java)
- Log4j
- Log4j2
- Logback
- jboss-logging

其中日志的抽象层，即日志门面有 `JCL`、`SLF4J`、`jboss-logging`，相应的实现类有：`Log4j`、`JUL`、`Log4j2`、`Logback`。

面向接口编程即挑选一个门面，然后再挑选一个实现类来使用，JCL框架很久没更新了，比较老，而 `jboss-logging` 只在一些特定的框架里使用，所以门面一般就选用 `SLF4J`。

日志实现类中，`Log4j` 和 `Logback` 都是是 `SLF4J` 的作者的作品，适配性更好，`JUL`是`Java.util`工具包自带的，功能比较简陋，`Log4j2` 其实是Apache借 `Log4j` 之名开发的，实现的也非常好，但是目前支持的框架还不多。

`SpringBoot`的底层是`Spring`框架，而`Spring`框架默认采用`JCL`（我们以前开发`Spring`框架要导`commons-logging`依赖），`SpringBoot`则做了个包装，它选用了 `SLF4J` 门面和 `Logback` 实现类。

`SpringBoot`中由于已经做了大量的包装工作，日志使用较为简单，在使用之前，先来了解关于日志框架的底层实现机制。

6.1.2 SpringBoot日志实现原理

调用记录日志的方法，不应该直接调用日志实现类，而应该调用日志抽象层里面的方法，由于多态机制，调用抽象层的方法将会自动调用实现类的中的方法实现功能。

如何使用`SLF4J`这个日志门面呢？可以到它的官网（<http://www.slf4j.org/>）中看看，首页中有使用手册：



Hello World

As customary in programming tradition, here is an example illustrating the simplest way to output "Hello world" using SLF4J. It begins by getting a logger with the name "HelloWorld". This logger is in turn used to log the message "Hello World".

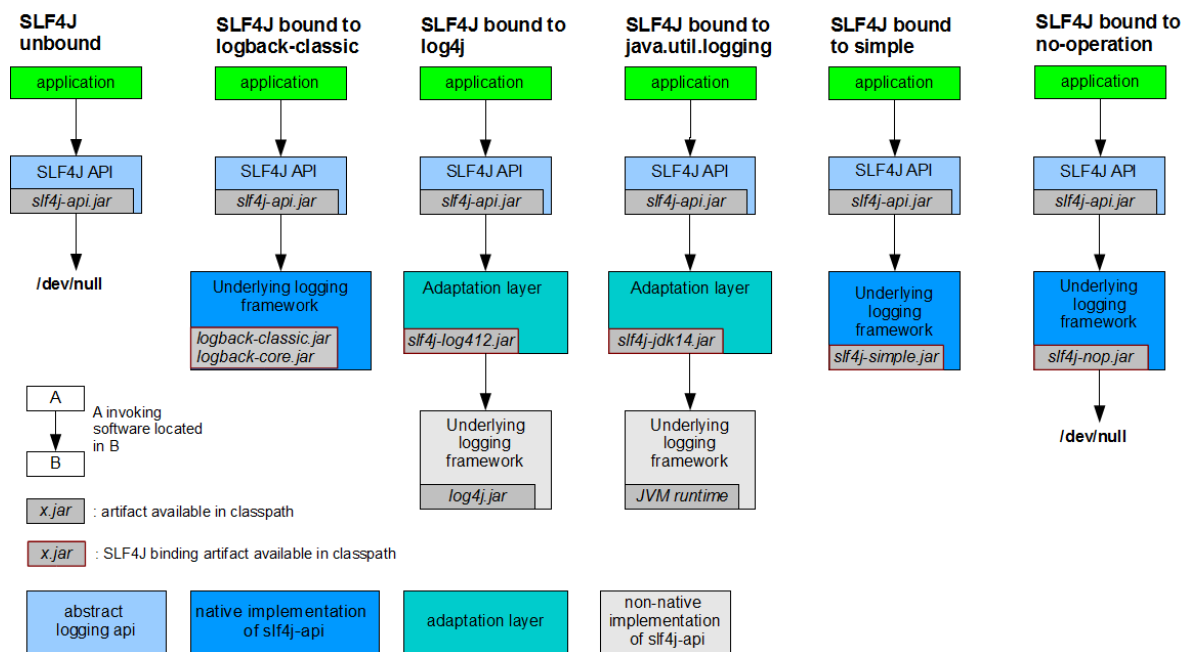
```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld {
    public static void main(String[] args) {
        Logger logger = LoggerFactory.getLogger(HelloWorld.class);
        logger.info("Hello World");
    }
}
```

可以看出，如果要使用 `slf4j`，我们需要导入`slf4j`的依赖包，在代码中通过 `LoggerFactory.getLogger` 方法，传递要记录日志的类，以此获得 `logger` 对象，然后就可以通过 `logger` 来进行日志输出了。

然而 `slf4j` 本身是抽象层，所以还需要一个实现层的包，不过编码的时候始终是面向接口编程，具体底层选择哪个实现类不用关心，只要提供一个实现层的jar包。

但是这个实现层的jar包不是任意放一个就行的，在用户手册页下方官网给出了实现层以及适配层的关系。



但是这里还有一个问题，每一个日志的实现框架都有自己的配置文件。使用SLF4J的话，配置文件需要针对日志的实现层，因为SLF4J提供了接口，具体日志怎么实现，这些配置还是要看实现日志接口的框架来做。

另外，SpringBoot选择SLF4J和Logback作为日志框架，而在其他的很多框架中，底层是有自己选择的实现类的，例如Spring4底层使用 `commons-logging`，在Spring5中使用了 `spring-jcl` 做门面，使用 `JUL` 作为实现，同时支持 `SLF4J`，Hibernate底层使用了 `jboss-logging`，等等。

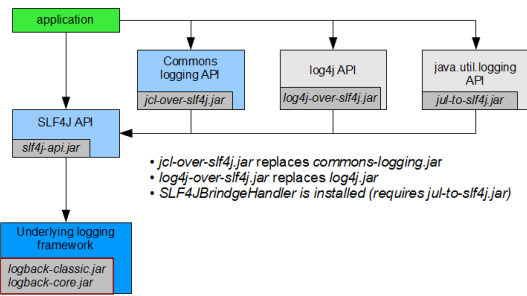
在这种情况下，可以统一日志记录框架，即使框架里使用了其他的日志实现类，也可以通过处理来统一使用 `SLF4J` 进行日志输出。这个该如何做到呢？

还是在SLF4J的官网的首页中，下方有一个链接（<http://www.slf4j.org/legacy.html>）

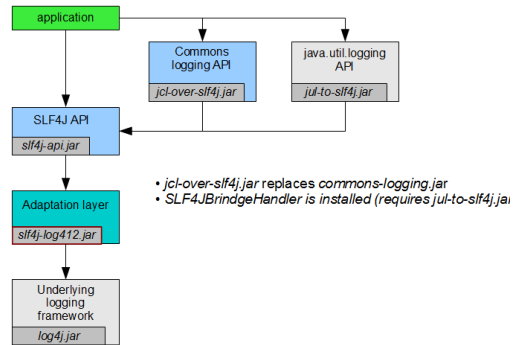
In case an externally-maintained component you depend on uses a logging API other than SLF4J, such as commons logging, log4j or java.util.logging, have a look at SLF4J's binary-support for **legacy APIs**.

其中介绍了如何将其他的日志框架统一重定向到SLF4J的API上：

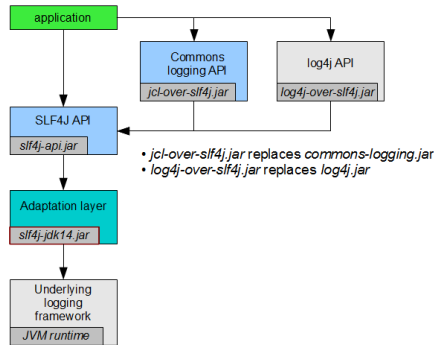
SLF4J bound to logback-classic with redirection of commons-logging, log4j and java.util.logging to SLF4J



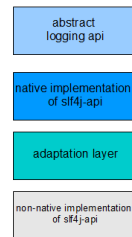
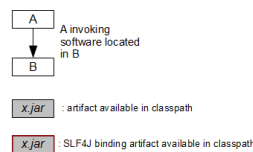
SLF4J bound to log4j with redirection of commons-logging and java.util.logging to SLF4J



SLF4J bound to java.util.logging with redirection of commons-logging and log4j to SLF4J



These diagrams illustrate *all* possible redirections for various bindings for reasons of convenience and expediency. Redirections should be performed only when necessary. For instance, it makes no sense to redirect java.util.logging to SLF4J if java.util.logging is not being used in your application.



那么总结来说，如何让系统中所有的日志都重定向到slf4j：

- 1) 将系统中其他的日志框架都排除掉
- 2) 用中间的桥接包替换原有的日志框架，例如如果使用jcl，那就使用jcl-over-slf4j.jar
- 3) 导入slf4j其他的实现层框架

SpringBoot就是这么来做的，能够自动适配所有的日志框架。所以现在可以很方便的使用日志而不需要操心上面的这些部分了。

6.1.3 SpringBoot中的日志使用

1) 创建项目

```
@SpringBootApplication
public class Springboot04Application {

    public static void main(String[] args) { SpringApplication.run(Springboot04Application.class, args); }

}
```

当创建好SpringBoot项目后，运行项目时可以观察到有日志内容输出，说明SpringBoot中已经默认配置好了日志框架，这意味着可以直接使用API获取日志实现类来进行日志输出。

2) 获取logger对象

获取logger对象有两种方式：

A. 通过 `LoggerFactory` 来获取

Logger

```
Logger java.util.logging
Logger jdk.internal.instrumentation
Logger jdk.nashorn.internal.runtime.log...
Logger org.slf4j
Logger ch.qos.logback.classic
Logger org.apache.logging.log4j
Logger com.sun.javafx.logging
Logger com.sun.org.slf4j.internal
```

```
public static final Logger LOGGER =
    LoggerFactory.getLogger(Springboot04Application.class);
```

B.通过 Lombok 获取

POM中添加Lombok依赖:

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
</dependency>
```

在需要创建日志对象的类上添加注解即可使用日志对象进行日志输出:

```
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@Slf4j
public class Springboot04Application {

    public static void main(String[] args) {
        SpringApplication.run(Springboot04Application.class, args);
        log.info("通过Lombok获取logger对象");
    }

    private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(Springboot04Application.class);
}
```

3) 进行日志输出

```
log.trace("trace日志...");
log.debug("debug日志...");
log.info("info日志...");
log.warn("warn日志...");
log.error("error日志...");
```

运行程序, 访问后观察输出结果:

```
INFO 18168 --- [main] c.i.s.Springboot04Application : Started Spri
INFO 18168 --- [main] c.i.s.Springboot04Application : info日志...
WARN 18168 --- [main] c.i.s.Springboot04Application : warn日志...
ERROR 18168 --- [main] c.i.s.Springboot04Application : error日志...
```

会发现只输出了info、warn和error日志, 这里涉及到日志级别, 这也是日志框架的一个好处, 日志框架可以调整输出的日志的级别, 日志级别由低到高: `trace < debug < info < warn < error`

从输出结果可以看出SpringBoot的默认指定的级别是 `info` 级别。

4) SpringBoot的日志输出配置

在项目配置文件（`application.properties`）中进行调整和修改

A.调整日志级别



这里通过 `level` 调整，在输入代码时可以观察到这里的level是一个Map类型的数据，指定的是一个包的或者每一个类用什么级别。图示中的配置表示调整为只指定的包的级别，调整后将会输出：

```
TRACE 5236 --- [main] c.i.s.Springboot04Application : trace日志...
DEBUG 5236 --- [main] c.i.s.Springboot04Application : debug日志...
INFO 5236 --- [main] c.i.s.Springboot04Application : info日志...
WARN 5236 --- [main] c.i.s.Springboot04Application : warn日志...
ERROR 5236 --- [main] c.i.s.Springboot04Application : error日志...
```

如果要调整SpringBoot的默认日志级别可以通过 `logging.level.root` 来设置。

B.设置日志输出位置

logging配置中还有两个配置可以用来指定日志输出文件

```
# 指定日志输出文件名
logging.file.name=my.log

# 指定日志输出文件的路径
logging.file.path=/spring/log
```

注意：

- 如果这两个参数都不指定，默认只在控制台输出
- 如果只指定了`logging.file.name`，则会输出日志到指定的文件中，同时：
 - `logging.file.name=springboot.log` --> 当前项目下创建日志文件
 - `logging.file=D:/springboot.log` --> 指定文件绝对路径
- 如果只指定了`logging.file.path`，则会输出到指定目录的`spring.log`文件中，同时：
 - `logging.file.path=/spring/log` --> 将会在项目所在的磁盘的根目录下创建spring文件夹和log文件夹，并且使用 `spring.log` 作为默认日志文件名。

C.设置输出的日志格式



这里主要有两个部分：`console` 用来设置控制台输出的日志的整体格式，`file` 用来设置输出到文件中的日志的整体格式。另外两个 `dateformat` 和 `level` 是对日志输出中的日期和日志级别两个部分单独设置。

格式说明：

- `%d` 表示日期时间，后面使用 `{yyyy-MM-dd}` 指定具体格式
- `%thread` 表示线程名
- `%-5level` 表示日志级别左对齐占5个字符宽度
- `%logger{50}` 表示logger的名字最长为50个字符，超过则按句点分隔并简写
- `%msg` 表示日志的消息内容
- `%n` 表示换行符号
- 新版中还可以使用 `%clr()` 包裹对其中的内容按照运行级别进行染色处理

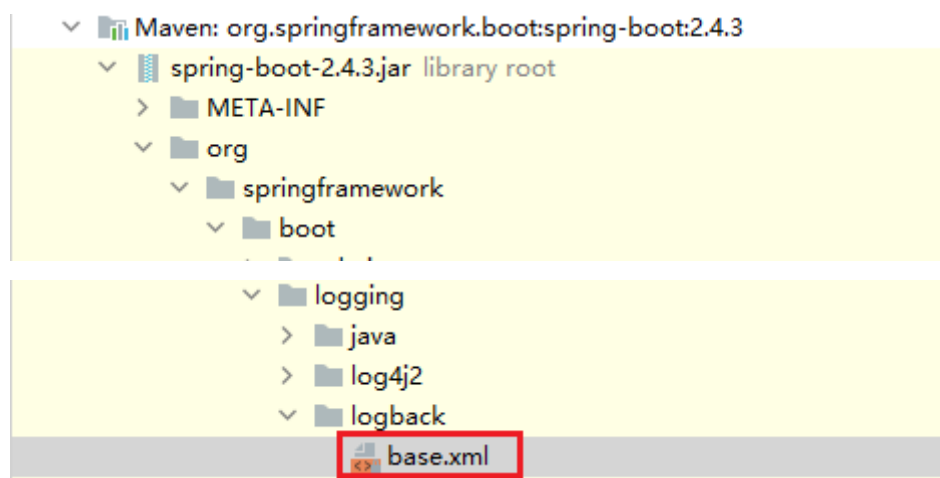
例如：

```
logging.pattern.console=%d{yyyy-MM-dd hh:mm:ss:SSS} %clr(%-5level) [%thread]
%clr(%logger{20}) - %msg%n
```

```
2021-03-15 05:59:33:807 TRACE [main] c.i.s.Springboot04Application - trace日志...
2021-03-15 05:59:33:807 DEBUG [main] c.i.s.Springboot04Application - debug日志...
2021-03-15 05:59:33:807 INFO [main] c.i.s.Springboot04Application - info日志...
2021-03-15 05:59:33:807 WARN [main] c.i.s.Springboot04Application - warn日志...
2021-03-15 05:59:33:807 ERROR [main] c.i.s.Springboot04Application - error日志...
```

5) SpringBoot的默认配置

SpringBoot的默认配置都在SpringBoot的包下：



```
<included>
  <include resource="org/springframework/boot/logging/logback/defaults.xml" />
  <property name="LOG_FILE" value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-${java.io.tmpdi
  <include resource="org/springframework/boot/logging/logback/console-appender.xml" />
  <include resource="org/springframework/boot/logging/logback/file-appender.xml" />
  <root level="INFO">
    <appender-ref ref="CONSOLE" />
    <appender-ref ref="FILE" />
  </root>
```

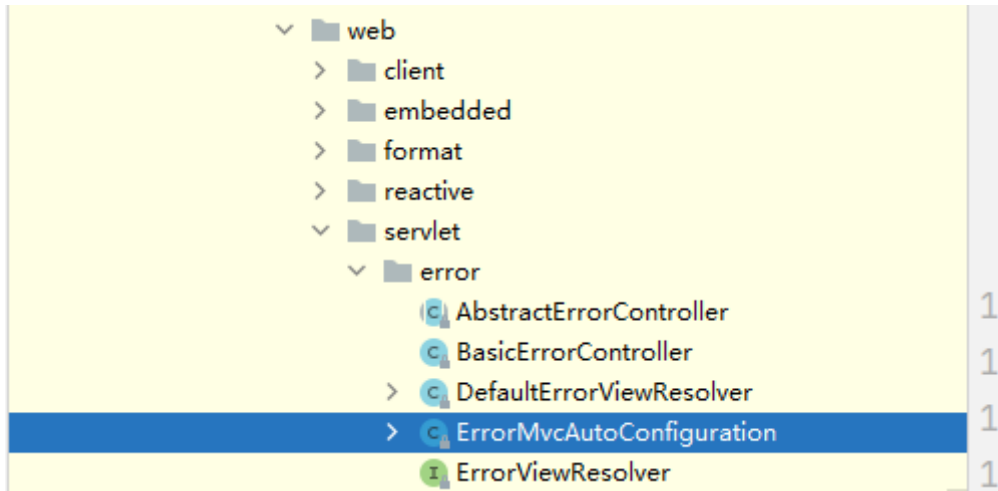
6.2 异常处理

6.2.1 SpringBoot处理机制

SpringBoot默认提供了错误处理机制，当发生错误时，例如访问一个不存在的页面，将会返回一个默认的空白错误页面，包含了发生时间、错误码、错误信类型、消息等信息。

而如果使用了其他客户端来访问SpringBoot，发现返回了一个JSON字符串，可以看出SpringBoot的处理还是不错的。

产生这种效果的原理在于 `spring-boot-autoconfigure` 包中有一个类：



理解这个类的工作原理可以解决两个问题：

- 1) 现在的默认返回的错误页不好，需要搭配整体网站风格来定制错误页面
- 2) 返回的JSON字符串信息不够明确，需要能够定制错误返回的JSON内容

6.2.2 SpringBoot异常处理

6.2.2.1 模拟异常环境

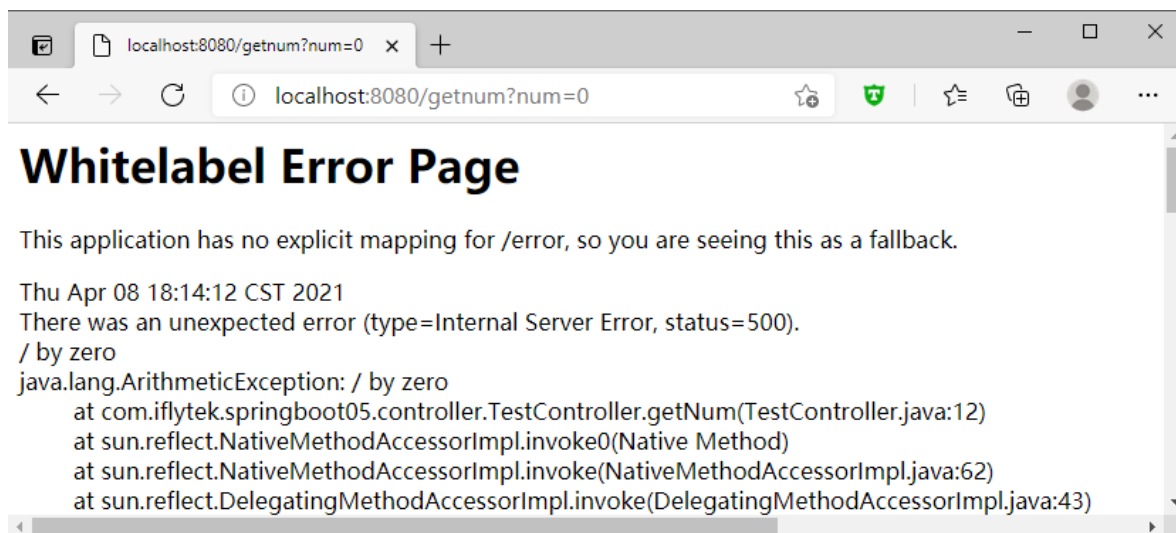
- 1) 创建SpringBoot项目

引入Web功能和Thymeleaf模板引擎

- 2) 创建控制器，模拟可能会产生异常的方法

```
@Controller
public class TestController {
    @GetMapping("/getnum")
    @ResponseBody
    public String getNum(int num){
        return 100 / num + "";
    }
}
```

此时访问 `http://localhost:8080/getnum?num=0` 时将会产生除0异常，并且由SpringBoot默认错误页处理：

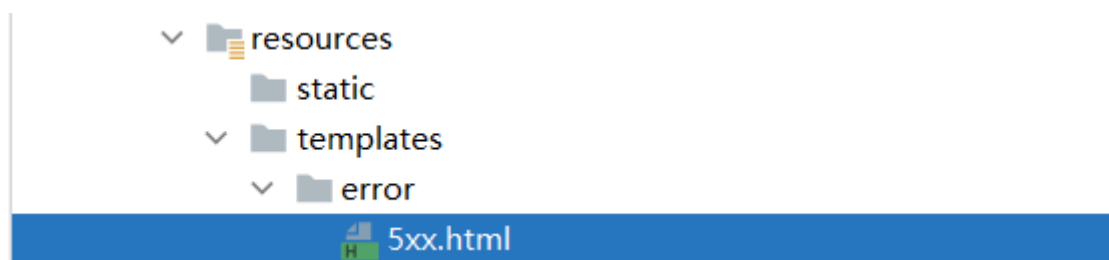


6.2.2.2 定制错误页

错误页的处理流程可以总结为三步：

- 1、模板引擎将会根据错误码在模板引擎的文件夹中查找 `/error/错误码` 对应的html文件进行解析处理
- 2、没有模板引擎或者模板引擎未能在 `error` 文件夹下找到错误码对应的html文件时，将会到静态资源目录下的 `error` 文件夹中查找以错误码命名的错误页
- 3、以上过程都没有解析成功时，返回SpringBoot的默认错误页Whitelable。

因此可以在 `templates` 目录下创建 `error` 目录，然后根据需要创建相应的错误码对应的html页面，或者使用 `4xx` 或 `5xx` 命名来进行模糊匹配：



并获取相关异常信息：

```
<body>
  <h1>服务器出问题了，正在紧急维护中，请稍后再访问...</h1>
  <h2>发生时间: [[${timestamp}]]</h2>
  <h2>信息: [[${message}]]</h2>
</body>
```

再次访问：



6.2.2.3 全局异常处理

配置错误页可以大致对一系列的错误进行定制化的提示，不过在实际项目中，可能会根据不同的异常需要展示不同的信息内容，另外还存在一些非功能性的异常，例如当用户查询不存在时抛出属于业务性质的自定义的异常，此时需要对不同的异常进行一些特殊的处理后再进行错误页的展示，这就需要进行异常处理。

从Spring4.3开始，提供一个注解 `@ControllerAdvice`，表示针对 `Controller` 做切面增强，可以用来实现全局异常处理切面逻辑。另外还有一个 `@ExceptionHandler` 注解，配合用来表示在抛出某个指定的异常时由该注解标注的方法来进行异常处理。

自定义异常类：

```
package com.iflytek.springboot05.exception;

public class UserNotFoundException extends RuntimeException{

    public UserNotFoundException(String username) {
        super("用户: [" + username + "]不存在!");
    }
}
```

全局异常处理类：

```
@ControllerAdvice
public class GlobalExceptionHandler {

    // 针对指定异常进行处理，返回视图
    @ExceptionHandler(value = UserNotFoundException.class)
    public ModelAndView unExceptionHandler(){
        ModelAndView modelAndView=new ModelAndView();
        modelAndView.setViewName("errorpage");
        Map<String, Object> map = new HashMap<>();
        map.put("code", 500);
        map.put("message", "内部错误");
        modelAndView.addObject("data", map);
        return modelAndView;
    }

    // 针对指定异常进行处理，返回JSON数据
    @ExceptionHandler(value = RuntimeException.class)
    @ResponseBody
    public Map<String, Object> rExceptionHandler(){
        Map<String, Object> map=new HashMap<>();
        map.put("code", "500");
        map.put("message", "系统错误!");
        return map;
    }
}
```

errorpage.html (在templates目录下)：

```
<h3>异常代码: [[${data.code}]]</h3>
<h3>异常信息: [[${data.message}]]</h3>
```

访问测试：



6.3 热部署

6.3.1 什么是热部署

后端开发人员修改了代码后，例如修改了返回值或者输出页面，因为修改了源码，需要重新启动项目才能让所作的修改生效，如果是之前的Spring环境下还需要重新启动服务器，等待服务器运行起来加载Web应用之后才能访问测试。这样每次启动很麻烦，热部署就是能够在修改代码后不重启的情况下立刻生效。

所谓热部署，就是在应用正在运行的时候升级软件，却不需要重新启动应用。

对于Java应用程序来说，热部署就是在运行时更新Java类文件。大多数基于Java的Web应用服务器，包括EJB服务器和Servlet容器，都支持热部署。

6.3.2 SpringBoot配置热部署

热部署作为开发阶段的特性，由spring-boot-devtools模块提供，用于在修改类、配置文件和页面等静态资源后，自动编译Spring Boot应用和加载应用和页面静态资源，从而提高开发流程自动化程度提升开发效率。

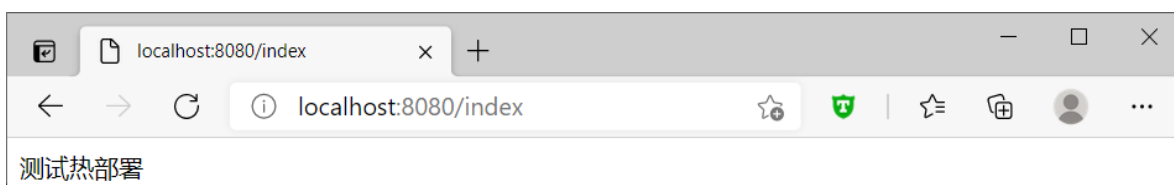
① 引入依赖：

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

② 编写控制器方法：

```
@GetMapping("/index")
@ResponseBody
public String getIndex(){
    return "测试热部署";
}
```

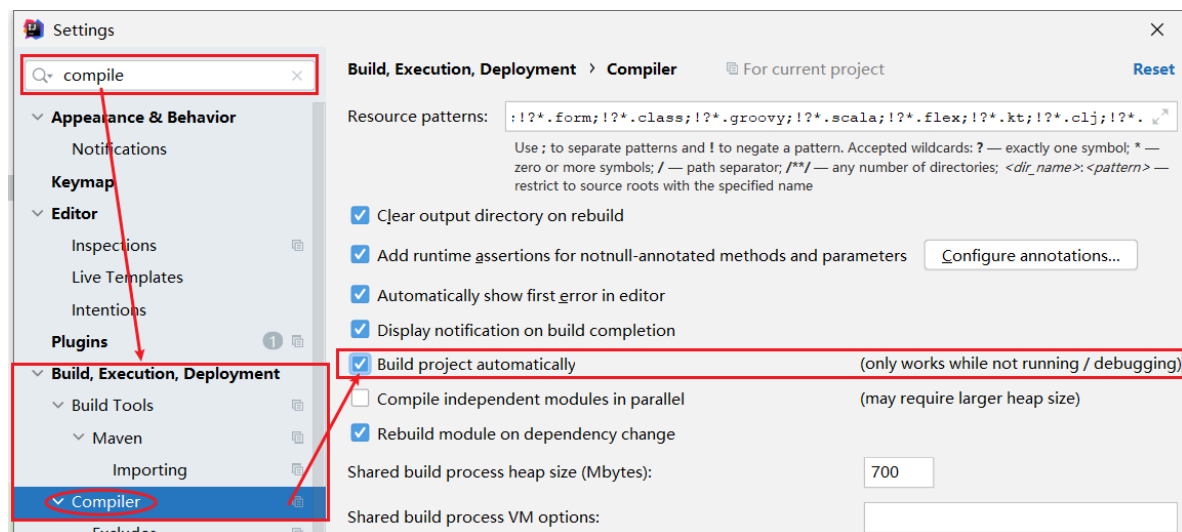
③ 运行项目后，访问：



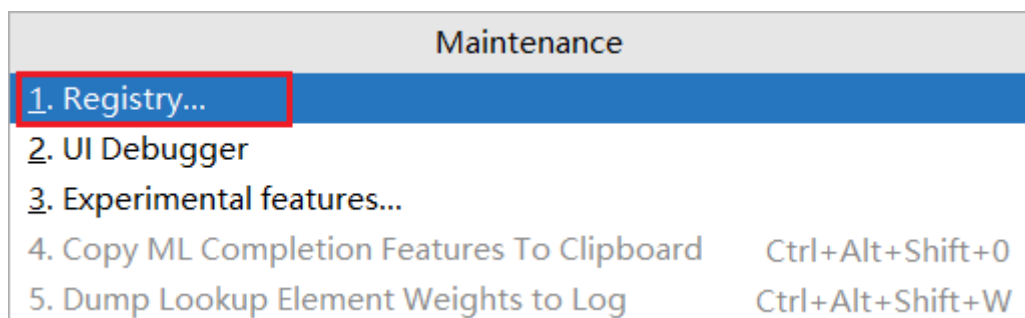
④ 修改控制器代码

```
@GetMapping("/index")
@ResponseBody
public String getIndex(){
    return "测试热部署--修改后";
}
```

⑤ 不重启项目，直接访问测试，此时并不会生效，原因并不是导入坐标的问题，而是IDEA本身默认情况下不会进行自动编译，所以没有触发SpringBoot的热部署，需要设置IDEA自动编译：



注意此时自动编译是在程序未运行或未调试时有效，因此需要继续配置使其能够在运行时也可以自动编译，使用快捷键 **CTRL + SHIFT + ALT + /** 打开维护对话框：



选择1后进入注册属性，其中找到 `compiler.automake.allow.when.app.running` 勾选上：



⑥ 完成自动编译的配置后，当进行代码的修改时，等待几秒钟后，IDEA会自动进行编译完成重新部署。

⑦ 另外在引入 `spring-boot-devtools` 依赖后，可能还需要在插件plugin处添加一个配置：

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <fork>true</fork> <!-- 默认值为false，可能需要设置为true才能启用热部署功能 -->
  </configuration>
</plugin>
```

实际上不添加时热部署也是可以生效的，默认情况下 `false` 表示Maven采用运行自身的JVM虚拟机运行插件，而通过 `true` 则告知Maven启动一个新的JVM虚拟机进程运行插件。启动新JVM虚拟机执行插件是比较耗费资源的，其使用的场景主要有：

- 1) 使用不同的JDK版本来运行插件，例如指定系统里安装的比较老的jdk1.5版本来运行项目
- 2) 采用不同的JVM配置来运行插件，例如指定了JVM运行的内存环境与默认值不一致时

6.3.3 小结

这种热部署由于需要自动构建项目，会需要很多的系统资源，另外每次还需要等待几秒才会自动编译构建，开发体验或许不佳。

所以可以在添加热部署的依赖 `spring-boot-devtools` 后，不设置自动构建，仍然采用手动构建的方式（`CTRL+F9` 或者小锤子图标进行项目编译构建、`CTRL+SHIFT+F9` Rebuild-项目重新构建），这样会在手动编译后自动进行部署。

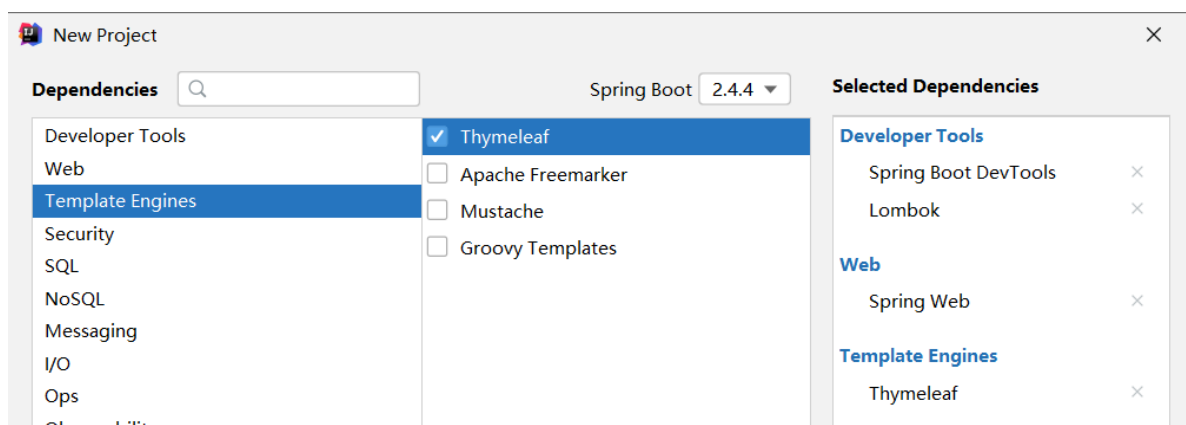
七、《图书管理系统》项目实战一

背景需求说明

使用SpringBoot框架，结合模板引擎技术和前端框架进行前端页面展示，完成《图书管理系统》SpringBoot版本，由于目前SpringBoot的持久层还没有进行处理，所以当前项目中采用模拟数据方式。

项目中模板引擎以 `Thymeleaf` 为例，前端框架以LayUI为例。

7.1 创建项目



应用热部署，需要勾选 `spring-boot-devtools`

New Project

Project name:

图书管理系统-SpringBoot

Project location:

D:\Projects\98 备课\安信工18级\2021春季\图书管理系统-SpringBoot

...

▼ More Settings

Module name:

图书管理系统-SpringBoot

Content root:

D:\Projects\98 备课\安信工18级\2021春季\图书管理系统-SpringBoot

📁

Module file location:

D:\Projects\98 备课\安信工18级\2021春季\图书管理系统-SpringBoot

📁

Project format:

.idea (directory based)

▼

Previous

Finish

Cancel

Help

关闭模板引擎缓存:

```
spring:
  thymeleaf:
    cache: false
```

7.2 导入前端框架

以 **LayUI** 为例，下载官方资源包后拷贝到静态资源目录下（以及其他的静态资源，如图片等）：

7.3 登录功能

7.3.1 创建登录页面

参考《企业级应用开发与设计》课程中LayUI中的课程案例 `login.html`。

注意，其中引入外部资源文件时，结合ThymeLeaf的表达式和SpringBoot中的静态资源处理（因为LayUI所在目录static默认就是静态资源查找目录，不需要特殊处理）

引入css文件:

```
<link rel="stylesheet" type="text/css" th:href="@{/lib/layui/css/layui.css}"/>
```

引入图片文件:

```

```

引入js文件:

```
<script th:src="@{/lib/layui/layui.js}"></script>
```

另外注意表单提交地址中, 使用Thymeleaf获取Web应用目录:

```
<form class="layui-form" th:action="${#request.getContextPath()} + '/login'"
method="post">
```

7.3.2 创建控制器访问登录页

```
@GetMapping("/login")
public String getLogin(){
    return "login";
}
```

7.3.3 获取登录表单请求模拟登录处理

```
@Data
public class User {

    private String loginname;
    private String password;
    private String username;
}
```

```
@PostMapping("/login")
public ModelAndView doLogin(User user) {
    ModelAndView mav = new ModelAndView();
    // 预设: 管理员 (admin 123456) 读者 (zhangsan 123456)
    if (user.getLoginname().equals("admin") && user.getPassword().equals("123456")){
        mav.setViewName("redirect:/admin/index");
    } else if (user.getLoginname().equals("zhangsan") &&
user.getPassword().equals("123456")){
        mav.setViewName("redirect:/reader/index");
    } else {
        mav.setViewName("login");
        mav.addObject("msg", "用户名或密码输入错误!");
    }
    return mav;
}
```

7.3.4 登录页面接受登录失败消息

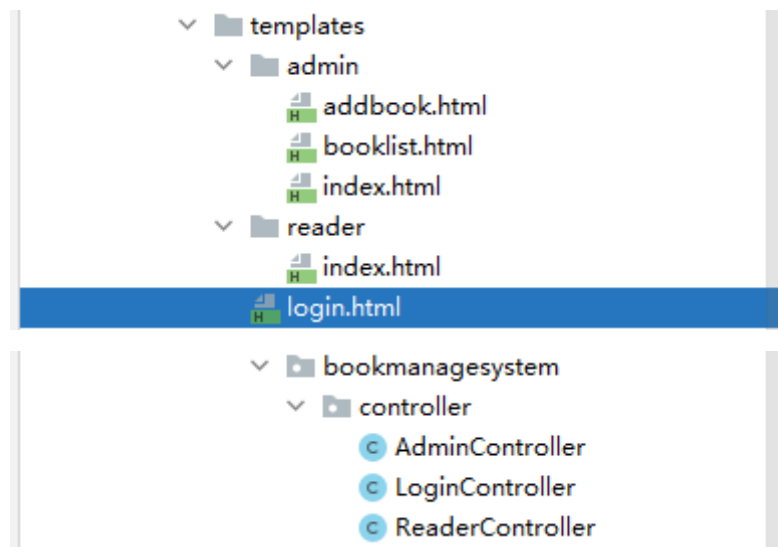
```
<script th:inline="javascript"> <!--注意要在js中使用Thymeleaf行内表达式获取域对象中的数据的写法-->
    layui.use(['layer', 'form'], function () {
        var layer = layui.layer
```

```

        , form = layui.form
        , jq = layui.jquery;
        jq(function(){
            let msg = [{${msg}}];
            if (msg != null && msg != "" && msg.length > 0){
                layer.alert(msg, {icon:5});
            }
        });
    });
</script>
或者
<script>
    ...
    let msg = ' [{${msg}} ]';
    ...
    });
</script>

```

7.3.5 处理登录成功后的首页跳转



```

@Controller
@RequestMapping("/admin")
public class AdminController {

    @GetMapping("/index")
    public String getIndex(){
        return "admin/index";
    }
}

```

```

@Controller
@RequestMapping("/reader")
public class ReaderController {

    @GetMapping("/index")
    public String getIndex(){
        return "reader/index";
    }
}

```

7.4 管理员模块

7.4.1 前端页面结构

- 首页页面
 - 图书管理
 - 图书列表页
 - 添加图书页
 - 读者管理
 - 读者列表页
 - 添加读者页
 - 审核注册页

首页见示例代码。

7.4.2 图书列表功能

7.4.2.1 前端页面

页面代码见示例文件。

注意，LayUI的数据表格在渲染时，如果使用了方法渲染的方式，其中在设置表头的 `cols` 参数里，采用的配置参数形式是：

```
cols: [[
  {field:'index', width:80, title: '序号'}
  ,{field:'name', width:'30%', title: '书名', sort: true}
  ...
]]
```

而这里两个中括号连接在一起的写法 `[[]]` 又会被Thymeleaf模板引擎解析，从而导致解析失败异常，解决方法有两种：

1. 最简单直接的将两个中括号分开即可：

```
cols: [ [
  ...
] ]
```

2. 在script标签里应用 `th:inline="none"`，不过此时在js中也无法使用行内表达式方便的获取域对象数据：

```
<script th:inline="none">
</script>
```

7.4.2.2 后端控制器

- 1、管理员首页内联框架页面：


```

@GetMapping("/booklist")
public String getBooklist(){
    return "admin/booklist";
}

@GetMapping("/addbook")
public String getAddbook(){
    return "admin/addbook";
}

```

2、图书列表页中图书数据获取及查询

```

@GetMapping("/booklist/all")
@ResponseBody
public Map<String, Object> getAll(int page, int limit, String bookname, String
authorname, String category){
    System.out.println("收到查询参数: 页码:" + page + " 书名:" + bookname + " 作者:" +
authorname + " 分类:" + category);
    Map<String, Object> map = new HashMap<>();
    // 此处应当调用IBookService相关方法查询指定参数的图书数据
    // int count = bookService.queryCountByParam(bookname, authorname, category); //
查询符合条件的图书数量
    // List<Book> books = bookService.queryByParam(bookname, authorname, category,
page, limit);
    List<Book> books = new ArrayList<>();
    for (int i = 0; i < limit; i++) {
        // 模拟假数据
        Book book = new Book();
        book.setId(i);
        book.setBookname("test" + page + "-" + i);
        book.setAuthor("test");
        book.setCategory("test");
        books.add(book);
    }

    // 按照LayUI表格数据格式传递结果数据
    map.put("code", 0);
    map.put("msg", "");
    map.put("count", 1000);
    map.put("data", books);

    return map;
}

```

7.4.3 添加图书功能

7.4.3.1 前端页面

页面代码见示例文件。

注意其中补充了一个div，用户获取添加结果信息：

```
<div class="layui-input-block" th:text="${r.lt}"></div>
```

7.4.3.2 后端控制器

此处仍然是模拟实现

```

@PostMapping("/addbook")
public ModelAndView doAddbook(Book book) {
    System.out.println("add-->" + book);
    ModelAndView mav = new ModelAndView("/admin/addbook");
    // 此处调用BookService方法将book存储到数据库中
    // bookService.insert(book);
    mav.addObject("r1t", "图书信息添加完成");
    return mav;
}

```

7.4.3 操作功能（修改和删除）

```

@PostMapping("/delete/{id}")
@ResponseBody
public Map<String, Object> doDelete(@PathVariable("id") int id) {
    System.out.println("---->删除图书: " + id);
    Map<String, Object> r1tMap = new HashMap<>();
    // 此处调用BookService方法将对应id的图书信息删除
    // int r1t = bookService.delete(id);
    r1tMap.put("code", 1);
    r1tMap.put("msg", "图书信息删除完成!");
    return r1tMap;
}

@GetMapping("/edit/{id}")
public ModelAndView getEdit(@PathVariable("id") int id) {
    System.out.println("---->修改图书: " + id);
    ModelAndView mav = new ModelAndView();
    // 此处调用BookService方法查询对应id的图书信息后转发到编辑页面中
    // Book book = bookService.queryById(id);
    // 模拟假数据
    Book book = new Book();
    book.setId(id);
    book.setBookname("测试要编辑的图书");
    book.setAuthor("test");
    book.setCategory("test");

    mav.setViewName("/admin/editbook");
    mav.addObject("book", book);
    return mav;
}

```

其他功能

由同学们参考已实现的功能模块进行实现。

八、SpringBoot多环境配置、性能优化、AOP集成

8.1 多环境配置

8.1.1 背景

通过前面的学习已经知道SpringBoot项目的默认配置文件是存放在 `main/resources` 下，且以 `application` 作为名称开头的 `propertie` 或者是 `yaml/yml` 格式的文件。在配置文件中可以配置整个SpringBoot所需的配置，例如端口，context-path，项目名称等，包括以后在工作中更进一步的新技术开发相关配置（例如SpringCloud的注册中心等等的配置信息），另外还可以配置自己自定义的配置项。

在正常的开发过程中会经历多个阶段，不同的阶段对应了不同的环境，例如测试环境、开发环境、生产环境等等，而每个环境下的配置却不一定相同，例如数据库、端口信息等，实际开发中不可能总是来修改配置文件信息然后重新启动项目，这就需要开发者配置对应于多个环境的配置文件，并在不同环境下启用不同的配置。

按照一般开发过程中的阶段，环境可以有这么几个：本地开发（dev）、项目组测试（test）、部署到预实际生产环境（pre），生产环境（prod），因此一般就会对应地创建这些配置文件。然后在默认配置文件中来指定启用哪个配置，或者启动项目时通过给定参数告知加载哪个配置，这种配置方式也是比较传统的 `静态配置方式`。

相对应的还有基于SpringCloud搭建的分布式配置中心，称为 `动态配置方式`，属于比较高级的技术，这里不考虑。

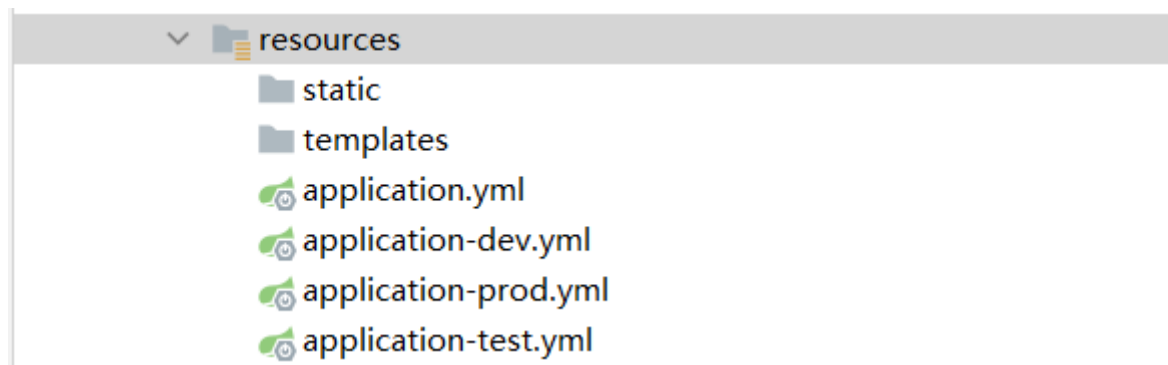
8.1.2 多环境配置过程（以yaml为例）

8.1.2.1 创建多个环境对应的配置文件

需要注意：多环境的配置文件的格式是固定的！

要求：

- 1) 如果SpringBoot当中使用的是yaml文件，那么多环境配置的文件也必须得是yaml文件，properties同理，简而言之，就是配置文件的类型必须保持统一。
- 2) 固定格式为：`application-*.yaml/properties`，其中 `*` 号部分是可以自定义的部分



8.1.2.2 定义配置内容

`application-dev.yaml`

```
server:
  port: 8081
info: 这是开发环境
```

`application-prod.yaml`

```
server:
  port: 8082
info: 这是生产环境
```

application-test.yml

```
server:
  port: 8083
info: 这是测试环境
```

8.1.2.3 确定当前使用的环境

在主配置文件 `application.yml` 当中使用 `spring.profiles.active` 属性来指定当前的环境配置, `active` 的值为上面创建的配置文件 * 号部分, 例如当前使用开发环境:

```
spring:
  profiles:
    active: dev
```

8.1.2.4 创建控制器进行测试

```
@Controller
public class TestController {

    @Value("${info:'未正确加载配置'}")
    private String info;

    @GetMapping("/info")
    @ResponseBody
    public String getInfo(){
        return info;
    }
}
```

Tomcat started on port(s): **8081** (http) with context path ''



8.1.2.5 运行时环境选择

现在在开发中可以很方便修改配置进行测试, 但是在工作中将一个项目打成jar包之后发布, SpringBoot 项目使用 `java -jar xxx.jar` 来启动项目。那么在将项目部署到生产环境, 然后发现了点问题又要部署到测试环境, 是不是需要在IDEA中修改一下配置文件, 然后重新打包部署呢?

这里有个更好的解决办法, 就是在执行jar时, 使用"--配置项=值", 来设置配置项:

```
java -jar xxxxx.jar --spring.profiles.active=test
```

8.2 性能优化

常规的性能优化策略从几个方面考虑:

- 包扫描优化
- JVM参数优化
- 内置服务器优化

8.2.1 包扫描优化

在启动类当中使用 `@SpringBootApplication` 注解来自动获取应用的配置信息，这样虽然方便，但是也有一些副作用。使用了这个注解会触发自动配置（auto-configuration）和组件扫描（component-scanning），通过阅读源代码知道该注解是一个复合注解，实际上应用了 `@Configuration`、`@EnableAutoConfiguration` 和 `@ComponentScan` 三个注解，在方便开发的同时却带了性能影响。

- 会导致项目启动时间变长，启动一个大型的应用项目或者将要做大量的集成测试启动应用程序时，影响会较为明显
- 会创建一些不必要的多余的Java Bean实例
- 增加CPU和内存损耗

优化的思路：移除 `@SpringBootApplication` 和 `@ComponentScan` 两个注解来禁用注解扫描，然后在需要的bean上手动使用 `@ComponentScan` 去定位具体功能需求的包进行定点扫描。

8.2.2 JVM参数优化

JVM中有两个参数和程序执行的内存相关：

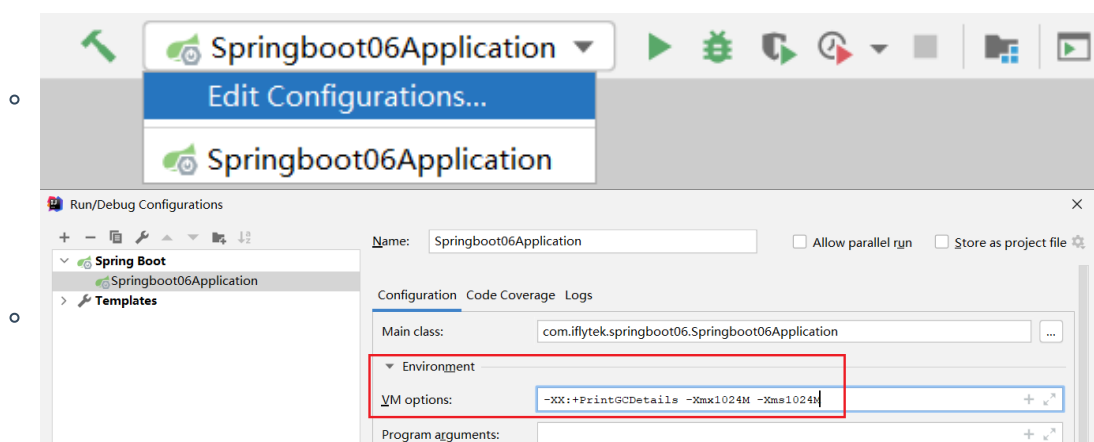
`-Xmx` : JVM最大堆内存大小

`-Xms` : JVM初始堆内存大小

JVM的优化策略是：在生产项目中，初始化堆内存和最大堆内存这两个值应该配置相同，并根据运行服务器硬件环境适当去配置。如果不一致的话，会造成频繁的进行垃圾回收次数。**最根本就是减少GC回收的次数**

JVM启动配置

- 外部启动
 - 使用cmd命令提示行当中的java命令实现：`java -server -Xms64M -Xmx64M -jar xxxx.jar`
- IDE中设置运行参数



- `PrintGCDetails`表示打印GC的详细日志，最大堆内存为1024M，初始堆内存为1024M

当配置好JVM内存参数后，通过测试项目的吞吐量观察内存配置是否合理（常用的性能测试工具有JMeter、LoadRunner等）。

吞吐量 = CPU在用户应用程序运行的时间 / （CPU在用户应用程序运行的时间 + CPU垃圾回收的时间）

简而言之，吞吐量就是每秒去完成的请求数，所以JVM的内存参数影响了GC回收次数，直接影响吞吐量。

进行 JVM 调优是一个谨慎细致的过程，需要慢慢的试，直到当前最优为止。

8.2.3 内置服务器优化

优化策略是根据项目采用合适的容器。

现在最流行也最常用的是Tomcat，默认是支持JSP的（SpringBoot官方并不推荐，特别在引入Thymeleaf或者FreeMarker后基本不考虑使用JSP了），当然也可以去优化Tomcat，关闭Tomcat对JSP的支持。

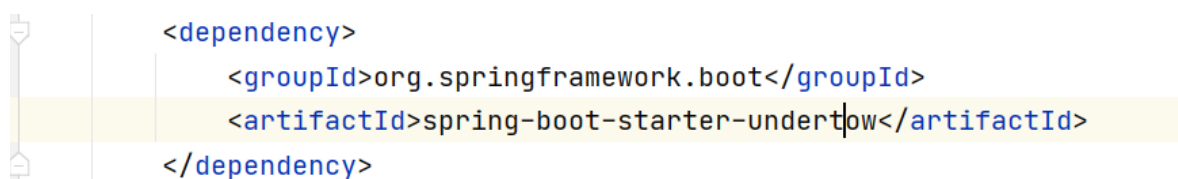
还可以在项目中使用其他容器，例如 **Undertow**、**Jetty**，效率上来看Undertow 优于 Jetty 优于 Tomcat。

优化过程：

1、首先在POM中排除Tomcat



2、导入Undertow依赖



启动观察：

: Undertow started on port(s) 8081 (http)

8.3 集成AOP技术

8.3.1 概念回顾

8.3.1.1 是什么

来自百度百科：

“在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。”

简单的说它就是把我们程序重复的代码抽取出来，在需要执行的时候，使用动态代理的技术，在不修改源码的基础上，对我们的已有方法进行增强。

8.3.1.2 名词解释

- Joinpoint(连接点): 所谓连接点是指那些被拦截到的点。在 spring 中,这些点指的是方法,因为 spring 只支持方法类型的连接点。例如我们的业务层接口中,你看到的方法都是连接点,这些方法就是连接我们的业务和增强方法之间的点。如何把增强方法比如记录日志方法加到业务中来呢?就是通过这些方法。
- Pointcut(切入点): 所谓切入点是指我们要对哪些 Joinpoint 进行拦截的定义。也就是将要被增强的连接点。例如某个接口中有很多个方法,这些方法都是连接点,但是我们进行增强的时候做了判断,某个方法不进行增强,那么它就不属于切入点了。
- Advice(通知/增强): 所谓通知是指拦截到 Joinpoint 之后所要做的事情就是通知。有四种类型:前置通知、后置通知、异常通知、最终通知 还有一种明确调用切入点方法并自定义通知执行时机的环绕通知
- Introduction(引介): 引介是一种特殊的通知。在不修改类代码的前提下, Introduction 可以在运行期为类动态地添加一些方法或 Field。
- Target(目标对象): 代理的目标对象,也就是被代理对象,就是包含连接点的对象。
- Weaving(织入): 是指把增强应用到目标对象来创建新的代理对象的过程。spring 采用动态代理织入,而 AspectJ 采用编译期织入和类装载期织入。
- Proxy (代理): 一个类被 AOP 织入增强后,就产生一个结果代理类。
- Aspect(切面): 是切入点和通知(引介)的结合。在Spring中我们通过配置的方式来做一些操作,就需要描述出切入点和通知的关系,通常是一个类,里面可以定义切入点和通知

8.3.2 SpringBoot中的AOP集成

在前面的课程学习中,其实已经应用过了,例如日志框架的引入、全局异常处理的 `ControllerAdvice` 注解。

而在SpringBoot的项目中通常还需要掌握使用AOP集成日志框架后进行自定义日志处理。

8.3.2.1 导入AOP依赖

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

8.3.2.2 添加测试控制器方法

```
@Controller
public class UserController {

    @GetMapping("/find")
    @ResponseBody
    public String findUser(int id){
        String name = "张三"; // 模拟根据用户id查找用户的操作
        return "模拟查找结果: " + name + "...";
    }
}
```

8.3.2.3 编写切面类

接下来就来编写一个日志切面,这就是一个普通的Java类,它的作用就是在这里配置一些通知,并声明通知和切入点之间的关系,比如这里的查找用户的findUser方法。

- 1) 创建一个切面类 `\aspect\WebLogAspect.java` :

```

@Aspect // 表明这是一个切面
@Component // 注入容器
@Slf4j // 引入日志工具类
public class WebLogAspect {

```

2) 编写切入点表达式

```

@Pointcut("execution(* com.iflytek.springboot06.controller.*.*(..))")
public void logPointCut(){
    // 该方法无方法体,主要为了让同类中其他方法使用此切入点
}

```

3) 添加一个通知 (就是进行日志记录的方法)

```

@Before("logPointCut()")
public void beforeLog() {
    // 获取请求
    ServletRequestAttributes attributes = (ServletRequestAttributes)
    RequestContextHolder.getRequestAttributes();
    HttpServletRequest request = attributes.getRequest();

    // 进行日志记录
    log.info("访问URL: " + request.getRequestURI().toString());
    log.info("请求的方法: " + request.getMethod());
    log.info("访问者地址: " + request.getRemoteAddr());
    Map<String, String[]> parameterMap = request.getParameterMap(); // 请求的参数
    for (Map.Entry<String, String[]> entry : parameterMap.entrySet()){
        log.info("请求参数: " + entry.getKey() + " 参数值: " +
        Arrays.toString(entry.getValue()));
    }
}

```

4) 测试



观察输出:

```

15116 --- [ XNIO-1 task-1] c.i.springboot06.aspect.WebLogAspect : 访问URL: /find
15116 --- [ XNIO-1 task-1] c.i.springboot06.aspect.WebLogAspect : 请求的方法: GET
15116 --- [ XNIO-1 task-1] c.i.springboot06.aspect.WebLogAspect : 访问者地址: 0:0:0:0:0:0:1
15116 --- [ XNIO-1 task-1] c.i.springboot06.aspect.WebLogAspect : 请求参数: id 参数值: [1]

```

5) 还可以继续添加一个后置通知观察返回结果

```

/*
这里可以通过给这个通知方法添加一个参数,并在注解中关联参数,之后就可以在这个方法中来输出看看切入点到底
返回了什么
*/
@AfterReturning(pointcut = "logPointCut()", returning = "result")
public void afterLog(Object result){
    log.info("控制器方法返回: " + result);
}

```

测试结果:

```

[ XNIO-1 task-1] c.i.springboot06.aspect.WebLogAspect : 控制器方法返回: 模拟查找结果: 张三...

```