

一、SpringBoot入门

介绍

为什么要使用

Spring框架的引入减轻了Java企业级应用开发的压力。

但是它也是有缺点的：大量的框架配置、Bean之间的注入或依赖关系管理、项目的依赖管理等。

SpringBoot主要是为了解决Spring框架本身的一些缺点而出现的。

SpringBoot概述

SpringBoot对Spring的缺点进行改善和优化，基于 **约定优先于配置** 的思想

让开发人员不必在配置和业务逻辑之间进行思维的切换。

特点

- 为基于spring的开发提供更快的入门体验
- 开箱即用，没有代码生成，也无需XML配置，同时也支持修改默认值进行自定义
- 提供了一些大型项目中常见的非功能性特性，如嵌入式服务器、安全、指标、健康检测、外部配置等
- 不是对Spring功能的增强，而是提供了一种快速使用Spring的方式

核心功能

- 起步依赖Starter
 - 起步依赖本质上是一个Maven项目对象模型（Project Object Model，POM），定义了对其他库的传递依赖，这些东西加在一起即支持某项功能。
 - 简单的说，起步依赖就是将具备某种功能的坐标打包到一起，并提供一些默认的功能。
- 自动配置AutoConfiguration
 - SpringBoot的自动配置是一个运行时（更准确地说，是应用程序启动时）的过程，考虑了众多因素，才决定Spring配置应该用哪个，不该用哪个。该过程是Spring自动完成的。
 - SpringBoot可以自动判断哪个对象内部需要什么东西，然后自动完成注入和配置：例如配置SqlSessionFactory，它需要一个DataSource，之前我们都是自己来配置，然后把数据源注入到SqlSessionFactory中的，那在SpringBoot中，可以判断出SqlSessionFactory是一定要有数据源的，那就自动配置好并进行注入。

SpringBoot项目创建

创建一个SpringBoot项目的过程可以分为几步走：

1. 创建一个Maven项目
2. 导入项目需要的起步依赖Starter

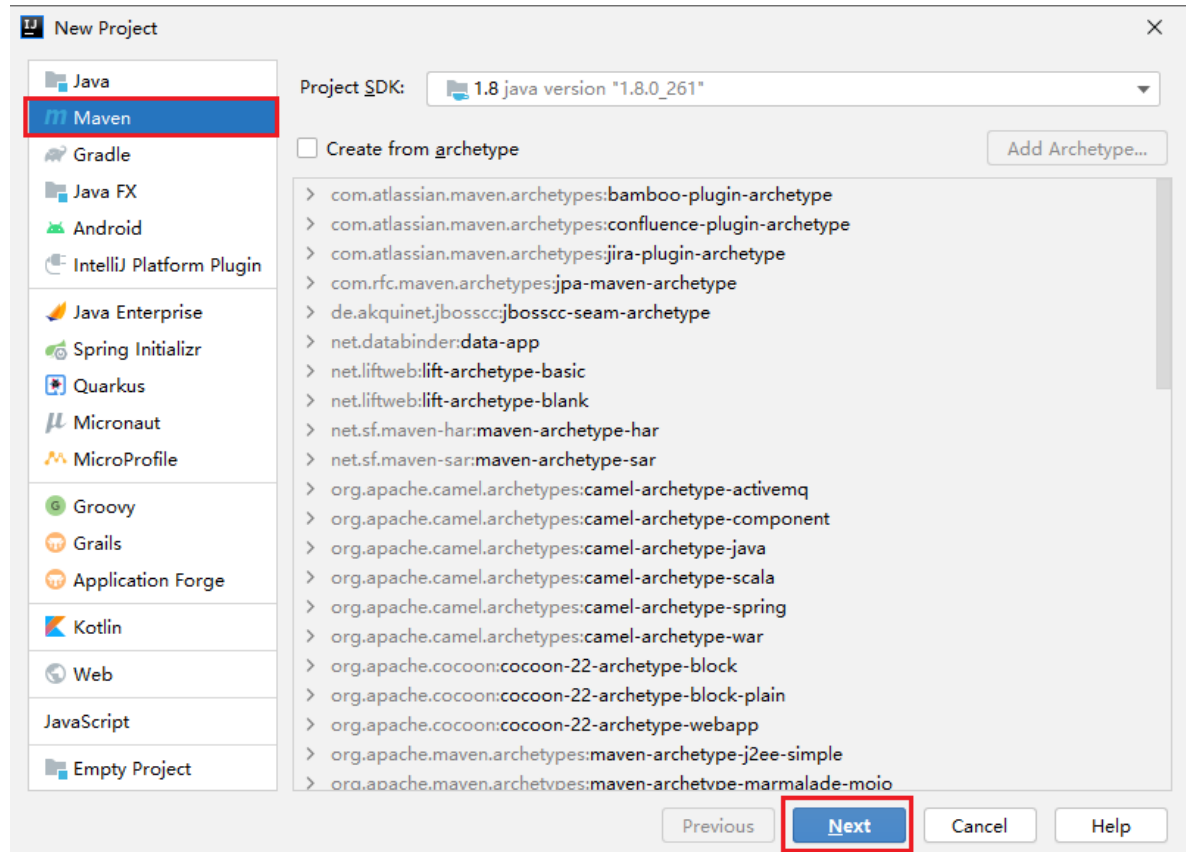
3. 编写入口引导类

而有些开发工具对SpringBoot项目支持的较好，例如IDEA，可以直接创建SpringBoot项目（注意：IDEA社区版需要额外安装插件）。

普通方式创建SpringBoot项目（Eclipse或IDEA）

1、创建Maven项目

创建项目时，不管是Web工程还是普通Java工程，都只需要创建基本的Java程序就可以了，不需要选择骨架来创建项目



2、添加起步依赖

SpringBoot要求，项目要继承SpringBoot的起步依赖（spring-boot-starter-parent），这意味着后续我们开发的SpringBoot工程所添加的起步依赖都是这个Paranet的子依赖。

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.0.RELEASE</version>
</parent>
```

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <parent>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-parent</artifactId>
10        <version>2.1.0.RELEASE</version>
11    </parent>
12
13    <groupId>com.iflytek</groupId>
14    <artifactId>SpringBoot01</artifactId>
15    <version>1.0-SNAPSHOT</version>

```

```

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

```

```

7 <parent>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-parent</artifactId>
10    <version>2.1.0.RELEASE</version>
11</parent>
12
13<groupId>com.iflytek</groupId>
14<artifactId>SpringBoot01</artifactId>
15<version>1.0-SNAPSHOT</version>
16
17<dependencies>
18    <dependency>
19        <groupId>org.springframework.boot</groupId>
20        <artifactId>spring-boot-starter-web</artifactId>
21    </dependency>
22</dependencies>
23
24</project>

```

Maven Dependencies:

- org.springframework.boot:spring-boot-starter-web:2.1.0.RELEASE
- org.springframework.boot:spring-boot-starter:2.1.0.RELEASE
- org.springframework.boot:spring-boot-starter-json:2.1.0.RELEASE
- org.springframework.boot:spring-boot-starter-tomcat:2.1.0.RELEASE
- org.hibernate.validator:hibernate-validator:6.0.13.Final
- org.springframework:spring-web:5.1.2.RELEASE
- org.springframework:spring-webmvc:5.1.2.RELEASE

配置到此结束。

3、编写引导类

这里要编写一个引导类，主要就是要提供一个入口函数，就是main函数，让它负责来进行引导。

```

src
├── main
│   └── java
│       ├── com
│       │   ├── iflytek
│       │   └── springboot
│       └── MySpringBootApplication

```

```

@SpringBootApplication
public class MySpringBootApplication { // 类名任意，例如AppRun等，注意可读性。
    public static void main(String[] args) {
        SpringApplication.run(MySpringBootApplication.class);
    }
}

```

- 通过SpringBoot来执行这个引导类：
`SpringApplication.run(MySpringBootApplication.class);`
- 这个类仅仅是个普通的Java类，名字叫什么改变不了它的性质，需要通过
`@SpringBootApplication` 注解来标识它是SpringBoot的引导类（注意注解不要写错了）

4、启动运行该程序

现在已经可以运行该程序了，将在控制输出字符版本信息：

[illegible]

此外，观察启动日志，显示tomcat已经启动，其web应用名称为空：

```
main] o.s.b.w.servlet.ServletRegistrationBean : Servlet dispatcherServlet mapped to [/]
main] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'characterEncodingFilter' to: [//*]
main] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [//*]
main] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'formContentFilter' to: [//*]
main] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: [//*]
main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path '/'
main] c.i.springboot.MySpringBootApplication : Started MySpringBootApplication in 2.817 seconds (JVM running for 3.365)
```

那服务器启动了，是否可以访问呢？在浏览器中输入地址进行访问 <http://localhost:8080/>：



Whitelabel Error Page

This application has no explicit mapping for `/error`, so you are seeing this as a fallback.

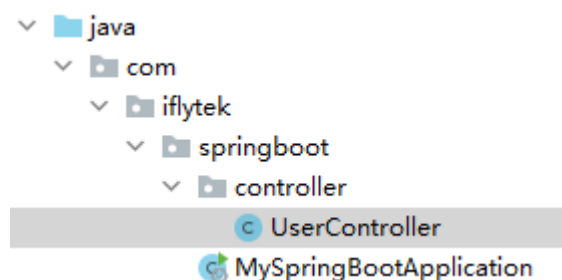
Mon Mar 01 17:56:07 CST 2021

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

No message available

看到这可以看出，服务器的确是启动了的，只不过由于没有编写任何页面所以报错，但是是实在连上了的。

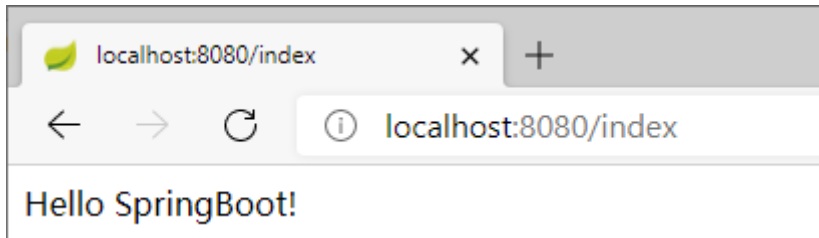
5、编写控制器Controller



```
@Controller
public class UserController {

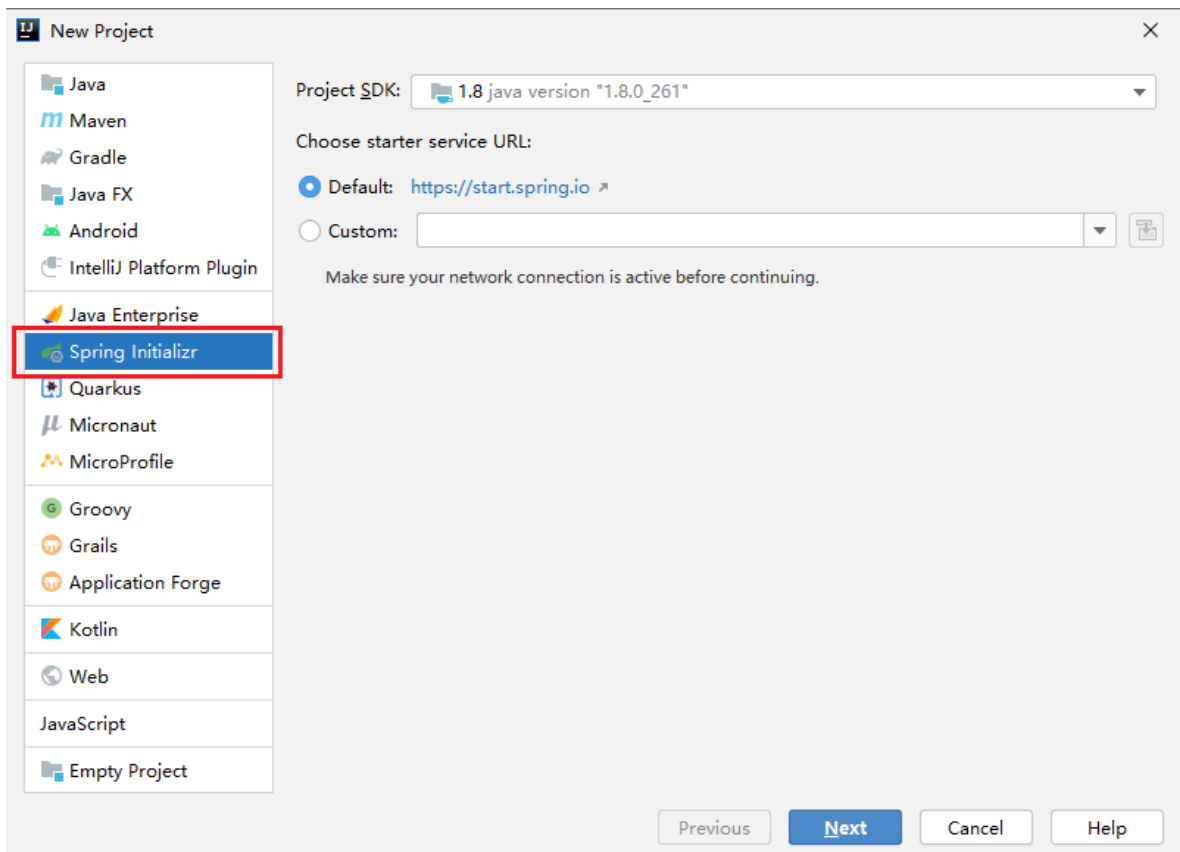
    @GetMapping("/index")
    @ResponseBody
    public String getIndex(){
        return "Hello SpringBoot!";
    }
}
```


然后重启项目并访问：



IDEA快速创建SpringBoot项目

1、创建Spring Initializr项目



 New Project ✕

Spring Initializr Project Settings

Group:

com.iflytek

Artifact:

springboot02

Type:

☒ Maven ☐ Gradle

Language:

☒ Java ☐ Kotlin ☐ Groovy

Packaging:

☒ Jar ☐ War

Java version:

8

Version:

0.0.1-SNAPSHOT

Name:

springboot02

Description:

Demo project for Spring Boot

Package:

com.iflytek.springboot02


Previous

Next

Cancel

Help

2、勾选需要的起步依赖

 New Project ✕

Dependencies

Spring Boot 2.4.3

Selected Dependencies

Developer Tools

Web

Template Engines

Security

SQL

NoSQL

Messaging

I/O

Ops

Observability

Testing

Spring Cloud

Spring Cloud Security

Spring Cloud Tools

Spring Cloud Config

Spring Cloud Discovery

Spring Cloud Routing

Spring Cloud Circuit Breaker

Spring Cloud Messaging

Pivotal Cloud Foundry

Amazon Web Services

☒ Spring Web

☐ Spring Reactive Web

☐ Rest Repositories

☐ Spring Session

☐ Rest Repositories HAL Explorer

☐ Spring HATEOAS

☐ Spring Web Services

☐ Jersey

☐ Vaadin

Spring Web

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

[Building a RESTful Web Service](#)

[Serving Web Content with Spring MVC](#)

[Building REST services with Spring](#)

Web

Spring Web

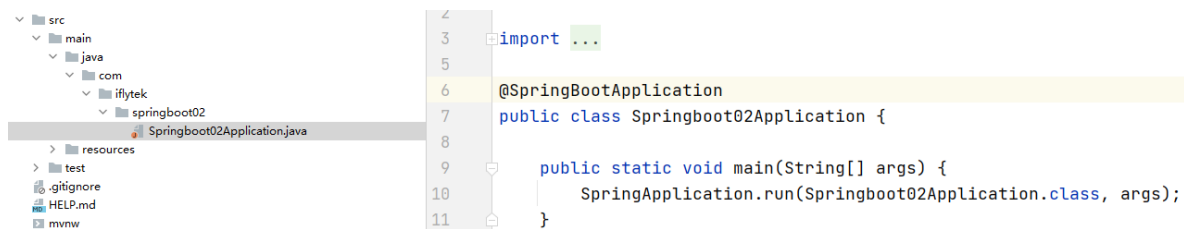
Previous

Next

Cancel

Help

3、完成创建



创建好项目之后可以发现，默认已经给我们导入了需要的起步依赖，另外在java文件夹中也提前创建了一个类，类名是根据项目名称修改来的。可以看到其中就是我们之前入门中写过的代码。

测试也很简单，一样的添加一个Controller进行测试就行了

起步依赖

首先从 `<parent>` 中看：

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.4.3</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

按住 **CTRL** 点击进入查看：

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.4.3</version>
</parent>
```

发现其中又依赖了 `spring-boot-dependencies`，继续进入其中可发现：

```
<properties>
  <activemq.version>5.16.1</activemq.version>
  <antlr2.version>2.7.7</antlr2.version>
  <appengine-sdk.version>1.9.86</appengine-sdk.version>
  <artemis.version>2.15.0</artemis.version>
  <aspectj.version>1.9.6</aspectj.version>
```

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot</artifactId>
  <version>2.4.3</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-test</artifactId>
  <version>2.4.3</version>
</dependency>

```

可以看到这里面配置了很多的依赖的版本，在下面还有很多的依赖管理，并且根据spring-boot的版本，每个导入的依赖中也对版本做了一些约定。

另外，在 `parent` 中的 `<build>` 节点里还有：

```

<resource>
  <directory>${basedir}/src/main/resources</directory>
  <filtering>true</filtering>
  <includes>
    <include>**/application*.yml</include>
    <include>**/application*.yaml</include>
    <include>**/application*.properties</include>
  </includes>
</resource>

```

可以看到它会包含系统的资源目录下的application开头的yml、yaml和properties的文件，这些是spring-boot的相关配置，当没有的时候就是按照默认的配置，如果有需要自定义的部分，我们就可以创建以application开头的配置文件。

再看看 `web` 部分，当导入了web功能的起步依赖后，就不需要导入其他的依赖了，是因为在web中已经导入了相关的依赖：

```

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

```



```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>5.3.4</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.3.4</version>
  <scope>compile</scope>
</dependency>
```

另外还有tomcat相关的依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
  <version>2.4.3</version>
  <scope>compile</scope>
</dependency>
```

自动配置原理

SpringBoot的自动配置就是把一些默认的东西加载进去，不用手动来配置东西。

1) 关键注解@SpringBootApplication

从最关键的注解 `@SpringBootApplication` 出发：

```

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
        type = FilterType.CUSTOM,
        classes = {TypeExcludeFilter.class}
    )}, @Filter(
        type = FilterType.CUSTOM,
        classes = {AutoConfigurationExcludeFilter.class}
    })
)
public @interface SpringBootApplication {

```

2) 配置标记注解@SpringBootConfiguration

继续看到关键注解 `SpringBootConfiguration` 中：

```

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

```

可以看到它上面还有一个 `@Configuration` 注解，这个注解在学习Spring的时候已经学习过了，它的作用就是标记该类是Spring的配置类。

3) 核心注解@EnableAutoConfiguration

另外在 `SpringBootApplication` 注解上还有一个注解 `EnableAutoConfiguration`，从名字上可以看出它的作用，它就是SpringBoot实现自动配置的核心注解。

进入该注解源码中，其定义上有一个导入注解：

```

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

```

这里的 `@Import` 注解的作用就是在当前 `EnableAutoConfiguration` 注解下引入其他的配置类，让当前被注解的类也具备被导入类的功能。那么这个被导入的类做了什么呢？

3-1. 关键类 `AutoConfigurationImportSelector`

该类用于处理自动配置，其实现了 `DeferredImportSelector` 接口，该接口是 `ImportSelector` 接口的子接口，而 `ImportSelector` 接口主要作用是收集需要导入的配置类，其中包含一个方法 `selectImports`，就是用来选择并返回需要被导入的类名称。

在类 `AutoConfigurationImportSelector` 中对该方法进行了实现，并且其中调用了方法 `getAutoConfigurationEntry`，：

```

@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(annotationMetadata);
    return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}

```

而在此方法中又调用了 `getCandidateConfigurations`（获取候选配置）：

```

protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
    configurations = removeDuplicates(configurations);
}

```

这个方法中使用 `SpringFactoriesLoader.loadFactoryNames` 去加载信息，注意看下面的断言提示：

```

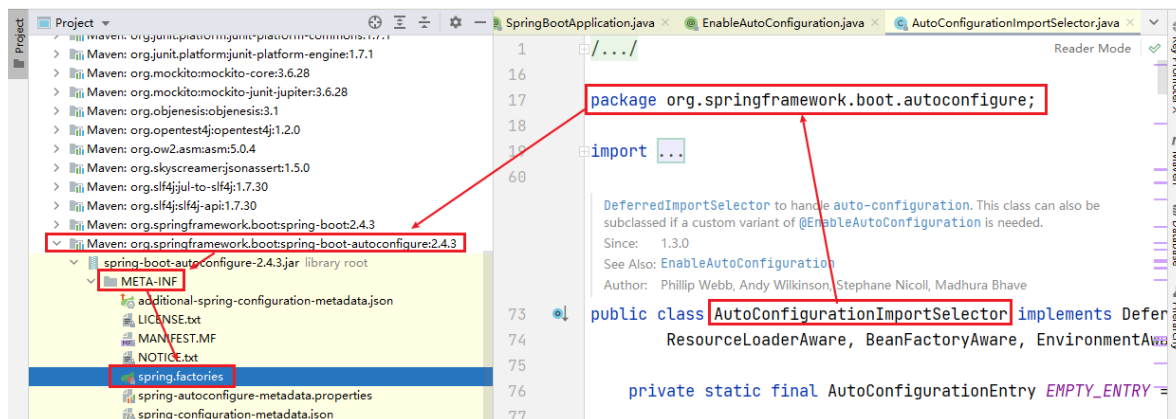
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(),
        getBeanClassLoader());
    Assert.notEmpty(configurations, "message: \"No auto configuration classes found in META-INF/spring.factories. If you \"
        + \"are using a custom packaging, make sure that file is correct.\"");
    return configurations;
}

```

可以看出如果这里加载出问题了，将会报一个错误信息，其中提到如果使用了 "custom packaging" 自定义包，需要确保在 `META-INF/spring.factories` 中存在自动配置的类型信息。

3-2. 自动配置类型表

这里的 `META-INF/spring.factories` 文件是指当前类所在的包中的配置文件，到加载的外部库中去查找它：



```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
```

在这些自动配置类中，有一些和Servlet相关（即和web相关）：

```
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\
```

而这里的类，在其定义上又明确表示在一个特殊的类之后再自动配置：

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class, TaskExecutionAutoConfiguration.class, ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {

    @AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
    @Configuration(proxyBeanMethods = false)
    @ConditionalOnWebApplication(type = Type.SERVLET)
    @ConditionalOnClass(DispatcherServlet.class)
    @AutoConfigureAfter(ServletWebServerFactoryAutoConfiguration.class)
    public class DispatcherServletAutoConfiguration {
```

这个类其实也在之前的自动配置类列表中：

```
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfiguration,\
```

这个类就是专门给Servlet web服务端程序进行自动配置的。

3-3.ServerProperties

点击进入该类，发现该类上有一个注解--启用配置属性，并指定配置的属性来自 `ServerProperties` 类：

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication
@EnableConfigurationProperties(ServerProperties.class)
public class EmbeddedWebServerFactoryCustomizerAutoConfiguration {

```

而在这个 `ServerProperties` 服务端属性类之中：

```

@ConfigurationProperties(prefix = "server", ignoreUnknownFields = true)
public class ServerProperties {

    /**
     * Server HTTP port.
     */
    private Integer port;

    /**
     * Network address to which the server should bind.
     */
    private InetAddress address;

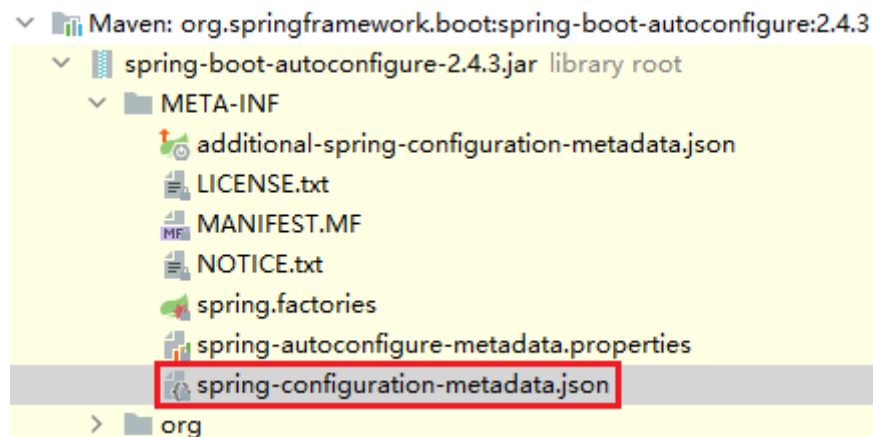
```

其上有一个表示配置属性的注解，其中有一个前缀信息 `prefix`，这里的前缀再和类中的属性进行结合，例如 `server.port`、`server.address` 等，就表示了一个配置属性了。

而这些配置属性中的一部分是有默认值的，例如Tomcat服务器默认端口8080，在哪里进行了默认配置呢？

3-4.自动配置默认数据

其实在之前的 `META-INF` 下，就有一个存放了默认数据的文件：



其中有很多的json字符串，可以搜索 `server.port` 找到server的端口等信息：

```

{
  "name": "server.port",
  "type": "java.lang.Integer",
  "description": "Server HTTP port.",
  "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties",
  "defaultValue": 8080
},

```

其他的数据也都是类似的，在这里事先准备好了很多的默认数据，然后通过自动配置相关的类和注解，引入完之后被相关的类进行加载，最终完成了SpringBoot的自动配置。

4) 修改默认配置

了解了整个过程及原理，就可以自己手动来修改默认配置文件了。

4-1.在resources下建立配置文件 `application.properties`

如果已经自动创建了，可以省略。

4-2.修改默认端口

```
# 服务器的端口
server.port=8081

# Web应用名称
server.servlet.context-path=/baidu
```

4-3.启动应用

```
: Tomcat started on port(s): 8081 (http) with context path '/baidu'
```

5) 注解ComponentScan

最后，在SpringBootApplication上还有一个注解：

```
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

这个注解在Spring中也学习过，它的作用就是在Spring在创建容器时要扫描的包，没有指定basePackages属性采用默认值，就会扫描本包或者子包中的注解

6) 小结

可以看到 `@SpringBootApplication` 一个注解起到了3个作用：

1. `Configuration` 注解
2. 自动配置
3. 注解扫描

二、SpringBoot配置文件

配置文件类型介绍

SpringBoot是基于约定的，很多的配置都有默认值，但是如果想使用自己的配置替换默认配置，例如修改服务器端口号，指定web应用名称，还有我们要配置数据源，那么数据库驱动、数据库地址、用户名密码这些信息肯定没有默认配置，这就需要通过配置文件的方式来告诉SpringBoot怎么连接处理。

SpringBoot可以使用 `application.properties` 或者 `application.yml` (yaml) 文件进行配置。

SpringBoot默认会自动包含resources目录下加载以application开头的properties或者yaml、yml文件。properties属性文件内部内容就是键值对形式的数据，这里不再多说，主要说说yaml文件如何配置。

YAML

YAML是"YAML Ain't a Markup Language" (YAML不是一种标记语言) 的递归缩写。在开发的这种语言时，YAML 的意思其实是："Yet Another Markup Language" (仍是一种标记语言)，但为了强调这种语言以数据做为中心，而不是以标记语言为重点，而用反向缩略语重命名。

YAML是一种直观的能够被电脑识别的数据序列化格式，并且易于阅读，可以被支持YAML库的不同的编程语言程序导入，以数据为核心，比传统的xml方式更加简洁。扩展名可以是 `yaml` 或者 `yml`。

如果有多个同名不同后缀名的文件，例如application.yaml、application.yml、application.properties，那么加载谁呢？

按照 `spring-boot-starter-parent` 的POM文件中的资源描述顺序，先包含yaml、然后是yml，最后是properties，意味着后加载的会把前面加载的配置覆盖掉，所以一般只使用一个配置文件就可以了。

YML语法

YAML的语法形式有点类似 `JSON`，但是是使用空格来描述层级关系，另外，使用缩进表示层级关系。缩进时不允许使用Tab键，只允许使用空格，缩进的空格数目不重要，只要相同层级的元素左侧对齐即可，大小写敏感。

普通数据

- 形式：`key: value`，注意冒号后有一个空格
- 示例：

```
# 普通数据
name: zhangsan
```

对象数据

- 形式：层级形式、行内形式
- 1、层级形式：

```
对象名：
  属性1: 值1
  属性2: 值2
```

- 示例：

```
# 对象数据
student:
  name: zhangsan
  age: 20
  major: Java

# 按层级形式配置server对象数据
server:
  port: 8082
  servlet:
    context-path: /iflytek
```

- 2、行内形式：`对象名: {属性1: 值1, 属性2: 值2}`
 - 总之，注意冒号和值之间有一个空格
 - 示例：

```
# 行内对象配置
student: {name: zhangsan, age: 20, major: Java}

# 行内server配置
server: {port: 8082, servlet: {context-path: /iflytek}}
```

- 一般第一种表示形式使用的较多，采用缩进来表示层级关系

数组、集合数据

- 集合（或数组）中存放普通字符串数据
 - 示例：

```
# 形式1配置集合数据
city1:
  - beijing
  - tianjin
  - chongqing
  - shanghai

# 形式2配置集合数据
city2: [beijing,tianjin,chongqing,shanghai]
```

- 集合（或数组）种存放对象数据
 - 示例：

```
# 配置集合数据（集合存放对象）
students1:
  - name: zhangsan
    age: 20
    major: Java
  - name: lisi
    age: 19
    major: PHP

students2: [{name: zhangsan, age: 20, major: Java}, {name: lisi, age: 19,
major: PHP}]
```

- Map数据
 - Map数据的表示形式和对象数据的表示形式相同。

YML配置文件属性映射

前面的数据配置中，类似 `server.port` 这样的数据，SpringBoot本身就能识别，但是如果是自己的数据比如 `student`、`city` 这些数据，springboot是不能识别的，那如何把这些数据映射到自己的业务代码中？

建立一个控制器Controller：

```
@Controller
public class TestController {

    private String name;
    private int age;
    private String major;

    @ResponseBody
    @RequestMapping("/test1")
    public String getInfo(){
        return "name:" + name + ",age:" + age + ",major:" + major;
    }
}
```


对于配置数据：

```
# 对象数据
student:
  name: zhangsan
  age: 20
  major: Java
```

将数据值注入到控制器的属性中。

有两种方式：1、使用 `@Value` 注解 2、使用 `@ConfigurationProperties` 注解

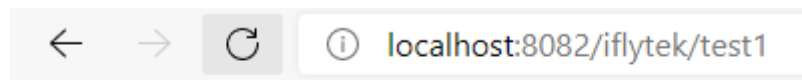
使用@Value注解

`@Value` 注解是Spring中的注解，在字段、方法或构造器的参数级别上使用，用来指定一个默认值表达式，在这个默认值表达式中可以使用SpEL（Spring Expression Language）读取配置文件中的属性值。

使用注解将其注入到控制器的属性中：

```
@Value("${student.name}")
private String name;
@Value("${student.age}")
private int age;
@Value("${student.major}")
private String major;
```

启动项目后访问：



name:zhangsan,age:20,major:Java

还可以在使用时添加默认值：

```
@Value("${student.name:lisi}")
private String name;
@Value("${student.age:18}")
private int age;
@Value("${student.major:PHP}")
private String major;
```

当对表达式获取属性值时，若没有查找到该属性，则使用冒号后的默认值注入。

优点在于更加精确的匹配，缺点就是比较繁琐，如果配置信息特别多，那么Value注解需要写多个。

使用@ConfigurationProperties注解

新建一个Controller，定义private属性，提供getter和setter，并在类上添加

`@ConfigurationProperties` 注解：

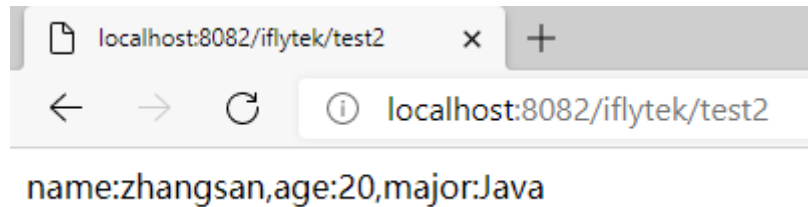
```
@Controller
@ConfigurationProperties(prefix = "student")
public class TestController2 {

    private String name;
    private int age;
    private String major;
```

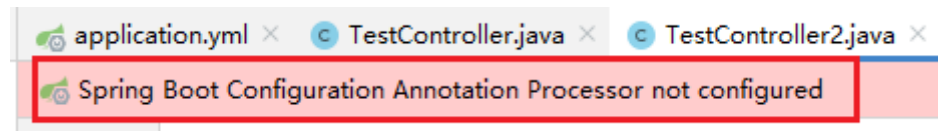
这个注解中需要提供一个 `prefix`，就是前缀，在类里面要获得的是 `student.name`，`student.major`，那么这里的 `student` 就是一个前缀，因此在注解上标记前缀信息。

之后添加访问方法进行测试：

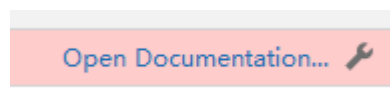
```
@ResponseBody
@RequestMapping("/test2")
public String getInfo(){
    return "name:" + name + ",age:" + age + ",major:" + major;
}
```



但是注意，此时IDEA会给出一个警告：



但是我们的运行结果仍然可以映射成功，这个信息是说这个配置的注解的执行器还没有配置，可以点击右边的打开文档，查看执行器的配置信息：



在打开的官方文档中有如下说明：

3.1. Configuring the Annotation Processor

To use the processor, include a dependency on `spring-boot-configuration-processor`.

With Maven the dependency should be declared as optional, as shown in the following example:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
  <optional>true</optional>
</dependency>
```

因此，拷贝依赖信息到项目的POM文件中。

```
<!-- 配置@ConfigurationProperties注解的执行器 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
  <optional>true</optional>
</dependency>
```

其他

- 1、`@Value` 注解不支持松散绑定（松散语法），`@ConfigurationProperties` 注解则支持，例如在配置文件中使用 `first-name` 这种带短横线的表示方式，而在类中使用 `firstName` 命名，对于后者则也可以获取配置的值。
- 2、`@Value` 注解需要对每个字段都添加，`@ConfigurationProperties` 只需要写一次

使用场景：

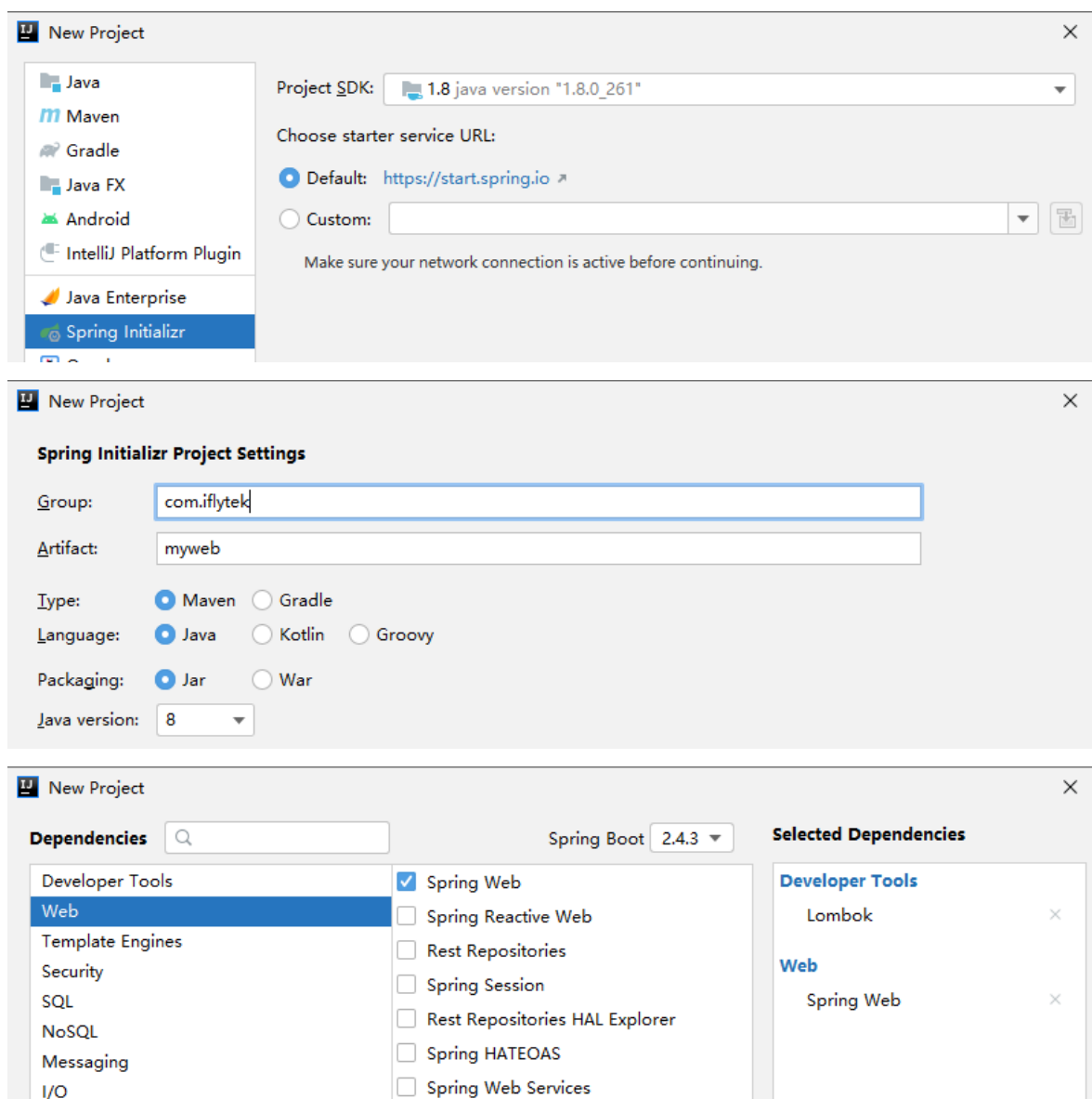
如果在业务代码中需要获取配置文件中的某个值，可以使用 `@Value` 注解，而如果专门编写一个 `JavaBean` 并通过配置文件进行映射，则应该使用 `@ConfigurationProperties` 注解。

三、SpringBoot进行Web开发

Web开发过程

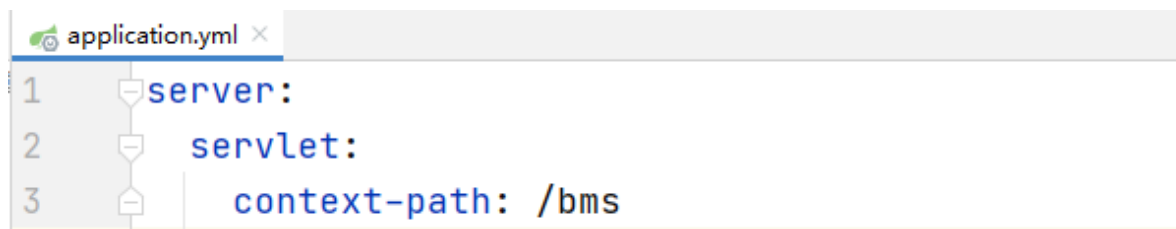
1.回顾SpringBoot开发步骤

1) 创建SpringBoot应用，其中选择我们需要的模块。例如Web、或者连接数据库的MyBatis等



目前先不考虑底层中关于JDBC操作或者MyBatis框架的支持，因此不使用它们的依赖，暂时采用虚拟数据模拟效果。

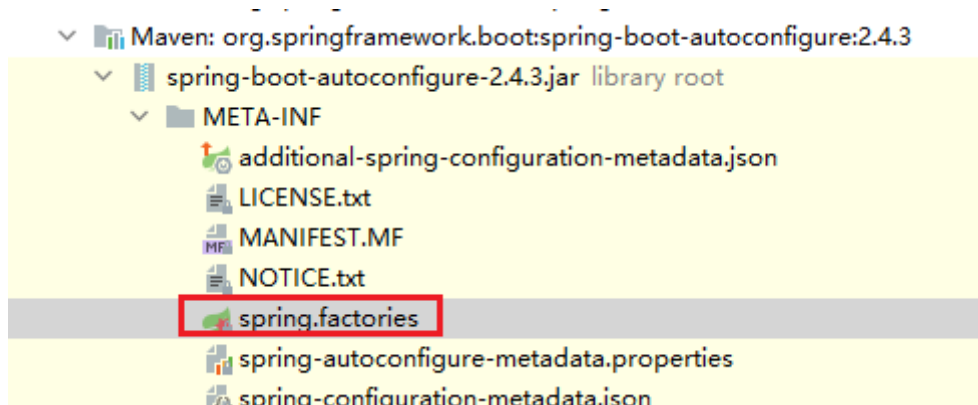
2) SpringBoot默认已经将这些场景配置好了（在外部库的spring-boot-autoconfigure中），只需要在配置文件中指定少量配置就可以运行



之后就可以来编写业务代码了。SpringBoot的自动配置同样作用于SpringMVC。

2.WebMVC的自动配置

- 1) SpringBoot实现自动配置的相关类都在 `spring-boot-autoconfigure` 包中
- 2) 其中包含了自动配置类型表



其中包含了针对Web Mvc进行自动配置的类：

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

而这个自动配置类将在另一个自动配置完成配置后进行配置：

```
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class, TaskExec  
ValidationAutoConfiguration.class })  
public class WebMvcAutoConfiguration {
```

从名字上看的出， `DispatcherServletAutoConfiguration` 就是给SpringMVC的前端控制器进行自动配置的类，在这个类中通过 `@Bean` 注解完成在容器中添加各种组件的配置，例如 `DispatcherServlet`、`MultipartResolver`。

而这些组件一旦配置在容器中后，有需要各种各样的属性，属性的值从何处获取呢？在这些配置类上面通常会有 `EnableConfigurationProperties` 注解：

```
@Configuration(proxyBeanMethods = false)  
@Conditional(DefaultDispatcherServletCondition.class)  
@ConditionalOnClass(ServletRegistration.class)  
@EnableConfigurationProperties(WebMvcProperties.class)  
protected static class DispatcherServletConfiguration {  
  
properties for Spring MVC.  
Since: 2.0.0  
Author: Phillip Webb, Sébastien Deleuze, Stephane Nicoll, Eddú Meléndez, Brian Clozel  
  
@ConfigurationProperties(prefix = "spring.mvc")  
public class WebMvcProperties {
```

这里就是针对SpringMVC框架的各种属性信息，使用 `ConfigurationProperties` 注解结合 `prefix` 前缀，来支持对各种属性的配置，而SpringBoot的默认配置就在 `META-INF` 下的 `spring-configuration-metadata.json` 文件中：

```
{
  {
    "name": "spring.mvc.view.prefix",
    "type": "java.lang.String",
    "description": "Spring MVC view prefix.",
    "sourceType": "org.springframework.boot.autoconfigure.web.servlet.
  },
  {
    "name": "spring.mvc.view.suffix",
    "type": "java.lang.String",
    "description": "Spring MVC view suffix.",
    "sourceType": "org.springframework.boot.autoconfigure.web.servlet.
  },
}
```

可以直接使用默认配置的属性值，也可以进行自定义修改。

另外，在 `WebMvcAutoConfiguration` 自动配置类中，还有一个内部类：

`WebMvcAutoConfigurationAdapter`，其上导入了一个类：

```
@Configuration(proxyBeanMethods = false)
@EnableWebMvcConfiguration
@EnableConfigurationProperties({ WebMvcProperties.class,
    org.springframework.boot.autoconfigure.web.ResourceProperties.class, WebProper
@Order(0)
public static class WebMvcAutoConfigurationAdapter implements WebMvcConfigurer {
```

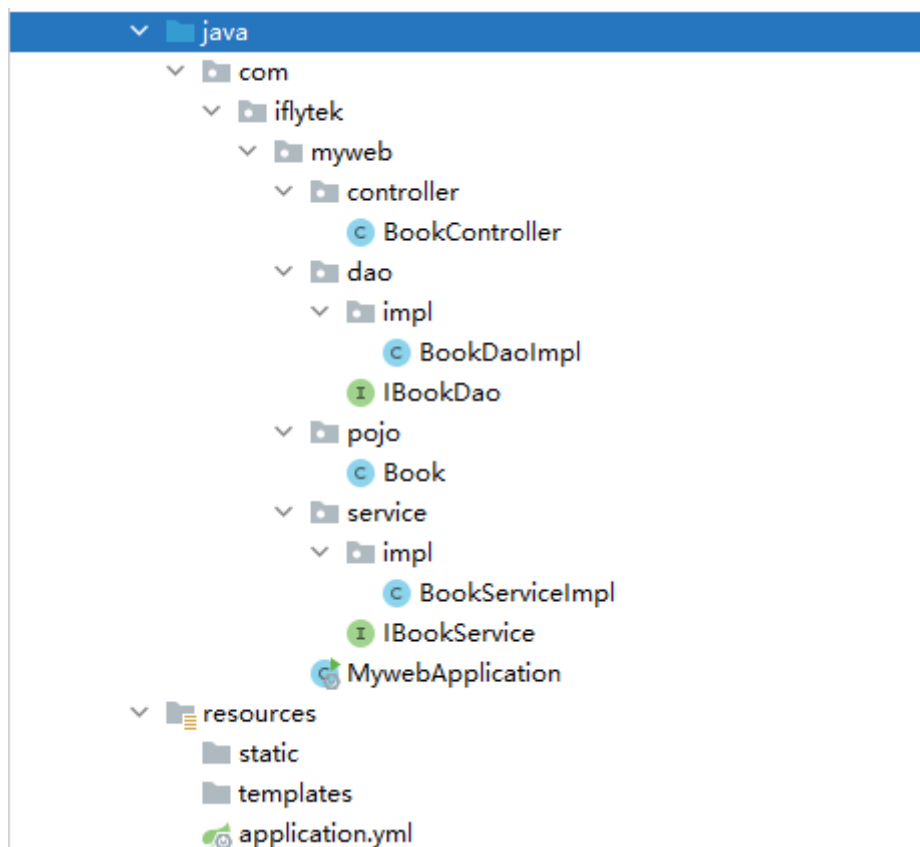
该类继承自 `DelegatingWebMvcConfiguration`：

```
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(WebProperties.class)
public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration im
```

`DelegatingWebMvcConfiguration` 类是对SpringMVC进行配置的一个代理类，可以结合默认配置和用户自定义的配置决定SpringMVC运行时最终使用的配置。

它的父类 `WebMvcConfigurationSupport` 为 Spring MVC 提供缺省配置，如何添加用户自定义配置呢？根据官网的描述，SpringBoot支持为SpringMVC提供自定义组件的功能，可以扩展添加自己定义的拦截器、视图解析器、格式化工具等等（<https://docs.spring.io/spring-boot/docs/2.4.3/reference/html/spring-boot-features.html#boot-features-spring-mvc-auto-configuration>）。

3.完成Web开发业务逻辑



pojo.Book

```
@Data
public class Book {
    private int id;
    private String name; // 名称
    private float price; // 价格
    private String author; // 作者
    private String category; // 分类
    private String publisher; // 出版社
    private Date publishdate; // 出版日期
    private int count; // 数量
    private String cover; // 封面
}
```

service.IBookService

```
public interface IBookService {
    List<Book> queryAll();
    Book queryById(int id);
}
```

service.impl.BookServiceImpl

```
@Service
public class BookServiceImpl implements IBookService {

    @Autowired
    private IBookDao dao;

    @Override
    public List<Book> queryAll() {
        return dao.queryAll();
    }
}
```

```

@Override
public Book queryById(int id) {
    return dao.queryById(id);
}
}

```

dao.IBookDao

```

public interface IBookDao {
    List<Book> queryAll();
    Book queryById(int id);
}

```

dao.impl.BookDaoImpl (不考虑数据库, 采用预设的模拟数据)

```

@Repository
public class BookDaoImpl implements IBookDao {

    private List<Book> dataSource = new ArrayList<>(); // 模拟数据源

    {
        String[] testBookNames = {"Java从入门到精通", "改变世界的科学家: 爱因斯坦、霍金、爱迪生、牛顿等", "藏在地图里的国家地理·世界", "Java核心技术 卷I 基础知识"};
        Book book = null;
        for(int i = 0; i < testBookNames.length * 10; i++){
            book = new Book();
            book.setId(i);
            book.setName(testBookNames[i % testBookNames.length]);
            book.setAuthor("test");
            book.setCategory("test");
            book.setCount(100);
            book.setCover("test");
            book.setPrice(100);
            book.setPublisher("test");
            book.setPublishdate(new Date());
            dataSource.add(book);
        }
    }

    @Override
    public List<Book> queryAll() {
        return dataSource;
    }

    @Override
    public Book queryById(int id) {
        for (int i = 0; i < dataSource.size(); i++){
            if (dataSource.get(i).getId() == id){
                return dataSource.get(i);
            }
        }
        return null;
    }
}

```

controller.BookController

```
// 此处控制器方法上均使用@ResponseBody注解，将查询结果直接以字符串形式返回，省略了创建页面展示数据的过程
// 因为在SpringBoot中基本不会再使用JSP展示数据，后续将会学习模板引擎技术
// 所以在控制器中暂时只关注于访问控制
@Controller
@RequestMapping("/book")
public class BookController {

    @Autowired
    private IBookService service;

    @GetMapping("/all")
    @ResponseBody
    public List<Book> getAll(){
        return service.queryAll();
    }

    @GetMapping("/info/{id}")
    @ResponseBody
    public Book getBook(@PathVariable int id) {
        return service.queryById(id);
    }
}
```

静态资源配置

在Web项目开发中，经常会用到各种静态资源，例如html、css、js文件等，后面将会引入学习的模板引擎中也会涉及静态资源，而SpringBoot对于静态资源的存放是有要求的。

按照SpringBoot对SpringMVC的自动配置原理分析，SpringBoot对SpringMVC的相关配置都在 `WebMvcAutoConfiguration` 类中，而该类中有一个添加资源处理的方法，其中添加了两个资源处理器：

```
@Override
protected void addResourceHandlers(ResourceHandlerRegistry registry) {
    super.addResourceHandlers(registry);
    if (!this.resourceProperties.isAddMappings()) {
        logger.debug("Default resource handling disabled");
        return;
    }
    ServletContext servletContext = getServletContext();
    addResourceHandler(registry, pattern: "/webjars/**", locations: "classpath:/META-INF/resources/webjars/");
    addResourceHandler(registry, this.mvcProperties.getStaticPathPattern(), (registration) -> {
        registration.addResourceLocations(this.resourceProperties.getStaticLocations());
        if (servletContext != null) {
            registration.addResourceLocations(new ServletContextResource(servletContext, SERVLET_LOCATION));
        }
    });
}
```

这两个方法就添加了对静态资源的映射规则：

- 1) 对所有 `/webjars/**` 下的资源的请求都去 `classpath:/META-INF/resources/webjars/` 路径下查找
- 2) 对所有 `StaticPathPattern` 下的资源的请求都去 `StaticLocations` 路径下去查找。

关于静态资源位置

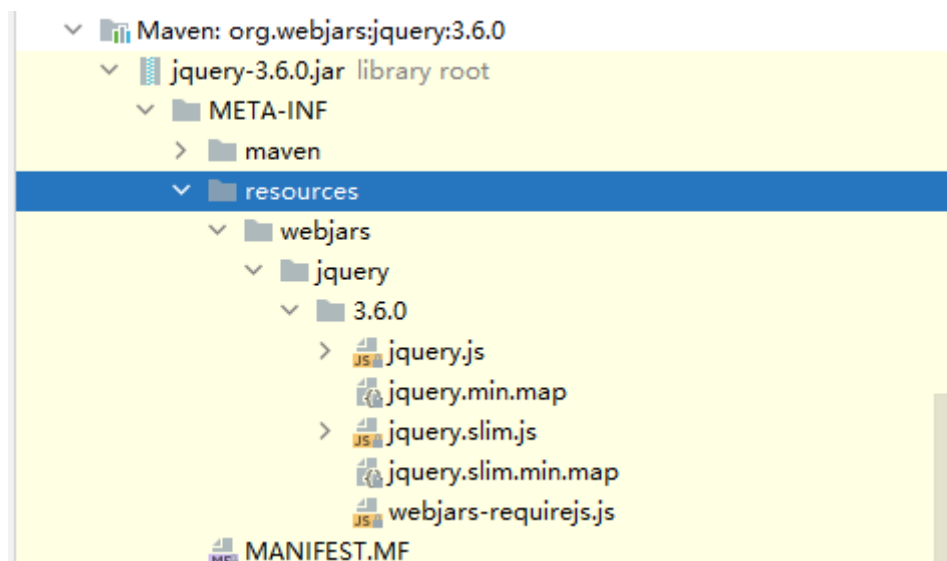
1.webjars

webjars 指的是以 **jar** 包的方式引入静态资源，在 www.webjars.org 网站上，提供了很多的jar包，将常用的各种静态资源以jar包方式重新封装，之后提供各种依赖坐标供项目引入：

Popular WebJars			<input checked="" type="checkbox"/> NPM
			Add a WebJar <input type="text" value="Search"/>
Name	Versions	Build Tool:	SBT / Play 2 / Maven / Ivy / Grape / Gradle / Buildr
Swagger UI	3.45.0 <input type="button" value="+"/>	<dependency> <groupId>org.webjars</groupId> <artifactId>swagger-ui</artifactId> <version>3.45.0</version> </dependency>	
npm	5.0.0-2 <input type="button" value="+"/>	<dependency> <groupId>org.webjars</groupId> <artifactId>npm</artifactId> <version>5.0.0-2</version> </dependency>	
jquery	3.6.0 <input type="button" value="+"/>	<dependency> <groupId>org.webjars</groupId> <artifactId>jquery</artifactId> <version>3.6.0</version> </dependency>	
Bootstrap	5.0.0-beta2 <input type="button" value="+"/>	<dependency> <groupId>org.webjars</groupId> <artifactId>bootstrap</artifactId> <version>5.0.0-beta2</version> </dependency>	

复制其中的依赖坐标到项目POM文件中就可以引入该静态资源依赖：

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>jquery</artifactId>
  <version>3.6.0</version>
</dependency>
```



这个时候如果访问 <http://localhost:8080/webjars/jquery/1.11.3/jquery.js> 就可以访问到这个资源，所以如果需要 **jQuery** 功能，引入 **jquery-webjar**，在访问的时候只要写webjars下面资源的名称即可。

2.StaticPathPattern

```
addResourceHandler(registry, this.mvcProperties.getStaticPathPattern(), (registration) -> {
```

对该方法调用进行跟踪，发现其调用的是 **WebMvcProperties** 中的方法：

```
public String getStaticPathPattern() {
    return this.staticPathPattern;
}
```

而这个属性值则是：

Path pattern used for static resources.

```
private String staticPathPattern = "/*";
```

当对 `/*` 进行访问请求，也就是访问当前项目的任何资源时，例如访问 `localhost:8080/abc`，如果此时 `abc` 没有映射路径去处理（例如没有一个控制方法是映射了这个路径），那么就默认去静态资源里找 `abc`。

而这个时候查找的资源路径是通过调用方法完成添加的：

```
addResourceHandler(registry, this.mvcProperties.getStaticPathPattern(), (registration) -> {  
    registration.addResourceLocations(this.resourceProperties.getStaticLocations());  
    if (servletContext != null) {
```

该方法指向了 `WebProperties` 类中的属性：

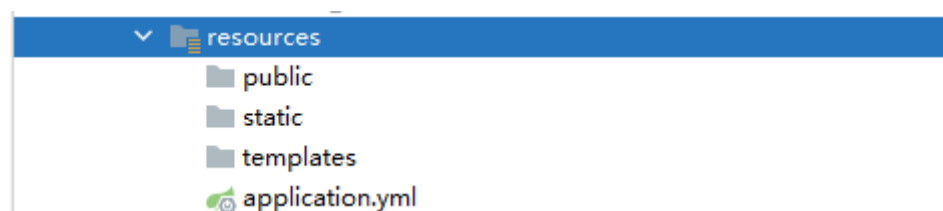
```
public String[] getStaticLocations() {  
    return this.staticLocations;  
}
```

这个属性是 `WebProperties` 类的静态内部类中的字符串数组：

```
public static class Resources {  
  
    private static final String[] CLASSPATH_RESOURCE_LOCATIONS = { "classpath:/META-INF/resources/",  
                                                                    "classpath:/resources/",  
                                                                    "classpath:/static/",  
                                                                    "classpath:/public/" };  
  
    Locations of static resources. Defaults to classpath:/META-INF/resources/, /resources/,  
    /static/, /public/.
```

```
private String[] staticLocations = CLASSPATH_RESOURCE_LOCATIONS;
```

这几个路径就指定了项目中的默认静态资源路径，在项目中 `java` 文件夹和 `resources` 文件夹就代表了 `classpath`，按照这几个路径表示，还可以在 `resources` 文件夹下继续建立 `META-INF/resources` 文件夹和 `resources` 文件夹，只不过这样的路径会稍显复杂，且不利于编码处理，所以一般不使用，通常使用的静态资源路径就是 `resources/static` 和 `resources/public`



静态资源目录配置

通过对静态资源位置的分析，目前可以知道，对于无法映射的路径请求，都将转向静态资源目录下查找资源，当然也可以通过修改默认配置来指定对哪些资源的访问需要进行查找静态资源，以及到哪里查找静态资源。

1. 配置 `staticPathPattern`

该属性是 `WebMvcProperties` 类的属性，该类有配置属性注解：

```
@ConfigurationProperties(prefix = "spring.mvc")  
public class WebMvcProperties {
```

因此可以在配置文件中结合前缀实现修改，例如默认为 `/*`，可以配置为只包含 `/static/*` 的资源才去查找静态资源：

```
spring:
  mvc:
    static-path-pattern: /static/**
```

2.配置staticLocation

`staticLocations` 属性是 `WebProperties` 类的内部类 `Resources` 的属性，注意配置的路径：

```
spring:
  web:
    resources:
      static-locations: classpath:/hello/, classpath:/iflytek/
```

注意：早期的SpringBoot版本中，该路径配置是 `spring.resources.static-locations`，新版本中已经弃用了。

欢迎页配置

在 `WebMvcAutoConfiguration` 中还包含了欢迎页的处理映射：

```
@Bean
public WelcomePageHandlerMapping welcomePageHandlerMapping(ApplicationContext applicationContext,
    FormattingConversionService mvcConversionService, ResourceUrlProvider mvcResourceUrlProvider) {
    WelcomePageHandlerMapping welcomePageHandlerMapping = new WelcomePageHandlerMapping(
        new TemplateAvailabilityProviders(applicationContext), applicationContext, getWelcomePage(),
        this.mvcProperties.getStaticPathPattern());
    welcomePageHandlerMapping.setInterceptors(getInterceptors(mvcConversionService, mvcResourceUrlProvider));
    welcomePageHandlerMapping.setCorsConfigurations(getCorsConfigurations());
    return welcomePageHandlerMapping;
}
```

在 `getWelcomePage` 方法中查询静态资源路径是否存在欢迎页：

```
private Resource getWelcomePage() {
    for (String location : this.resourceProperties.getStaticLocations()) {
        Resource indexHtml = getIndexHtml(location);
        if (indexHtml != null) {
            return indexHtml;
        }
    }

    private Resource getIndexHtml(String location) {
        return getIndexHtml(this.resourceLoader.getResource(location));
    }

    private Resource getIndexHtml(Resource location) {
        try {
            Resource resource = location.createRelative("index.html");
            if (resource.exists() && (resource.getURL() != null)) {
                return resource;
            }
        }
    }
}
```

可以看出静态资源目录下所有的 `index.html` 都可以被映射，并找第一个。如果访问了 `localhost:8080/` --> 这就会去找 `index.html` 页面。

所以只需要在静态资源路径下给定一个 `index.html` 即可自动完成欢迎页的配置：



欢迎您！这是SpringBoot的欢迎页。

注意，前面的静态资源配置会影响此处，注意排查。

图标配置

请注意：在SpringBoot 2.4.0版本开始，网页标题图标的配置属性已经被弃用！

在之前的版本中（如果项目创建时使用的是早期版本），对于网页图标的配置是通过 `FaviconConfiguration` 类完成的，该类实现了Spring框架中的 `ResourceLoaderAware` 接口，其类中指定配置了处理 `favicon.icon` 文件的映射。

在SpringBoot2.2.0版本开始已经移除了该类，可能是考虑到网页图标属于前端开发设计内容，后端并不关心。

如果需要配置，在现在的版本中可以做如下操作：

1. 在静态资源目录中（例如 `resources/public`）添加一个 `favicon.ico` 图标文件（可以在线制作 <https://tool.lu/favicon/>）
2. 前端页面上编写配置代码

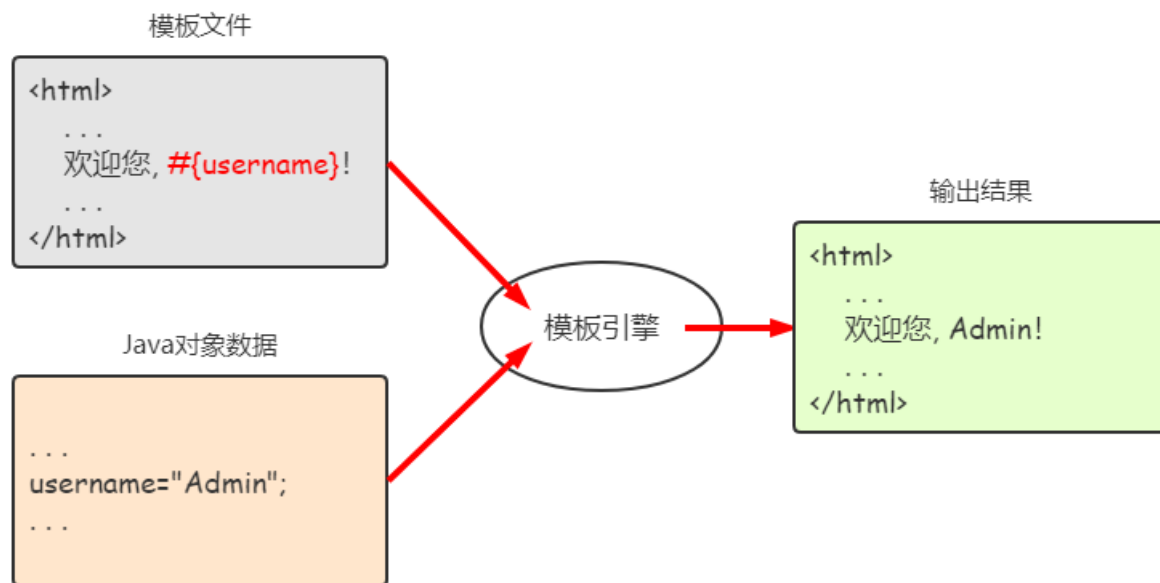
```
<link rel="shortcut icon" href="/bms/favicon.ico"/>
<link rel="bookmark" href="/bms/favicon.ico"/>
```

四、模板引擎Thymeleaf

0.模板引擎

除了REST风格的网络服务之外，SpringMVC还支持动态HTML内容。SpringMVC还支持多种模板技术，例如 `Thymeleaf`、`FreeMarker`、`JSP` 等，而且其他一些模板引擎也有Spring的集成版本。

可以看出，Java Web中学习过的 `JSP` 也是一种模板技术，不同的模板引擎的思想都是一样的：



模板引擎就是为了解决用户界面与业务数据（内容）分离而产生的，它可以生成特定格式的文档，用于网站的模板引擎就会生成一个标准的HTML文档。

模板引擎不属于特定技术领域，它是跨领域跨平台的概念。在ASP下有模板引擎，在PHP下也有模板引擎，在C#下也有，甚至JavaScript、WinForm开发都会用到模板引擎技术。不同的模板引擎思想基本都一致，只不过不同的模板引擎之间，语法会有些差异。

在SpringBoot中包含了下列这些模板引擎的自动配置支持，可以更加方便的配置开发：

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

另外官网也提到，如果可以的话避免使用JSP，和嵌入的Servlet容器一起运行时会产生一些问题。

1.Thymeleaf模板引擎引入

简介

Thymeleaf是个XML/XHTML/HTML5模板引擎，可以用于Web与非Web应用。

Thymeleaf的主要目标在于提供一种可被浏览器正确显示的、格式良好的模板创建方式，因此也可以用作静态建模。你可以使用它创建经过验证的XML与HTML模板。相对于编写逻辑或代码，开发者只需将标签属性添加到模板中即可。接下来，这些标签属性就会在DOM（文档对象模型）上执行预先制定好的逻辑。Thymeleaf的可扩展性也非常棒。你可以使用它定义自己的模板属性集合，这样就可以计算自定义表达式并使用自定义逻辑。这意味着Thymeleaf还可以作为模板引擎框架。

特性

1. Thymeleaf 在有网络和无网络的环境下皆可运行，它可以让美工在浏览器查看页面的静态效果，也可以让程序员在服务器查看带数据的动态页面效果。这是由于它支持html原型，在html标签里增加额外的属性来达到 **模板+数据** 的展示方式。浏览器解释html时会忽略未定义的标签属性，所以Thymeleaf的模板可以静态地运行；当有数据返回到页面时，Thymeleaf标签会动态地替换掉静态内容，使页面动态显示。
2. Thymeleaf 开箱即用的特性。它提供标准和Spring标准两种方言，可以直接套用模板实现JSTL、OGNL表达式效果，避免每天套模板、改jstl、改标签的困扰。同时开发人员也可以扩展和创建自

定义的方言。

3. Thymeleaf 提供Spring标准方言和一个与 SpringMVC 完美集成的可选模块，可以快速的实现表单绑定、属性编辑器、国际化等功能。

使用

要在SpringBoot项目中引入模板引擎，就需要导入相应的起步依赖，在官方手册列举的起步依赖中也可以找到Thymeleaf的起步依赖信息：

Spring Boot Reference Documentation

Phillip Webb · Dave Syer · Josh Long · Stéphane Nicoll · Rob Winch · Andy Wilkinson · Marcel Overdijk · Christian Dupuis · Sébastien Deleuze · Michael Simons · Vedran Pavić · Jay Bryant · Madhura Bhavne · Eddú Meléndez · Scott Frederick

2.4.3

The reference documentation consists of the following sections:

Legal	Legal information.
Documentation Overview	About the Documentation, Getting Help, First Steps, and more.
Getting Started	Introducing Spring Boot, System Requirements, Servlet Containers, Installing Spring Boot, Database Drivers, and more.
Using Spring Boot	Build Systems, Structuring Your Code, Configuration, Spring Beans and Dependency Injection, Thymeleaf, and more.
Spring Boot Features	Profiles, Logging, Security, Caching, Spring Integration, Testing, and more.

1. Build Systems

1.1. Dependency Management

1.2. Maven

1.3. Gradle

1.4. Ant

1.5. Starters

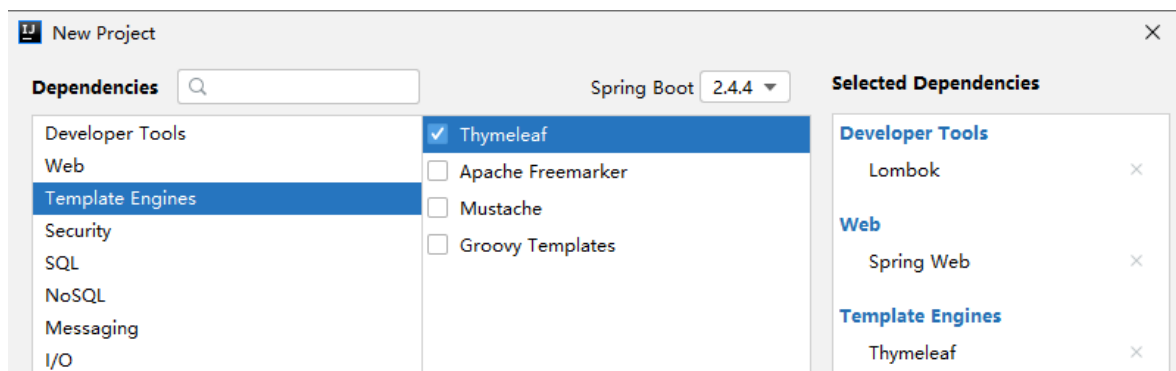
`spring-boot-starter-thymeleaf`

Starter for building MVC web applications using Thymeleaf views

可以在现有的SpringBoot项目的POM文件中加入Thymeleaf的起步依赖：

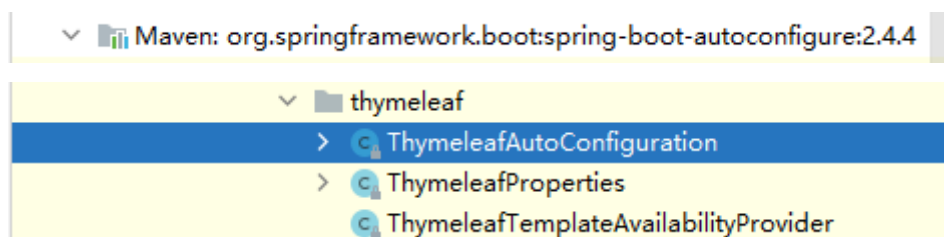
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

或者在新建项目时，在IDEA向导中就选择好对应的组件：



2. 自动配置

SpringBoot支持Thymeleaf的自动配置，在 `spring-boot-autoconfigure` 包中就包含了Thymeleaf的自动配置类：



```
@Configuration(
    proxyBeanMethods = false
)
@EnableConfigurationProperties({ThymeleafProperties.class})
@ConditionalOnClass({TemplateMode.class, SpringTemplateEngine.class})
@AutoConfigureAfter({WebMvcAutoConfiguration.class, WebFluxAutoConfiguration.class})
public class ThymeleafAutoConfiguration {
```

该类中包含了Thymeleaf的组件的配置，一些默认的配置属性则在 `ThymeleafProperties` 类中：

```
@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;
    private boolean cache;
```

对于这些属性规则，使用默认配置就可以把Thymeleaf用起来了。

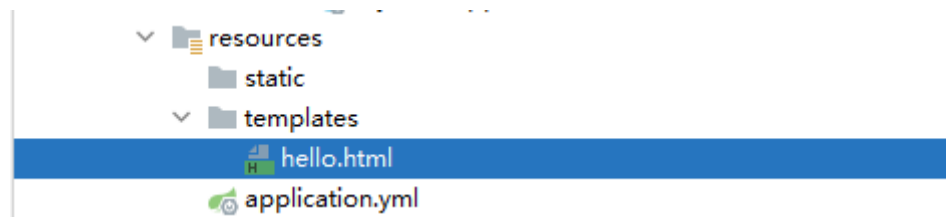
开发时建议将缓存 `cache` 设置为false，否则因为缓存的原因可能会导致热部署时的数据不会实时更新，开发完成上线后可去除：

```
spring:
  thymeleaf:
    cache: false # 禁用缓存
```


可以看到在这些属性配置里，有 `prefix`、`suffix`，这些是在SpringMVC里配置视图解析器时采用的，也就是说对于Thymeleaf模板引擎来说，只需要将 `*.html` 页面放到 `classpath:/templates` 目录下，即可以被模板引擎解析得到，之后可以自动完成渲染。

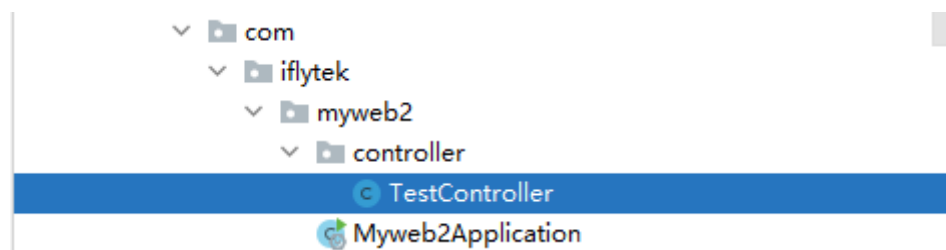
测试使用

1) 在指定资源目录下创建html页面



```
<body>
  <h1>访问成功了! </h1>
</body>
```

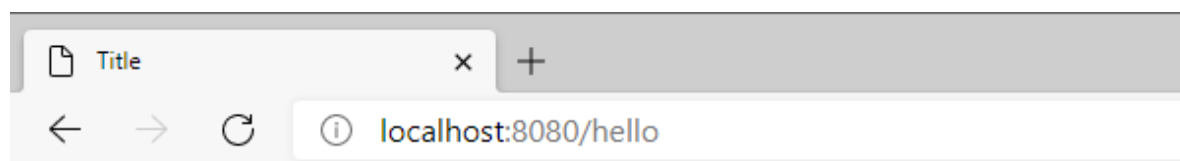
2) 添加控制器方法



```
@Controller
public class TestController {

    @GetMapping("/hello")
    public String getHello() {
        return "hello";
    }
}
```

3) 访问测试



访问成功了!

3.Thymeleaf使用

0) 数据准备

现在通过一个简单的案例来学习Thymeleaf的语法，首先明确要完成的任务：

访问一个控制器Controller，其中调用Service执行查询操作获得一些数据（可以使用假数据模拟实现），之后在页面中显示出来。

- 创建实例类Student

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Student {
    private int id;
    private String name;
    private int age;
    private String major;
}
```

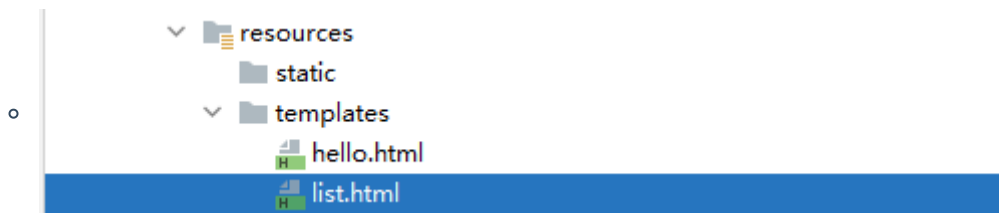
- 添加控制器方法，其中模拟查询数据过程

```
@Controller
@RequestMapping("/student")
public class StudentController {

    @GetMapping("/list")
    public String getList(Model model){
        List<Student> students = new ArrayList<>();
        Student s = new Student(1, "张三", 18, "Java");
        model.addAttribute("stu", s); // 添加一个学生对象属性
        students.add(s);
        s = new Student(2, "李四", 19, "Java");
        students.add(s);
        s = new Student(3, "王五", 21, "Java");
        students.add(s);
        s = new Student(4, "刘柳", 20, ".NET");
        students.add(s);
        model.addAttribute("stuList", students); // 添加一个学生集合属性

        return "list";
    }
}
```

- 在 templates 目录下建立 list.html



这里先添加一个对象到 model 中，再添加了一个集合到 model 中，后续进行对比。

这里使用SpringMVC的model对象添加属性数据，默认添加到 request域对象 里，之前的Web项目中可以到转发的jsp页面中使用EL表达式 `${}` 去读取出来，那么thymeleaf里如何读取与对象中的数据呢？

1) 导入名称空间

首先在html页面上要使用thymeleaf需要在html标签上导入它的名称空间：

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

导入名称空间主要为了代码提示。

2) 使用Thymeleaf语法编写模板文件

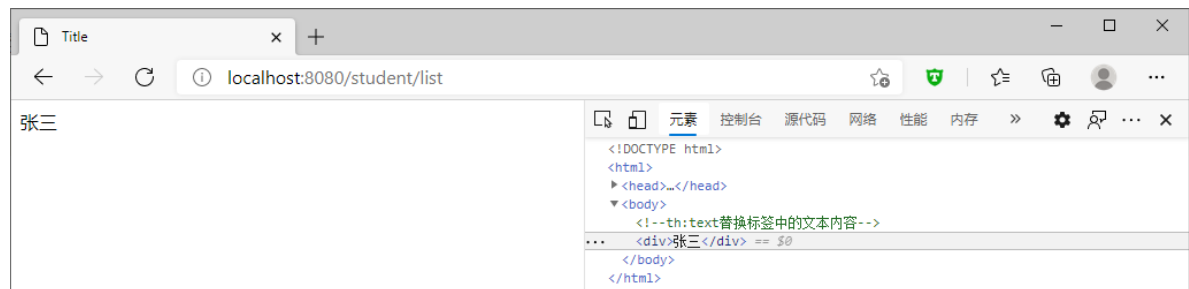
Thymeleaf的特征是拓展属性，通过导入名称空间后，使用前缀 **th** (th:xx)，通过拓展属性和服务端进行数据交互。

例如，显示域对象中 **stu** 对象的name属性：

```
<!--th:text替换标签中的文本内容-->
<div th:text="${stu.name}">学生姓名</div>
```

th:text 的作用是改变当前元素里面的文本内容

运行效果：



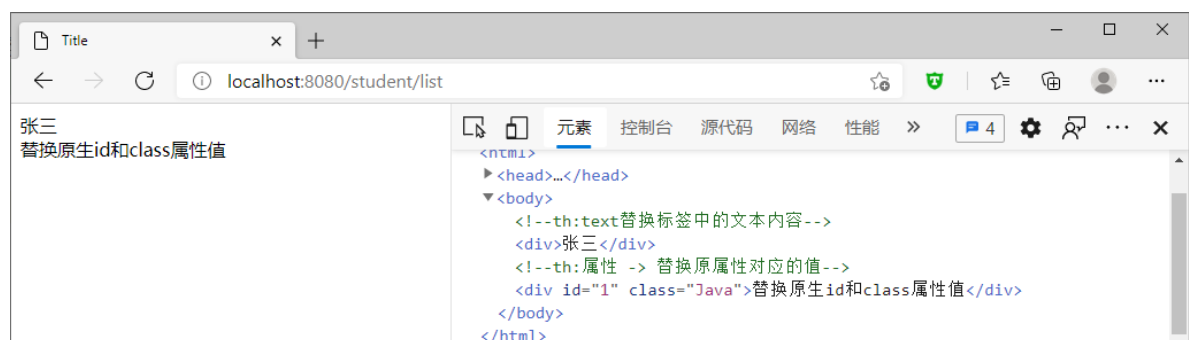
4.Thymeleaf语法规则

th:html属性

th: 后接html属性，其作用即为替换原生html属性所对应的值，例如 **th:id="xxx"** 的作用就是用对应的值替换html标签中的id属性值：

```
<!--th:属性 -> 替换原属性对应的值-->
<div id="div1" class="rect" th:id="${stu.id}" th:class="${stu.major}">替换原生id和class属性值</div>
```

运行效果：



在官方线上文档（<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#attribute-precedence>）中，按照 **属性优先级** 顺序对属性进行了分类列举：

Order	Feature	Attributes
1	Fragment inclusion	th:insert th:replace
2	Fragment iteration	th:each
3	Conditional evaluation	th:if th:unless th:switch th:case
4	Local variable definition	th:object th:with
5	General attribute modification	th:attr th:attrprepend th:attrappend
6	Specific attribute modification	th:value th:href th:src ...
7	Text (tag body modification)	th:text th:utext
8	Fragment specification	th:fragment
9	Fragment removal	th:remove

其中：

1. 片段包含，类似于 `jsp:include`
2. 遍历迭代，类似于 `c:forEach`
3. 条件判断，类似于 `c:if`
4. 定义变量，类似于 `c:set`
5. 属性修改，支持在属性前或者属性后追加
6. 修改指定的属性值
7. 修改标签体内容，`text` 会将特殊字符进行转义，`utext` 则不会转义
8. 片段声明，声明后的片段可以结合1中的 `insert` 和 `replace` 进行包含，例如：

在A.html中声明一个片段：

```
<footer th:fragment="copyright">&copy;ifytek</footer>
```

在B.html中包含该片段：

```
<div th:insert="A::copyright"></div>
或
<div th:replace="A::copyright"></div>
```

表达式

前面的代码中 `th:text="${stu.name}"` 使用了表达式 `${xxx}`，类似于JSP中的EL表达式，那么在Thymeleaf中可以编写哪些表达式？

官网在线文档的第4章《Standard Expression Syntax》(<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#standard-expression-syntax>)中进行了总结，共有如下类型：

- Simple expressions：简单表达式
- Literals：字面量
- Text operations：文本操作

- Arithmetic operations: 数学运算
- Boolean operations: 布尔运算
- Comparisons and equality: 比较运算
- Conditional operators: 条件判断
- Special tokens: 特殊标记

1) Simple expressions 简单表达式

- Variable Expressions: `${...}` : 变量表达式, 获取变量值 (OGNL)
 - 可以获取对象的属性, 或者调用方法
 - 可以使用内置的基本对象 (<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#expression-basic-objects>) :

- `#ctx` : 上下文对象, 可用于获取其他内置对象
- `#vars` : 上下文变量
- `#locale` : 上下文区域对象
- `#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.

例如:

```
<!--表达式使用内置对象-->
<!--getScheme()获取协议, getServerName()获取服务器名, getServerPort()服务器端口,
getContextPath()获取APP根路径-->
<a th:href="${#httpServletRequest.getScheme() + '://' +
#httpServletRequest.getServerName() + ':' + #request.getServerPort() +
#request.getContextPath() + '/'}"
id="contextPath">Web应用根目录</a>
```

- 可以使用内置的工具对象 (<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#expression-utility-objects>), 具体对象的使用方法参考官网文档的附录B部分 (<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#appendix-b-expression-utility-objects>) :

- `#execInfo` : 获取页面模板的处理信息
- `#messages` : 在变量表达式中获取外部消息的方法, 与使用 `#{...}` 语法获取的方法相同
- `#uris` : 转义URL/URI部分的方法
- `#conversions` : 用于执行已配置的转换服务的方法
- `#dates` : 时间操作和时间格式化等

```
model.addAttribute("now", new Date());
```

```
<div th:text="${#dates.createNow()}">使用工具对象创建当前时间</div>
<div th:text="${#dates.format(now, 'yyyy年MM月dd日 HH:mm:ss')}">获取域对象中的时间进行格式化</div>
```

- `#calendars` : 和 `#dates` 类似, 使用日历对象进行更复杂的时间处理
- `#numbers` : 格式化数字对象的方法
- `#strings` : 字符串工具对象

```
<div th:text="${#strings.startsWith(stu.major, 'J')}">字符串工具对象</div>
```

- **#objects** : 一般对象类, 通常用来判断非空
 - **#bools** : 常用的布尔方法
 - **#arrays** : 数据工具类
 - **#lists** : List集合工具类, 获取大小、判空、判断是否包含、排序
 - **#sets** : Set集合工具类, 获取大小、判断、判断是否包含
 - **#maps** : Map集合工具类, 获取大小、判空、判断是否包含key、value
 - **#aggregates** : 在数组或集合上创建聚合的方法, 例如对数组或集合计算和值或平均值
 - **#ids** : 处理可能重复的id属性的方法,
- Selection Variable Expressions: ***{...}**, 选择变量表达式, 和 **\${...}** 在功能上是一样的, 但是有一个补充功能, 配合 **th:object** 使用, 可以方便的直接获取对象的属性
 - 官方文档 (<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#expressions-on-selections-asterisk-syntax>)

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

- Which is exactly equivalent to:

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

- Message Expressions: **#{...}**, 获取国际化内容
- Link URL Expressions: **@{...}**, 定义URL链接, 其中可以方便的拼接参数字符串
- Fragment Expressions: **~{...}**, 片段引用表达式

2) 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**, ...

3) Text operations文本操作

- String concatenation: **+**
- Literal substitutions: **|The name is \${name}|**

4) Arithmetic operations数学运算

- Binary operators: **+**, **-**, *****, **/**, **%**
- Minus sign (unary operator): **-**

5) Boolean operations布尔运算

- Binary operators: **and**, **or**
- Boolean negation (unary operator): **!**, **not**

6) Comparisons and equality 比较运算

- Comparators: `>`, `<`, `>=`, `<=` (`gt`, `lt`, `ge`, `le`)
- Equality operators: `==`, `!=` (`eq`, `ne`)

7) Conditional operators 条件判断

- If-then: `(if) ? (then)`
- If-then-else: `(if) ? (then) : (else)`
- Default: `(value) ?: (defaultvalue)`

迭代遍历

在JSP中引入JSTL之后，使用核心标签库里的 `c:forEach` 标签，可以对一个集合或数组进行遍历输出，在ThymeLeaf中也有类似的语法。

在官方文档第6章《Iteration》（<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#iteration>）中有介绍使用方法：

```
<tr th:each="prod : ${prods}">
  <td th:text="${prod.name}">Onions</td>
  <td th:text="${prod.price}">2.41</td>
  <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
</tr>
```

使用的规则是： `th:each = "变量名 : ${集合}"`，需要注意： `th:each` 每次遍历都会将其所在的标签生成一次

例如：

```
<!-- 将会生成多个div -->
<div th:each="s : ${stuList}" th:text="${s.name}"></div>
```

```
<!-- 将会生成多个div -->
<div>张三</div>
<div>李四</div>
<div>王五</div>
<div>刘柳</div>
```

```
<!-- 将会生成多个span -->
<div>
  <span th:each="s : ${stuList}" th:text="${s.name} + ', ' + ${s.major}"></span>
</div>
```

```
▼ <div>
  <span>张三, Java</span>
  <span>李四, Java</span>
  <span>王五, Java</span>
  <span>刘柳, .NET</span>
</div>
```

使用ThymeLeaf完成学生信息的展示：

```

<table border="1" cellpadding="10" cellspacing="0">
  <tr><th>序号</th><th>姓名</th><th>年龄</th><th>专业</th></tr>
  <tr th:each="s,stat : ${stuList}" th:object="${s}">
    <td th:text="${stat.count}"></td><!-- 获取迭代状态中的序号，index从0开始，count从1开
始-->
    <td th:text="*{name}"></td><!-- 使用*{}选择变量表达式，搭配th:object一起使用直接获取属
性-->
    <td th:text="${s.age}"></td><!-- 普通变量表达式-->
    <td>[[*{major}]]</td> <!-- 行内形式-->
  </tr>
</table>

```

补充说明：

1. 想要获取列表的序号，可以通过在 `th:each` 中添加一个状态变量来获取，状态也是一个对象，包含多个属性值（<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#keeping-iteration-status>）

Status variables are defined within a `th:each` attribute and contain the following data:

- The current *iteration index*, starting with 0. This is the `index` property.
- The current *iteration index*, starting with 1. This is the `count` property.
- The total amount of elements in the iterated variable. This is the `size` property.
- The *iter variable* for each iteration. This is the `current` property.
- Whether the current iteration is even or odd. These are the `even/odd` boolean properties.
- Whether the current iteration is the first one. This is the `first` boolean property.
- Whether the current iteration is the last one. This is the `last` boolean property.

2. 使用表达式的时候都要写在标签的属性中并使用双引号括起来，直接在标签体中使用表达式是不行的，例如 `<div>${user.name}</div>` 是不会被正确解析的。

但是官方也考虑到了这种情况，使用了一种特殊的写法来支持在标签体中写表达式。官方文档的第12章《inlining》

（<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#inlining>）中介绍了行内写法：

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