SpringBoot2 基础篇

命名规范

- 类的名称UpperCamel
- 包的名称lowCamel

环境要求

- Java 8 - 14 以上的环境（Spring5 基于Java 8 实现）
- Spring Framework 5.2.9.RELEASE+
- Maven 3.3+

预备知识

- Spring 生态圈

  参考网址： spring.io

- SpringBoot
  - 简化了"配置地狱"问题，专注业务代码即可
  - 底层是Spring框架，整合了Spring整个系列技术栈
  - 参照 SpringBoot 官方文档进行学习（project -> learn -> 最新版本 [GA] ）
  - 查看框架的变化： https://github.com/spring-projects/spring-boot/wiki#release-notes （尤其要注意大版本的变化 [1.0->2.0] 要看整个框架的变化，中版本 [1.1->1.2] 看开发文档即可，小版本 [1.1.1->1.1.2] 无需关心）

配置

- Maven：

  在修改 conf 中 setting.xml 文件，加入如下的 jdk 版本和镜像配置

```
<mirrors>
      <mirror>
        <id>nexus-aliyun</id>
        <mirrorOf>central</mirrorOf>
        <name>Nexus aliyun</name>
        <url>http://maven.aliyun.com/nexus/content/groups/public</url>
      </mirror>
  </mirrors>

  <profiles>
         <profile>
              <id>jdk-1.8</id>
              <activation>
                <activeByDefault>true</activeByDefault>
                <jdk>1.8</jdk>
              </activation>
              <properties>
                <maven.compiler.source>1.8</maven.compiler.source>
                <maven.compiler.target>1.8</maven.compiler.target>

<maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
              </properties>
         </profile>
  </profiles>
```

入门

- **相关注解（一般可以理解向容器中注册组件）**
  - @SpringBootApplication： 表示一个 SpringBoot 应用
    - @SpringBootConfiguration
      @EnableAutoConfiguration
      @ComponentScan（"包名"）

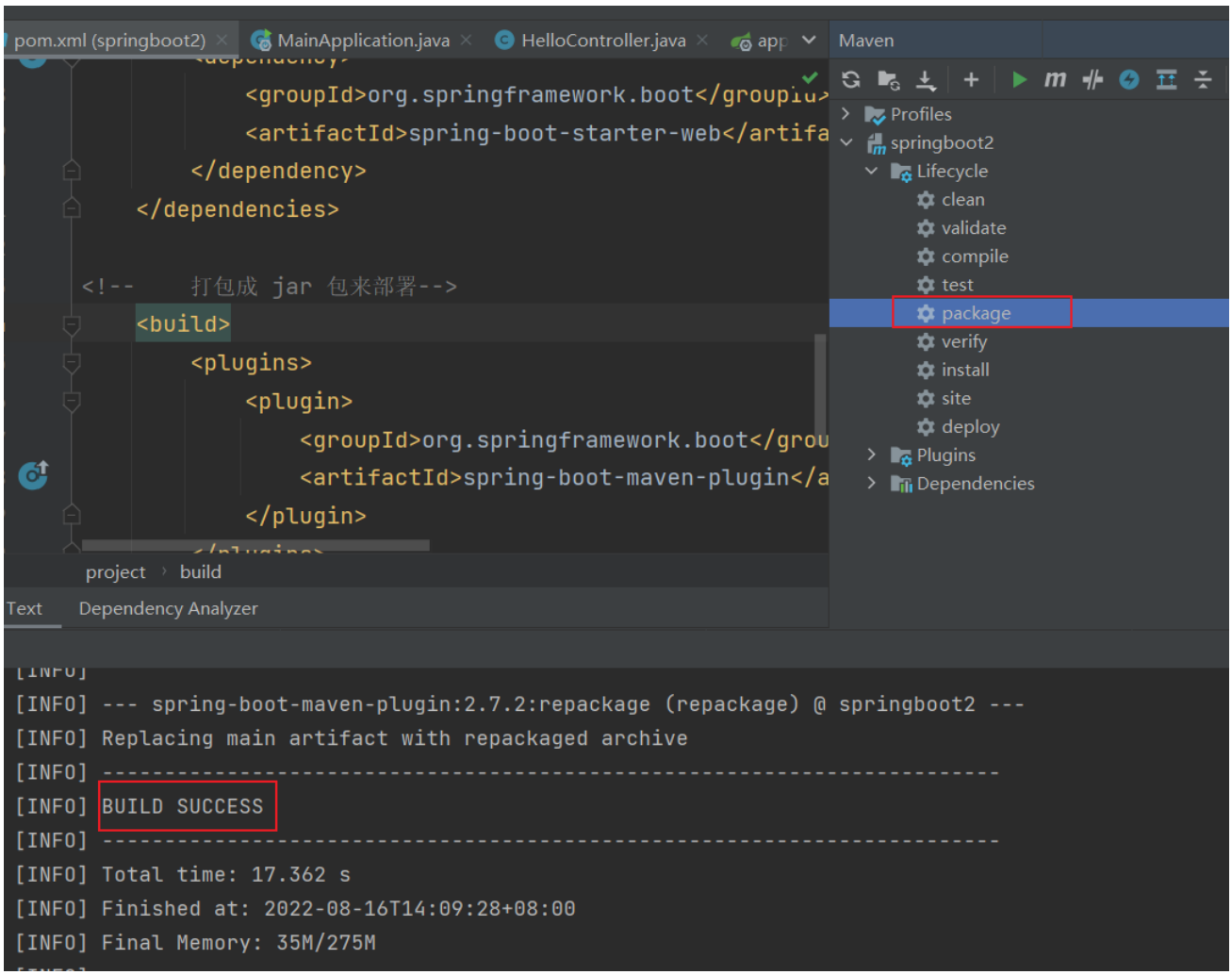
      一个封装了该三个
  - Controller： 合并 @Controller 和 @ResponseBody
    - @Controller： web 层
    - @ResponseBody： 将数据写回给浏览器
- **引入依赖**

```
<!--    使用 SpringBoot 功能需要引入 SpringBoot 父项目依赖-->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.2</version>
    </parent>
<!--    引入 web 场景来开发 web -->
<dependencies>
      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
  </dependencies>
```

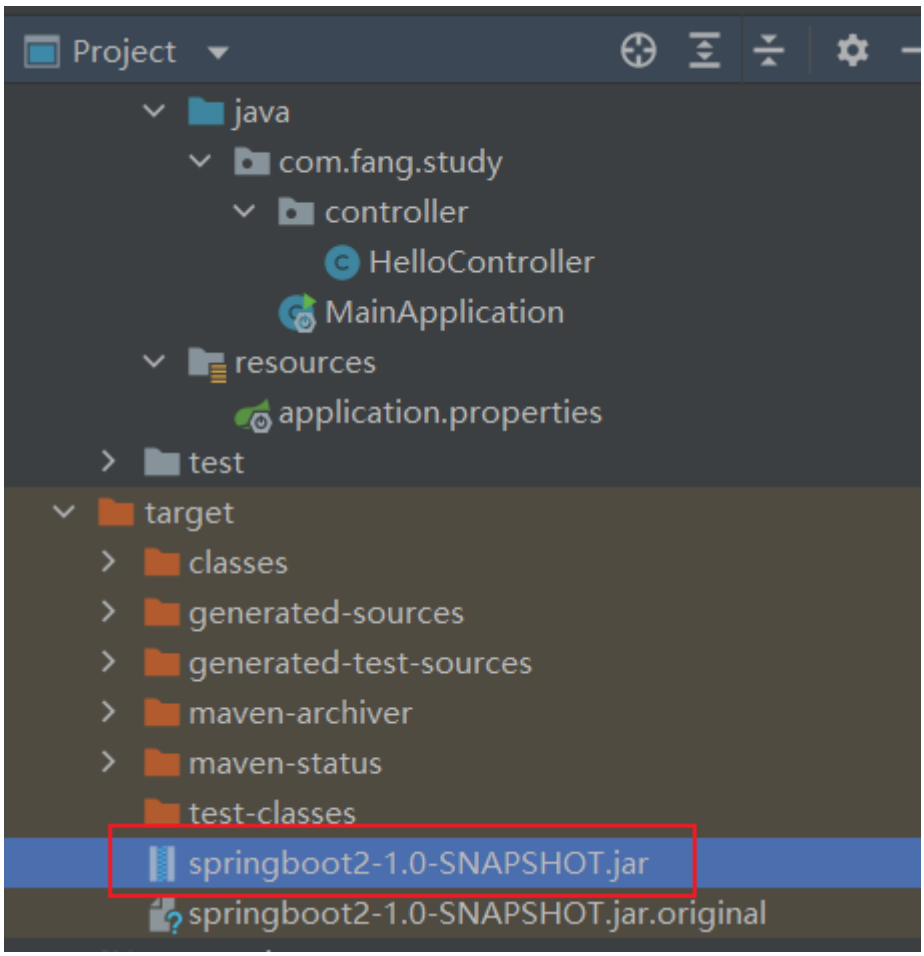
- **SpringBoot 简化了配置**
- 在 resources 中新建一个 application.properties/.yml 文件来**全局管理**所有配置
    - 修改 tomcat 配置（比如：端口号）
    - 修改 SpringMVC 配置（web 上）
    - 参照官方文档： https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties
- **运行**
- 编写一个主程序
- 直接运行注解了 @SpringBootApplication 类下的 main 函数（主程序）即可
- **部署**
- 在 pom.xml 添加插件

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

- 打包成 jar 包



- 打开命令行运行该 jar 包




```
# 使用命令行运行
java -jar springboot2-1.0-SNAPSHOT.jar
```

1. SpringBoot 的特点

1.1 依赖管理

- 每个 SpringBoot 都有一个父工程

```
<!--    使用 SpringBoot 功能需要引入 SpringBoot 父项目依赖-->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.2</version>
    </parent>

<!-- 父项目的父项目-->
 <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>
    <version>2.7.2</version>
  </parent>
```

作用：

- 用父项目来做依赖管理
  - 子项目继承了父项目，使用其他依赖就不需要版本号 (比如 web 依赖就不需要版本号)，因为父项目几乎声明了所有开发中常用的依赖版本号，即自动版本仲裁机制（已经有了默认版本号）
  - 如果想自定义版本号，在 pom.xml 中添加如下来修改重写

```
<!-- 修改自己想要的版本(每个依赖都有对应属性[查看 spring-boot-dependencies
中]，此处为 mysql.version)-->
<properties>
        <mysql.version>5.1.43</mysql.version>
 </properties>
```

- starter 场景启动器
  - 例如：spring-boot-starter-*，*代表场景，只要引入该 starter，相关场景的依赖都会自动引入
  - 所支持的 starter: https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.build-systems.starters

```
<!-- 所有场景启动器最底层的依赖 -->
 <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
      <version>2.7.2</version>
      <scope>compile</scope>
 </dependency>
```

1.2 自动配置

- 自动配置好 Tomcat
  - 引入了 Tomcat 依赖
  - 配置了 Tomcat
- 自动配好了SpringMVC
  - 引入了 SpringMVC 全套组件
  - 自动配好 SpringMVC 常用组件（功能）
    - 字符编码问题
    - 拦截器
    - 视图解析器
    - SpringBoot 帮我们配置好了所有 web 开发的常见场景

```
//打印 spring MVC 相关组件
ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class,
args);
 String[] names = run.getBeanDefinitionNames();
 for (String name : names) {
     System.out.println(name);
 }
```

- 默认的包结构
  - 参考链接： https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.structuring-your-code
  - 主程序所在的包及其子包里面所有的组件都会被默认扫描，之外的不会被扫描

    - 无需以前包扫描
    - 如果需要修改包扫描，可以在注解添加属性 @SpringBootApplication(scanBasePackage="包名")

      或者使用 @ComponentScan 制定扫描路径

- 各种配置（application.properties文件当中内容）拥有默认值
    - 默认配置最终都是映射到某一个类上
    - 配置文件的值最终会绑定某个类上，这个类在容器中会创建对象（对象有默认值）
- 按需加载所有自动配置项
    - starter 引入了哪个场景就加载哪个场景，相关配置才会自动开启
    - SpringBoot 所有自动配置功能，包含所有场景（点开IDEA显示红色就是未被导入）

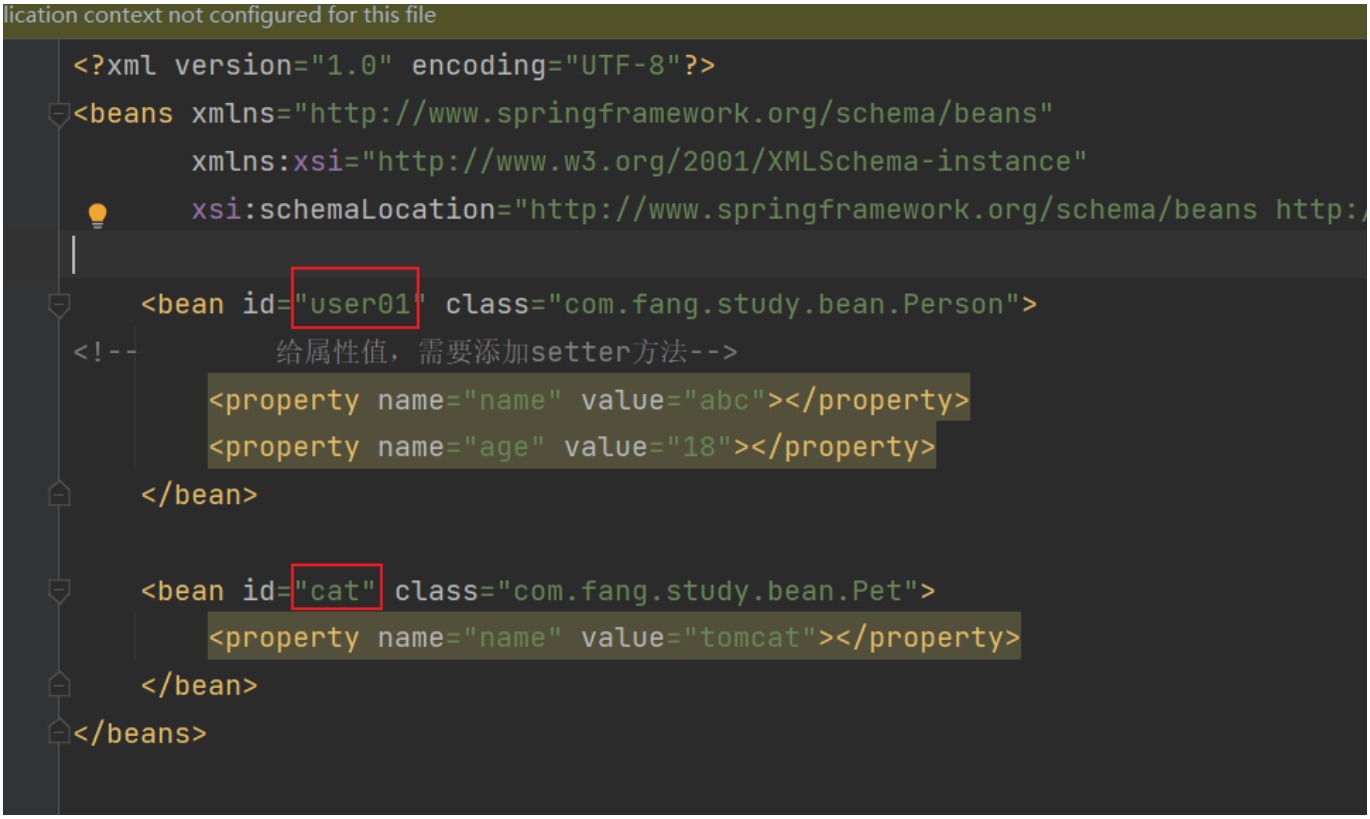
```
<!--在springboot-starter中 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-autoconfigure</artifactId>
  <version>2.7.2</version>
  <scope>compile</scope>
</dependency>
```

2. 容器功能

2.1 组件添加

2.1.1 @Configuration

- 对比以前的 Spring，在容器中注册组件，需要创建配置文件，使用 bean 标签



- SpringBoot 使用 @Configuration 和 @Bean 注解，使用 @Bean 标注来向容器中注册组件，是**单实例**
  - **没有 @Configuration 注解，那么 @Bean 就不会生效**
  - 配置类（带 @Configuration 注解）本身就是组件
  - @SpringBootApplication： 标识主配置类
  - @Configuration
    - proxyBeanMethods = true，默认就是 true，**单实例的保证**，**组件依赖模式** ------ Full 模式
    - proxyBeanMethods = false 不用每次都去容器里面找，直接创建即可，不保证单实例 ------ Lite模式
    - **Full/Lite 模式（SpringBoot2 特有）**
  - 外部无论对配置类中的组件注册方方调用多少次，仍是之前注册在容器中的单实例对象，原因在于 proxyBeanMethods = true

```java
/**
 * @Configuration 告诉 SpringBoot 这是一个 配置类，相当于原来 spring.xml 的配置文件
 * @author bestFang666
 * @date 2022/8/16 16:17
 */
@Configuration(proxyBeanMethods = true)
public class MyConfig {

    /**
     * 方法名对应 bean 标签的 id
     * @Bean 给容器注册组件,默认以方法名作为组件名，可以为该注解赋值，从而改变组件名 @Bean（“abc”）
     * @return 组件自容器中的实例（保存的对象）
     */
    @Bean(name = "bbb")
    public Person person01() {
        return new Person( name: "aaa", age: 18);
    }
```

2.1.2 @Component/@Controller/@Service/@Repository

继承原来 Spring 老一套

- @Component： 类上标注表示该类是一个组件
- @Controller： 类上标注表示该类是一个 web 控制器组件
    - RestController 和 Controller： http://www.wjhsh.net/ggjun-p-11311876.html
- @Service： 类上（标注在 service 实现类上）标注表示该类是一个业务逻辑组件
- @Repository： 类上标注表示该类是一个数据库层的组件

2.1.3 @Import

- 此注解**写在组件上**

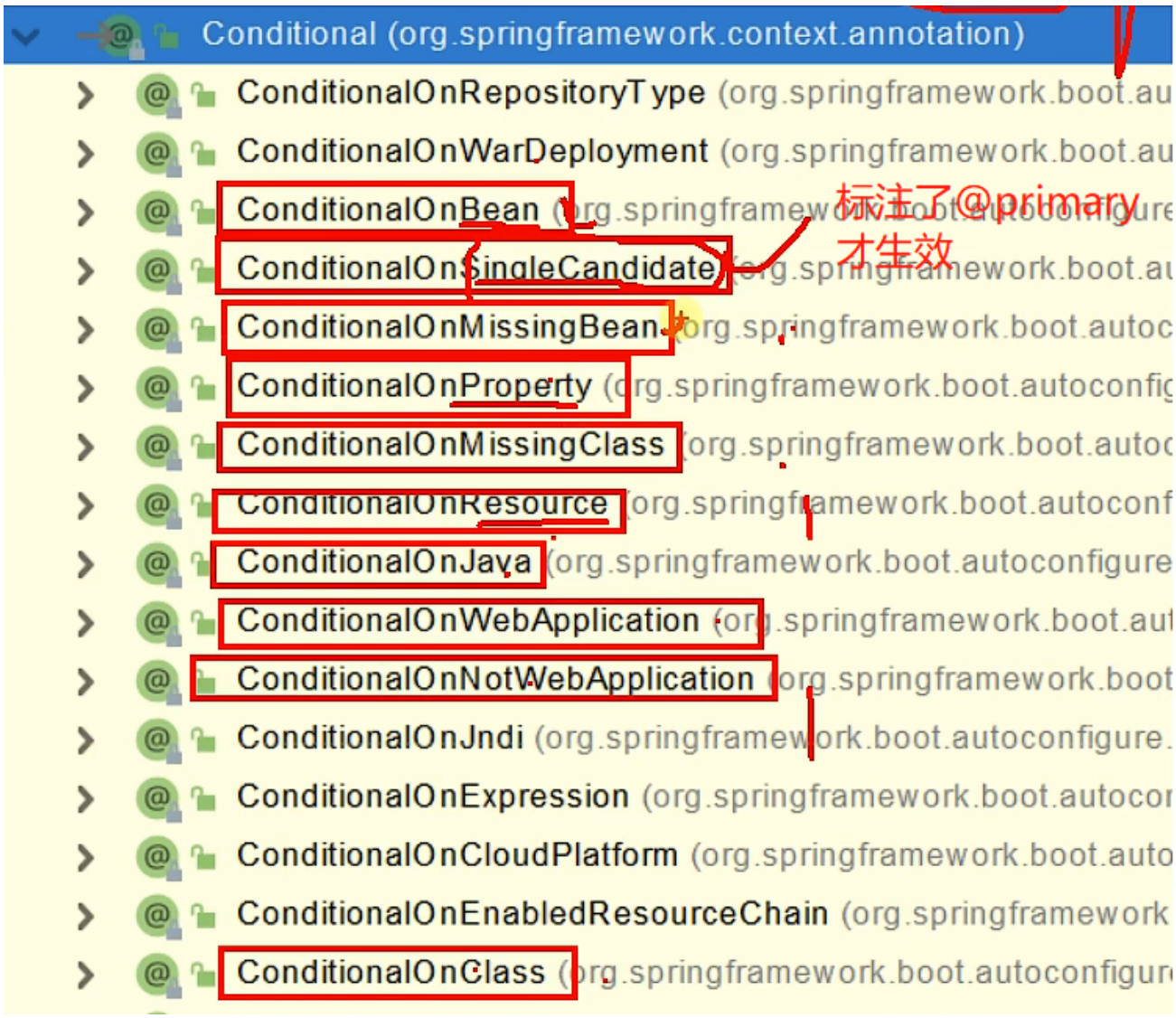
```java
@Import({Person.class, Pet.class})
@Configuration(proxyBeanMethods = true)
public class MyConfig {

    /**
     * 方法名对应 bean 标签的 id
     * @Bean 给容器注册组件,默认以方法名作为组件名，可以为该注解赋值，从而改变组件名 @Bean（“abc”）
     * @return 组件自容器中的实例（保存的对象）
     */
    @Bean(value = "你好啊")
    public Person person01() {
        return new Person( name: "aaa", age: 18);
    }
```

- @ import({A.class, B.class, ...})：给容器中自动创建 A, B 这两个类型的组件, 默认组件的名字是**全类名**（com.fang.XXX.xxx），与 @Bean 区别开，@Bean 注册的组件是**方法名**

2.1.4 @Conditional

- 条件装配：满足 Conditional 指定的条件，进行组件注入（注册）

  常用



- ConditionalOnXXX 对 XXX 生效

  例如：如果容器中没有 tomcat 组件，那么”你好啊“组件就不会自动注入容器（此处**不存在“你好啊”组件**，原因是向容器中注入组件是按照代码顺序逻辑，“你好啊”注入时候“tomcat”还没注入）

```java
@ConditionalOnBean(name = "tomcat")
@Bean(value = "你好啊")
public Person person01() {
    return new Person( name: "aaa", age: 18);
}

@Bean
public Pet tomcat(){
    return new Pet( name: "tomcat");
}
}
```

2.2 原生配置文件引入

2.2.1 @ImportResource

- 在**随便一个类上**加入该注解既可以将配置文件（eg. beans.xml）中的组件注入容器当中

```java
@Import({Person.class, Pet.class})
@ImportResource("classpath:beans.xml")
@Configuration(proxyBeanMethods = true)
public class MyConfig {

    /**
     * 方法名对应 bean 标签的 id
     * @Bean 给容器注册组件,默认以方法名作为组
     * @return 组件自容器中的实例（保存的对象
     */
```

2.3 配置绑定

- 使用 Java 读取 properties 文件中的内容，然后**把它封装到 JavaBean** 中，以供使用

2.3.1 @Component + @ConfigurationProperties

- @ConfigurationProperties(prefix = "") 是跟 .properties 配置文件绑定
- @Component 组件注入
- 读取的话一定要有 **Getter** 方法
- 在**对应的类**上标识

```java
/**
 * @Component 要注入为组件，才能拥有 SpringBoot 所提供的强大功能
 * @ConfigurationProperties 仅仅是一个注解，读取 application.properties 中的内容
 *                          prefix: 与自己所定义的类的属性名对应之外的部分
 * @author bestFang666
 * @date 2022/8/17 11:09
 */
@Component
@ConfigurationProperties(prefix = "mycar")
public class Car {
    private String brand;
    private Double price;
```

2.3.2 @EnableConfigurationProperties + @ConfigurationProperties

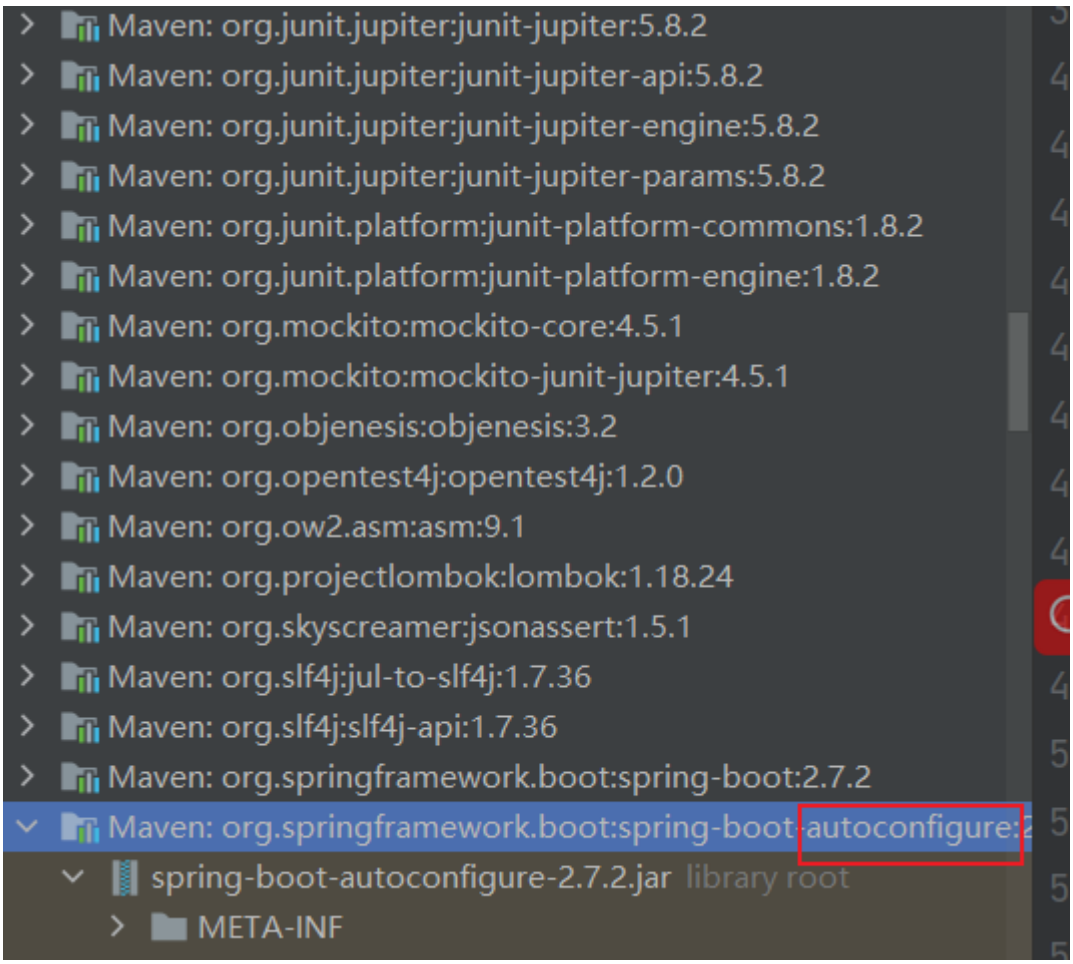
- @EnableConfigurationProperties(A.class) 在**配置类**上标识
- @ConfigurationProperties 在**对应的类**上标识
- 开启配置绑定功能，并且组件自动注入到容器中
- 对于使用**第三方类**时，可以使用该方法，不再使用2.3.1中方法

3. 实践

3.1 最佳实践

- 引入场景依赖 ------ starter-xxx
    - 查看依赖，当前层级看不到，往上一层级看

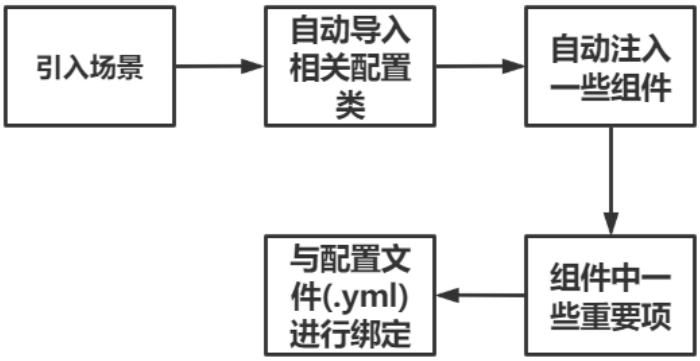
- 查看自动配置了哪些，**两种方法**
  - 自己分析，引入了某个场景，对应的自动配置就生效了
    - 举例：



  - 开启自动配置报告，在配置文件（.properties）添加 debug=true
- 是否要修改
  - 参照文档来修改配置项（改 .properties 文件中的内容）
  - 自己分析，看相关 xxx.properties 绑定了哪些
  - 自定义加入或者替换组件
  - 自定义器 XXXXXCustomizer

总结

- 使用SpringBoot过程



4. 开发小技巧

4.1 lombok

- 简化了 JavaBean 的开发，不需要给 pojo 写 getter&setter 方法和一些构造器方法
- 在 pom.xml 中导入依赖

```
<!--        引入 lombok-->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
        </dependency>
```

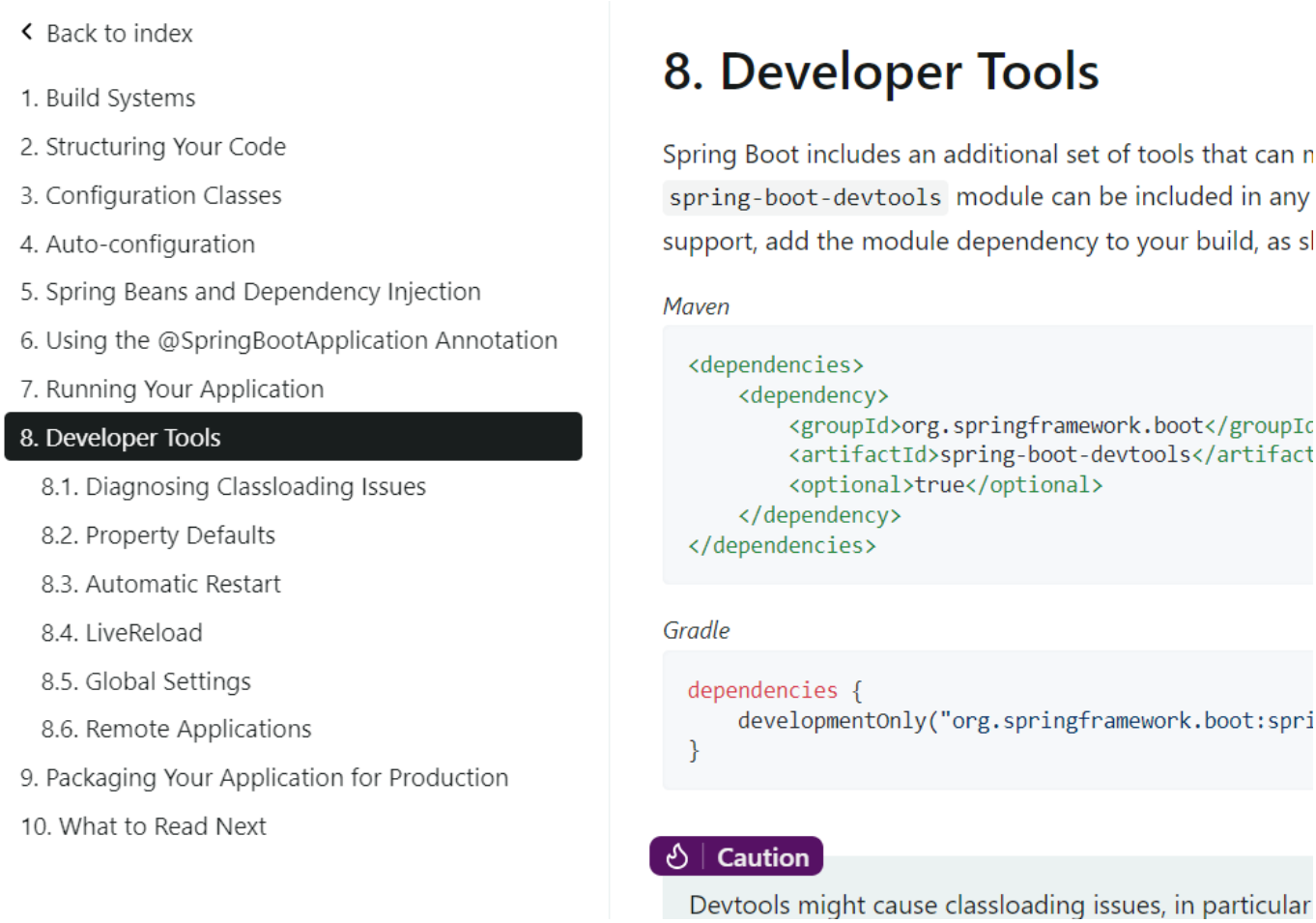
- 安装 lombok 插件
- 直接在 pojo 上面写上注解，不自动生成就自己写
  - @Data： 生成 getter/setter 方法
  - @ToString： 生成 ToString 方法
  - @AllArgsConstructor： 生成全参构造器
  - @NoArgsConstructor： 生成无参构造器
  - @EqualsAndHashCode： 生成 equals 和 hashcode 方法
- @Slf4j： 日志注解
  - 调用 log.info("打印xxx") ------ 打印日志

4.2 dev-tools

- 参考链接： https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.devtools

The reference documentation consists of the following sections:

Legal	Legal information.
Getting Help	Resources for getting help.
Documentation Overview	About the Documentation, First Steps, and more.
Getting Started	Introducing Spring Boot, System Requirements, Servl
Upgrading Spring Boot Applications	Upgrading from 1.x, Upgrading to a new feature relea
Using Spring Boot	Build Systems, Structuring Your Code, Configuration,
Core Features	Profiles, Logging, Security, Caching, Spring Integratio
Web	Servlet Web, Reactive Web, GraphQL, Embedded Con
Data	SQL and NOSQL data access.
IO	Caching, Quartz Scheduler, REST clients, Sending ema
Messaging	JMS, AMQP, Apache Kafka, RSocket, WebSocket, and



- 导入依赖

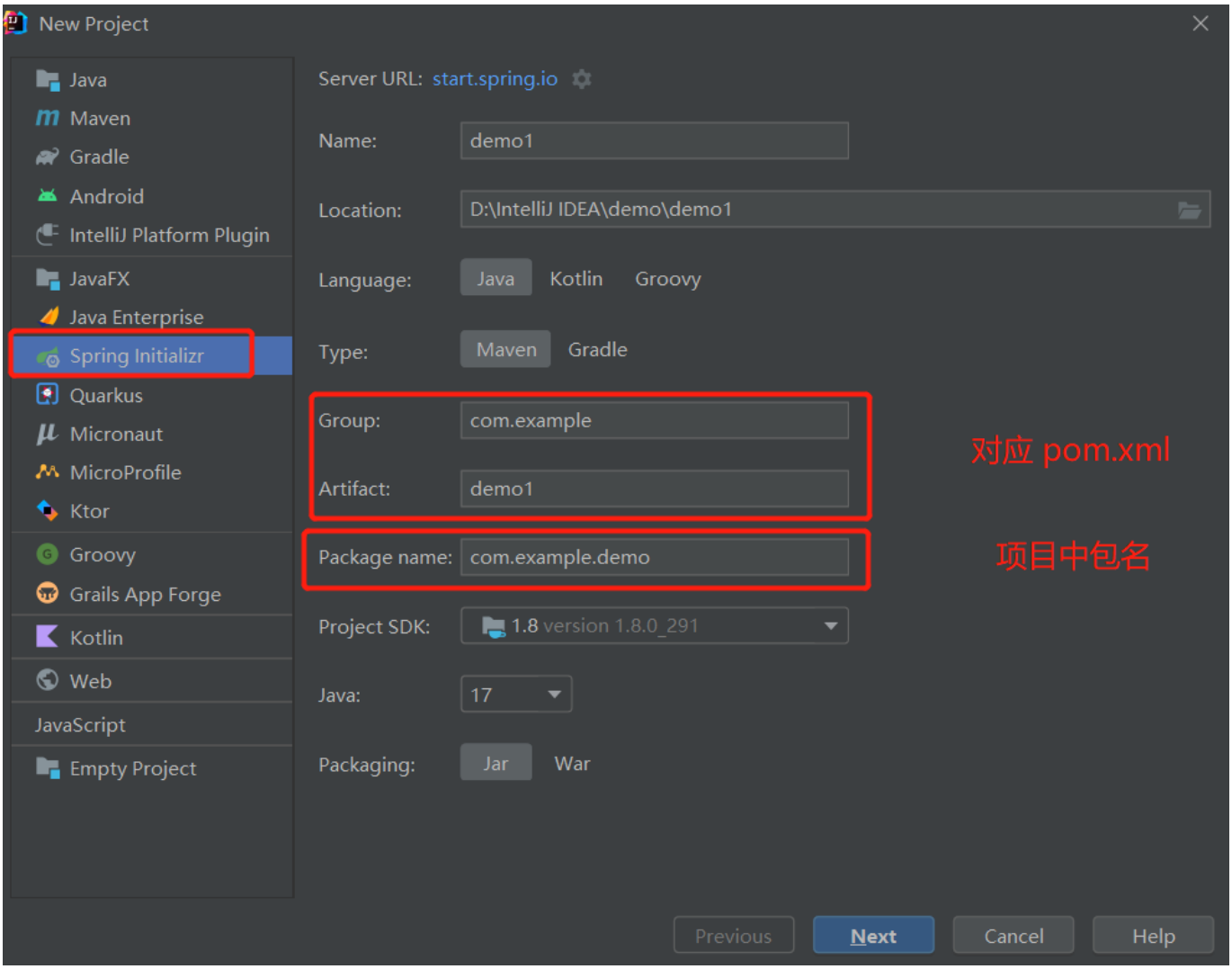
```
<!-- 项目的热更新 -->
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

- 重启项目

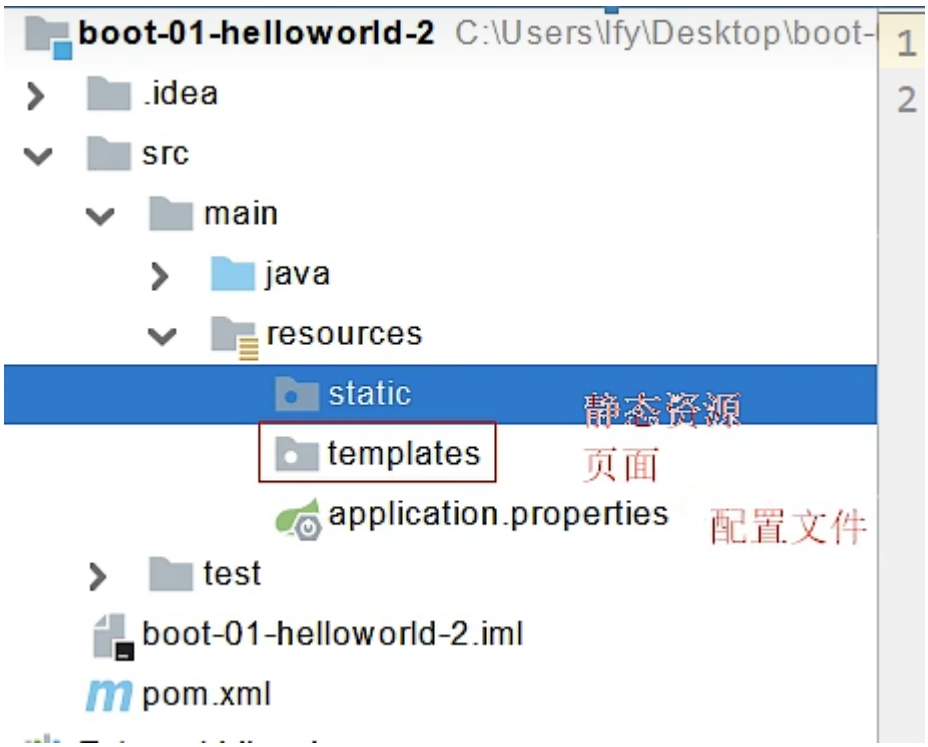
```
# 直接使用该命令重载
Crtl + F9
# 不用去按重启按钮（原来方式）
shift + F10
```

4.3 Spring Initailizr

- 项目初始化向导



- 可以**<u>按需去勾选相关的场景</u>**，他会自动给你所给需求下载一个项目供开发者使用
- 创建之后，包含了**依赖**、项目的**目录结构**，创建了**主程序**



SpringBoot2 核心技术篇

1. 配置文件

- 如果有多个配置文件，按顺序优先选择第一个

1.1 properties

- 配置一个全局文件

```
# 修改服务端口号
server.port = 8080

mycar.brand = BYD
mycar.price = 100000

# 可以查看那些自动配置类生效了
debug = true
```

1.2 yaml

- 非常适合用来做**以数据为中心**的配置文件，**轻量**，只是与 .properties 写法不同
- 注意层级之间的对齐

- 字符串无需加引号，如果加上，单引号代表转义（即可以输出转义字符[\n 输出 '\n']），双引号代表不转义（转义字符按照原来输出[\n 输出为换行]）

```
# k 和 v 之间有空格
# 字面量
k: v

# 对象/键值对
# 行内写法
k: {k1: v1, k2: v2, k3: v3}
# 等价于
k:
   k1: v1
   k2: v2
   k3: v3

# 数组集合（map 不是集合）
# 行内写法
k: [v1, v2, v3]
# 等价于(- 和 v 有空格)
k:
 - v1
 - v2
 - v3
```

- 配置提示
  - 配置文件中**<u>没有自定类的自动提示</u>**，加入下面依赖便于开发
  - 需要使用注解@ConfigurationProperties(prefix = "person")才能与配置文件进行绑定，下面依赖才生效

```
<!--        自定义类与配置文件进行绑定-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
            <optional>true</optional>
        </dependency>
<!-- 轻量打包 -->
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <!--  除去非业务逻辑（比如：插件、代码识别 etc）的依赖，打包时候会忽略下面的配置-->
                    <excludes>
                        <exclude>
                            <groupId>org.springframework.boot</groupId>
                            <artifactId>spring-boot-configuration-processor</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

2. web 开发

- 参考链接： https://docs.spring.io/spring-boot/docs/current/reference/html/web.html#web

Legal	Legal information.
Getting Help	Resources for getting help.
Documentation Overview	About the Documentation, First Steps, and more.
Getting Started	Introducing Spring Boot, System Requirements, Servlet Containers, Installing Spring
Upgrading Spring Boot Applications	Upgrading from 1.x, Upgrading to a new feature release, and Upgrading the Spring
Using Spring Boot	Build Systems, Structuring Your Code, Configuration, Spring Beans and Dependency
Core Features	Profiles, Logging, Security, Caching, Spring Integration, Testing, and more.
Web	Servlet Web, Reactive Web, GraphQL, Embedded Container Support, Graceful Shutc
Data	SQL and NOSQL data access.
IO	Caching, Quartz Scheduler, REST clients, Sending email, Spring Web Services, and n
Messaging	JMS, AMQP, Apache Kafka, RSocket, WebSocket, and Spring Integration.
Container Images	Efficient container images and Building container images with Dockerfiles and Clou
Production-ready Features	Monitoring, Metrics, Auditing, and more.

2.1 静态资源规则与定制化

2.1.1 静态资源目录

- 在类路径下（即 resources 文件下） `/static` (or `/public` or `/resources` or `/META-INF/resources`) 文件夹里面的动态资源都可以直接通过根路径来访问

  开发常放在 /static 目录下

  ```
  localhost:8080/abc.jpg
  ```

- 原理

  默认是 /**

  - **请求顺序**

  1. 找 Controller
  2. 找静态资源
  3. 404

2.1.2 静态资源访问前缀 && 路径

- 默认是**无前缀**，直接通过 /资源名 来访问

- 加前缀

  访问静态资源路径变为： /sta/资源名 （此处参照下面代码模块）

- 配路径

  参照如下代码，开发时候**一般不配置**

```yaml
spring:
  mvc:
    # 配置静态资源前缀 (默认是 /**)
    # 这个前缀会影响欢迎页 index.html 和网页logo favicon.icon的显示
    static-path-pattern: /sta/**

  web:
    resources:
      # 默认是在 resources 目录下四种文件夹（表示形式为集合）
      # 修改过后所有的静态资源都放在 static-locations 中
      static-locations: [classpath:/aaa/]
```

2.1.3 欢迎页 && 图标

1.1.5. Welcome Page

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

一定要按这个命名

1.1.6. Custom Favicon

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

- 注意： Favicon **用法好像有问题，运行不出来**

2.2 请求处理

2.2.1 Rest 风格

- @RequestMapping(value = "/user",method = RequestMethod.GET/DELETE/PUT/POST)

  value = {} 可以是个数组，也可以是单个值

  - **GET**： 获取/查询 ------ /getUser

- DELETE： 删除 ------ /deleteUser
- PUT： 修改 ------ /editUser
- **POST**： 保存 ------ /saveUser

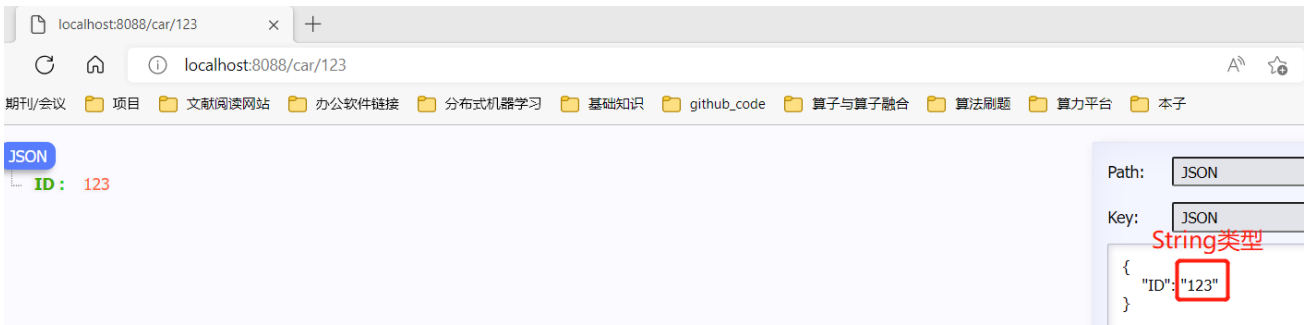
- 一般只使用注解@GetMapping（url）/ @PostMapping(url)
  - GetMapping： 获取/查询
  - PostMapping： 删除/修改/保存

2.2.2 注解方式

- **@PathVariable("key") object / @PathVariable Map <String, String>**

```
@RestController
public class TestController {

    @RequestMapping("/car/{id}")
    public Map<String,Object> test1(@PathVariable("id") String ID) {
        Map<String,Object> map =new HashMap<String,Object>();
        map.put("ID",ID);
        return map;

    }
}
```

```
# 请求链接
localhost:8080/car/123
```



详解：@PathVariable("id") String ID

- 读取请求链接上的 123 对应到 id 上面，然后赋值给 ID

- **@RequestHeader 获取请求头**

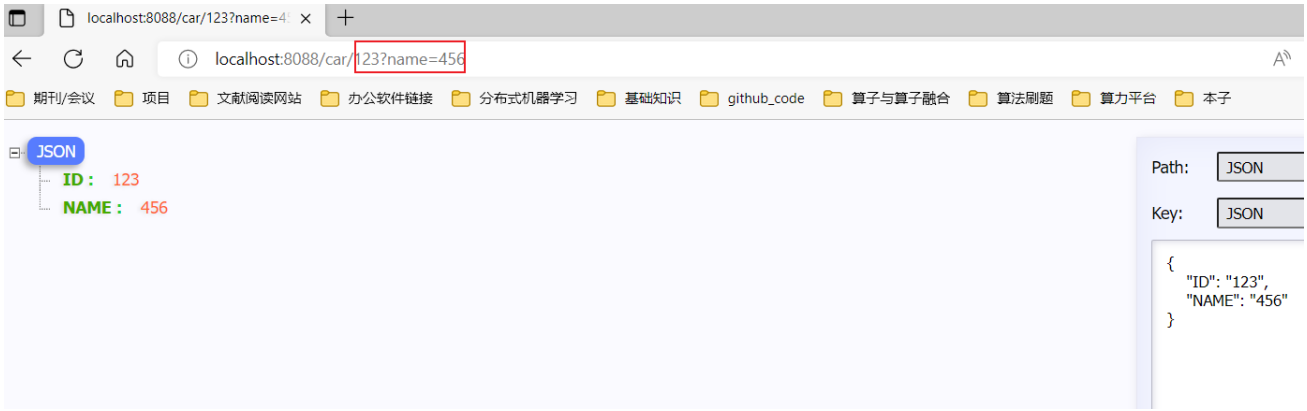
- @RequestHeader("key") object / @RequestHeader Map<String, String>, MultiValueMap<String, String>, or HttpHeaders
  - key 是对应请求中的字段

- **@RequestParam("key") object / @RequestParam Map<String, String> or MultiValueMap<String, String>**

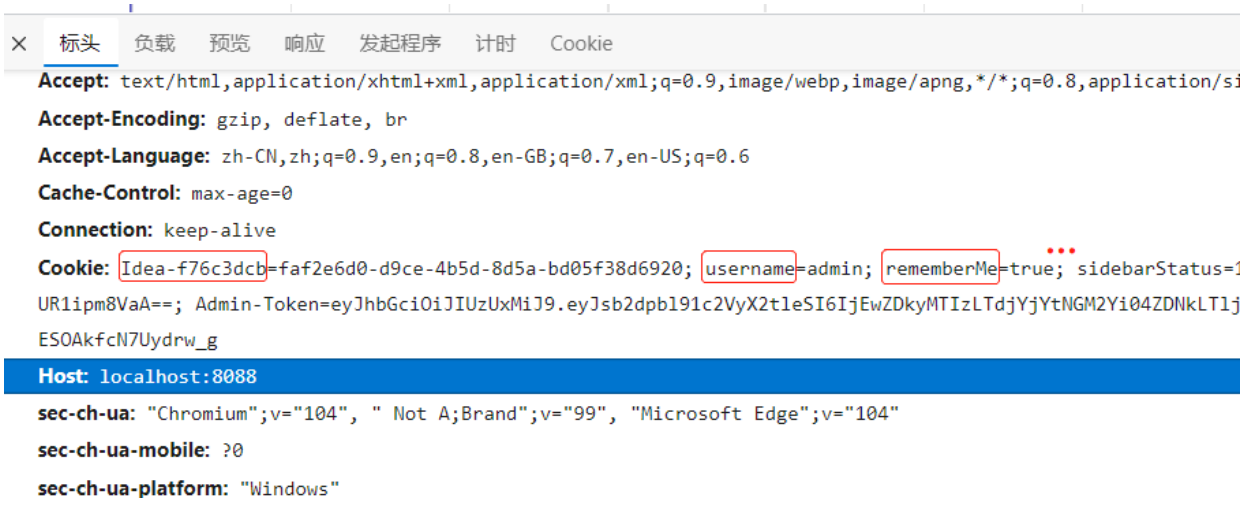
```
@RestController
public class TestController {

    @RequestMapping("/car/{id}")
    public Map<String,Object> test1(@PathVariable("id") String ID,
                                    @RequestParam("name") String NAME) {
        Map<String,Object> map =new HashMap<String,Object>();
        map.put("ID", ID);
        map.put("NAME", NAME);
        return map;

    }
}
```

访问链接：

```
# 请求链接
# ***标了什么参数就要设置什么参数，queryString 格式***
localhost:8080/car/123?name=456
```



- **@CookieValue("key") object** 获取 cookie 值
  - key 是 cookie 中的字段
  - **@CookieValue("key") Cookie object** 也可封装为一个 cookie 对象



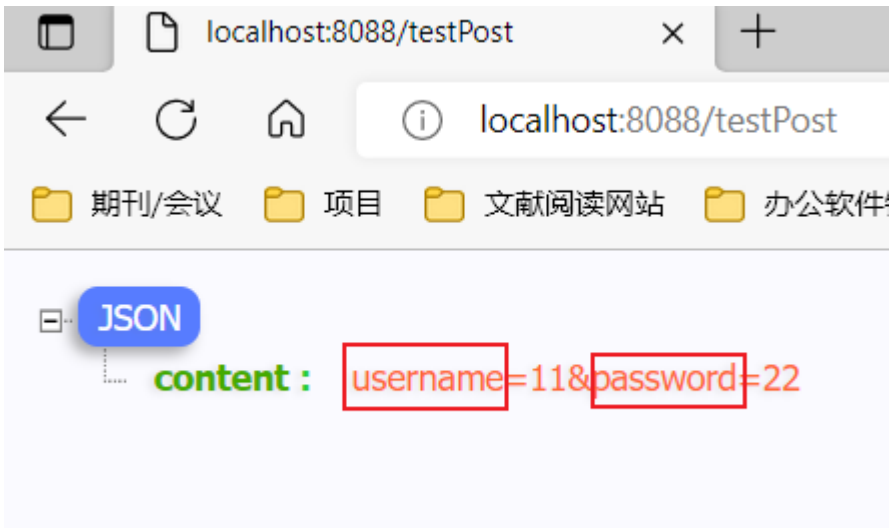
- **@RequestBody Object object** 获取请求体 [post]
  - 前端对应表单提交到后端
  - 使用 @PostMapping
  - 如果**前端表单中的 name 字段名和后端 pojo 实体类中的属性相同**，后端可以直接**以对象形式或者以对象中的属性直接接收**
    - 以下面例子为例，定义了一个 Person 对象，其中含有属性 username&password，那么 post 请求中的参数可以定义为 Person person（或者 String username, String password），**不用加注解**，因为 **SpringMVC 表单自动封装到Controller对象**
    - **注意：** 如果使用 @RequestBody Person person 来接收，那么前端发请求时候需要修改content-type 为 json 格式，因为 @RequestBody对应请求体，以 json 形式封装。

- **也可以使用 @RequestParam("表单中的 name 属性") Object object**

```
<!--测试post请求，action 绑定后端请求链接-->
    <form action="/testPost" method="post">
        <p>
            <label>姓名：</label><input type="text" name="username" value="11" />
        </p>
        <p>
            <label>密码：</label><input type="password" name="password" />
        </p>
        <p>
            <input type="submit" name="" value="提交">
            <input type="reset" name="" value="重置">
        </p>
    </form>
```

```
@PostMapping("/testPost")
    public Map<String, Object> test2 (@RequestBody String content) {
        Map<String,Object> map =new HashMap<String,Object>();
        map.put("content", content);
        return map;
    }
```

- 结果展示，按照前端中 name 字段发送给后端



- **@RequestAttribute** 获取request域属性
    - 用于页面转发，得到前一页面的请求数据

```java
@Controller
public class RequestController {

    @GetMapping("/origin")
    public String goToPage (HttpServletRequest request) {
        // 给请求中设置属性
        request.setAttribute("msg","跳转成功...");
        request.setAttribute("code", "200");
        // forward 表示跳转至那个请求
        return "forward:/success";
   /**
     * @ResponseBody 将前端传来的json格式的数据转为自己定义好的 javabean 对象
     * @param MSG
     * @param CODE
     * @return
     */
    @ResponseBody
    @GetMapping("/success")
    public Map<String, Object> endPage(@RequestAttribute("msg") String MSG,
                                       @RequestAttribute("code") Integer CODE){
        Map<String, Object> map =new HashMap<String, Object>();
        map.put("msg",MSG);
        map.put("code",CODE);
        return map;
    }
}
```

总结：@RequestAttribute("msg") String MSG 把请求中的属性值封装到指定对象中

- @MatrixVariable
    - 举例：

```java
//1、语法： 请求路径：/cars/sell;low=34;brand=byd,audi,yd
//2、SpringBoot默认是禁用了矩阵变量的功能
//      手动开启：原理。对于路径的处理。UrlPathHelper进行解析。
//              removeSemicolonContent（移除分号内容）支持矩阵变量的
//3、矩阵变量必须有url路径变量才能被解析
@GetMapping("/cars/{path}")
public Map carsSell(@MatrixVariable("low") Integer low,
                    @MatrixVariable("brand") List<String> brand,
                    @PathVariable("path") String path){
    Map<String,Object> map = new HashMap<>();

    map.put("low",low);
    map.put("brand",brand);
    map.put("path",path);
    return map;
}

// /boss/1;age=20/2;age=10

@GetMapping("/boss/{bossId}/{empId}")
public Map boss(@MatrixVariable(value = "age",pathVar = "bossId") Integer bossAge,
                @MatrixVariable(value = "age",pathVar = "empId") Integer empAge){
    Map<String,Object> map = new HashMap<>();

    map.put("bossAge",bossAge);
    map.put("empAge",empAge);
    return map;

}
```

    - 应用场景举例：
        - 当把 cookie 禁用了，无法得到 sessionid， 从而无法得到 session， 最后无法得到里面的数据（比如：无法得到网页中的数据）

- **@RequestParam和@RequestBody区别**

    1. @RequestParam 接收的参数是来自 requestHeader 中，即**请求头**
    2. @RequestBody 接收的参数是来自 requestBody 中，即**请求体**
    3. @RequestParam用来处理 `Content-Type` 为 `application/x-www-form-urlencoded` 编码的内容
    4. 一般用于处理非 `Content-Type: application/x-www-form-urlencoded` 编码格式的数据，比如： `application/json` 、 `application/xml` 等类型的数据。
    5. GET请求中，因为没有 HttpEntity，所以 @RequestBody 并不适用。

POST请求中，通过 HttpEntity 传递的参数，**必须要在请求头中声明数据的类型Content-Type**

6. 当前端请求的 Content-Type 是 Json 时，可以用 @RequestBody 这个注解来解决；@RequestParam 用来处理 Content-Type: 为 application/x-www-form-urlencoded 编码的内容，提交方式 GET、POST。

2.3 视图接信息与模板引擎

- SpringBoot 不支持 JSP
- src/main/resources/templates 下面放页面
- 视图解析

```
# forward 与 redirect 的区别，参考链接：
https://blog.csdn.net/weixin_37766296/article/details/80375106
# 防止发 post 请求时候，表单重复提交
使用 重定向 redirect, eg: redirect:/aaa
```

- 模板引擎： Thymeleaf (供后台服务开发使用，不支持**前后端分离**)

```
# 简介及基础使用参见如下链接：
# （第一季 -> 05 -> 2 和 3）
https://www.yuque.com/atguigu/springboot/vgzmgh#lzseb
# ThymeLeaf 本身的网页默认放在 templates 文件夹下面，访问时候不用加.html后缀
https://blog.csdn.net/wenmou_/article/details/124414888
# 使用参加官网
https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html
```

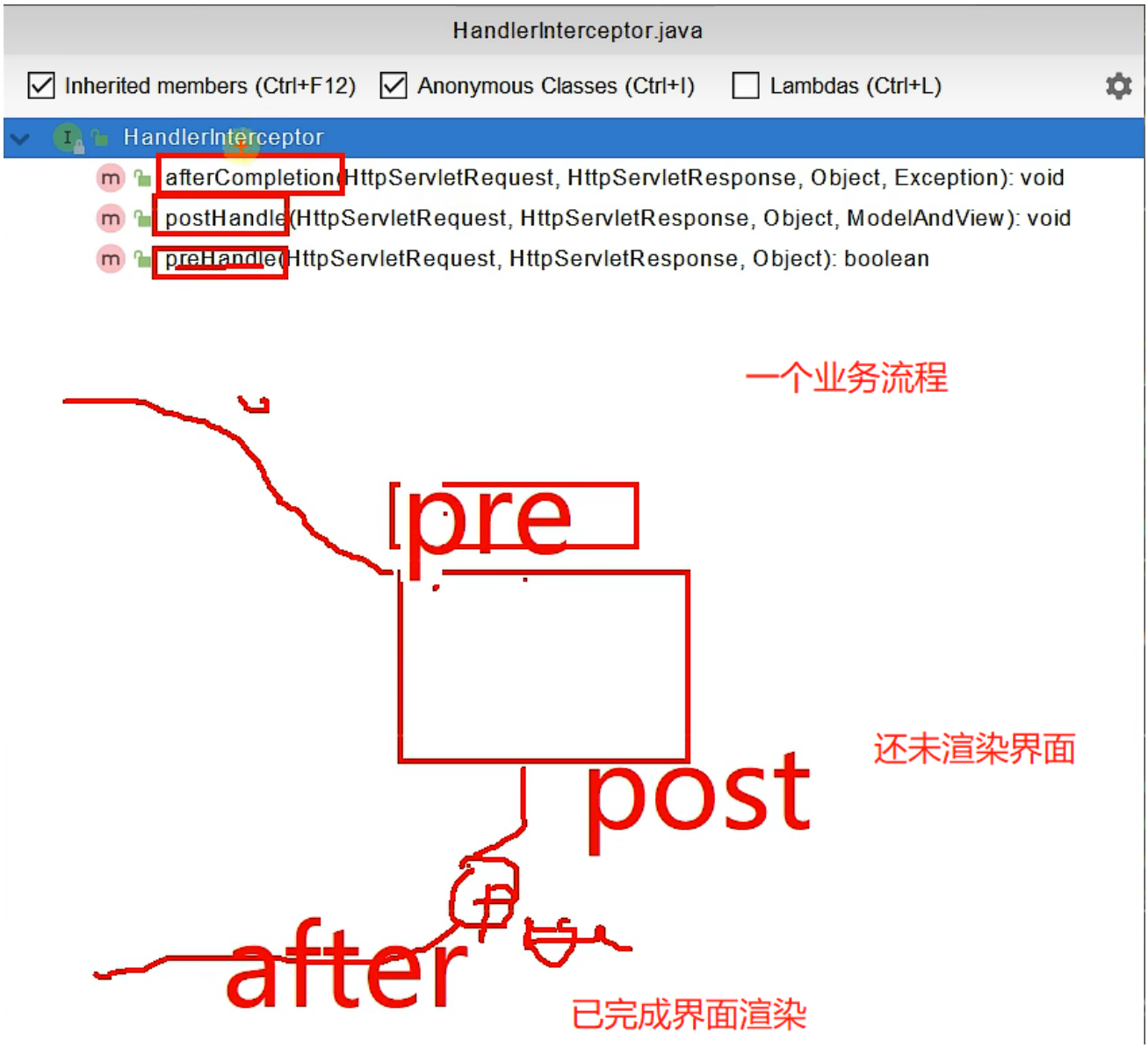
2.4 后台管理系统一些常识

- 发送一个请求，对象保存在 session 中，即给方法增加一个参数 HTTPSession session， 把对象保存在当中（使用 setAttribute() 方法）
    - 使用场景举例：登录成功后，把用户信息保存在 session 中，**前端使用session.object 取得**
- 发送一个请求，数据保存在 model 中，即给方法增加一个参数 Model model，把相关数据保存在当中（使用 addAttribute() 方法）
    - 使用场景举例：登录成功后，把请求的状态码和 message 提示保存在当中，**前端可以直接取得**

- Session 和 model 中的数据前后端都可以使用，随便用，二者没有强调特殊场景
- 写页面时候，记得**抽取公共页面**，方便调用

2.5 拦截器

- 接口名： HandleInterceptor



- 自定义拦截器接口，首先需要实现原生接口，重写 pre（目标方法执行前）、post（目标方法执行后且页面渲染前）、after （页面渲染后）三个方法（**返回值为 boolean 类型，true 表示放行，false 表示拦截**）
- **拦截器使用步骤**
  - 定义好拦截器实现 HandlerInterceptor 接口

- 拦截器注册带容器中（**定义一个配置类**实现 webMVCConfigurer 并标注 @Configuration 注解，调用 addInterceptor 方法）
- 指定拦截规则

  举例：

```
@Configuration
public class AdminWebConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoginInterceptor()) 添加一个拦截器
                .addPathPatterns("/**")                  拦截所有，静态资源也会拦截
                .excludePathPatterns("/","/login"); //不拦截哪些
    }
}
```

2.6 单文件与多文件上传

- 使用注解 @RequestPart("表单中的name") MutipartFile object 来从前端获取到后端
- MutipartFile 该类型自动封装前端传过来的文件
- 前端代码

```
<form method="post" action="/upload" enctype="multipart/form-data">
<!-- 多文件上传，添加属性 multiple -->
    <input type="file" name="filename"></br>
    <input type="submit" value="提交">
</form>
```

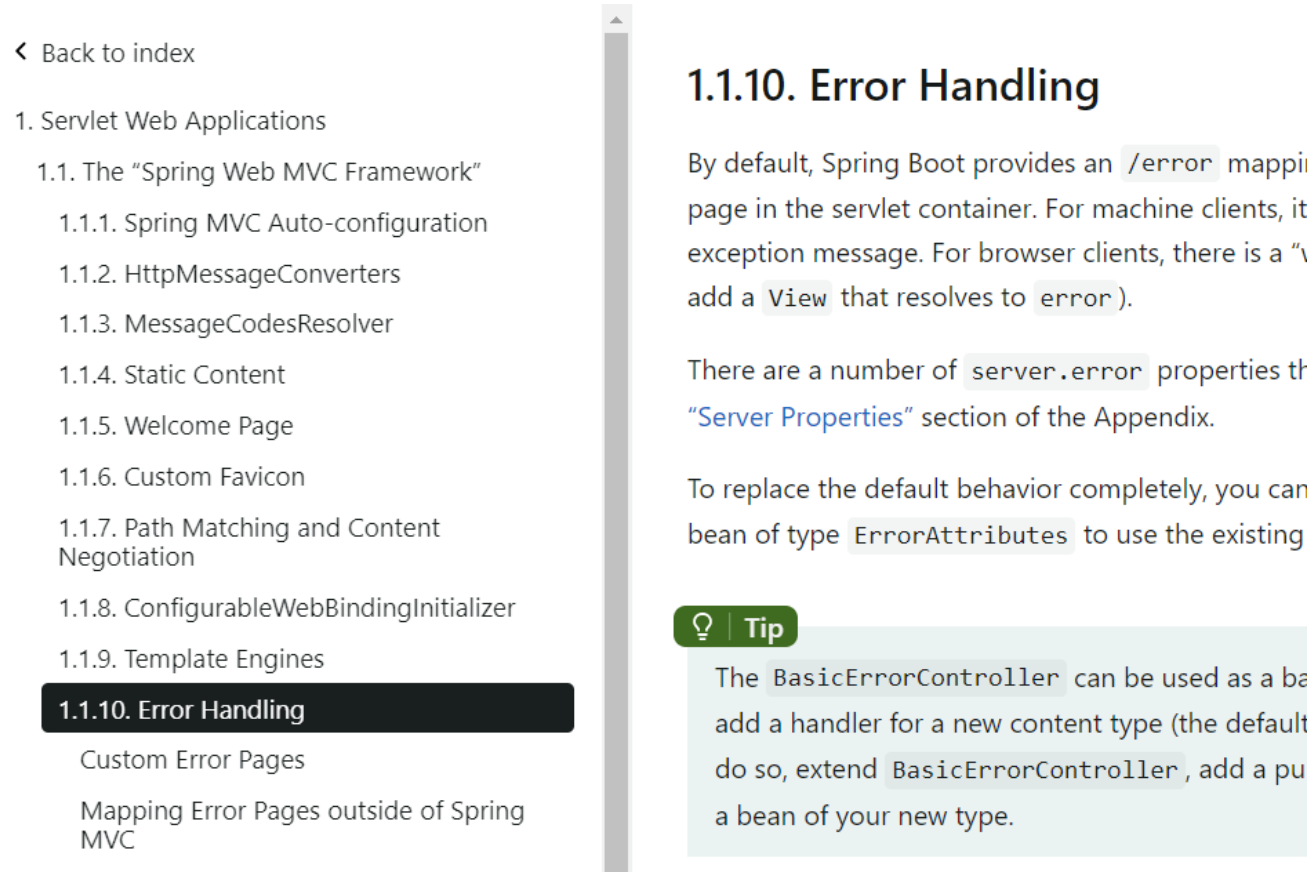
- 注意修改单个/整体文件的最大 size， 在 application.properties/.yml 中配置

```
# 最大单个/整体文件大小
  servlet:
    multipart:
      max-file-size: 10MB
      max-request-size: 100MB
```

- **文件上传自动配置类**-MultipartAutoConfiguration-MultipartProperties

2.7 异常处理

- 官方文档链接： https://docs.spring.io/spring-boot/docs/current/reference/html/web.html#web.servlet.spring-mvc.error-handling



- 使用方法：
  - 在 templates 或者存放静态资源的文件夹中创建 error 文件夹，同时放入自定义的 html 文件（自己可以直接读取一些字段值，比如：message、trace等），出异常会自动跳转，不用写 Controller 请求跳转

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` directory. Error pages can either be static HTML (that is, added under any of the static resource directories) or be built by using templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your directory structure would be as follows:

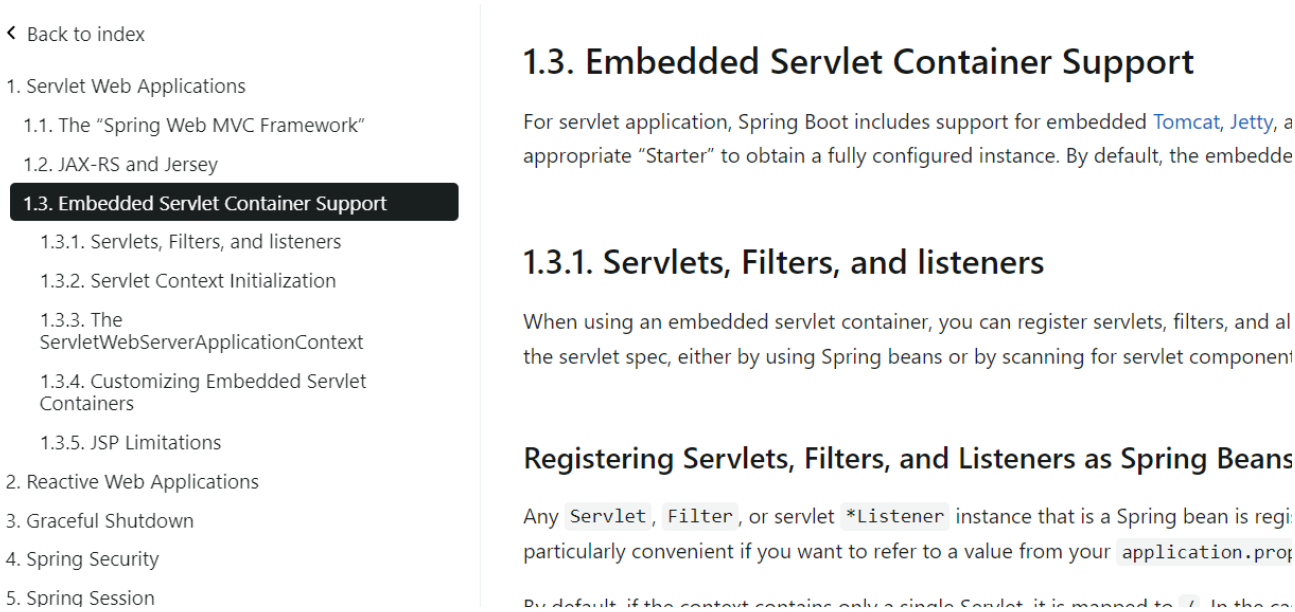
```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

To map all `5xx` errors by using a FreeMarker template, your directory structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.ftlh
             +- <other templates>
```

2.8 Web 原生组件注入（Servlet、Fliter、Listener）

- 官方文档：https://docs.spring.io/spring-boot/docs/current/reference/html/web.html#web.servlet.embedded-container



- 使用方法：
  - 使用原生的 Servlet API

Scanning for Servlets, Filters, and listeners

自定义的类要实现原生接口

When using an embedded container, automatic registration of classes annotated with `@WebServlet`, `@WebFilter`, and `@WebListener` can be enabled by using `@ServletComponentScan`.

写在主类上面，即标有@SpringBootApplication

Tip

`@ServletComponentScan` has no effect in a standalone container, where the container's built-in discovery mechanisms are used instead.

注意：@WebServlet 效果：直接响应，**没有经过Spring的拦截器**

- 使用 RegistrationBean

```
@Configuration
public class MyRegistConfig {

    @Bean
    public ServletRegistrationBean myServlet(){
        MyServlet myServlet = new MyServlet();

        return new ServletRegistrationBean(myServlet, ...urlMappings: "/my","/my02");
    }


    @Bean
    public FilterRegistrationBean myFilter(){

        MyFilter myFilter = new MyFilter();
//        return new FilterRegistrationBean(myFilter,myServlet());
        FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(myFilter);
        filterRegistrationBean.setUrlPatterns(Arrays.asList("/my","/css/*"));
        return filterRegistrationBean;
    }

    @Bean
    public ServletListenerRegistrationBean myListener(){
        MySwervletContextListener mySwervletContextListener = new MySwervletContextListener();
        return new ServletListenerRegistrationBean(mySwervletContextListener);
    }
}
```

发请求

过滤请求、静态资源

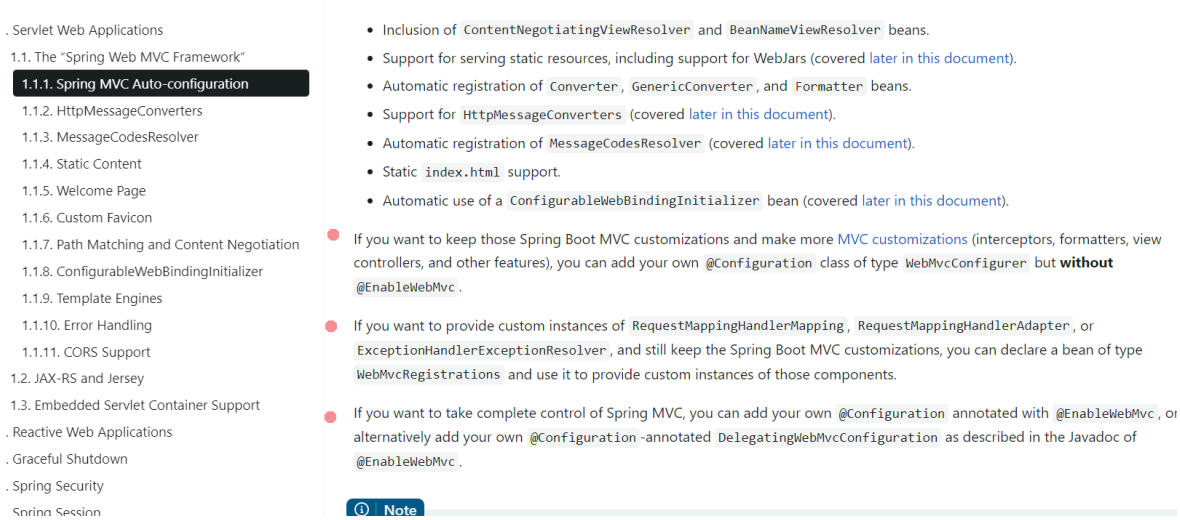
项目监听

注意： @Configuration(proxyBeanMethods = true) 保证单实例

2.9 定制化组件

- 常见方式
  - 修改配置文件
  - @configuration 与 @Bean 结合
  - 绑定配置文件

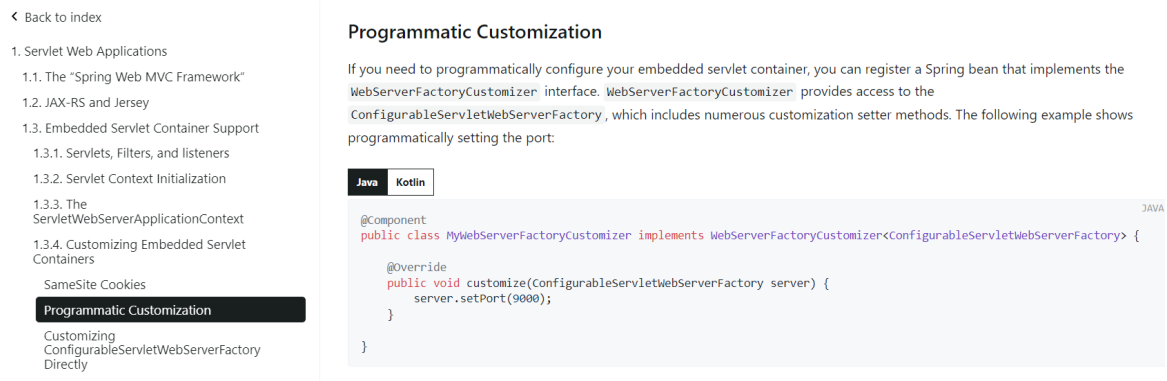
- **web 应用编写一个配置类实现 WebMvcConfigurer 来定制化 web 功能（例如：见2.5拦截器） + @Bean 来向容器中注入组件**
  - 参见官方链接： https://docs.spring.io/spring-boot/docs/current/reference/html/web.html#web.servlet.spring-mvc.auto-configuration



注意：@EnableMvc 表示**全面接管** SpringMVC， 所有东西都要**重写**，要**慎用**

- @EnableWebMvc + WebMvcConfigurer —— @Bean 可以全面接管SpringMVC，所有规则全部自己重新配置； 实现定制和扩展功能
  - 原理
  - 1、WebMvcAutoConfiguration 默认的SpringMVC的自动配置功能类。静态资源、欢迎页.....
  - 2、一旦使用 @EnableWebMvc 、。会 @Import(DelegatingWebMvcConfiguration.**class**)
  - 3、**DelegatingWebMvcConfiguration** 的 作用，只保证SpringMVC最基本的使用
    - 把所有系统中的 WebMvcConfigurer 拿过来。所有功能的定制都是这些 WebMvcConfigurer 合起来一起生效
    - 自动配置了一些非常底层的组件。**RequestMappingHandlerMapping**、这些组件依赖的组件都是从容器中获取
    - **public class** DelegatingWebMvcConfiguration **extends WebMvcConfigurationSupport**
  - 4、**WebMvcAutoConfiguration** 里面的配置要能生效 必须 @ConditionalOnMissingBean(**WebMvcConfigurationSupport.class**)
  - 5、@EnableWebMvc 导致了 **WebMvcAutoConfiguration** **没有生效。**

- XXXCustomizer
  - 官方文档链接： https://docs.spring.io/spring-boot/docs/current/reference/html/web.html#web.servlet.embedded-container.customizing.programmatic



3. 数据访问

3.1 SQL

- **<u>版本说明</u>**

  ```
  MySQL 版本：5.7.31
  ```

- 引入 JDBC 场景，自动导入了相关的自动配置类
  - 底层默认数据源
  - spring - jdbc
  - spring - tx 事务

```
# JDBC 场景
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

  - **<u>缺少了数据驱动</u>**
    - 不默认导入，版本需要自定义选择（**<u>可以不修改，向下兼容</u>**）

```
<!--  引入数据库驱动,有两种修改方式，默认版本是 8.0.29，需要与当前所匹配，-->
 <!-- 方式一 不指定版本，在 properties 标签中指定版本（maven属性就近依赖）-->
 <properties>
        <mysql.version>5.1.49</mysql.version>
 </properties>
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <!-- 方式二直接指定版本如下（maven就近依赖） -->
            <version>5.1.47</version>
        </dependency>
```

- 修改相关配置项（数据源）来改变绑定的 javabean 的值
  - 驱动类名改为： com.mysql.cj.jdbc.Driver， 原 com.mysql.jdbc.Driver **已过时**
  - **在 `mysql-connector-java 5` 以后的版本中(不包括5) 使用的都是 `com.mysql.cj.jdbc.Driver`**

```
spring:
  # 配置数据源
  datasource:
    # 默认类型
    type: com.zaxxer.hikari.HikariDataSource
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ry?
useUnicode=true&characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&useSSL=t
rue&serverTimezone=GMT%2B8
    username: root
    password: 123456789
```

3.1.1 使用 Druid 数据源

- 官方参考网址： https://github.com/alibaba/druid
- 引入场景

```
<!-- 第三方 starter 命名：*-spring-boot-starter-->
<dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>druid-spring-boot-starter</artifactId>
      <version>1.1.17</version>
</dependency>
```

- 修改相关配置项

```yaml
spring:
  # 配置数据源
  datasource:
    url: jdbc:mysql://localhost:3306/ry?
useUnicode=true&characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&useSSL=t
rue&serverTimezone=GMT%2B8
    username: root
    password: 123456789
    driver-class-name: com.mysql.cj.jdbc.Driver

    druid:
      aop-patterns: com.atguigu.admin.*  #监控SpringBean
      filters: stat,wall     # 底层开启功能，stat（sql监控），wall（防火墙）

      stat-view-servlet:   # 配置监控页功能
        enabled: true
        login-username: admin
        login-password: 123456789
        resetEnable: false

      web-stat-filter:  # 监控web
        enabled: true
        urlPattern: /*
        exclusions: '*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*'
```

3.1.2 Mybatis 整合

- Mybatis 官方参考文档链接： https://mybatis.org/mybatis-3/zh/getting-started.html
- 导入场景（第三方）

```
# 开源项目
https://github.com/mybatis/spring-boot-starter/tree/mybatis-spring-boot-2.2.2
```

```xml
<!--        引入 Mybatis 场景-->
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
        </dependency>
```

```
# Mybatis 场景里面映入了 SQL 场景，所以不用引入
# spring-boot-starter-data-jdbc
```

- 快速使用：
  - 导入场景
  - 编写 Mapper 接口，并且要加上注解 @Mapper （**每一个都加**）
    - 也可使用包扫描方式，在主程序（启动）上加注解 **@MapperScan("存Mapper的包名")**
  - 编写对应 Mapper 接口的 sql 映射文件，**sql.xml 中的 id 对应 Mapper 的方法名**
  - 在配置文件（.yml）中配置 Mapper 配置文件（**sql 映射文件**）的位置
  - 在配置文件（.yml）中配置 Mybatis 全局配置文件位置（建议**不要再创建配置文件**，直接在 **mybatis.configuration** 中对相关项进行配置即可）

```yaml
# 配置mybatis规则
mybatis:
#  config-location: classpath:mybatis/mybatis-config.xml
  mapper-locations: classpath:mybatis/mapper/*.xml
  configuration:
    map-underscore-to-camel-case: true

 可以不写全局；配置文件，所有全局配置文件的配置都放在configuration配置项中即可
```

- 相关注解的使用
  - 对于复杂的 sql ， 可以使用编写相关的 sql 映射文件
  - 对于简单的 sql，可以不用编写 sql 映射文件，直接使用注解，直接在相关接口中的方法上加上 @Select("sql 语句") 等注解即可

3.1.3 MyBatisPlus 整合

- MyBatisPlus 官方参考文档链接： https://baomidou.com/
- 场景导入

```xml
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter-test</artifactId>
    <version>3.5.2</version>
</dependency>
```

```
# MybatisPlus 引入了 SQL 场景，同时也引入了 Mybatis 场景
# 和 Mybatis 与 Spring 整合的场景，因此不用引入
# spring-boot-starter-data-jdbc 和 mybatis-spring-boot-starter
```

- 相关注解：
  - pojo 上标 @TableName("表名") ------> 类与对应表进行映射
    - **默认情况**下，不使用该注解，类名（驼峰式）与表名相同
  - 表中**不存在的字段**，在 pojo 中相应的属性上标注解 @TableField(exist = false)，进行查询时候回自动忽略相应字段
- 优点：
  - 对于 Mapper 接口，可以**直接继承 BaseMapper**， 具备了**基本的 CRUD** 功能
  - **mapper-locations 自动配置好的**，有默认值：classpath*:/mapper/**/*.xml（任意包的类路径下的所有mapper文件夹下任意路径下的所有xml都是sql映射文件）， **建议以后sql映射文件，放在 mapper下**
  - 对于 service 接口继承 IService< pojo中的类 > 就拥有了基本的功能，对应的 service 实现类需要继承 ServiceImpl<对应Mapper接口, pojo中的类> 从避免需要实现 service 接口中的众多方法，这样就可以直接调用来使用即可（比如：里面含有分页方法，page() ）

```
// 举例
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper,User> implements
UserService {

}

public interface UserService extends IService<User> {

}
```

3.2 NoSQL

3.2.1 Redis

- 官方中文参考文档： http://redis.cn/documentation.html
- 导入对应场景

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

- 配置相关配置项

```yaml
spring:
  redis:
      host: #  主机地址
      port:  # 段口
      password:  # 密码
      # 默认是 Lettuce，导入 jedis 就可以进行修改了
      # 从而使用对应的模板来对 redis 操作
      client-type:
      # 如果使用的是 jedis 就加下面的配置
      jedis:
        pool:
          max-active: 10
```

- **面试题（Filter 与 Interceptor 区别）：**

```java
/**
 * Filter、Interceptor 几乎拥有相同的功能？
 * 1、Filter是Servlet定义的原生组件。好处，脱离Spring应用也能使用
 * 2、Interceptor是Spring定义的接口。可以使用Spring的自动装配等功能
 *
 */
@Autowired
RedisUrlCountInterceptor redisUrlCountInterceptor;
```

自动注入