

SpringBoot3 全栈指南教程——尚硅谷学习 笔记 2023 年

- [SpringBoot3 全栈指南教程——尚硅谷学习笔记 2023 年](#)
- [一、Spring Boot 3-核心特性](#)
 - [第 1 章 SpringBoot3-快速入门](#)
 - [1.1 简介](#)
 - [1.1.1 前置知识](#)
 - [1.1.2 环境要求](#)
 - [1.1.3 SpringBoot 是什么](#)
 - [1.2 快速体验](#)
 - [1.2.1 开发流程](#)
 - [1.2.1.1 创建项目](#)
 - [1.2.1.2 导入场景](#)
 - [1.2.1.3 主程序](#)
 - [1.2.1.4 业务](#)
 - [1.2.1.5 测试](#)
 - [1.2.1.6 打包](#)
 - [1.2.2 特性小结](#)
 - [1.2.2.1 简化整合](#)
 - [1.2.2.2 简化开发](#)
 - [1.2.2.3 简化配置](#)
 - [1.2.2.4 简化部署](#)
 - [1.2.2.5 简化运维](#)
 - [1.2.2.6 Spring Initializr 创建向导](#)
 - [1.3 应用分析](#)
 - [1.3.1 依赖管理机制](#)
 - [1.3.2 自动配置机制](#)
 - [1.3.2.1 初步理解](#)
 - [1.3.2.2 完整流程](#)
 - [1.3.2.3 如何学好 SpringBoot](#)
 - [1.4 核心技能](#)
 - [1.4.1 常用注解](#)
 - [1.4.1.1 组件注册](#)

- 1.4.1.2 条件注解
 - 1.4.1.3 属性绑定
 - 1.4.2 YAML 配置文件
 - 1.4.2.1. 基本语法
 - 1.4.2.2 示例
 - 1.4.2.3 细节
 - 1.4.2.4. 小技巧 : lombok
 - 1.4.3 日志配置
 - 1.4.3.1 简介
 - 1.4.3.2 日志格式
 - 1.4.3.3 记录日志
 - 1.4.3.4 日志级别
 - 1.4.3.5 日志分组
 - 1.4.3.6 文件输出
 - 1.4.3.7 文件归档与滚动切割
 - 1.4.3.8 自定义配置
 - 1.4.3.9 切换日志组合
 - 1.4.3.10 最佳实战
- 第 2 章 SpringBoot3-Web 开发
 - 2.1 WebMvcAutoConfiguration 原理
 - 2.1.1 生效条件
 - 2.1.2 效果
 - 2.1.3 WebMvcConfigurer 接口
 - 2.1.4 静态资源规则源码
 - 2.1.5 EnableWebMvcConfiguration 源码
 - 2.1.6 为什么 容器中放一个 WebMvcConfigurer 就能配置底层行为
 - 2.1.7 WebMvcConfigurationSupport
 - 2.2 Web 场景
 - 2.2.1 自动配置
 - 2.2.2 默认效果
 - 2.3 静态资源
 - 2.3.1 默认规则
 - 2.3.1.1 静态资源映射
 - 2.3.1.2 静态资源缓存

- 2.3.1.3 欢迎页
 - 2.3.1.4 Favicon
 - 2.3.1.5 缓存实验
 - 2.3.2 自定义静态资源规则
 - 2.3.2.1 配置方式
 - 2.3.2.2 代码方式
- 2.4 路径匹配
 - 2.4.1 Ant 风格路径用法
 - 2.4.2 模式切换
- 2.5 内容协商
 - 2.5.1 多端内容适配
 - 2.5.1.1 默认规则
 - 2.5.1.2 效果演示
 - 2.5.1.3 配置协商规则与支持类型
 - 2.5.2 自定义内容返回
 - 2.5.2.1 增加 yaml 返回支持
 - 2.5.2.2 思考：如何增加其他
 - 2.5.2.3 HttpResponseMessage 的示例写法
 - 2.5.3 内容协商原理 HttpResponseMessage
 - 2.5.3.1 @ResponseBody 由 HttpResponseMessage 处理
 - 2.5.3.2 WebMvcAutoConfiguration 提供几种默认 HttpResponseMessage
- 2.6 模板引擎
 - 2.6.1 Thymeleaf 整合
 - 2.6.2 基础语法
 - 2.6.2.1 核心用法
 - 2.6.2.2 语法示例
 - 2.6.2.3 属性设置
 - 2.6.2.4 遍历
 - 2.6.2.5 判断
 - 2.6.2.6 属性优先级
 - 2.6.2.7 行内写法

- 2.6.2.8 变量选择
 - 2.6.2.9 模板布局
 - 2.6.2.10 devtools
 - 2.7 国际化
 - 2.8 错误处理
 - 2.8.1 默认机制
 - 2.8.2 自定义错误响应
 - 2.8.2.1 自定义 json 响应
 - 2.8.2.2 自定义页面响应
 - 2.8.3 最佳实战
 - 2.9 嵌入式容器
 - 2.9.1 自动配置原理
 - 2.9.2 自定义
 - 2.9.3 最佳实践
 - 2.10 全面接管 SpringMVC
 - 2.10.1 WebMvcAutoConfiguration 到底自动配置了哪些规则
 - 2.10.2 @EnableWebMvc 禁用默认行为
 - 2.10.3 WebMvcConfigurer 功能
 - 2.11 最佳实践
 - 2.11.1 三种方式
 - 2.11.2 两种模式
 - 2.12 Web 新特性
 - 2.12.1 ProblemDetails
 - 2.12.2 函数式 Web
 - 2.12.2.1 场景
 - 2.12.2.2 核心类
 - 2.12.2.3 示例
- 第 3 章 SpringBoot3-数据访问
 - 3.1 创建 SSM 整合项目
 - 3.2 配置数据源
 - 3.3 配置 MyBatis
 - 3.4 CRUD 编写
 - 3.5 自动配置原理
 - 3.6 快速定位生效的配置

- 3.7 扩展：整合其他数据源
 - 3.7.1 Druid 数据源
- 3.8 附录：示例数据库
- 第 4 章 SpringBoot3-基础特性
 - 4.1 SpringApplication
 - 4.1.1 自定义 banner
 - 4.1.2 自定义 SpringApplication
 - 4.1.3 FluentBuilder API
 - 4.2 Profiles
 - 4.2.1 使用
 - 4.2.1.1 指定环境
 - 4.2.1.2 环境激活
 - 4.2.1.3 环境包含
 - 4.2.2 Profile 分组
 - 4.2.3 Profile 配置文件
 - 4.3 外部化配置
 - 4.3.1 配置优先级
 - 4.3.2 外部配置
 - 4.3.3 导入配置
 - 4.3.4 属性占位符
 - 4.4 单元测试 JUnit5
 - 4.4.1 整合
 - 4.4.2 测试
 - 4.4.2.1 组件测试
 - 4.4.2.2 注解
 - 4.4.2.3 断言
 - 4.4.2.4 嵌套测试
 - 4.4.2.5 参数化测试
- 第 5 章 SpringBoot3-核心原理
 - 5.1 事件和监听器
 - 5.1.1 生命周期监听
 - 5.1.1.1 监听器
SpringApplicationRunListener
 - 5.1.1.2 生命周期全流程
 - 5.1.2 事件触发时机

- 5.1.2.1 各种回调监听器
- 5.1.2.2 完整触发流程
- 5.1.2.3 SpringBoot 事件驱动开发
- 5.2 自动配置原理
 - 5.2.1 入门理解
 - 5.2.1.1 自动配置流程
 - 5.2.1.2 SPI 机制
 - 5.2.1.3 功能开关
 - 5.2.2 进阶理解
 - 5.2.2.1 @SpringBootApplication
 - 5.2.2.2 完整启动加载流程
- 5.3 自定义 starter
 - 5.3.1 业务代码
 - 5.3.2 基本抽取
 - 5.3.3 使用@EnableXxx 机制
 - 5.3.4 完全自动配置

一、Spring Boot 3-核心特性

第 1 章 SpringBoot3-快速入门

1.1 简介

1.1.1 前置知识

- Java17
- Spring、SpringMVC、MyBatis
- Maven、IDEA

1.1.2 环境要求

| 环境&工具 | 版本 |
|------------|--------|
| SpringBoot | 3.1.3+ |

| 环境&工具 | 版本 |
|--------------------|-----------|
| IDEA | 2022.3.3+ |
| Java | 17+ |
| Maven | 3.8.1+ |
| Tomcat | 10.1.12+ |
| Servlet | 5.0.0+ |
| GraalVM Community | 22.3+ |
| Native Build Tools | 0.9.19+ |

1.1.3 SpringBoot 是什么

SpringBoot 帮我们简单、快速地创建一个独立的、生产级别的 **Spring 应用**（说明：**SpringBoot 底层是 Spring**）。

大多数 SpringBoot 应用只需要编写少量配置即可快速整合 Spring 平台以及第三方技术。

特性：

- 快速创建 独立 Spring 应用。
 - SSM：导包、配置、启动运行。
- 直接 嵌入 Tomcat、Jetty 或 Undertow（无需部署 war 包）【Servlet 容器】。
 - Linux、Java、Tomcat、MySQL：war 放到 Tomcat 的 webapps 下。
 - jar、Java 环境：java -jar。
- 重点：提供可选的 starter，简化应用整合。
 - 场景启动器（starter）：web、json、邮件、oss（对象存储）、异步、定时任务、缓存.....
 - 导很多包，控制好版本。
 - 为每一种场景准备了一个依赖：**web-starter、mybatis-starter**。
- 重点：按需自动配置 Spring 以及第三方库。
 - 如果这些场景要使用（生效）。这个场景的所有配置都会自动配置好。
 - 约定大于配置：每个场景都有很多默认配置。
 - 自定义：配置文件中修改几项就可以。
- 提供 生产级特性：如监控指标、健康检查、外部化配置等。

- 监控指标、健康检查（k8s）、外部化配置。
- 无代码生成、无 xml。

总结：简化开发，简化配置，简化整合，简化部署，简化监控，简化运维。

1.2 快速体验

场景：浏览器发送 /hello 请求，返回"Hello, Spring Boot 3!"

1.2.1 开发流程

1.2.1.1 创建项目

maven 项目

```
<!-- 所有 Spring Boot 项目都必须继承自 spring-boot-starter-parent -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.1.3</version>
</parent>
```

1.2.1.2 导入场景

场景启动器

```
<dependencies>
  <!-- Web 开发的场景启动器 -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```


1.2.1.3 主程序

```
package com.myxh.springboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * @author MYXH
 * @date 2023/9/11
 * @description 启动 SpringBoot 项目的主入口程序
 */
// 这是一个 SpringBoot 应用
@SpringBootApplication
public class MainApplication
{
    public static void main(String[] args)
    {
        SpringApplication.run(MainApplication.class, args);
    }
}
```

1.2.1.4 业务

```
package com.myxh.springboot.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * @author MYXH
 * @date 2023/9/11
 */
@RestController
public class HelloController
{
    @GetMapping("/hello")
    public String hello()
    {
        return "Hello, Spring Boot 3!";
    }
}
```

1.2.1.5 测试

默认启动访问：localhost:8080

1.2.1.6 打包

```
<build>
  <plugins>
    <!-- SpringBoot 应用打包插件-->
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

`mvn clean package` 把项目打成可执行的 jar 包。

`java -jar boot3-01-demo-1.0-SNAPSHOT.jar` 启动项目。

1.2.2 特性小结

1.2.2.1 简化整合

导入相关的场景，拥有相关的功能的场景启动器。

默认支持的所有场景：<https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.build-systems.starters>

- 官方提供的场景：命名为 `spring-boot-starter-*`。
- 第三方提供场景：命名为 `*-spring-boot-starter`。

场景一导入，万物皆就绪。

1.2.2.2 简化开发

无需编写任何配置，直接开发业务。

1.2.2.3 简化配置

`application.properties`：

- 集中式管理配置，只需要修改这个文件就行。

- 配置基本都有默认值。
- 能写的所有配置都在：<https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties>

1.2.2.4 简化部署

打包为可执行的 jar 包。

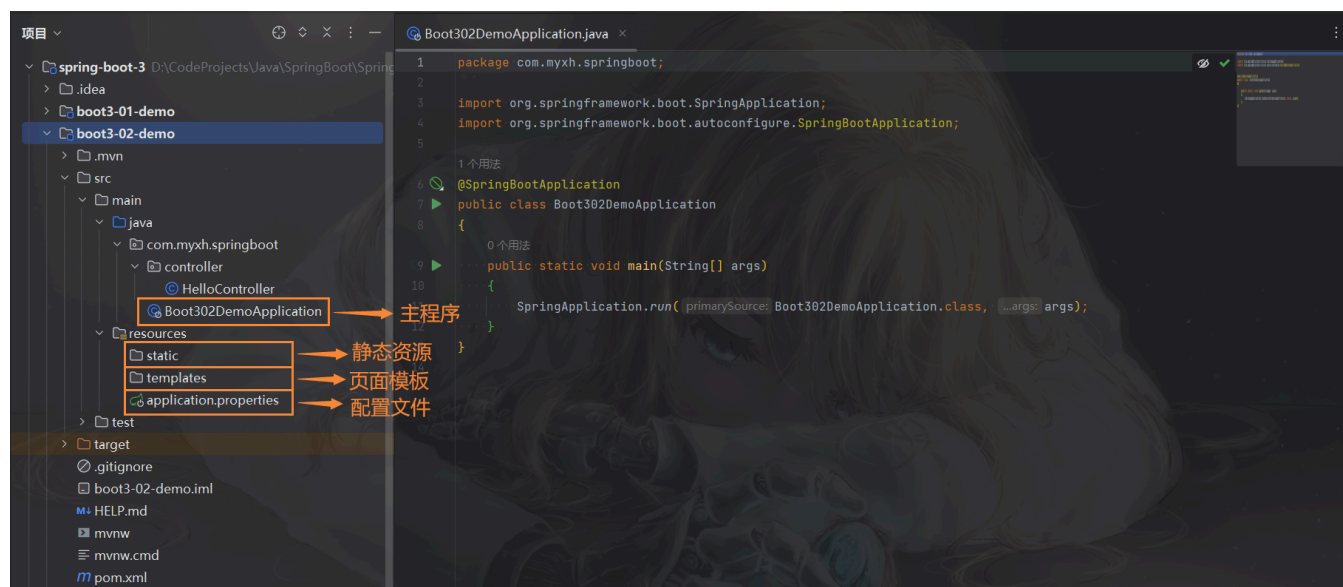
Linux 服务器上有 Java 环境。

1.2.2.5 简化运维

修改配置（外部放一个 application.properties 文件）、监控、健康检查.....

1.2.2.6 Spring Initializr 创建向导

一键创建好整个项目结构。



1.3 应用分析

1.3.1 依赖管理机制

思考：

1、为什么导入 `starter-web` 所有相关依赖都导入进来？

- 开发什么场景，导入什么**场景启动器**。
- **maven 依赖传递原则**。A-B-C：A 就拥有 B 和 C。

- 导入场景启动器，场景启动器自动把这个场景的所有核心依赖全部导入进来。

2、为什么版本号都不用写？

- 每个 boot 项目都有一个父项目 `spring-boot-starter-parent`。
- parent 的父项目是 `spring-boot-dependencies`。
- 父项目版本仲裁中心，把所有常见的 jar 的依赖版本都声明好了。
- 比如：`mysql-connector-j`。

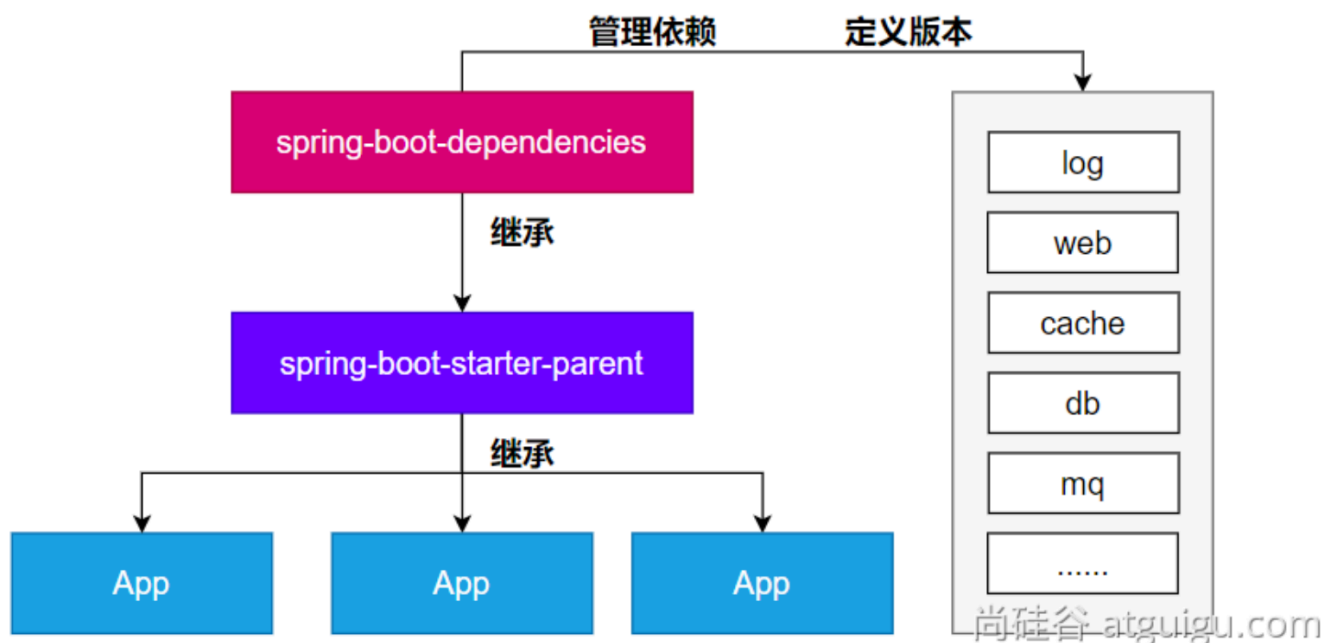
3、自定义版本号。

- 利用 maven 的就近原则。
 - 直接在当前项目 `properties` 标签中声明父项目用的版本属性的 key。
 - 直接在**导入依赖的时候**声明版本。

4、第三方的 jar 包。

- boot 父项目没有管理的需要自行声明好。

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.2.16</version>
</dependency>
```



1.3.2 自动配置机制

1.3.2.1 初步理解

- 自动配置的 Tomcat、SpringMVC 等。
 - **导入场景**，容器中就会自动配置好这个场景的核心组件。
 - 以前：DispatcherServlet、ViewResolver、CharacterEncodingFilter...
 - 现在：自动配置好的这些组件。
 - 验证：容器中有了什么**组件**，就具有什么功能。

```

package com.myxh.springboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * @author MYXH
 * @date 2023/9/11
 * @description 启动 SpringBoot 项目的主入口程序
 */
// 主程序: com.myxh.springboot
// @SpringBootConfiguration
// @EnableAutoConfiguration
// @ComponentScan("com.myxh.springboot")
// @SpringBootApplication(scanBasePackages = "com.myxh.springboot")

// 这是一个 SpringBoot 应用
@SpringBootApplication
public class MainApplication
{
    public static void main(String[] args)
    {
        // Java10: 局部变量类型的自动推断
        var ioc = SpringApplication.run(MainApplication.class, args);

        // 1、获取容器中所有组件的名字
        String[] beanNames = ioc.getBeanDefinitionNames();

        // 2、挨个遍历
        /*
        dispatcherServlet、beanNameViewResolver、characterEncodingFilter、mu
        SpringBoot 把以前配置的核心组件现在都给自动配置好了
        */
        for (String beanName : beanNames)
        {
            System.out.println("beanName = " + beanName);
        }
    }
}

```

- 默认的包扫描规则。
 - `@SpringBootApplication` 标注的类就是主程序类。
 - **SpringBoot** 只会扫描主程序所在的包及其下面的子包，自动的

component-scan 功能。

- 自定义扫描路径。
 - `@SpringBootApplication(scanBasePackages = "com.myxh.springboot")`
 - `@ComponentScan("com.myxh.springboot")` 直接指定扫描的路径。
- 配置默认值。
 - 配置文件的所有配置项是和某个类的对象值进行一一绑定的。
 - 绑定了配置文件中每一项值的类：属性类。
 - 比如：
 - `ServerProperties` 绑定了所有 Tomcat 服务器有关的配置。
 - `MultipartProperties` 绑定了所有文件上传相关的配置。
 - 参照官方文档 <https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties.server>, 或者参照绑定的属性类。
- 按需加载自动配置。
 - 导入场景 `spring-boot-starter-web`。
 - 场景启动器除了会导入相关功能依赖，导入一个 `spring-boot-starter`，是所有 starter 的 starter，基础核心 starter。
 - `spring-boot-starter` 导入了一个包 `spring-boot-autoconfigure`。包里面都是各种场景的 `AutoConfiguration` 自动配置类。
 - 虽然全场景的自动配置都在 `spring-boot-autoconfigure` 这个包，但是不是全都开启的。
 - 导入哪个场景就开启哪个自动配置。

总结：导入场景启动器、触发 `spring-boot-autoconfigure` 这个包的自动配置生效、容器中就会具有相关场景的功能。

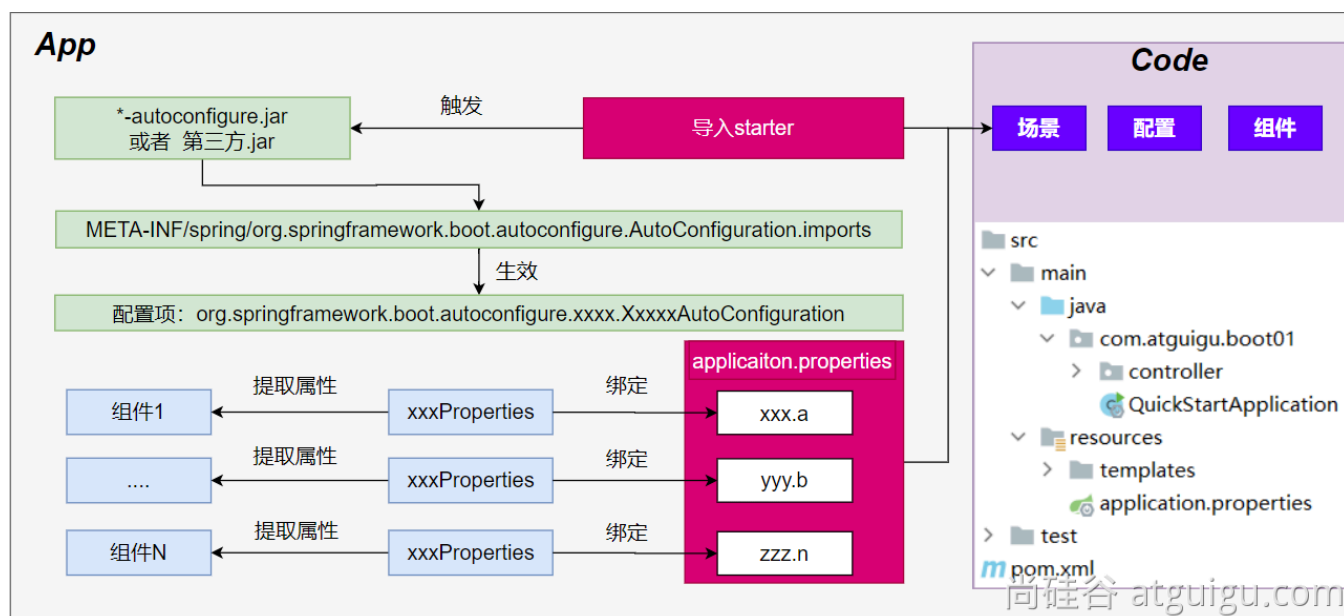
1.3.2.2 完整流程

思考：

1、SpringBoot 怎么实现导一个 starter、写一些简单配置，应用就能跑起来，无需关心整合。

2、为什么 Tomcat 的端口号可以配置在 **application.properties** 中，并且 Tomcat 能启动成功？

3、导入场景后哪些**自动配置**能生效？



自动配置流程细节梳理：

1、导入 `starter-web`：导入了 web 开发场景。

- 1、场景启动器导入了相关场景的所有依赖：`starter-json`、`starter-tomcat`、`springmvc`。
- 2、每个场景启动器都引入了一个 `spring-boot-starter`，核心场景启动器。
- 3、**核心场景启动器**引入了 `spring-boot-autoconfigure` 包。
- 4、`spring-boot-autoconfigure` 里面囊括了所有场景的所有配置。
- 5、只要这个包下的所有类都能生效，那么相当于 SpringBoot 官方写好的整合功能就生效了。
- 6、SpringBoot 默认却扫描不到 `spring-boot-autoconfigure` 下写好的所有配置类。（这些配置类做了整合操作），**默认只扫描主程序所在的包。**

2、主程序：`@SpringBootApplication`。

- 1、`@SpringBootApplication` 由三个注解组成 `@SpringBootConfiguration`、`@EnableAutoConfiguration`、`@ComponentScan`。
- 2、SpringBoot 默认只能扫描自己主程序所在的包及其下面的子包，扫描不到 `spring-boot-autoconfigure` 包中官方写好的配置类。

- 3、`@EnableAutoConfiguration` : SpringBoot 开启自动配置的核心。
 - ① 是由 `@Import(AutoConfigurationImportSelector.class)` 提供功能：批量给容器中导入组件。
 - ② SpringBoot 启动会默认加载 146 个配置类。
 - ③ 这 146 个配置类来自于 `spring-boot-autoconfigure` 下 `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 文件指定的。
 - ④ 项目启动的时候利用 `@Import` 批量导入组件机制把 `autoconfigure` 包下的 146 `xxxAutoConfiguration` 类导入进来（自动配置类）。
- 4、按需生效：
 - 虽然导入了 146 个自动配置，并不是这 146 个自动配置类都能生效。
 - 每一个自动配置类，都有条件注解 `@ConditionalOnXxx`，只有条件成立，才能生效。

3、`xxxAutoConfiguration` 自动配置类。

- 1、给容器中使用 `@Bean` 放一堆组件。
- 2、每个自动配置类都可能都有这个注解 `@EnableConfigurationProperties(ServerProperties.class)`，用来把配置文件中配的指定前缀的属性值封装到 `xxxProperties` 属性类中。
- 3、以 Tomcat 为例：把服务器的所有配置都是以 `server` 开头的。配置都封装到了属性类中。
- 4、给容器中放的所有组件的一些核心参数，都来自于 `xxxProperties`。`xxxProperties` 都是和配置文件绑定。
- 5、只需要改配置文件的值，核心组件的底层参数都能修改。

4、写业务，全程无需关心各种整合（底层这些整合写好了，而且也生效了）。

核心流程总结：

1、导入 `starter`，就会导入 `autoconfigure` 包。

2、`autoconfigure` 包里面有一个文件

`META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`，里面指定的所有启动要加载的自动配置类。

3、`@EnableAutoConfiguration` 会自动的把上面文件里面写的所有自动配置类都导入进来。
`xxxAutoConfiguration` 是有条件注解进行按需加载。

4、`xxxAutoConfiguration` 给容器中导入一堆组件，组件都是从 `xxxProperties` 中提取属性值。

5、`xxxProperties` 又是和**配置文件**进行了绑定。

效果：导入 `starter`、修改配置文件，就能修改底层行为。

1.3.2.3 如何学好 SpringBoot

框架的框架、底层基于 Spring。能调整每一个场景的底层行为。100%项目一定会用到底层自定义。

摄影：

- 傻瓜：自动配置好。
- **单反**：焦距、光圈、快门、感光度.....
- 傻瓜+**单反**：

1、理解**自动配置原理**。

- ① **导入 starter -> 生效 xxxAutoConfiguration -> 组件 -> xxxProperties -> 配置文件。**

2、理解**其他框架底层**。

- ① **拦截器。**

3、可以随时**定制化任何组件**。

- ① **配置文件。**
- ② **自定义组件。**

普通开发：导入 `starter`，Controller、Service、Mapper、偶尔修改配置文件。

高级开发：自定义组件、自定义配置、自定义 starter。

核心：

- 这个场景自动配置导入了哪些组件，能不能 Autowired 进来使用。
- 能不能通过修改配置改变组件的一些默认参数。
- 需不需要自己完全定义这个组件。

- 场景定制化。

最佳实战：

- 选场景，导入到项目。
 - 官方：starter。
 - 第三方：去仓库搜。
- 写配置，改配置文件关键项。
 - 数据库参数（连接地址、账号密码.....）。
- 分析这个场景导入了哪些能用的组件。
 - 自动装配这些组件进行后续使用。
 - 不满意 SpringBoot 提供的自动配好的默认组件。
 - 定制化。
 - 改配置。
 - 自定义组件。

整合 redis：

- 选场景：`spring-boot-starter-data-redis`。
 - 场景 `AutoConfiguration` 就是这个场景的自动配置类。
- 写配置：
 - 分析到这个场景的自动配置类开启了哪些属性绑定关系。
 - `@EnableConfigurationProperties(RedisProperties.class)`。
 - 修改 redis 相关的配置。
- 分析组件：
 - 分析到 `RedisAutoConfiguration` 给容器中放了 `StringRedisTemplate`。
 - 给业务代码中自动装配 `StringRedisTemplate`。
- 定制化：
 - 修改配置文件。
 - 自定义组件，自己给容器中放一个 `StringRedisTemplate`。

1.4 核心技能

1.4.1 常用注解

SpringBoot 摒弃 XML 配置方式，改为全注解驱动。

1.4.1.1 组件注册

@Configuration、@SpringBootConfiguration

@Bean、@Scope

@Controller、@Service、@Repository、@Component

@Import

@ComponentScan

步骤：

- 1、**@Configuration** 编写一个配置类。
- 2、在配置类中，自定义方法给容器中注册组件。配合**@Bean**。
- 3、或使用**@Import** 导入第三方的组件。

1.4.1.2 条件注解

如果注解指定的条件成立，则触发指定行为。

@ConditionalOnXxx

@ConditionalOnClass：如果类路径中存在这个类，则触发指定行为。

@ConditionalOnMissingClass：如果类路径中不存在这个类，则触发指定行为。

@ConditionalOnBean：如果容器中存在这个 Bean（组件），则触发指定行为。

@ConditionalOnMissingBean：如果容器中不存在这个 Bean（组件），则触发指定行为。

场景：

- 如果存在 **FastsqlException** 这个类，给容器中放一个 **Cat** 组件，命名 cat1。
- 否则，就给容器中放一个 **Dog** 组件，命名 dog1。
- 如果系统中有 **dog1** 这个组件，就给容器中放一个 User 组件，名 zhangsan。
- 否则，就放一个 User，名叫 lisi。

@ConditionalOnBean（value=组件类型，name=组件名字）：判断容器中是否有这个类型的组件，并且名字是指定的值。

@ConditionalOnRepositoryType (org.springframework.boot.autoconfigure.data)

@ConditionalOnDefaultWebSecurity (org.springframework.boot.autoconfigure.security)

@ConditionalOnSingleCandidate (org.springframework.boot.autoconfigure.condition)

@ConditionalOnWebApplication (org.springframework.boot.autoconfigure.condition)

@ConditionalOnWarDeployment (org.springframework.boot.autoconfigure.condition)

@ConditionalOnJndi (org.springframework.boot.autoconfigure.condition)

@ConditionalOnResource (org.springframework.boot.autoconfigure.condition)

@ConditionalOnExpression (org.springframework.boot.autoconfigure.condition)

@ConditionalOnClass (org.springframework.boot.autoconfigure.condition)

@ConditionalOnEnabledResourceChain ([org.springframework.boot.autoconfigure.web](https://docs.spring.io/spring-boot/docs/current/api/org.springframework.boot.autoconfigure.web/))

@ConditionalOnMissingClass (org.springframework.boot.autoconfigure.condition)

@ConditionalOnNotWebApplication (org.springframework.boot.autoconfigure.condition)

@ConditionalOnProperty (org.springframework.boot.autoconfigure.condition)

@ConditionalOnCloudPlatform (org.springframework.boot.autoconfigure.condition)

@ConditionalOnBean (org.springframework.boot.autoconfigure.condition)

@ConditionalOnMissingBean (org.springframework.boot.autoconfigure.condition)

@ConditionalOnMissingFilterBean (org.springframework.boot.autoconfigure.web.servlet)

@Profile (org.springframework.context.annotation)

@ConditionalOnInitializedRestarter (org.springframework.boot.devtools.restart)

@ConditionalOnGraphQLSchema (org.springframework.boot.autoconfigure.graphql)

@ConditionalOnJava (org.springframework.boot.autoconfigure.condition)

1.4.1.3 属性绑定

@ConfigurationProperties : 声明组件的属性和配置文件哪些前缀开始项进行绑定。

@EnableConfigurationProperties : 快速注册注解 :

- **场景** : SpringBoot 默认只扫描自己主程序所在的包。如果导入第三方包, 即使组件上标注了 **@Component**、**@ConfigurationProperties** 注解也没用。因为组件都扫描不进来, 此时使用这个注解就可以快速进行属性绑定并把组件注册进容器。

将容器中任意**组件 (Bean)** 的**属性值**和**配置文件的配置项的值**进行绑定。

- 1、给容器中注册组件 (**@Component**、**@Bean**) 。
- 2、使用 **@ConfigurationProperties** 声明组件和配置文件的哪些配置项进行绑定。

1.4.2 YAML 配置文件

痛点 : SpringBoot 集中化管理配置, **application.properties**。

问题 : 配置多以后难阅读和修改, **层级结构辨识度不高**。

YAML 是 “YAML Ain’t a Markup Language” (YAML 不是一种标记语言)。在开发的这种语言时, YAML 的意思其实是 : “Yet Another Markup Language” (是另一种标记语言) 。

- 设计目标, 就是**方便人类读写**。
- **层次分明**, 更适合做配置文件。

使用 **.yaml** 或 **.yml** 作为文件后缀。

1.4.2.1. 基本语法

- 大小写敏感。
- 使用**缩进**表示**层级**关系, **k: v**, 使用空格分割 **k, v**。
- 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。
- **#** 表示**注释**, 从这个字符一直到行尾, 都会被解析器忽略。

支持的写法 :

- **对象** : **键值对**的集合, 例如 : 映射 (map)、哈希 (hash)、字典 (dictionary) 。

- **数组**：一组按次序排列的值，例如：序列（sequence）、列表（list）。
- **纯量**：单个的、不可再分的值，例如：字符串、数字、bool、日期。

1.4.2.2 示例

```
package com.myxh.springboot.bean;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.List;
import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/11
 */
@Component
// 和配置文件 person 前缀的所有配置进行绑定
@ConfigurationProperties(prefix = "person")
// 自动生成无参构造器
@NoArgsConstructor
// 自动生成全参构造器
@AllArgsConstructor
// 自动生成 JavaBean 属性的 getter/setter
@Data
public class Person
{
    private String name;
    private Integer age;
    private Date birthDay;
    private Boolean like;
    // 嵌套对象
    private Child child;
    // 数组（里面是对象）
    private List<Dog> dogs;
    // Map
    private Map<String, Cat> cats;
}
```

```
package com.myxh.springboot.bean;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.List;

/**
 * @author MYXH
 * @date 2023/9/11
 */
@Component
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Child
{
    private String name;
    private Integer age;
    private Date birthDay;
    // 数组
    private List<String> text;
}
```



```

package com.myxh.springboot.bean;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.stereotype.Component;

/**
 * @author MYXH
 * @date 2023/9/11
 */
@Component
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Dog
{
    private String name;
    private Integer age;
}

```

```

package com.myxh.springboot.bean;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.stereotype.Component;

/**
 * @author MYXH
 * @date 2023/9/11
 */
@Component
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Cat
{
    private String name;
    private Integer age;
}

```

properties 表示法。

```
server.port=8080

spring.servlet.multipart.max-file-size=10MB

# 配置 Redis
spring.data.redis.host=localhost
spring.data.redis.port=6379

# properties 表示复杂对象
person.name=张三
person.age=35
person.birthDay=1988/01/01 00:00:00
person.like=true
person.child.name=李四
person.child.age=12
person.child.birthDay=2011/01/01
person.child.text[0]=hello
person.child.text[1]=world
person.dogs[0].name=小黑
person.dogs[0].age=2
person.dogs[1].name=小白
person.dogs[1].age=1
person.cats.cat1.name=小蓝
person.cats.cat1.age=2
person.cats.cat2.name=小灰
person.cats.cat2.age=1
```

yaml 表示法。

1、k: v, k v 之前是空格区分
2、属性有层级关系，使用下一行，空两个空格
3、左侧对齐的代表同一层级的属性

server:

port: 8080

port: 8081

spring:

servlet:

multipart:

max-file-size: 10MB

配置 Redis

data:

redis:

host: localhost

port: 6379

下边是一个单独文档

yaml 表示复杂对象

person:

name: 张三

age: 35

birth-day: 1988/01/01 00:00:00

like: true

child:

name: 李四

age: 12

birth-day: 2011/01/01

text: ["he\nllo", 'wor\nld']

text:

- "he\nllo"

- 'wor\nld'

- |

cats:

cat1:

name: 小蓝

age: 2

对象也可用 {} 表示

```

    cat2: {name: 小灰,age: 1}
- >
  cats:
    cat1:
      name: 小蓝
      age: 2

# 对象也可用 {} 表示
  cat2: {name: 小灰,age: 1}

dogs:
# 数组也可用 - 表示
- name: 小黑
  age: 2
- name: 小白
  age: 1

cats:
  cat1:
    name: 小蓝
    age: 2

# 对象也可用 {} 表示
  cat2: { name: 小灰, age: 1 }

```

1.4.2.3 细节

- birthDay 推荐写为 birth-day。
- 文本：
 - 单引号不会转义【\n 则为普通字符串显示】。
 - 双引号会转义【\n 会显示为换行符】。
- 大文本：
 - | 开头，大文本写在下层，保留文本格式，换行符正确显示。
 - > 开头，大文本写在下层，折叠换行符。

多文档合并：

- 使用 \-\-\- 可以把多个 yaml 文档合并在一个文档中，每个文档区依然认为内容独立。

1.4.2.4. 小技巧 : lombok

简化 JavaBean 开发。自动生成构造器、getter/setter、自动生成 Builder 模式等。

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

使用 `@Data` 等注解。

1.4.3 日志配置

规范：项目开发不要编写 `System.out.println()`，应该用日志记录信息。

| 日志门面 | 日志实现 |
|--|---------------------------|
| JCL (Jakarta Commons Logging) | Log4j |
| SLF4j (Simple Logging Facade for Java) | JUL (java.util.logging) |
| jboss-logging | Log4j2 |
| | Logback |

尚硅谷 atguigu.com

1.4.3.1 简介

1、Spring 使用 `commons-logging` 作为内部日志，但底层日志实现是开放的。可对接其他日志框架。

- ① spring5 及以后 `commons-logging` 被 spring 直接自己实现了。

2、支持 `jul`，`log4j2`，`logback`。SpringBoot 提供了默认的控制台输出配置，也可以配置输出为文件。

3、`logback` 是默认使用的。

4、虽然日志框架很多，但是不用担心，使用 SpringBoot 的默认配置就能工作的很好。

SpringBoot 怎么把日志默认配置好的。

1、每个 `starter` 场景，都会导入一个核心场景 `spring-boot-starter`。

- 2、核心场景引入了日志的所用功能 `spring-boot-starter-logging`。
- 3、默认使用了 `logback + slf4j` 组合作为默认底层日志。
- 4、日志是系统一启动就要用，`xxxAutoConfiguration` 是系统启动好了以后放好的组件，后来用的。
- 5、日志是利用**监听器机制**配置好的。`ApplicationListener`。
- 6、日志所有的配置都可以通过修改配置文件实现。以 `logging` 开始的所有配置。

1.4.3.2 日志格式

```
2023-09-14 20:24:43.709 INFO 96528 --- [main] o.s.b.w.e.t.TomcatWebServer : Tomcat i
2023-09-14 20:24:43.712 INFO 96528 --- [main] o.a.c.c.AprLifecycleListener : Loaded A
```

默认输出格式：

- 时间和日期：毫秒级精度。
- 日志级别：`ERROR`，`WARN`，`INFO`，`DEBUG`，`TRACE`。
- 进程 ID。
- ---：消息分割符。
- 线程名：使用[]包含。
- Logger 名：通常是产生日志的**类名**。
- 消息：日志记录的内容。

注意：`logback` 没有 `FATAL` 级别，对应的是 `ERROR`。

默认值：参照：`spring-boot` 包 `additional-spring-configuration-metadata.json` 文件。

默认输出格式

值

：`%clr(%d${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd'T'HH:mm:ss.SSSXXX}){faint} %clr(${LOG_LEVEL_PATTERN:-%5p}){faint} %clr(%-15.15s) %m%n`

可修改为：`%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level [%thread] %logger{15} ==> %msg%n`。

1.4.3.3 记录日志

```
Logger logger = LoggerFactory.getLogger(getClass());
```

或者使用 Lombok 的 `@Slf4j` 注解。

1.4.3.4 日志级别

- 由低到高：ALL, TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF。
 - 只会打印指定级别及以上级别的日志。
 - ALL：打印所有日志。
 - TRACE：追踪框架详细流程日志，一般不使用。
 - DEBUG：开发调试细节日志。
 - INFO：关键、感兴趣信息日志。
 - WARN：警告但不是错误的信息日志，比如：版本过时。
 - ERROR：业务错误日志，比如出现各种异常。
 - FATAL：致命错误日志，比如 jvm 系统崩溃。
 - OFF：关闭所有日志记录。
- 不指定级别的所有类，都使用 root 指定的级别作为默认级别。
- SpringBoot 日志默认级别是 INFO。

1、在 `application.properties/yaml` 中配置 `logging.level.<logger-name>=<level>` 指定日志级别。

2、`level` 可取值范围：TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF，定义在 `LogLevel` 类中。

3、root 的 `logger-name` 叫 `root`，可以配置 `logging.level.root=warn`，代表所有未指定日志级别都使用 root 的 warn 级别。

1.4.3.5 日志分组

比较有用的技巧是：

将相关的 `logger` 分组在一起，统一配置。SpringBoot 也支持。比如：Tomcat 相关的日志统一设置。

```
logging.group.tomcat=org.apache.catalina,org.apache.coyote,org.apache.tomcat
logging.level.tomcat=trace
```

SpringBoot 预定义两个组。

| Name | Loggers |
|------|--|
| web | <code>org.springframework.core.codec</code> , <code>org.springframework.http</code> , <code>org.springframework.web</code> , <code>org.springframework.boot.actuate.endpoint.web</code> , <code>org.springframework.boot.web.servlet.ServletContextInitializerBeans</code> |
| sql | <code>org.springframework.jdbc.core</code> , <code>org.hibernate.SQL</code> , <code>org.jooq.tools.LoggerListener</code> |

1.4.3.6 文件输出

SpringBoot 默认只把日志写在控制台，如果想额外记录到文件，可以在 `application.properties` 中添加 `logging.file.name` 或 `logging.file.path` 配置项。

| <code>logging.file.name</code> | <code>logging.file.path</code> | 示例 | 效果 |
|--------------------------------|--------------------------------|---------------------|---------------------------------------|
| 未指定 | 未指定 | 无 | 仅控制台输出。 |
| 指定 | 未指定 | <code>my.log</code> | 写入指定文件。可以加路径。 |
| 未指定 | 指定 | <code>./log</code> | 写入指定目录，文件名为 <code>spring.log</code> 。 |
| 指定 | 指定 | 无 | 以 <code>logging.file.name</code> 为准。 |

1.4.3.7 文件归档与滚动切割

归档：每天的日志单独存到一个文档中。

切割：每个文件 10MB，超过大小切割成另外一个文件。

- 1、每天的日志应该独立分割出来存档。如果使用 `logback`（SpringBoot 默认整合），可以通过 `application.properties/yaml` 文件指定日志滚动规则。
- 2、如果是其他日志系统，需要自行配置（添加 `log4j2.xml` 或 `log4j2-spring.xml`）。
- 3、支持的滚动规则设置如下。

| 配置项 | 描述 |
|---|---|
| <code>logging.logback.rollingpolicy.file-name-pattern</code> | 日志存档的文件名格式（默认值： <code>\${LOG_FILE_PATTERN}</code> ） |
| <code>logging.logback.rollingpolicy.clean-history-on-start</code> | 应用启动时是否清除以前存档（默认值： <code>false</code> ） |
| <code>logging.logback.rollingpolicy.max-file-size</code> | 存档前，每个日志文件的最大大小（默认值： <code>10MB</code> ） |
| <code>logging.logback.rollingpolicy.total-size-cap</code> | 日志文件被删除之前，可以容纳的最大大小（默认值： <code>1GB</code> ）。如果磁盘存储超过 1GB 日志后就会删除旧日志文件 |
| <code>logging.logback.rollingpolicy.max-history</code> | 日志文件保存的最大天数(默认值： <code>7</code>)。 |

1.4.3.8 自定义配置

通常配置 `application.properties` 就够了。当然也可以自定义。比如：

| 日志系统 | 自定义 |
|-------------------------|---|
| Logback | <code>logback-spring.xml</code> , <code>logback-spring.groovy</code> , <code>logback.xml</code> , <code>logback.groovy</code> |
| Log4j2 | <code>log4j2-spring.xml</code> or <code>log4j2.xml</code> |
| JDK (Java Util Logging) | <code>logging.properties</code> |

如果可能，建议在日志配置中使用 `-spring` 变量（例如，`logback-spring.xml` 而不是 `logback.xml`）。如果使用标准配置文件，spring 无法完全控制日志初始化。

最佳实战：自己要写配置，配置文件名加上 `xxx-spring.xml`。

1.4.3.9 切换日志组合

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- 如果第三方框架使用了其他日志框架，如 jul，可以排除掉这个框架的默认日志 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

log4j2 支持 yaml 和 json 格式的配置文件。

| 格式 | 依赖 | 文件名 |
|------|--|-----------------------------|
| YAML | com.fasterxml.jackson.core:jackson-databind、com.fasterxml.jackson.dataformat:jackson-dataformat-yaml | log4j2.yaml 或 log4j2.yml |
| JSON | com.fasterxml.jackson.core:jackson-databind | log4j2.json 或 log4j2.jsn |

1.4.3.10 最佳实战

- 1、导入任何第三方框架，先排除它的日志包，因为 Boot 底层控制好了日志。
- 2、修改 `application.properties` 配置文件，就可以调整日志的所有行为。如果不够，可以编写日志框架自己的配置文件放在类路径下就行，比如 `logback-spring.xml`，`log4j2-spring.xml`。
- 3、如需对接**专业日志系统**，也只需要把 logback 记录的日志灌倒 **kafka** 之类的中间件，这和 SpringBoot 没关系，都是日志框架自己的配置，**修改配置文件即可**。

4、业务中使用 `slf4j-api` 记录日志，不要再用 `System.out.println()` 了

第 2 章 SpringBoot3-Web 开发

SpringBoot 的 Web 开发能力，由 **SpringMVC** 提供。

2.1 WebMvcAutoConfiguration 原理

2.1.1 生效条件

```
// 在这些自动配置之后
@Configuration(after = { DispatcherServletAutoConfiguration.class,
    TaskExecutionAutoConfiguration.class,
    ValidationAutoConfiguration.class })
// 如果是 Web 应用就生效，类型有 SERVLET、REACTIVE（响应式 Web）
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })
// 容器中没有这个 Bean，才生效，默认就是没有
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
// 优先级
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@ImportRuntimeHints(WebResourcesRuntimeHints.class)
public class WebMvcAutoConfiguration
{

}
```

2.1.2 效果

1、放了两个 Filter：

- ① `HiddenHttpMethodFilter`：页面表单提交 Rest 请求（GET、POST、PUT、DELETE）。
- ② `FormContentFilter`：表单内容 Filter，GET（数据放 URL 后面）、POST（数据放请求体）请求可以携带数据，PUT、DELETE 的请求体数据会被忽略。

2、给容器中放了 `WebMvcConfigurer` 组件：给 SpringMVC 添加各种定制功能。

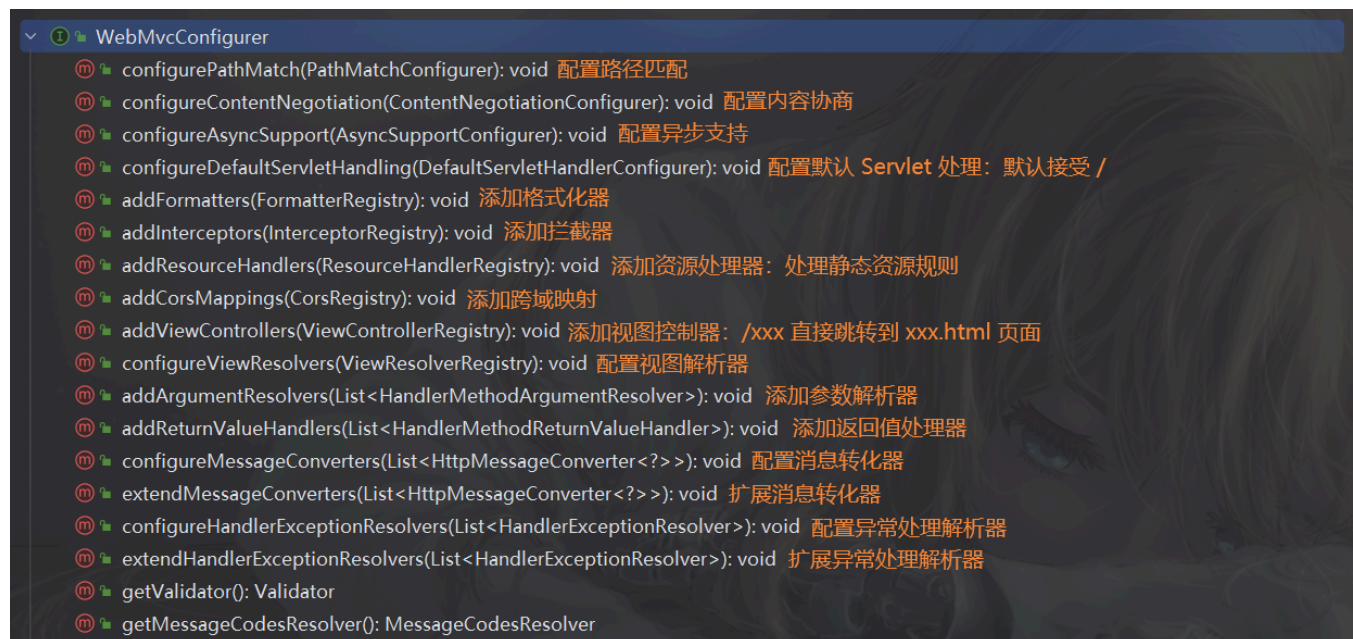
- ① 所有的功能最终会和配置文件进行绑定。
- ② `WebMvcProperties`：`spring.mvc` 配置文件。

- ③ WebProperties : `spring.web` 配置文件。

```
@Configuration(proxyBeanMethods = false)
// 额外导入了其他配置
@Import(EnableWebMvcConfiguration.class)
@EnableConfigurationProperties({ WebMvcProperties.class, WebProperties.class })
@Order(0)
public static class WebMvcAutoConfigurationAdapter implements WebMvcConfigurer, ServletContextAware {
}
}
```

2.1.3 WebMvcConfigurer 接口

提供了配置 SpringMVC 底层的所有组件入口。



2.1.4 静态资源规则源码

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry)
{
    if (!this.resourceProperties.isAddMappings())
    {
        logger.debug("Default resource handling disabled");

        return;
    }

    addResourceHandler(registry, this.mvcProperties.getWebjarsPathPattern(),
        "classpath:/META-INF/resources/webjars/");

    addResourceHandler(registry, this.mvcProperties.getStaticPathPattern(),
        (registration) -> {
            registration.addResourceLocations(this.resourceProperties.getStaticLocations());

            if (this.servletContext != null)
            {
                ServletContextResource resource = new ServletContextResource(this.servletContext, SERV
                registration.addResourceLocations(resource);
            }
        });
}
```

1、规则一：访问 `/webjars/**` 路径就去 `classpath:/META-INF/resources/webjars/` 下找资源。

- ① Maven 导入依赖。

2、规则二：访问 `/**` 路径就去 静态资源默认四个位置找资源。

- ① `classpath:/META-INF/resources/`
- ② `classpath:/resources/`
- ③ `classpath:/static/`
- ④ `classpath:/public/`

3、规则三：静态资源默认都有缓存规则的设置。

- ① 所有缓存的设置，直接通过配置文件：`spring.web`。
- ② `cachePeriod`：缓存周期，多久不用找服务器要新的，默认没有缓存周期，以秒

为单位。

- ③ **cacheControl** : **HTTP 缓存控制**, <https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Caching> 。
- ④ **useLastModified** : 是否使用最后一次修改, 配合 HTTP Cache 规则。

如果浏览器访问了一个静态资源 **index.js**, 如果服务这个资源没有发生变化, 下次访问的时候就可以直接让浏览器用自己缓存中的东西, 而不用给服务器发请求。

```
registration.setCachePeriod(getSeconds(this.resourceProperties.getCache().getPeriod()));
registration.setCacheControl(this.resourceProperties.getCache().getCacheControl().toHttpCacheControl());
registration.setUseLastModified(this.resourceProperties.getCache().isUseLastModified());
```

2.1.5 EnableWebMvcConfiguration 源码

```
/*
SpringBoot 给容器中放 WebMvcConfigurationSupport 组件
如果自己放了 WebMvcConfigurationSupport 组件, SpringBoot 的 WebMvcAutoConfiguration 都会失效
*/
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(WebProperties.class)
public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration implements ResolvableConfiguration {
}
}
```

1、**HandlerMapping** : 根据请求路径 `/xxx` 找那个 handler 能处理请求。

- ① **WelcomePageHandlerMapping** :
 - (1) 访问 `/**` 路径下的所有请求, 都在以前四个静态资源路径下找, 欢迎页也一样。
 - (2) 找 `index.html` : 只要静态资源的位置有一个 `index.html` 页面, 项目启动默认访问。

2.1.6 为什么容器中放一个 WebMvcConfigurer 就能配置底层行为

1、**WebMvcAutoConfiguration** 是一个自动配置类, 它里面有一个 **EnableWebMvcConfiguration** 。

2、**EnableWebMvcConfiguration** 继承于 **DelegatingWebMvcConfiguration**, 这两个都生效。

- 3、`DelegatingWebMvcConfiguration` 利用依赖注入把容器中所有 `WebMvcConfigurer` 注入进来。
- 4、别人调用 `DelegatingWebMvcConfiguration` 的方法配置底层规则，而它调用所有 `WebMvcConfigurer` 的配置底层方法。

2.1.7 WebMvcConfigurationSupport

提供了很多的默认设置。

判断系统中是否有相应的类：如果有，就加入相应的 `HttpMessageConverter`

```
jackson2Present = ClassUtils.isPresent("com.fasterxml.jackson.databind.ObjectMapper", classLoader);
ClassUtils.isPresent("com.fasterxml.jackson.core.JsonGenerator", classLoader);
jackson2XmlPresent = ClassUtils.isPresent("com.fasterxml.jackson.dataformat.xml.XmlMapper", classLoader);
jackson2SmilePresent = ClassUtils.isPresent("com.fasterxml.jackson.dataformat.smile.SmileFactory", classLoader);
```

2.2 Web 场景

2.2.1 自动配置

- 1、整合 web 场景。

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

- 2、引入了 `autoconfigure` 功能。
- 3、`@EnableAutoConfiguration` 注解使用 `@Import(AutoConfigurationImportSelector.class)` 批量导入组件。
- 4、加载 `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 文件中配置的所有组件。
- 5、所有自动配置类如下。

```

org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration
org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactoryCustomizerAutoConfiguration
// =====以下是响应式 Web 场景=====
org.springframework.boot.autoconfigure.web.reactive.HandlerAutoConfiguration
org.springframework.boot.autoconfigure.web.reactive.ReactiveMultipartAutoConfiguration
org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryAutoConfiguration
org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration
org.springframework.boot.autoconfigure.web.reactive.WebSessionIdResolverAutoConfiguration
org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration
org.springframework.boot.autoconfigure.web.reactive.function.client.ClientHttpConnectorAutoConfiguration
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration
// =====
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

```

6、绑定了配置文件的一堆配置项。

- ① SpringMVC 的所有配置 `spring.mvc`。
- ② Web 场景通用配置 `spring.web`。
- ③ 文件上传配置 `spring.servlet.multipart`。
- ④ 服务器的配置 `server`，比如：编码方式

2.2.2 默认效果

默认配置：

- 1、包含了 `ContentNegotiatingViewResolver` 和 `BeanNameViewResolver` 组件，方便视图解析。
- 2、默认的静态资源处理机制：静态资源放在 `static` 文件夹下即可直接访问。
- 3、自动注册了 `Converter`，`GenericConverter`，`Formatter` 组件，适配常见数据类型转换和格式化需求。
- 4、支持 `HttpMessageConverters`，可以方便返回 `json` 等数据类型。
- 5、注册 `MessageCodesResolver`，方便国际化及错误消息处理。
- 6、支持静态 `index.html`。

7、自动使用 `ConfigurableWebBindingInitializer`，实现消息处理、数据绑定、类型转化、数据校验等功能。

重要：

- 如果想保持 **boot mvc** 的默认配置，并且自定义更多的 mvc 配置，比如：**interceptors, formatters, view controllers** 等。可以使用 **@Configuration** 注解添加一个 **WebMvcConfigurer** 类型的配置类，并且不要标注 **@EnableWebMvc**。
- 如果想保持 boot mvc 的默认配置，但要自定义核心组件实例，比如：**RequestMappingHandlerMapping, RequestMappingHandlerAdapter**, 或 **ExceptionHandlerExceptionResolver**，给容器中放一个 **WebMvcRegistrations** 组件即可。
- 如果想全面接管 Spring MVC，**@Configuration** 标注一个配置类，并加上 **@EnableWebMvc** 注解，实现 **WebMvcConfigurer** 接口。

2.3 静态资源

2.3.1 默认规则

2.3.1.1 静态资源映射

静态资源映射规则在 `WebMvcAutoConfiguration` 中进行了定义：

- 1、`/webjars/**` 的所有路径资源都在 `classpath:/META-INF/resources/webjars/`。
- 2、`/**` 的所有路径资源都在 `classpath:/META-INF/resources/`、`classpath:/resources/`、`classpath:/static/`、`classpath:/public/`。
- 3、所有静态资源都定义了 **缓存规则**。【浏览器访问过一次，就会缓存一段时间】，但此功能参数无默认值。
 - ① `period`：缓存间隔，默认 0 秒。
 - ② `cacheControl`：缓存控制，默认无。
 - ③ `useLastModified`：是否使用 lastModified 头，默认 false。

2.3.1.2 静态资源缓存

如前面所述

1、所有静态资源都定义了 缓存规则 。【浏览器访问过一次，就会缓存一段时间】， 但此功能参数无默认值。

- ① `period` : 缓存间隔， 默认 0 秒。
- ② `cacheControl` : 缓存控制， 默认无。
- ③ `useLastModified` : 是否使用 lastModified 头， 默认 false。

2.3.1.3 欢迎页

欢迎页规则在 `WebMvcAutoConfiguration` 中进行了定义：

- 1、在静态资源目录下找 `index.html` 模板页。
- 2、没有就在 `templates` 下找 `index.html` 模板页。

2.3.1.4 Favicon

- 1、在静态资源目录下找 `favicon.ico` 。

2.3.1.5 缓存实验

```
server.port=8080

# 1、spring.web:
# ① 配置国际化的区域信息
# ② 静态资源策略（开启、处理链、缓存）

# 开启静态资源映射规则
spring.web.resources.add-mappings=true

# 设置缓存
spring.web.resources.cache.period=3600

# 缓存详细合并项控制，覆盖 period 配置
# 浏览照第一次请求服务器，服务器告诉浏览器此资源缓存 7200 秒，7200 秒以内的所有此资源访问不用发给服务器请求
spring.web.resources.cache.cachecontrol.max-age=7200

# 共享缓存
spring.web.resources.cache.cachecontrol.cache-public=true

# 使用资源 last-modified 时间，来对比服务器和浏览器的资源是否相同没有变化，相同返回 304
spring.web.resources.cache.use-last-modified=true
```

2.3.2 自定义静态资源规则

自定义静态资源路径、自定义缓存规则。

2.3.2.1 配置方式

`spring.mvc`：静态资源访问前缀路径。

`spring.web`：

- 静态资源目录。
- 静态资源缓存策略。

```
# 2、spring.mvc
# ① 自定义 webjars 路径前缀
spring.mvc.webjars-path-pattern=/webjars/**

# ② 静态资源访问路径前缀
spring.mvc.static-path-pattern=/static/**
```

2.3.2.2 代码方式

容器中只要有一个 WebMvcConfigurer 组件，配置的底层行为都会生效。

@EnableWebMvc，禁用 boot 的默认配置。

```
package com.myxh.springboot.web.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.http.CacheControl;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import java.util.concurrent.TimeUnit;

/**
 * @author MYXH
 * @date 2023/9/18
 */
// 禁用 Spring Boot 的默认配置
// @EnableWebMvc
// 这是一个配置类，给容器中放一个 WebMvcConfigurer 组件，就能自定义底层
@Configuration
public class MyConfig implements WebMvcConfigurer
{
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry)
    {
        // 保留默认规则
        WebMvcConfigurer.super.addResourceHandlers(registry);

        // 新增自定义规则
        registry.addResourceHandler("/static/**")
            .addResourceLocations("classpath:/image/", "classpath:/static/")
            .setCacheControl(CacheControl.maxAge(7200, TimeUnit.SECONDS));
    }
}
```

```
package com.myxh.springboot.web.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.CacheControl;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import java.util.concurrent.TimeUnit;

/**
 * @author MYXH
 * @date 2023/9/18
 */
// 禁用 Spring Boot 的默认配置
// @EnableWebMvc
// 这是一个配置类，给容器中放一个 WebMvcConfigurer 组件，就能自定义底层
@Configuration
public class MyConfig
{
    @Bean
    public WebMvcConfigurer webMvcConfigurer()
    {
        return new WebMvcConfigurer()
        {
            /**
             * 配置静态资源
             *
             * @param registry 注册表
             */
            @Override
            public void addResourceHandlers(ResourceHandlerRegistry registry)
            {
                // 保留默认规则
                WebMvcConfigurer.super.addResourceHandlers(registry);

                // 新增自定义规则
                registry.addResourceHandler("/static/**")
                    .addResourceLocations("classpath:/image/", "classpath:/static/")
                    .setCacheControl(CacheControl.maxAge(7200, TimeUnit.SECONDS));
            }
        };
    }
}
```

2.4 路径匹配

Spring5.3 之后加入了更多的请求路径匹配的实现策略。

以前只支持 **AntPathMatcher** 策略, 现在提供了 **PathPatternParser** 策略, 并且可以指定到底使用那种策略。

2.4.1 Ant 风格路径用法

Ant 风格的路径模式语法具有以下规则：

- `*`：表示任意数量的字符。
- `?`：表示任意一个字符。
- `**`：表示任意数量的目录。
- `{}`：表示一个命名的模式占位符。
- `[]`：表示 字符集合，例如 `[a-z]` 表示小写字母。

例如：

- `*.html` 匹配任意名称，扩展名为 `.html` 的文件。
- `/folder1/*/*.java` 匹配在 `folder1` 目录下的任意两级目录下的 `.java` 文件。
- `/folder2/**/*.jsp` 匹配在 `folder2` 目录下任意目录深度的 `.jsp` 文件。
- `/ {type} / {id} .html` 匹配任意文件名为 `{id}.html`，在任意命名的 `{type}` 目录下的文件。

注意：Ant 风格的路径模式语法中的特殊字符需要转义，例如：

- 要匹配文件路径中的星号，则需要转义为 `*`。
- 要匹配文件路径中的问号，则需要转义为 `\\\\\\?`。

2.4.2 模式切换

AntPathMatcher 与 **PathPatternParser**。

PathPatternParser 在 jmh 基准测试下，有 6~8 倍吞吐量提升，降低 30%~40% 空间分配率。

PathPatternParser 兼容 **AntPathMatcher** 语法，并支持更多类型的路径模式。

PathPatternParser “**” 多段匹配的支持仅允许在模式末尾使用。

```

package com.myxh.springboot.web.controller;

import jakarta.servlet.http.HttpServletRequest;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

/**
 * @author MYXH
 * @date 2023/9/18
 */
@Slf4j
@RestController
public class HelloController
{
    /**
     * 默认使用新版 PathPatternParser 进行路径匹配
     * 不能匹配 ** 在中间的情况，其他情况和 antPathMatcher语法兼容
     *
     * @param request 请求
     * @param path 路径
     * @return uri
     */
    @GetMapping("/a*/b?/**/{p1:[a-f]+}/**")
    public String hello(HttpServletRequest request, @PathVariable("p1") String path)
    {
        log.info("路径变量 p1: {}", path);
        String uri = request.getRequestURI();

        return uri;
    }
}

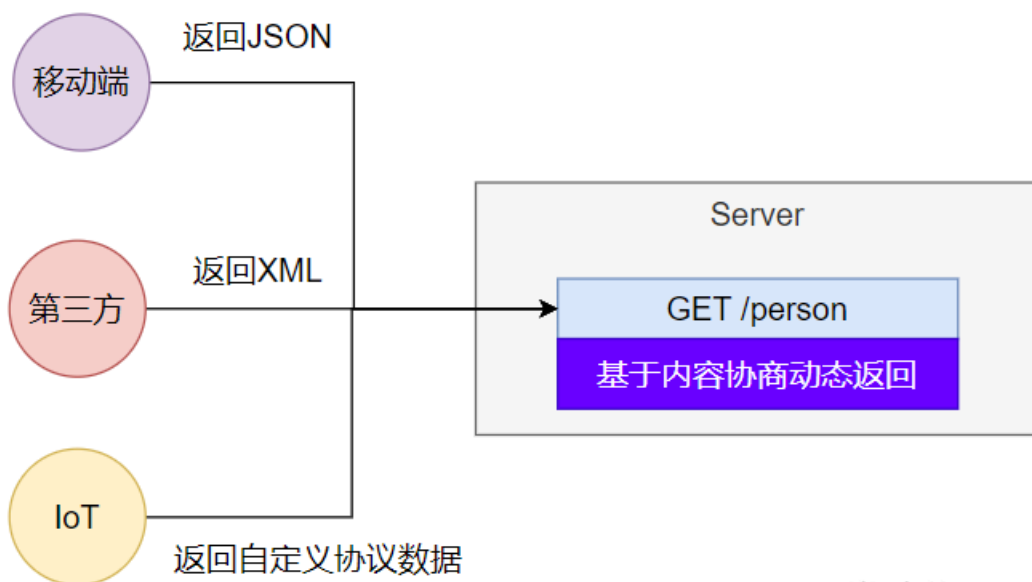
```

总结：

- 使用默认的路径匹配规则，是由 `PathPatternParser` 提供的。
- 如果路径中间需要有 `**`，替换成 ant 风格路径。

2.5 内容协商

一套系统适配多端数据返回。



尚硅谷 atguigu.com

2.5.1 多端内容适配

2.5.1.1 默认规则

1、SpringBoot 多端内容适配。

- ① 基于请求头内容协商：（默认开启）
 - (1)客户端向服务端发送请求，携带 HTTP 标准的 **Accept 请求头**。
 - [1] **Accept:** application/json 、 text/xml 、 text/yaml 。
 - [2] 服务端根据客户端**请求头期望的数据类型**进行**动态返回**。
 - ② 基于请求参数内容协商：（需要开启）
 - [1] 发送请求 GET /projects/spring-boot?format=json 。
 - [2] 匹配到 @GetMapping("/projects/spring-boot") 。
 - [3] 根据**参数协商**，优先返回 json 类型数据 **【需要开启参数匹配设置】** 。
 - [4] 发送请求 GET /projects/spring-boot?format=xml ， 优先返回 xml 类型数据。

2.5.1.2 效果演示

请求同一个接口，可以返回 json 和 xml 不同格式数据。

1、引入支持写出 xml 内容依赖。


```
<!-- 支持返回 XML 格式数据 -->
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

2、标注注解。

```
package com.myxh.springboot.web.bean;

import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlElement;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

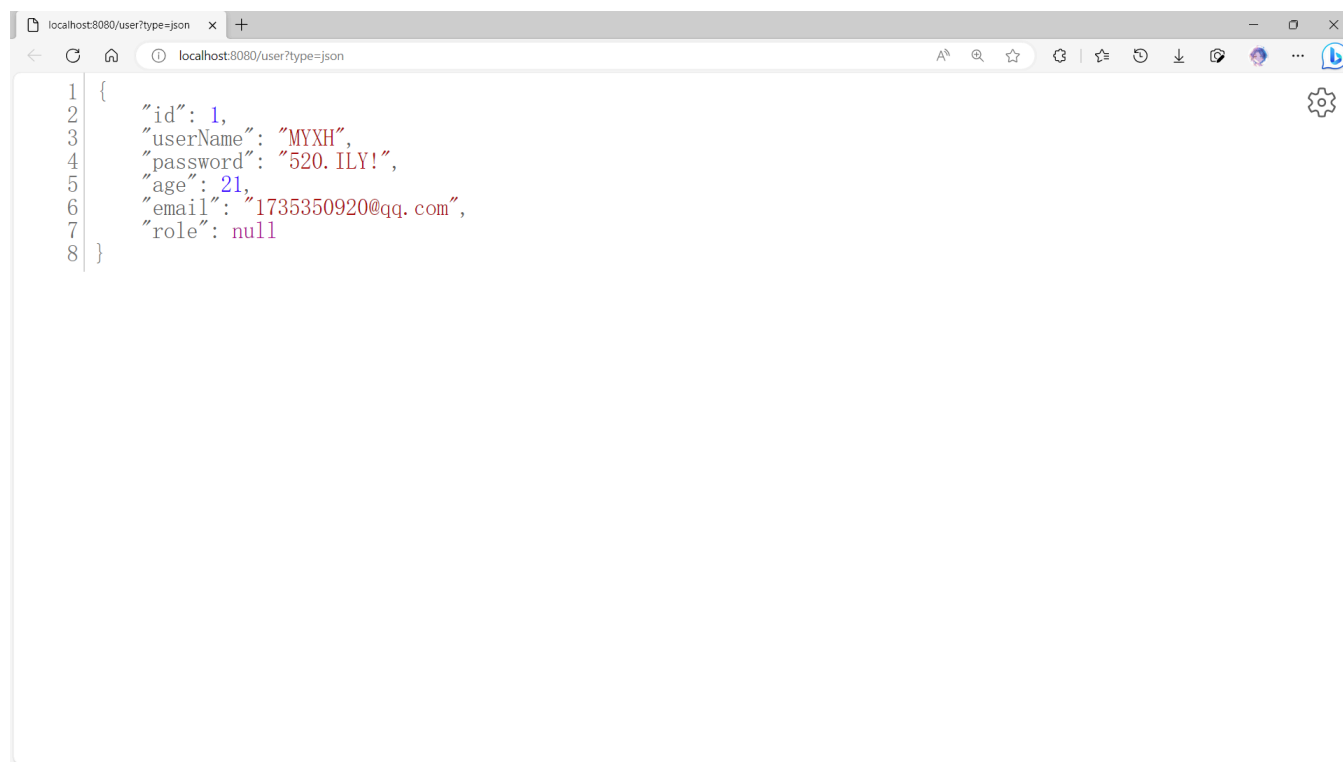
/**
 * @author MYXH
 * @date 2023/9/18
 */
// 可以写出为 xml 文档
@JacksonXmlElement
@Component
// 和配置文件 person 前缀的所有配置进行绑定
@ConfigurationProperties(prefix = "user")
// 自动生成无参构造器
@NoArgsConstructor
// 自动生成全参构造器
@AllArgsConstructor
// 自动生成 JavaBean 属性的 getter/setter
@Data
public class User
{
    private Long id;
    private String userName;
    private String password;
    private Integer age;
    private String email;
    private String role;
}
```

3、开启基于请求参数的内容协商。

```
# 开启基于请求参数的内容协商功能，默认参数名：format，默认此功能不开启
spring.mvc.contentnegotiation.favor-parameter=true

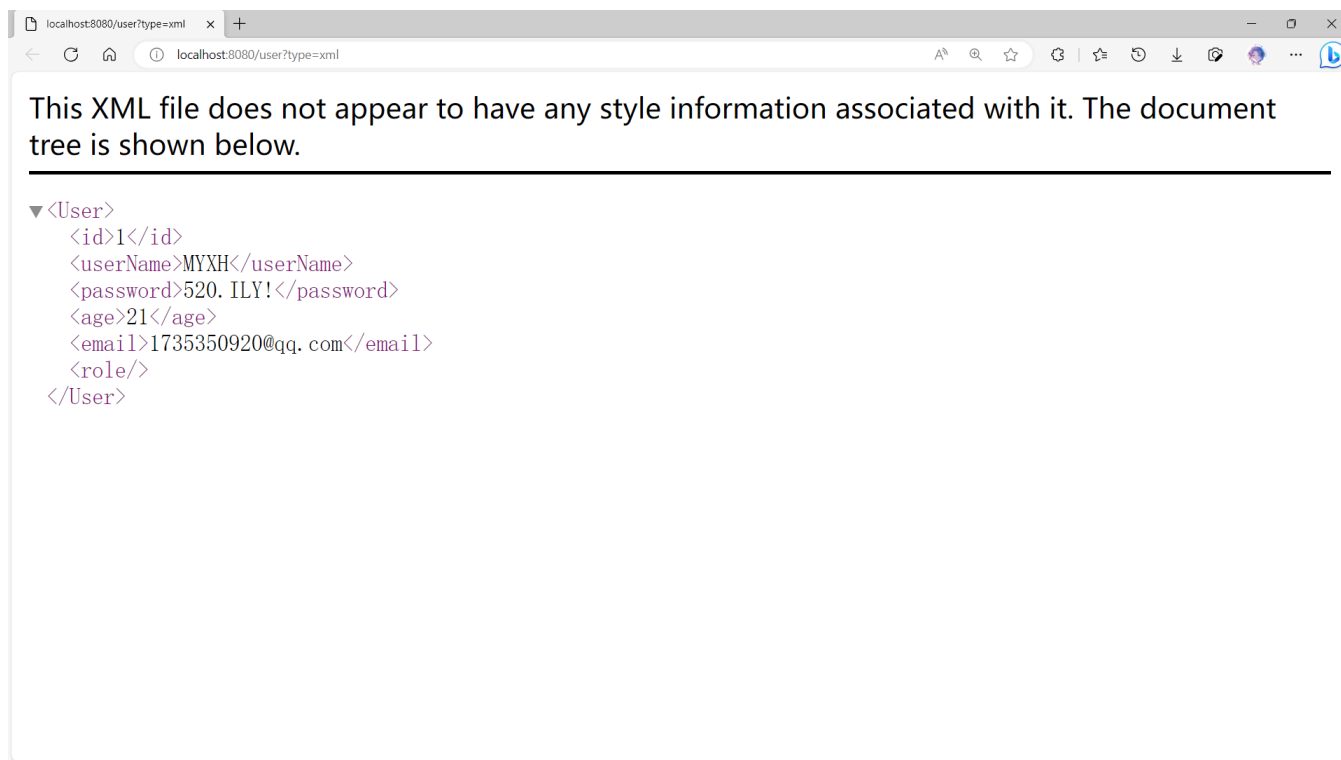
# 指定内容协商时使用的参数名，默认是 format
spring.mvc.contentnegotiation.parameter-name=type
```

4、效果。



A screenshot of a web browser window. The address bar shows the URL `localhost:8080/user?type=json`. The browser's developer tools are open, displaying a JSON object in the console. The JSON object is a user profile with the following fields: `id` (1), `userName` ("MYXH"), `password` ("520. ILY!"), `age` (21), `email` ("1735350920@qq.com"), and `role` (null). The browser interface includes standard navigation buttons, a search bar, and a settings icon in the top right corner.

```
1 {
2   "id": 1,
3   "userName": "MYXH",
4   "password": "520. ILY!",
5   "age": 21,
6   "email": "1735350920@qq.com",
7   "role": null
8 }
```



2.5.1.3 配置协商规则与支持类型

1、修改内容协商方式。

```
# 开启基于请求参数的内容协商功能，默认参数名：format，默认此功能不开启
spring.mvc.contentnegotiation.favor-parameter=true

# 指定内容协商时使用的参数名，默认是 format
spring.mvc.contentnegotiation.parameter-name=type
```

2、大多数 MediaType 都是开箱即用的。也可以自定义内容类型，例如：

```
# 增加一种新的内容类型
spring.mvc.contentnegotiation.media-types.yaml=text/yaml
spring.mvc.contentnegotiation.media-types.yml=text/yml
```

2.5.2 自定义内容返回

2.5.2.1 增加 yaml 返回支持

导入依赖。

```
<!-- 支持返回 YAML 格式数据 -->
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-yaml</artifactId>
</dependency>
```

把对象写出成 YAML。

```
package com.myxh.springboot.web.controller;

import com.myxh.springboot.web.bean.User;
/**
 * @author MYXH
 * @date 2023/9/18
 */
public class HelloController
{
    public static void main(String[] args) throws JsonProcessingException
    {
        User user = new User();
        user.setId(1L);
        user.setUserName("MYXH");
        user.setPassword("520.ILY!");
        user.setAge(21);
        user.setEmail("1735350920@qq.com");

        YAMLFactory factory = new YAMLFactory().disable(YAMLGenerator.Feature.WRITE_DOC_START_MARKER);
        ObjectMapper mapper = new ObjectMapper(factory);

        String userYaml = mapper.writeValueAsString(user);
        System.out.println("userYaml = " + userYaml);
    }
}
```

编写配置。

```
# 增加一种新的内容类型
spring.mvc.contentnegotiation.media-types.yaml=text/yaml
spring.mvc.contentnegotiation.media-types.yml=text/yml
```

增加 `HttpMessageConverter` 组件，专门负责把对象写出为 yaml 格式。

```

package com.myxh.springboot.web.config;

import com.myxh.springboot.web.component.MyYamlHttpMessageConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.converter.HttpMessageConverter;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import java.util.List;

/**
 * @author MYXH
 * @date 2023/9/18
 */
// 禁用 Spring Boot 的默认配置
// @EnableWebMvc
// 这是一个配置类，给容器中放一个 WebMvcConfigurer 组件，就能自定义底层
@Configuration
public class MyConfig
{
    @Bean
    public WebMvcConfigurer webMvcConfigurer()
    {
        return new WebMvcConfigurer()
        {
            /**
             * 配置一个能把对象转为 yaml 的 messageConverter
             *
             * @param converters 最初是转换器的空列表
             */
            @Override
            public void configureMessageConverters(List<HttpMessageConverter<?>> converters)
            {
                converters.add(new MyYamlHttpMessageConverter());
            }
        };
    }
}

```

2.5.2.2 思考：如何增加其他

- 配置媒体类型支持：
 - `spring.mvc.contentnegotiation.media-types.yaml=text/yaml`。
- 编写对应的 `HttpMessageConverter`，要告诉 Boot 这个支持的媒体类型。

- 按照 `HttpMessageConverter` 的示例写法。
- 把 `MessageConverter` 组件加入到底层。
 - 容器中放一个 `WebMvcConfigurer` 组件，并配置底层的 `MessageConverter` 。

2.5.2.3 HttpResponseMessageConverter 的示例写法

```
package com.myxh.springboot.web.component;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.dataformat.yaml.YAMLFactory;
import com.fasterxml.jackson.dataformat.yaml.YAMLGenerator;
import org.springframework.http.HttpInputMessage;
import org.springframework.http.HttpOutputMessage;
import org.springframework.http.MediaType;
import org.springframework.http.converter.AbstractHttpMessageConverter;
import org.springframework.http.converter.HttpMessageNotReadableException;
import org.springframework.http.converter.HttpMessageNotWritableException;

import java.io.IOException;
import java.io.OutputStream;
import java.nio.charset.StandardCharsets;

/**
 * @author MYXH
 * @date 2023/9/18
 * @description
 * 自定义的 YAML HTTP 消息转换器
 * 用于将对象转换为 YAML 格式的内容
 */
public class MyYamlHttpMessageConverter extends AbstractHttpMessageConverter<Object>
{
    // 将对象转换为 YAML 的 ObjectMapper
    private final ObjectMapper objectMapper;

    public MyYamlHttpMessageConverter()
    {
        // 告诉 SpringBoot 这个 MessageConverter 支持哪种媒体类型
        super(new MediaType("text", "yaml", StandardCharsets.UTF_8),
            new MediaType("text", "yml", StandardCharsets.UTF_8));

        // 创建 YAMLFactory 并禁用写入文档起始标记
        YAMLFactory factory = new YAMLFactory().
            disable(YAMLGenerator.Feature.WRITE_DOC_START_MARKER);
        this.objectMapper = new ObjectMapper(factory);
    }

    /**
     * 判断该转换器是否支持将指定类型的对象转换为 YAML
     *
     * @param clazz 要测试支持的类
     */
}
```



```

    * @return 如果支持转换, 则返回 true; 否则返回 false
    */
    @Override
    protected boolean supports(Class<?> clazz)
    {
        // 只要是对象类型, 不是基本类型和数组类型, 都支持
        return !clazz.isPrimitive() && !clazz.isArray();
    }

    /**
     * 从 HTTP 输入消息中读取数据并将其转换为指定类型的对象
     * {@code @RequestBody} 把对象怎么读进来
     *
     * @param clazz 要返回的对象类型
     * @param inputMessage 要从中读取的 HTTP 输入消息
     * @return 转换后的对象
     * @throws IOException IO 异常
     * @throws HttpMessageNotReadableException Http 消息不可读异常
     */
    @Override
    protected Object readInternal(Class<?> clazz, HttpInputMessage inputMessage) throws IOException
    {
        return objectMapper.readValue(inputMessage.getBody(), clazz);
    }

    /**
     * 将指定对象写入 HTTP 输出消息
     * {@code @ResponseBody} 把对象怎么写出去
     *
     * @param methodReturnValue 要写入输出消息的对象
     * @param outputMessage 要写入的 HTTP 输出消息
     * @throws IOException IO 异常
     * @throws HttpMessageNotWritableException Http 消息不可写入异常
     */
    @Override
    protected void writeInternal(Object methodReturnValue, HttpOutputMessage outputMessage) throws
    {
        // 使用 try-with-resources 语句以自动关闭流
        try (OutputStream os = outputMessage.getBody())
        {
            this.objectMapper.writeValue(os, methodReturnValue);
        }
    }
}

```

2.5.3 内容协商原理 `HttpMessageConverter`

`HttpMessageConverter` 怎么工作？合适工作？

定制 `HttpMessageConverter` 来实现多端内容协商。

编写 `WebMvcConfigurer` 提供的 `configureMessageConverters` 底层，修改底层的 `MessageConverter`。

2.5.3.1 `@ResponseBody` 由 `HttpMessageConverter` 处理

标注了 `@ResponseBody` 的返回值 将会由支持它的 `HttpMessageConverter` 写给浏览器。

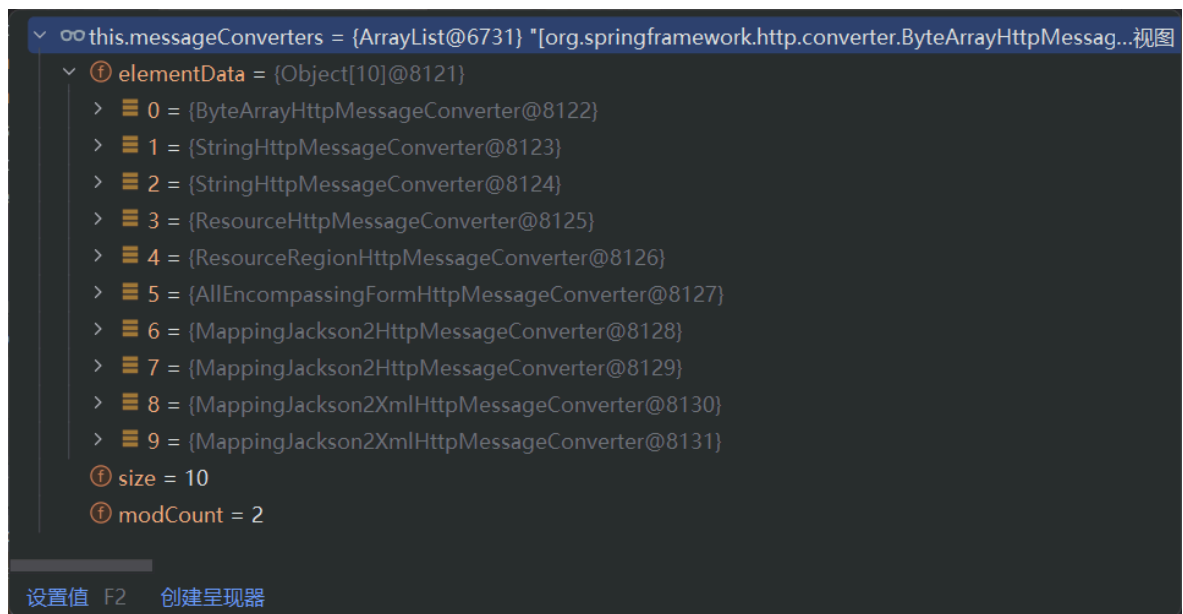
1、如果 controller 方法的返回值标注了 `@ResponseBody` 注解。

- ① 请求进来先来到 `DispatcherServlet` 的 `doDispatch()` 进行处理。
- ② 找到一个 `HandlerAdapter` 适配器，利用适配器执行目标方法。
- ③ `RequestMappingHandlerAdapter` 来执行，调用 `invokeHandlerMethod()` 来执行目标方法。
- ④ 目标方法执行之前，准备好两个东西。
 - (1) `HandlerMethodArgumentResolver`：参数解析器，确定目标方法每个参数值。
 - (2) `HandlerMethodReturnValueHandler`：返回值处理器，确定目标方法的返回值怎么处理。
- ⑤ `RequestMappingHandlerAdapter` 里面的 `invokeAndHandle()` 真正执行目标方法。
- ⑥ 目标方法执行完成，会返回返回值对象。
- ⑦ 找到一个合适的返回值处理器 `HandlerMethodReturnValueHandler`。
- ⑧ 最终找到 `RequestResponseBodyMethodProcessor` 能处理标注了 `@ResponseBody` 注解的方法。
- ⑨ `RequestResponseBodyMethodProcessor` 调用 `writeWithMessageConverters`，利用 `MessageConverter` 把返回值写出去。

上面解释：`@ResponseBody` 由 `HttpMessageConverter` 处理。

2、`HttpMessageConverter` 会先进行内容协商。

- ① 遍历所有的 `MessageConverter` 看谁支持这种内容类型的数据。
- ② 默认 `MessageConverter` 有以下 10 个。

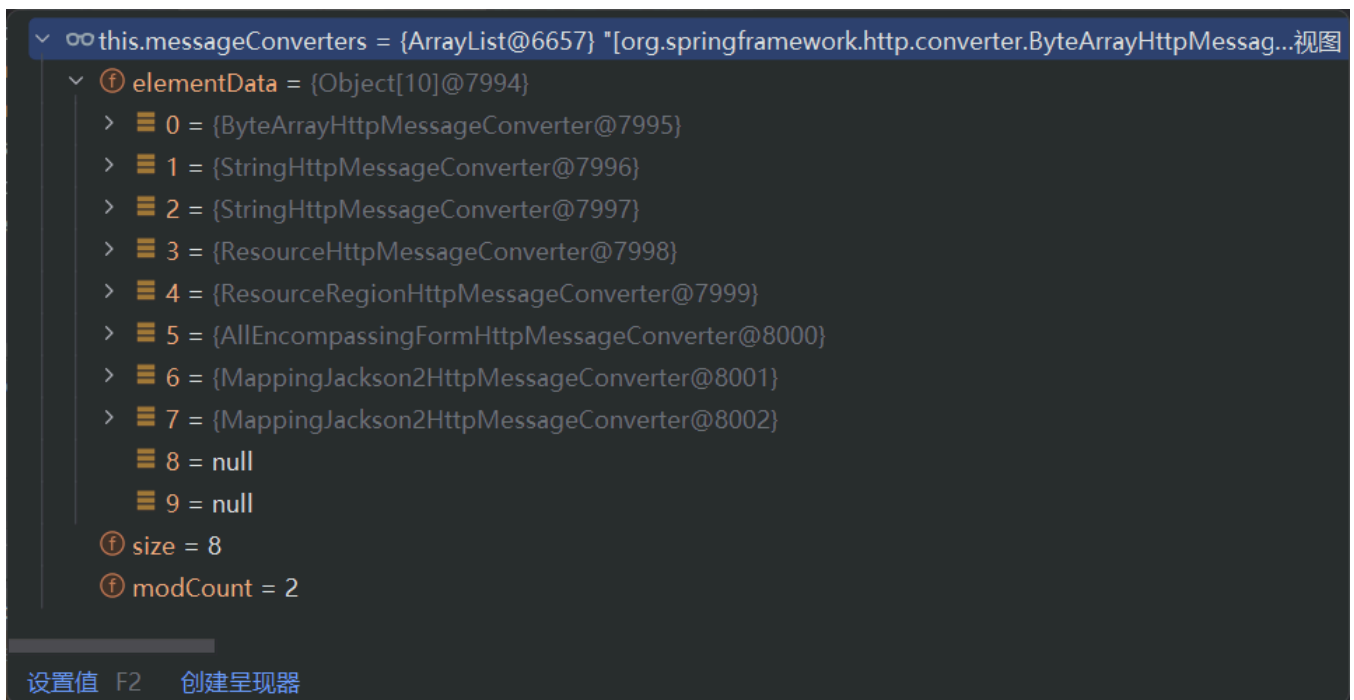


- ③ 最终因为要 json 所以 MappingJackson2HttpMessageConverter 支持写出 json。
- ④ jackson 用 ObjectMapper 把对象写出去。

2.5.3.2 WebMvcAutoConfiguration 提供几种默认 HttpMessageConverters

- EnableWebMvcConfiguration 通过 addDefaultHttpMessageConverters 添加了默认的 MessageConverter，如下：
 - ByteArrayHttpMessageConverter：支持字节数据读写。
 - StringHttpMessageConverter：支持字符串读写。
 - ResourceHttpMessageConverter：支持资源读写。
 - ResourceRegionHttpMessageConverter：支持分区资源写出。
 - AllEncompassingFormHttpMessageConverter：支持表单 xml/json 读写。
 - MappingJackson2HttpMessageConverter：支持请求响应体 Json 读写。

默认 MessageConverter 有以下 8 个：

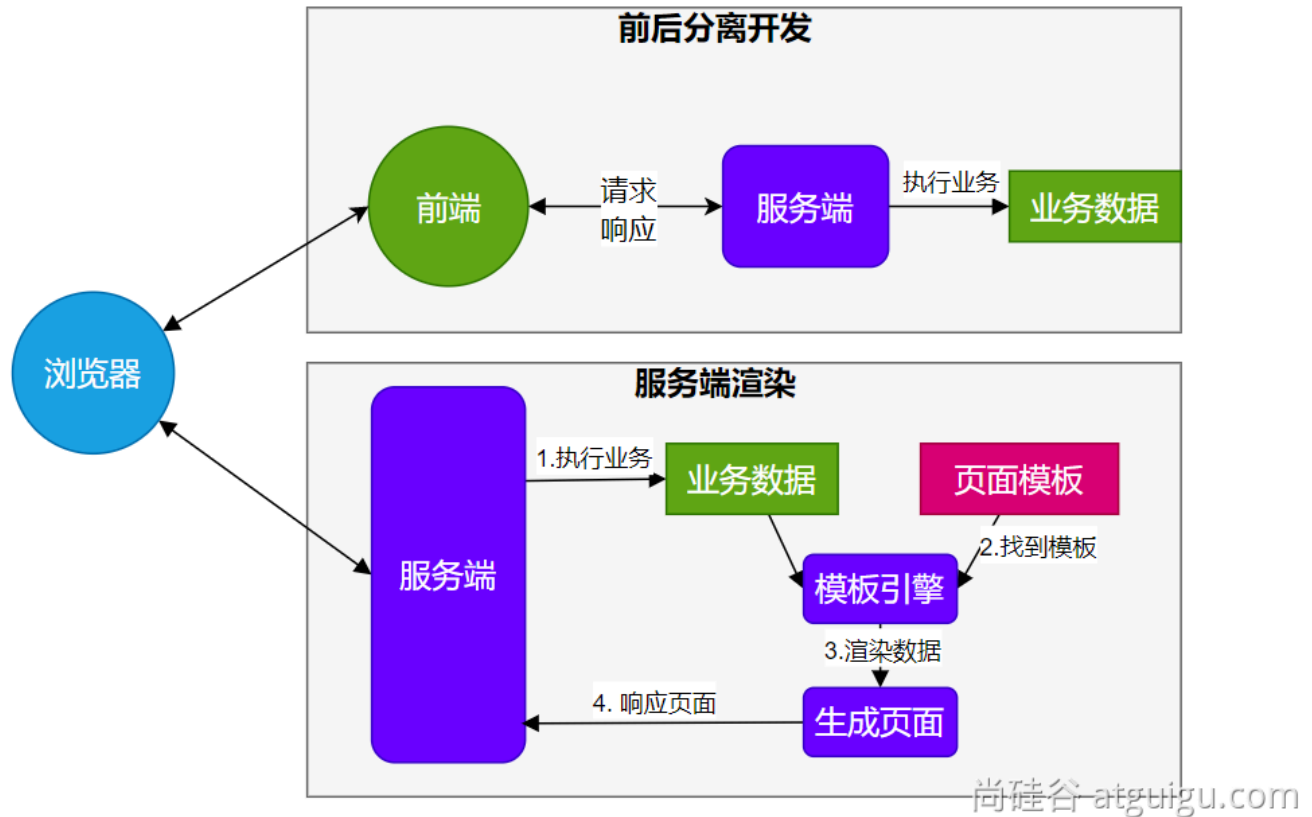


系统提供默认的 MessageConverter 功能有限，仅用于 json 或者普通返回数据。额外增加新的内容协商功能，必须增加新的 **HttpMessageConverter**。

2.6 模板引擎

由于 **SpringBoot** 使用了嵌入式 **Servlet** 容器。所以 **JSP** 默认是不能使用的。

如果需要**服务端页面渲染**，优先考虑使用**模板引擎**。



模板引擎页面默认放在 **src/main/resources/templates**。

SpringBoot 包含以下模板引擎的自动配置。

- FreeMarker
- Groovy
- **Thymeleaf**
- Mustache

Thymeleaf 官网：<https://www.thymeleaf.org/>。

2.6.1 Thymeleaf 整合

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

自动配置原理：

1、开启了 `org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration` 自动配置。

2、属性绑定在 `ThymeleafProperties` 中，对应配置文件 `spring.thymeleaf` 内容。

3、所有的模板页面默认在 `classpath:/templates/` 文件夹下。

4、默认效果。

- ① 所有的模板页面在 `classpath:/templates/` 下面找。
- ② 找后缀名为 `.html` 的页面。

2.6.2 基础语法

2.6.2.1 核心用法

`th:xxx` : 动态渲染指定的 html 标签属性值、或者 th 指令（遍历、判断等）。

- `th:text` : 标签体内文本值渲染。
 - `th:utext` : 不会转义，显示为 html 原本的样子。
- `th:属性` : 标签指定属性渲染。
- `th:attr` : 标签任意属性渲染。
- `th:if`、`th:each` ... : 其他 th 指令。
- 例如 :

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

  <body>
    <h1>你好, <span th:utext="${message}"></span></h1>

    <hr />

    <h2>1、Thymeleaf 基础语法</h2>

    <h3>1.1 th:text: 替换标签体的内容, 会转义</h3>
    <p th:text="${message}"></p>

    <h3>
      1.2 th:utext: 替换标签体的内容, 不会转义 html 标签, 真正显示为 html
      该有的样式
    </h3>
    <p th:utext="${message}"></p>

    <h3>1.3 th:任意 html 属性, 动态替换任意属性的值</h3>
    

    <h3>1.4 th:attr: 任意属性指定</h3>
    

    <h3>1.5 th: 其他指令</h3>
    
```

```

        alt="大户爱"
    />

<h3>1.6 @{} 专门用来取各种路径</h3>


<h3>1.7 字符串转换为大写</h3>
<p th:text="${#strings.toUpperCase(name)}"></p>

<h3>1.8 字符串拼接</h3>
<p th:text="${'前缀'+name+'后缀'}"></p>
<p th:text="|前缀${name}后缀|"></p>
</body>
</html>

```

表达式：用来动态取值。

- `${\{\}}`：变量取值；使用 model 共享给页面的值都直接用 `${}`。
- `@{\}`：url 路径。
- `#{\}`：国际化消息。
- `~{\}`：片段引用。
- `*{\}`：变量选择：需要配合 `th:object` 绑定对象。

系统工具&内置对象：详细文档。

- `param`：请求参数对象。
- `session`：session 对象。
- `application`：application 对象。
- `#execInfo`：模板执行信息。
- `#messages`：国际化消息。
- `#uris`：uri/url 工具。
- `#conversions`：类型转换工具。
- `#dates`：日期工具，是 `java.util.Date` 对象的工具类。
- `#calendars`：类似 `#dates`，只不过是 `java.util.Calendar` 对象的工具类。
- `#temporals`：JDK8+ `java.time` API 工具类。

- `#numbers` : 数字操作工具。
- `#strings` : 字符串操作。
- `#objects` : 对象操作。
- `#bools` : bool 操作。
- `#arrays` : array 工具。
- `- #lists` : list 工具。
- `#sets` : set 工具。
- `#maps` : map 工具。
- `#aggregates` : 集合聚合工具 (sum、avg) 。
- `#ids` : id 生成工具。

2.6.2.2 语法示例

表达式 :

- 变量取值 : `${...}`
- url 取值 : `@{...}`
- 国际化消息 : `#{...}`
- 变量选择 : `*{...}`
- 片段引用: `~{...}`

常见 :

- 文本 : `'one text'` , `'another one!'` , ...
- 数字 : `0` , `12` , `3.0` , `15.6` , ...
- 布尔 : `true` 、 `false`
- null: `null`
- 变量名 : `one` , `sometext` , `main` ...

文本操作 :

- 拼串 : `+`
- 文本替换 : `| The name is ${name} |`

布尔操作 :

- 二进制运算 : `and` , `or`
- 取反 : `!` , `not`

比较运算：

- 比较：`<`，`>`，`<=`，`>=`（`lt`，`gt`，`le`，`ge`）
- 等值运算：`==`，`!=`（`eq`，`ne`）

条件运算：

- if-then：`(if)?(then)`
- if-then-else：`(if)?(then):(else)`
- default：`(value)?:(defaultValue)`

特殊语法：

- 无操作：`_`

所有以上都可以嵌套组合。

2.6.2.3 属性设置

- 1、`th:href="@{/product/list}"`
- 2、`th:attr="class=${active}"`
- 3、`th:attr="src=@{/images/image.png},title=${logo},alt=#{logo}"`
- 4、`th:checked="${user.active}"`

2.6.2.4 遍历

语法：`th:each="元素名,迭代状态:${集合}"`

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>用户列表页</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
      rel="stylesheet"
      integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEVDwWykc2MPK8M2HN"
      crossorigin="anonymous"
    />
  </head>

  <body>
    <!-- 导航区使用公共部分进行替换 -->
    <!-- ~{ 模板名 :: 片段名 } -->
    <div th:replace="~{common :: myheader}"></div>

    <div class="container">
      <table class="table">
        <thead>
          <tr>
            <th scope="col">序号</th>
            <th scope="col">用户名</th>
            <th scope="col">密码</th>
            <th scope="col">年龄</th>
            <th scope="col">邮箱</th>
            <th scope="col">角色</th>
            <th scope="col">状态信息</th>
          </tr>
        </thead>

        <tbody>
          <tr th:each="user, stats : ${userList}" th:object="${user}">
            <th scope="row" th:text="${user.id}"></th>
            <!-- <td th:text="${user.userName}"></td> -->
            <td th:text="*{userName}"></td>
            <td>[[${user.password}]]</td>
            <td
              th:text="|${user.age}(${user.age >= 18 ? '成年' : '未成年'})|"
            ></td>
            <td
              th:if="${#strings.isEmpty(user.email)}"
              th:text="'联系不上'"
            >

```

```

    ></td>
    <td>
        th:if="${not #strings.isEmpty(user.email)}"
        th:text="${user.email}"
    ></td>
    <td th:switch="${user.role}">
        <button th:case="'root'" type="button" class="btn btn-primary">
            root 用户
        </button>
        <button th:case="'admin'" type="button" class="btn btn-secondary">
            管理员
        </button>
        <button th:case="'test'" type="button" class="btn btn-success">
            测试员
        </button>
        <button th:case="'user'" type="button" class="btn btn-light">
            用户
        </button>
    </td>
    <td>
        index (索引) : [[${stats.index}]]<br />
        count (计数) : [[${stats.count}]]<br />
        size (大小) : [[${stats.size}]]<br />
        current (当前对象) : [[${stats.current}]]<br />
        even(true)/odd(false) (奇数/偶数) : [[${stats.even}]]<br />
        first (第一个) : [[${stats.first}]]<br />
        last (最后的) : [[${stats.last}]]<br />
    </td>
</tr>
</tbody>
</table>
</div>

<script>
    src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"
    integrity="sha384-C6RzsynM9kWDrmNeT87bh950GNyZPhcTNXj1NW7RuBCsyN/o0jlpcV8Qyq46cDfL"
    crossorigin="anonymous"
</script>
</body>
</html>

```

iterStat 有以下属性：

- index：当前遍历元素的索引，从 0 开始。

- count : 当前遍历元素的索引, 从 1 开始。
- size : 需要遍历元素的总数量。
- current : 当前正在遍历的元素对象。
- even/odd : 是否为偶数/奇数行。
- first : 是否第一个元素。
- last : 是否最后一个元素。

2.6.2.5 判断

th:if

```
<td th:text="|${user.age}|(${user.age >= 18 ? '成年' : '未成年'})|"></td>
<td th:if="${#strings.isEmpty(user.email)}" th:text="'联系不上'"></td>
<td th:if="${not #strings.isEmpty(user.email)}" th:text="${user.email}"></td>
```

th:switch

```
<td th:switch="${user.role}">
  <button th:case="'root'" type="button" class="btn btn-primary">
    root 用户
  </button>
  <button th:case="'admin'" type="button" class="btn btn-secondary">
    管理员
  </button>
  <button th:case="'test'" type="button" class="btn btn-success">测试员</button>
  <button th:case="'user'" type="button" class="btn btn-light">用户</button>
</td>
```

2.6.2.6 属性优先级

- 片段。
- 遍历。
- 判断。

```
<tr th:each="user, stats : ${userList}" th:object="${user}">
  <th scope="row" th:text="${user.id}"></th>
  <!-- <td th:text="${user.userName}"></td> -->
  <td th:text="*{userName}"></td>
</tr>
```

| 顺序 | 特性 | 属性 |
|----|-----------|--|
| 1 | 片段包含。 | th:insert 、 th:replace |
| 2 | 遍历。 | th:each |
| 3 | 判断。 | th:if 、 th:unless 、 th:switch 、 th:case |
| 4 | 定义本地变量。 | th:object 、 th:with |
| 5 | 通用方式属性修改。 | th:attr 、 th:attrprepend 、 th:attrappend |
| 6 | 指定属性修改。 | th:value 、 th:href 、 th:src 、 ... |
| 7 | 文本值。 | th:text 、 th:utext |
| 8 | 片段指定。 | th:fragment |
| 9 | 片段移除。 | th:remove |

2.6.2.7 行内写法

[[...]] 或 [(...)] 。

```
<tr th:each="user, stats : ${userList}" th:object="${user}">
  <th scope="row" th:text="${user.id}"></th>
  <!-- <td th:text="${user.userName}"></td> -->
  <td th:text="*{userName}"></td>
  <td>[[${user.password}]]</td>
</tr>
```

2.6.2.8 变量选择

```
<tr th:each="user, stats : ${userList}" th:object="${user}">
  <th scope="row" th:text="${user.id}"></th>
  <td th:text="*{userName}"></td>
</tr>
```

等同于

```
<tr th:each="user, stats : ${userList}">
  <th scope="row" th:text="${user.id}"></th>
  <td th:text="${user.userName}"></td>
</tr>
```

2.6.2.9 模板布局

- 定义模板：`th:fragment`
- 引用模板：`~{templatename::selector}`
- 插入模板：`th:insert`、`th:replace`

```
<!-- 导航区使用公共部分进行替换 -->
<!-- ~{ 模板名 :: 片段名 } -->
<div th:replace="~{common :: myheader}"></div>
```

2.6.2.10 devtools

```
<!-- 热启动功能 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

修改页面后，`ctrl+F9` 刷新效果。

Java 代码的修改，如果 `devtools` 热启动了，可能会引起一些 bug，难以排查。

2.7 国际化

国际化的自动配置参照 `MessageSourceAutoConfiguration`。

实现步骤：

1、Spring Boot 在类路径根下查找 `messages` 资源绑定文件。文件名为：`messages.properties`。

2、多语言可以定义多个消息文件，命名为 `messages_区域代码.properties`。例如：

- ① `messages.properties`：默认。

- ② `messages_zh_CN.properties` : 中文环境。
- ③ `messages_en_US.properties` : 英语环境。

3、在程序中可以自动注入 `MessageSource` 组件，获取国际化的配置项值。

4、在页面中可以使用表达式 `#{}` 获取国际化的配置项值。

```
package com.myxh.springboot.web.controller;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.Locale;

/**
 * @author MYXH
 * @date 2023/9/18
 */
//适配服务端渲染，前后不分离模式
@Controller
public class WelcomeController
{
    // 国际化获取消息的组件
    @Autowired
    MessageSource messageSource;

    @GetMapping("/message")
    @ResponseBody
    public String message(HttpServletRequest request)
    {
        Locale locale = request.getLocale();

        // 利用代码的方式获取国际化配置文件中指定的配置项的值
        String login = messageSource.getMessage("login", null, locale);

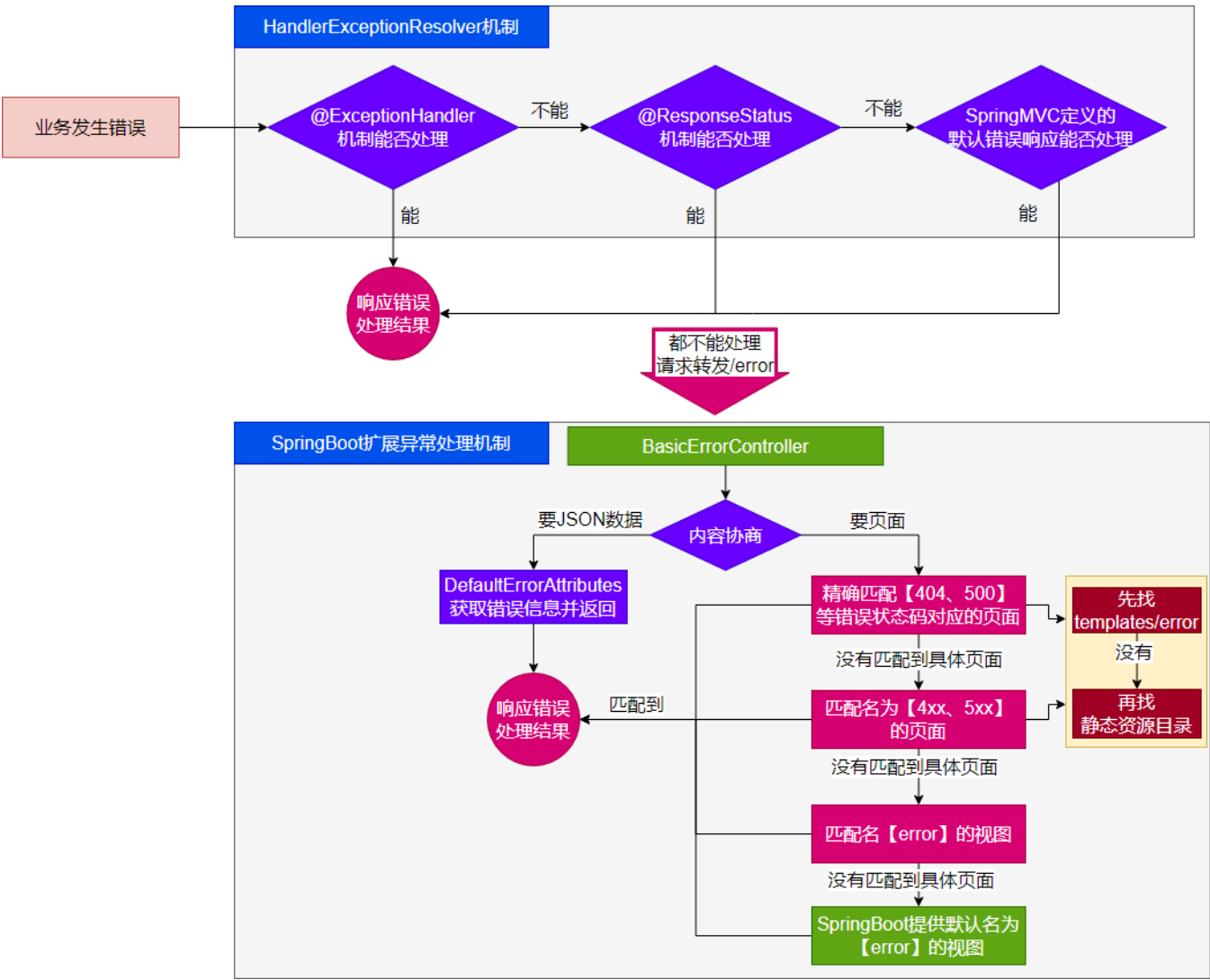
        return login;
    }
}
```


2.8 错误处理

2.8.1 默认机制

错误处理的自动配置都在 `ErrorMvcAutoConfiguration` 中，两大核心机制：

- 1、SpringBoot 会自适应处理错误，响应页面或 JSON 数据。
- 2、SpringMVC 的错误处理机制依然保留，MVC 处理不了，才会交给 boot 进行处理。



- 发生错误以后，转发给/error 路径，SpringBoot 在底层写好一个 `BasicErrorController` 的组件，专门处理这个请求。

```

// 返回 HTML
@RequestMapping(produces = MediaType.TEXT_HTML_VALUE)
public ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse response)
{
    HttpStatus status = getStatus(request);
    Map<String, Object> model = Collections
        .unmodifiableMap(getErrorAttributes(request, getErrorAttributeOptions(request,
            response.setStatus(status.value()));
    ModelAndView modelAndView = resolveErrorView(request, response, status, model);

    return (modelAndView != null) ? modelAndView : new ModelAndView("error", model);
}

//返回 ResponseEntity, JSON
@RequestMapping
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request)
{
    HttpStatus status = getStatus(request);

    if (status == HttpStatus.NO_CONTENT)
    {
        return new ResponseEntity<>(status);
    }

    Map<String, Object> body = getErrorAttributes(request, getErrorAttributeOptions(request,
        response.setStatus(status.value()));
    return new ResponseEntity<>(body, status);
}

```

- 错误页面是这么解析到的。

```

// 1、解析错误的自定义视图地址
ModelAndView modelAndView = resolveErrorView(request, response, status, model);

// 2、如果解析不到错误页面的地址，默认的错误页就是 error
return (modelAndView != null) ? modelAndView : new ModelAndView("error", model);

```

- 容器中专门有一个错误视图解析器。

```
@Bean
@ConditionalOnBean(DispatcherServlet.class)
@ConditionalOnMissingBean(ErrorViewResolver.class)
DefaultErrorViewResolver conventionErrorViewResolver()
{
    return new DefaultErrorViewResolver(this.applicationContext, this.resources);
}
```

- SpringBoot 解析自定义错误页的默认规则。

```

@Override
public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status, Map<String, Object> model)
{
    ModelAndView modelAndView = resolve(String.valueOf(status.value()), model);

    if (modelAndView == null && SERIES_VIEWS.containsKey(status.series()))
    {
        modelAndView = resolve(SERIES_VIEWS.get(status.series()), model);
    }

    return modelAndView;
}

private ModelAndView resolve(String viewName, Map<String, Object> model)
{
    String errorViewName = "error/" + viewName;
    TemplateAvailabilityProvider provider = this.templateAvailabilityProviders.getProvider(errorViewName,
        this.applicationContext);

    if (provider != null)
    {
        return new ModelAndView(errorViewName, model);
    }

    return resolveResource(errorViewName, model);
}

private ModelAndView resolveResource(String viewName, Map<String, Object> model)
{
    for (String location : this.resources.getStaticLocations())
    {
        try
        {
            Resource resource = this.applicationContext.getResource(location);
            resource = resource.createRelative(viewName + ".html");

            if (resource.exists())
            {
                return new ModelAndView(new HtmlResourceView(resource), model);
            }
        }
        catch (Exception ex)
        {
        }
    }
}

```

```

    }
}
return null;
}

```

- 容器中有一个默认的名为 error 的 view，提供了默认白页功能。

```

@Bean(name = "error")
@ConditionalOnMissingBean(name = "error")
public View defaultErrorView()
{
    return this.defaultErrorView;
}

```

- 封装了 JSON 格式的错误信息。

```

@Bean
@ConditionalOnMissingBean(value = ErrorAttributes.class, search = SearchStrategy.CURRENT)
public DefaultErrorAttributes errorAttributes()
{
    return new DefaultErrorAttributes();
}

```

规则：

1、解析一个错误页。

- ① 如果发生了 500、404、503、403 这些错误。
 - (1) 如果有模板引擎，默认在 `classpath:/templates/error/精确码.html`。
 - (2) 如果没有模板引擎，在静态资源文件夹下找 `精确码.html`。
- ② 如果匹配不到 `精确码.html` 这些精确的错误页，就去找 `5xx.html`，`4xx.html` 模糊匹配。
 - (1) 如果有模板引擎，默认在 `classpath:/templates/error/5xx.html`。
 - (2) 如果没有模板引擎，在静态资源文件夹下找 `5xx.html`。

2、如果模板引擎路径 `templates` 下有 `error.html` 页面，就直接渲染。

2.8.2 自定义错误响应

2.8.2.1 自定义 json 响应

使用 `@ControllerAdvice` + `@ExceptionHandler` 进行统一异常处理。

2.8.2.2 自定义页面响应

根据 boot 的错误页面规则，自定义页面模板。

2.8.3 最佳实战

- 前后分离。
 - 后台发生的所有错误，`@ControllerAdvice` + `@ExceptionHandler` 进行统一异常处理。
- 服务端页面渲染。
 - 不可预知的一些，HTTP 码表示的服务器或客户端错误。
 - 给 `classpath:/templates/error/` 下面，放常用精确的错误码页面。500.html，404.html。
 - 给 `classpath:/templates/error/` 下面，放通用模糊匹配的错误码页面。5xx.html，4xx.html。
 - 发生业务错误。
 - 核心业务，每一种错误，都应该代码控制，跳转到自己定制的错误页。
 - 通用业务，`classpath:/templates/error.html` 页面，显示错误信息。

在 HTML 页面、JSON 数据中，可用的 Model 数据如下。

```
model = {Collections$UnmodifiableMap@8803} "{timestamp=Thu Sep 21 18:21:24 CST 2023, status=5...视图
  m = {LinkedHashMap@8814} "{timestamp=Thu Sep 21 18:21:24 CST 2023, status=500, error=Intern...视图
    head = {LinkedHashMap$Entry@8816} "timestamp=Thu Sep 21 18:21:24 CST 2023"
    tail = {LinkedHashMap$Entry@8817} "path=/userList"
    accessOrder = false
    table = {HashMap$Node[16]@8818}
    entrySet = {LinkedHashMap$LinkedEntrySet@8819} "[timestamp=Thu Sep 21 18:21:24 CST 2023, ...视图
    size = 6
    modCount = 8
    threshold = 12
    loadFactor = 0.75
    keySet = null
    values = null
  keySet = null
  entrySet = null
  values = null
```

设置值 F2 创建呈现器

2.9 嵌入式容器

Servlet 容器：管理、运行 **Servlet 组件**（Servlet、Filter、Listener）的环境，一般指**服务器**。

2.9.1 自动配置原理

- SpringBoot 默认嵌入 Tomcat 作为 Servlet 容器。
- **自动配置类**是 **ServletWebServerFactoryAutoConfiguration**, **EmbeddedWebServerFactoryCustomizerAutoConfiguration**。
- 自动配置类开始分析功能，**xxxAutoConfiguration**。

```

@Configuration
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@ConditionalOnClass(ServletRequest.class)
@ConditionalOnWebApplication(type = Type.SERVLET)
@EnableConfigurationProperties(ServerProperties.class)
@Import({ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar.class,
    ServletWebServerFactoryConfiguration.EmbeddedTomcat.class,
    ServletWebServerFactoryConfiguration.EmbeddedJetty.class,
    ServletWebServerFactoryConfiguration.EmbeddedUndertow.class})
public class ServletWebServerFactoryAutoConfiguration
{

}

```

- 1、`ServletWebServerFactoryAutoConfiguration` 自动配置了嵌入式容器场景。
- 2、绑定了 `ServerProperties` 配置类，所有和服务有关的配置 `server`。
- 3、`ServletWebServerFactoryAutoConfiguration` 导入了嵌入式的三大服务器 `Tomcat`、`Jetty`、`Undertow`。
 - ① 导入 `Tomcat`、`Jetty`、`Undertow` 都有条件注解，系统中有这个类才行（也就是导了包）。
 - ② 默认 `Tomcat` 配置生效。给容器中放 `TomcatServletWebServerFactory`。
 - ③ 都给容器中 `ServletWebServerFactory` 放了一个 **Web 服务器工厂**（造 **Web 服务器**的）。
 - ④ **Web 服务器工厂**都有一个功能，`getWebServer` 获取 Web 服务器。
 - ⑤ `TomcatServletWebServerFactory` 创建了 `tomcat`。
- 4、`ServletWebServerFactory` 什么时候会创建 `webServer` 出来。
- 5、`ServletWebServerApplicationContext` ioc 容器，启动的时候会调用创建 Web 服务器。
- 6、Spring **容器刷新（启动）**的时候，会预留一个时机，刷新子容器，`onRefresh()`。
- 7、`refresh()` 容器刷新，十二个步骤的刷新，子容器会调用 `onRefresh()`。

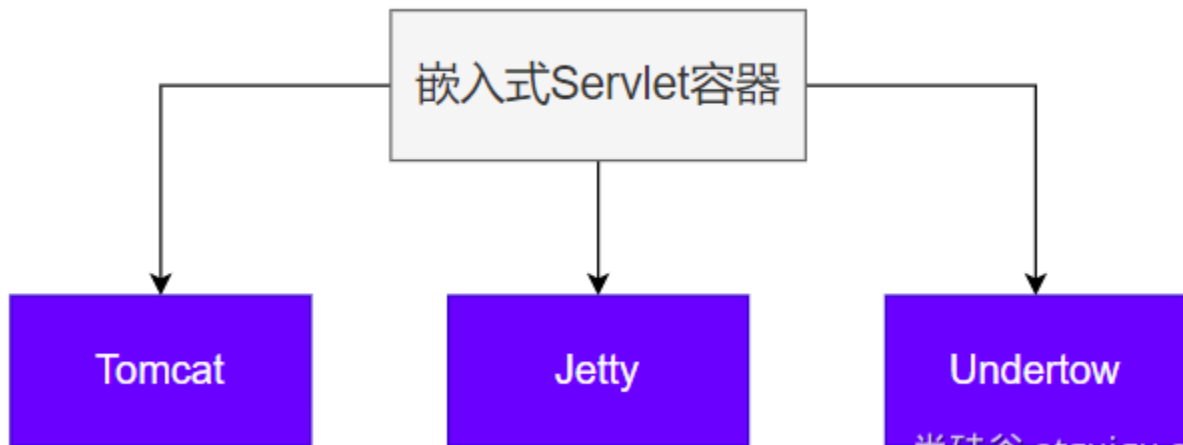

```
@Override
protected void onRefresh()
{
    super.onRefresh();

    try
    {
        createWebServer();
    }
    catch (Throwable ex)
    {
        throw new ApplicationContextException("Unable to start web server", ex);
    }
}
```

Web 场景的 Spring 容器启动，在 onRefresh 的时候，会调用创建 Web 服务器的方法。

Web 服务器的创建是通过 WebServerFactory 搞定的。容器中又会根据导了什么包条件注解，启动相关的服务器配置，默认 **EmbeddedTomcat** 会给容器中放一个 **TomcatServletWebServerFactory**，导致项目启动，自动创建出 Tomcat。

2.9.2 自定义



尚硅谷 atguigu.com

切换服务器。

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <!-- 排除 Tomcat 依赖项 -->
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<!-- 改为使用 Jetty -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>

```

2.9.3 最佳实践

用法：

- 修改 `server` 下的相关配置就可以修改**服务器参数**。
- 通过给容器中放一个 `ServletWebServerFactory`，来禁用掉 SpringBoot 默认放的服务器工厂，实现自定义嵌入任意**服务器**。

2.10 全面接管 SpringMVC

- SpringBoot 默认配置好了 SpringMVC 的所有常用特性。
- 如果需要全面接管 SpringMVC 的所有配置并**禁用默认配置**，仅需要编写一个 **WebMvcConfigurer** 配置类，并标注 **@EnableWebMvc** 即可。
- 全手动模式。
 - **@EnableWebMvc**：禁用默认配置。
 - **WebMvcConfigurer** 组件：定义 MVC 的底层行为。

2.10.1 WebMvcAutoConfiguration 到底自动配置了哪些规则

SpringMVC 自动配置场景配置了如下所有**默认行为**。

- 1、`WebMvcAutoConfiguration` Web 场景的自动配置类。

- ① 支持 RESTful 的 filter : HiddenHttpMethodFilter。
- ② 支持非 POST 请求，请求体携带数据 : FormContentFilter。
- ③ 导入 EnableWebMvcConfiguration :
 - (1) RequestMappingHandlerAdapter 。
 - (2) WelcomePageHandlerMapping : **欢迎页功能支持**（模板引擎目录、静态资源目录放 index.html），项目访问 / 就默认展示这个页面。
 - (3) RequestMappingHandlerMapping : 找每个请求由谁处理的映射关系。
 - (4) ExceptionHandlerExceptionResolver : 默认的异常解析器。
 - (5) LocaleResolver : 国际化解析器。
 - (6) ThemeResolver : 主题解析器。
 - (7) FlashMapManager : 临时数据共享。
 - (8) FormattingConversionService : 数据格式化、类型转化。
 - (9) Validator : 数据校验 JSR303 提供的数据校验功能。
 - (10) WebBindingInitializer : 请求参数的封装与绑定。
 - (11) ContentNegotiationManager : 内容协商管理器。
- ④ WebMvcAutoConfigurationAdapter 配置生效，它是一个 WebMvcConfigurer，定义 mvc 底层组件。
 - (1) 定义好 WebMvcConfigurer **底层组件默认功能**，所有功能详见列表。
 - (2) 视图解析器 : InternalResourceViewResolver 。
 - (3) 视图解析器 : BeanNameViewResolver，**视图名**（controller 方法的返回值字符串）就是组件名。
 - (4) 内容协商解析器 : ContentNegotiatingViewResolver 。
 - (5) 请求上下文过滤器 : RequestContextFilter，任意位置直接获取当前请求。
 - (6) 静态资源链规则。
 - (7) ProblemDetailsExceptionHandler : 错误详情。
 - [1] SpringMVC 内部场景异常被它捕获。
- ⑤ 定义了 MVC 默认的底层行为: WebMvcConfigurer 。

2.10.2 @EnableWebMvc 禁用默认行为

- 1、@EnableWebMvc 给容器中导入 DelegatingWebMvcConfiguration 组件，是 WebMvcConfigurationSupport 。
- 2、WebMvcAutoConfiguration 有一个核心的条件注解，@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)，容器中没有

`WebMvcConfigurationSupport` , `WebMvcAutoConfiguration` 才生效。

3、`@EnableWebMvc` 导入 `WebMvcConfigurationSupport` 导致 `WebMvcAutoConfiguration` 失效，导致禁用了默认行为。

`@EnableWebMvc` 禁用了 `Mvc` 的自动配置。

`WebMvcConfigurer` 定义 `SpringMVC` 底层组件的功能类。

2.10.3 WebMvcConfigurer 功能

定义扩展 `SpringMVC` 底层功能。

| 提供方法 | 核心参数 | |
|--|--|--|
| <code>addFormatters</code> | <code>FormatterRegistry</code> | 格式化器 ：支持属的数据类型转换。 |
| <code>getValidator</code> | 无 | 数据校验 ：校验 <code>C</code> 标注的参数合法性 |
| <code>addInterceptors</code> | <code>InterceptorRegistry</code> | 拦截器 ：拦截收到 |
| <code>configureContentNegotiation</code> | <code>ContentNegotiationConfigurer</code> | 内容协商 ：支持多 <code>HttpMessageConver</code> |
| <code>configureMessageConverters</code> | <code>List<HttpMessageConverter<?>></code> | 消息转换器 ：标注 <code>MessageConverter</code> |
| <code>addViewControllers</code> | <code>ViewControllerRegistry</code> | 视图映射 ：直接将业务逻辑的直接视 |
| <code>configureViewResolvers</code> | <code>ViewResolverRegistry</code> | 视图解析器 ：逻辑 |
| <code>addResourceHandlers</code> | <code>ResourceHandlerRegistry</code> | 静态资源处理 ：静 |
| <code>configureDefaultServletHandling</code> | <code>DefaultServletHandlerConfigurer</code> | 默认 Servlet ：可 <code>DispatcherServlet</code> |
| <code>configurePathMatch</code> | <code>PathMatchConfigurer</code> | 路径匹配 ：自定义 路径匹配，可以自 |

| 提供方法 | 核心参数 | |
|---|--|----------|
| <code>configureAsyncSupport</code> | <code>AsyncSupportConfigurer</code> | 异步支持： |
| <code>addCorsMappings</code> | <code>CorsRegistry</code> | 跨域： |
| <code>addArgumentResolvers</code> | <code>List<HandlerMethodArgumentResolver></code> | 参数解析器： |
| <code>addReturnValueHandlers</code> | <code>List<HandlerMethodReturnValueHandler></code> | 返回值解析器： |
| <code>configureHandlerExceptionResolvers</code> | <code>List</code> | 异常处理器： |
| <code>getMessageCodesResolver</code> | 无 | 消息码解析器：国 |

2.11 最佳实践

SpringBoot 已经默认配置好了 **Web 开发场景**常用功能，直接使用即可。

2.11.1 三种方式

| 方式 | 用法 | 效果 |
|-----|---|--|
| 全自动 | 直接编写控制器逻辑。 | 全部使用 自动配置默认效果 。 |
| 半自动 | 注解 <code>@Configuration</code> + 配置 <code>WebMvcConfigurer</code> + 配置 <code>WebMvcRegistrations</code> ， 不要标注 注解 <code>@EnableWebMvc</code> 。 | 保留自动配置效果，手动设置部分功能 ，定义 MVC 底层组件。 |
| 全手动 | 注解 <code>@Configuration</code> + 配置 <code>WebMvcConfigurer</code> ， 标注注解 <code>@EnableWebMvc</code> 。 | 禁用自动配置效果，全手动设置 。 |

总结：

给容器中写一个配置类 `@Configuration` **实现** `WebMvcConfigurer` **但是不要标注** `@EnableWebMvc` 注解，**实现半自动的效果**。

2.11.2 两种模式

- 1、前后分离模式：@RestController 响应 JSON 数据。
- 2、前后不分离模式：@Controller + Thymeleaf 模板引擎。

2.12 Web 新特性

2.12.1 ProblemDetails

RFC 7807: <https://www.rfc-editor.org/rfc/rfc7807>

错误信息返回新格式。

原理：

```
@Configuration(proxyBeanMethods = false)
// 配置一个属性 spring.mvc.problemDetails.enabled=true
@ConditionalOnProperty(prefix = "spring.mvc.problemDetails", name = "enabled", havingValue = "true")
static class ProblemDetailsErrorHandlingConfiguration
{
    @Bean
    @ConditionalOnMissingBean(ResponseEntityExceptionHandler.class)
    ProblemDetailsExceptionHandler problemDetailsExceptionHandler()
    {
        return new ProblemDetailsExceptionHandler();
    }
}
```

- 1、ProblemDetailsExceptionHandler 是一个 @ControllerAdvice 集中处理系统异常。
- 2、处理以下异常。如果系统出现以下异常，会被 SpringBoot 支持以 RFC 7807 规范方式返回错误数据。

```

@ExceptionHandler({
    // 请求方式不支持
    HttpRequestMethodNotSupportedException.class,
    HttpMediaTypeNotSupportedException.class,
    HttpMediaTypeNotAcceptableException.class,
    MissingPathVariableException.class,
    MissingServletRequestParameterException.class,
    MissingServletRequestPartException.class,
    ServletRequestBindingException.class,
    MethodArgumentNotValidException.class,
    NoHandlerFoundException.class,
    AsyncRequestTimeoutException.class,
    ErrorResponseException.class,
    ConversionNotSupportedException.class,
    TypeMismatchException.class,
    HttpMessageNotReadableException.class,
    HttpMessageNotWritableException.class,
    BindException.class
})

```

效果：

默认响应错误的 json。状态码 405。

```

{
  "timestamp": "2023-09-18T11:13:05.515+00:00",
  "status": 405,
  "error": "Method Not Allowed",
  "trace": "org.springframework.web.HttpRequestMethodNotSupportedException: Request method 'POST'",
  "message": "Method 'POST' is not supported.",
  "path": "/list"
}

```

开启 ProblemDetails 返回，使用新的 MediaType

Content-Type: application/problem+json + 额外扩展返回。

2.12.2 函数式 Web

SpringMVC 5.2 以后允许使用函数式的方式，定义 Web 的请求处理流程。

函数式接口。

Web 请求处理的方式：

- 1、**@Controller + @RequestMapping**：耦合式（路由、**业务耦合**）。
- 2、**函数式 Web**：分离式（路由、业务分离）。

2.12.2.1 场景

场景：User RESTful - CRUD。

- GET /user/1 获取 1 号用户。
- GET /users 获取所有用户。
- POST /user **请求体**携带 JSON，新增一个用户。
- PUT /user/1 **请求体**携带 JSON，修改 1 号用户。
- DELETE /user/1 **删除** 1 号用户。

2.12.2.2 核心类

- RouterFunction
- RequestPredicate
- ServerRequest
- ServerResponse

2.12.2.3 示例

```

package com.myxh.springboot.web.config;

import com.myxh.springboot.web.biz.UserBizHandler;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.MediaType;
import org.springframework.web.servlet.function.RequestPredicates;
import org.springframework.web.servlet.function.RouterFunction;
import org.springframework.web.servlet.function.RouterFunctions;
import org.springframework.web.servlet.function.ServerResponse;

/**
 * @author MYXH
 * @date 2023/9/18
 * @description
 * 场景: User RESTful - CRUD
 * GET /user/1 获取 1 号用户
 * GET /users 获取所有用户
 * POST /user 请求体携带 JSON, 新增一个用户
 * PUT /user/1 请求体携带 JSON, 修改 1 号用户
 * DELETE /user/1 删除 1 号用户
 */
@Configuration
public class WebFunctionConfig
{
    /**
     * 函数式 Web
     * 1、给容器中放一个 Bean: 类型是 RouterFunction<ServerResponse>, 集中所有路由信息
     * 2、每个业务准备一个自己的 Handler
     * <br>
     * 核心四大对象
     * 1、RouterFunction: 定义路由信息, 发什么请求, 谁来处理
     * 2、RequestPredicate: 定义请求, 请求谓语, 请求方式 (GET、POST), 请求参数
     * 3、ServerRequest: 封装请求完整数据
     * 4、ServerResponse: 封装响应完整数据
     *
     * @param userBizHandler 用户业务处理程序 (userBizHandler 会被自动注入进来)
     * @return routerFunction 路由器功能
     */
    @Bean
    public RouterFunction<ServerResponse> userRouter(UserBizHandler userBizHandler)
    {
        // 开始定义路由信息
        RouterFunction<ServerResponse> routerFunction = RouterFunctions.route()

```

```
        .GET("/user/{id}", RequestPredicates.accept(MediaType.ALL), userBizHandler::getUser)
        .GET("/users", userBizHandler::getUsers)
        .POST("/user", RequestPredicates.accept(MediaType.APPLICATION_JSON), userBizHandler::createUser)
        .PUT("/user{id}", RequestPredicates.accept(MediaType.APPLICATION_JSON), userBizHandler::updateUser)
        .DELETE("/user{id}", userBizHandler::deleteUser)
        .build();

    return routerFunction;
}
}
```

```
package com.myxh.springboot.web.biz;

import com.myxh.springboot.web.bean.User;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.servlet.function.ServerRequest;
import org.springframework.web.servlet.function.ServerResponse;

import java.util.Arrays;
import java.util.List;

/**
 * @author MYXH
 * @date 2023/9/18
 * @description 专门处理 User 有关的业务
 */
@Slf4j
@Service
public class UserBizHandler
{
    @Autowired
    private User user;

    /**
     * 查询指定 id 的用户
     *
     * @param request 请求
     * @return response 响应
     * @throws Exception 异常
     */
    public ServerResponse getUser(ServerRequest request) throws Exception
    {
        // 业务处理
        String id = request.pathVariable("id");

        user.setId(1L);
        user.setUserName("MYXH");
        user.setPassword("520.ILY!");
        user.setAge(21);
        user.setEmail("1735350920@qq.com");

        log.info("查询 {} 号用户信息成功", id);
    }
}
```

```

        // 构造响应
        ServerResponse response = ServerResponse
            .ok()
            // 凡是 body 中的对象, 就是以前 @ResponseBody 原理, 利用 HttpMessageConverter 写出为
            .body(user);

        return response;
    }

    /**
     * 获取所有用户
     *
     * @param request 请求
     * @return response 响应
     * @throws Exception 异常
     */
    public ServerResponse getUsers(ServerRequest request) throws Exception
    {
        // 业务处理
        List<User> userList = Arrays.asList(
            new User(1L, "MYXH", "520.ILY!", 21, "1735350920@qq.com", "root"),
            new User(2L, "root", "000000", 21, "root@qq.com", "root"),
            new User(3L, "admin", "123456", 21, "admin@qq.com", "admin"),
            new User(4L, "test", "test", 18, "test@qq.com", "test"),
            new User(5L, "张三", "123456", 18, "", "user")
        );

        log.info("查询所有用户信息成功");

        // 构造响应
        ServerResponse response = ServerResponse
            .ok()
            .body(userList);

        return response;
    }

    /**
     * 保存用户
     *
     * @param request 请求
     * @return response 响应
     * @throws Exception 异常
     */

```

```
public ServerResponse saveUser(ServerRequest request) throws Exception
{
    // 业务处理
    // 提取请求体
    User body = request.body(User.class);

    log.info("保存用户信息成功, 用户信息: {}", body);

    // 构造响应
    ServerResponse response = ServerResponse
        .ok()
        .build();

    return response;
}

/**
 * 更新指定 id 的用户
 *
 * @param request 请求
 * @return response 响应
 * @throws Exception 异常
 */
public ServerResponse updateUser(ServerRequest request) throws Exception
{
    // 业务处理
    // 提取请求体
    User body = request.body(User.class);

    log.info("更新用户信息成功, 用户信息: {}", body);

    // 构造响应
    ServerResponse response = ServerResponse
        .ok()
        .build();

    return response;
}

/**
 * 删除指定 id 的用户
 *
 * @param request 请求
 * @return response 响应
 */
```

```
    * @throws Exception 异常
    */
    public ServerResponse deleteUser(ServerRequest request) throws Exception
    {
        // 业务处理
        String id = request.pathVariable("id");

        log.info("删除 {} 号用户信息成功", id);

        // 构造响应
        ServerResponse response = ServerResponse
            .ok()
            .build();

        return response;
    }
}
```

第 3 章 SpringBoot3-数据访问

整合 SSM 场景

SpringBoot 整合 **Spring**、**SpringMVC**、**MyBatis** 进行数据访问场景开发。

3.1 创建 SSM 整合项目


```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.4</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.myxh.springboot</groupId>
    <artifactId>boot3-05-ssm</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>boot3-05-ssm</name>
    <description>boot3-05-ssm</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>3.0.2</version>
        </dependency>

        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

```

```

        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter-test</artifactId>
            <version>3.0.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>

```

3.2 配置数据源

```

# 1、先配置数据源信息
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.type=com.zaxxer.hikari.HikariDataSource
spring.datasource.url=jdbc:mysql://localhost:3306/spring_boot
spring.datasource.username=MYXH
spring.datasource.password=520.ILY!

```

安装 Free MyBatis Tool 或 MyBatisX 插件，生成 Mapper 接口的 xml 文件即可。

3.3 配置 MyBatis

```
# 2、配置整合 MyBatis
mybatis.mapper-locations=classpath:/mapper/*.xml

# 打开驼峰命名规则
mybatis.configuration.map-underscore-to-camel-case=true
```

3.4 CRUD 编写

- 编写 Bean。
- 编写 Mapper。
- 使用 `Free MyBatis Tool` 插件，快速生成 MapperXML。
- 测试 CRUD。

3.5 自动配置原理

SSM 整合总结：

- 1、导入 `mybatis-spring-boot-starter`。
- 2、配置数据源信息。
- 3、配置 MyBatis 的 **Mapper 接口扫描**与 **xml 映射文件扫描**。
- 4、编写 Bean, Mapper, 生成 xml, 编写 SQL 进行 CRUD。**事务**等操作依然和 **Spring** 中用法一样。

5、效果：

- ① 所有 SQL 写在 xml 中。
- ② 所有 **MyBatis** 配置写在 `application.properties` 下面。
- `jdbc` 场景的自动配置：
 - `mybatis-spring-boot-starter` 导入 `spring-boot-starter-jdbc`，`jdbc` 是操作数据库的场景。
 - `jdbc` 场景的几个自动配置。
 - `org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration`

- 数据源的自动配置。
 - 所有和数据源有关的配置都绑定在 `DataSourceProperties` 。
 - 默认使用 `HikariDataSource` 。
- `org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration`
 - 给容器中放了 `JdbcTemplate` 操作数据库。
- `org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration`
- `org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration`
 - 基于 XA 二阶提交协议的分布式事务数据源。
- `org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration`
 - 支持事务。
- 具有的底层能力：数据源、`JdbcTemplate`、事务。
- `MyBatisAutoConfiguration`：配置了 MyBatis 的整合流程。
 - `mybatis-spring-boot-starter` 导入 `mybatis-spring-boot-autoconfigure` (mybatis 的自动配置包) 。
 - 默认加载两个自动配置类：
 - `org.mybatis.spring.boot.autoconfigure.MybatisLanguageDriverAutoConfiguration`
 - `org.mybatis.spring.boot.autoconfigure.MybatisAutoConfiguration`
 - 必须在数据源配置好之后才配置。
 - 给容器中 `SqlSessionFactory` 组件，创建和数据库的一次会话。
 - 给容器中 `SqlSessionTemplate` 组件，操作数据库。
 - MyBatis 的所有配置绑定在 `MybatisProperties` 。
 - 每个 Mapper 接口的代理对象是怎么创建放到容器中，详见 **@MapperScan** 原理：
 - 利用 `@Import(MapperScannerRegistrar.class)` 批量给容器中注册组件，解析指定的包路径里面的每一个类，为每一个 Mapper 接口类，创建 Bean 定义信息，注册到容器中。

如何分析哪个场景导入以后，开启了哪些自动配置类。

寻找：`classpath:/META-INF/spring/`

`org.springframework.boot.autoconfigure.AutoConfiguration.imports` 文件中配置的所有值，就是要开启的自动配置类，但是每个类可能有条件注解，基于条件注解判断哪个自动配置类生效了。

3.6 快速定位生效的配置

```
# 开启调试模式，详细打印开启了哪些自动配置
# Positive（生效的自动配置），Negative（不生效的自动配置）
debug=true
```

3.7 扩展：整合其他数据源

3.7.1 Druid 数据源

暂不支持 **SpringBoot3**

- 导入 **druid-starter**。
- 写配置。
- 分析自动配置了哪些东西，怎么用。

Druid 官网：<https://github.com/alibaba/druid>

数据源基本配置

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.url=jdbc:mysql://localhost:3306/spring_boot
spring.datasource.username=MYXH
spring.datasource.password=520.ILY!
```

配置 StatFilter 监控

```
spring.datasource.druid.filter.stat.enabled=true
spring.datasource.druid.filter.stat.db-type=mysql
spring.datasource.druid.filter.stat.log-slow-sql=true
spring.datasource.druid.filter.stat.slow-sql-millis=2000
```

配置 WallFilter 防火墙

```
spring.datasource.druid.filter.wall.enabled=true
spring.datasource.druid.filter.wall.db-type=mysql
spring.datasource.druid.filter.wall.config.delete-allow=false
spring.datasource.druid.filter.wall.config.drop-table-allow=false
```

配置监控页，内置监控页面的首页是 /druid/index.html

```
spring.datasource.druid.stat-view-servlet.enabled=true
spring.datasource.druid.stat-view-servlet.login-username=admin
spring.datasource.druid.stat-view-servlet.login-password=admin
spring.datasource.druid.stat-view-servlet.allow=\\*
```

其他 Filter 配置不再演示

目前为以下 Filter 提供了配置支持，请参考文档或者根据 IDE 提示 (spring.datasource.druid.filter.*) 进行

StatFilter

WallFilter

ConfigFilter

EncodingConvertFilter

Slf4jLogFilter

Log4jFilter

Log4j2Filter

CommonsLogFilter

3.8 附录：示例数据库

```
# 创建数据库 spring_boot
CREATE DATABASE IF NOT EXISTS spring_boot;

# 选择数据库 spring_boot
USE spring_boot;

# 创建用户表 t_user

CREATE TABLE `t_user`
(
    `id`          BIGINT(20)    NOT NULL AUTO_INCREMENT COMMENT '编号',
    `login_name`  VARCHAR(200)  NULL DEFAULT NULL COMMENT '用户名称' COLLATE 'utf8_general_ci',
    `nick_name`   VARCHAR(200)  NULL DEFAULT NULL COMMENT '用户昵称' COLLATE 'utf8_general_ci',
    `password`    VARCHAR(200)  NULL DEFAULT NULL COMMENT '用户密码' COLLATE 'utf8_general_ci',
    PRIMARY KEY (`id`)
);

# 添加用户表 t_user 的数据
INSERT INTO `t_user`(`id`, `login_name`, `nick_name`, `password`)
VALUES (1, 'MYXH', '末影小黑xh', '520.ILY!'),
       (2, 'root', 'root 用户', '000000'),
       (3, 'admin', '管理员', '123456'),
       (4, 'test', '测试员', 'test');
```

第 4 章 SpringBoot3-基础特性

4.1 SpringApplication

4.1.1 自定义 banner

- 1、类路径添加 `banner.txt` 或设置 `spring.banner.location` 就可以定制 banner。
- 2、推荐网站：链接：[Spring Boot banner 在线生成工具](#)，制作下载英文 banner.txt，修改替换 banner.txt 文字实现自定义，个性化启动。

4.1.2 自定义 SpringApplication

```
package com.myxh.springboot.features;

import org.springframework.boot.Banner;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * 主程序类
 */
@SpringBootApplication
public class Boot306FeaturesApplication
{
    public static void main(String[] args)
    {
        // 1、SpringApplication: Boot 应用的核心 API 入口
        // SpringApplication.run(Boot306FeaturesApplication.class, args);

        // 1.1 自定义 SpringApplication 的底层设置
        SpringApplication springApplication = new SpringApplication(Boot306FeaturesApplication.class);

        // 1.2 程序化调整 SpringApplication 的参数，配置文件优先级高于程序化调整的优先级
        springApplication.setBannerMode(Banner.Mode.OFF);

        // 1.3 SpringApplication 运行起来
        springApplication.run(args);
    }
}
```


4.1.3 FluentBuilder API

```
package com.myxh.springboot.features;

import org.springframework.boot.Banner;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.context.ConfigurableApplicationContext;

/**
 * 主程序类
 */
@SpringBootApplication
public class Boot306FeaturesApplication
{
    public static void main(String[] args)
    {
        // 1、SpringApplication: Boot 应用的核心 API 入口
        // SpringApplication.run(Boot306FeaturesApplication.class, args);

        // 1.1 自定义 SpringApplication 的底层设置
        // SpringApplication springApplication = new SpringApplication(Boot306FeaturesApplication.class);

        // 1.2 程序化调整 SpringApplication 的参数, 配置文件优先级高于程序化调整的优先级
        // springApplication.setBannerMode(Banner.Mode.OFF);

        // 1.3 SpringApplication 运行起来
        // springApplication.run(args);

        // 2、Builder 方式构建 SpringApplication: 通过 FluentAPI 进行设置
        ConfigurableApplicationContext applicationContext = new SpringApplicationBuilder()
            .main(Boot306FeaturesApplication.class)
            .sources(Boot306FeaturesApplication.class)
            .bannerMode(Banner.Mode.OFF)
            .run(args);
    }
}
```

4.2 Profiles

环境隔离能力, 快速切换开发、测试、生产环境。

步骤:

- 1、**标识环境**：指定哪些组件、配置在哪个环境生效。
- 2、**切换环境**：这个环境对应的所有组件和配置就应该生效。

4.2.1 使用

4.2.1.1 指定环境

- Spring Profiles 提供一种**隔离配置**的方式，使其仅在**特定环境**生效。
- 任何 `@Component`，`@Configuration` 或 `@ConfigurationProperties` 可以使用 `@Profile` 标记，来指定何时被加载。（容器中的**组件**都可以被 `@Profile` 标记）

4.2.1.2 环境激活

- 1、配置激活指定环境，配置文件。

```
# 激活指定的一个或多个环境
spring.profiles.active=dev, test
```

- 2、也可以使用命令行激活。 `--spring.profiles.active=dev, test`。

- 3、还可以配置**默认环境**，不标注 `@Profile` 的组件永远都存在。

- ① 以前默认环境叫 default。
- ② `spring.profiles.default=dev`。

- 4、推荐使用激活方式激活指定环境。

4.2.1.3 环境包含

注意：

- 1、`spring.profiles.active` 和 `spring.profiles.default` 只能用到**无 profile**的文件中，如果在 `application-dev.properties/yaml` 中编写就是无效的。

- 2、也可以额外添加生效文件，而不是激活替换。比如：

```
# 包含指定环境，不管激活哪个环境，这个环境都有，总是要生效的环境
spring.profiles.include=dev, test
```

最佳实战：

- 生效的环境 = 激活的环境/默认环境 + 包含的环境。
- 项目里面这么用。
 - 基础的配置 `mybatis`、`log` : 写到包含环境中。
 - 需要动态切换变化的 `db`、`redis` : 写到激活的环境中。

4.2.2 Profile 分组

创建 `prod` 组, 指定包含 `db` 和 `mq` 配置。

```
# Profile 分组
spring.profiles.group.prod[0]=db
spring.profiles.group.prod[1]=mq
```

使用 `--spring.profiles.active=prod`, 就会激活 `prod`, `db`, `mq` 配置文件。

4.2.3 Profile 配置文件

- `application-{profile}.properties` 可以作为指定环境的配置文件。
- 激活这个环境, 配置就会生效。最终生效的所有配置是。
 - `application.properties` : 主配置文件, 任意时候都生效。
 - `application-{profile}.properties` : 指定环境配置文件, 激活指定环境生效。

profile 优先级 > application 优先级。

4.3 外部化配置

场景 : 线上应用如何快速修改配置, 并应用最新配置 ?

SpringBoot 使用配置优先级 + 外部配置简化配置更新、简化运维。

只需要给 `jar` 应用所在的文件夹放一个 `application.properties` 最新配置文件, 重启项目就能自动应用最新配置。

4.3.1 配置优先级

Spring Boot 允许将配置外部化, 以便可以在不同的环境中使用相同的应用程序代码。

可以使用各种外部配置源, 包括 `Java Properties` 文件、`YAML` 文件、环境变量和命令行参数。

`@Value` 可以获取值，也可以用 `@ConfigurationProperties` 将所有属性绑定到 `Java Object` 中。

以下是 **SpringBoot** 属性源加载顺序，后面的会覆盖前面的值，由低到高，高优先级配置覆盖低优先级。

- 1、**默认属性**（通过 `SpringApplication.setDefaultProperties` 指定的）。
- 2、`@PropertySource` 指定加载的配置（需要写在 `@Configuration` 类上才可生效）。
- 3、**配置文件**（`application.properties/yml` 等）。
- 4、`RandomValuePropertySource` 支持的 `random.*` 配置（如：`@Value("${random.int}")`）。
- 5、OS 环境变量。
- 6、Java 系统属性（`System.getProperties()`）。
- 7、JNDI 属性（来自 `java:comp/env`）。
- 8、`ServletContext` 初始化参数。
- 9、`ServletConfig` 初始化参数。
- 10、`SPRING_APPLICATION_JSON` 属性（内置在环境变量或系统属性中的 JSON）。
- 11、**命令行参数**。
- 12、测试属性。（`@SpringBootTest` 进行测试时指定的属性）。
- 13、测试类 `@TestPropertySource` 注解。
- 14、Devtools 设置的全局属性。（`$HOME/.config/spring-boot`）。

结论：配置可以写到很多位置，常见的优先级顺序：

- 命令行 > 配置文件 > **springapplication 配置**

配置文件**优先级**如下：（后面覆盖前面）

- 1、**jar 包内的** `application.properties/yml`。

2、**jar 包内的** `application-{profile}.properties/yml`。

3、**jar 包外的** `application.properties/yml`。

4、**jar 包外的** `application-{profile}.properties/yml`。

建议：用一种格式的配置文件。如果 `.properties` 和 `.yml` 同时存在，则 `.properties` 优先。

结论：包外 > 包内，同级情况：**profile 配置 > application 配置。**

所有参数均可由命令行传入，使用 `--参数项=参数值`，将会被添加到环境变量中，并优先于 配置文件。

比如 `java -jar app.jar --name="Spring"`，可以使用 `@Value("${name}")` 获取。

演示场景：

- 包内：`application.properties` `server.port=8080`。
- 包内：`application-dev.properties` `server.port=8081`
- 包外：`application.properties` `server.port=8180`
- 包外：`application-dev.properties` `server.port=8181`

启动端口？：命令行 > `8181` > `8180` > `8081` > `8080`。

4.3.2 外部配置

SpringBoot 应用启动时会自动寻找 `application.properties` 和 `application.yml` 位置，进行加载。顺序如下：（后面覆盖前面）

1、类路径：内部。

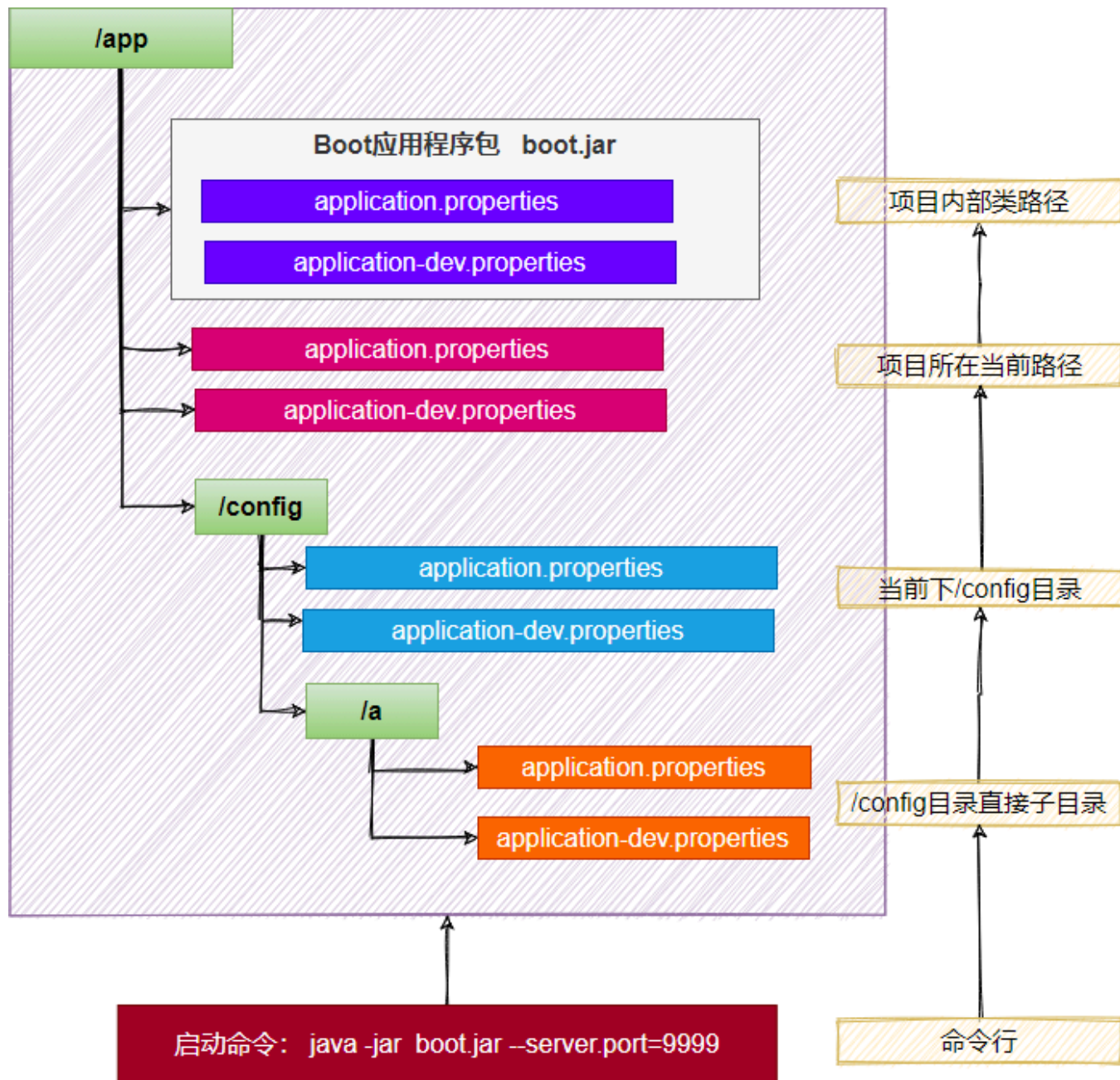
- ① 类根路径。
- ② 类下 `/config` 包。

2、当前路径：项目所在的位置。

- ① 当前路径。
- ② 当前下 `/config` 子目录。
- ③ `/config` 目录的直接子目录。

最终效果：优先级由高到低，前面覆盖后面。

- 命令行 > 包外 config 直接子目录 > 包外 config 目录 > 包外根目录 > 包内目录。
- 同级比较：
 - profile 配置 > 默认配置。
 - properties 配置 > yaml 配置。



规律：最外层的最优先。

- 命令行 > 所有。
- 包外 > 包内。
- config 目录 > 根目录。

- profile > application。

配置不同就都生效（互补），配置相同高优先级覆盖低优先级。

4.3.3 导入配置

使用 `spring.config.import` 可以导入额外配置。

```
# 导入指定的配置
spring.config.import=classpath:/a.properties

# 导入配置优先级低于配置文件的优先级
a=c
```

无论以上写法的先后顺序，`a.properties` 的值总是优先于直接在文件中编写的 `a`。

4.3.4 属性占位符

配置文件中可以使用 `${name:default}` 形式取出之前配置过的值。

```
server.port=8080

my.server.port=我的服务端口是: ${server.port}, 我的用户名是: ${my.username:末影小黑xh}
```

4.4 单元测试 JUnit5

4.4.1 整合

SpringBoot 提供一系列测试工具集及注解方便进行测试。

`spring-boot-test` 提供核心测试能力，`spring-boot-test-autoconfigure` 提供测试的一些自动配置。

只需要导入 `spring-boot-starter-test` 即可整合测试。

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

spring-boot-starter-test 默认提供了以下库供测试使用。

- JUnit 5
- Spring Test
- AssertJ
- Hamcrest
- Mockito
- JSONassert
- JsonPath

4.4.2 测试

4.4.2.1 组件测试

直接 `@Autowired` 容器中的组件进行测试。

4.4.2.2 注解

JUnit5 的注解与 JUnit4 的注解有所变化。

<https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations>

- **@Test** : 表示方法是测试方法，但是与 JUnit4 的 `@Test` 不同，他的职责非常单一不能声明任何属性，拓展的测试将会由 Jupiter 提供额外测试。
- **@ParameterizedTest** : 表示方法是参数化测试，下方会有详细介绍。
- **@RepeatedTest** : 表示方法可重复执行，下方会有详细介绍。
- **@DisplayName** : 为测试类或者测试方法设置展示名称。
- **@BeforeEach** : 表示在每个单元测试之前执行。
- **@AfterEach** : 表示在每个单元测试之后执行。
- **@BeforeAll** : 表示在所有单元测试之前执行。
- **@AfterAll** : 表示在所有单元测试之后执行。
- **@Tag** : 表示单元测试类别，类似于 JUnit4 中的 `@Categories`。
- **@Disabled** : 表示测试类或测试方法不执行，类似于 JUnit4 中的 `@Ignore`。

- **@Timeout** : 表示测试方法运行如果超过了指定时间将会返回错误。
- **@ExtendWith** : 为测试类或测试方法提供扩展类引用。

```

package com.myxh.springboot.features;

import com.myxh.springboot.features.service.HelloService;
import org.junit.jupiter.api.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

// 具备测试 SpringBoot 应用容器中所有组件的功能，测试类必须在主程序所在的包及其子包
@SpringBootTest
class Boot306FeaturesApplicationTests
{
    // 自动注入任意组件即可测试
    @Autowired
    HelloService helloService;

    @DisplayName("测试 sum() 方法 ")
    @Test
    void contextLoads()
    {
        Integer sum = helloService.sum(1, 2);
        Assertions.assertEquals(3, sum);
    }

    @DisplayName("测试1")
    @Test
    public void test1()
    {
        System.out.println("test1");
    }

    // 所有测试方法运行之前先运行这个，只打印一次
    @BeforeAll
    static void initAll()
    {
        System.out.println("hello");
    }

    // 每个测试方法运行之前先运行这个，每个方法运行打印一次
    @BeforeEach
    void init()
    {
        System.out.println("world");
    }
}

```

4.4.2.3 断言

| 方法 | 说明 |
|--------------------------|---------------------|
| assertEquals | 判断两个对象或两个原始类型是否相等。 |
| assertNotEquals | 判断两个对象或两个原始类型是否不相等。 |
| assertSame | 判断两个对象引用是否指向同一个对象。 |
| assertNotSame | 判断两个对象引用是否指向不同的对象。 |
| assertTrue | 判断给定的布尔值是否为 true。 |
| assertFalse | 判断给定的布尔值是否为 false。 |
| assertNull | 判断给定的对象引用是否为 null。 |
| assertNotNull | 判断给定的对象引用是否不为 null。 |
| assertArrayEquals | 数组断言。 |
| assertAll | 组合断言。 |
| assertThrows | 异常断言。 |
| assertTimeout | 超时断言。 |
| fail | 快速失败。 |

4.4.2.4 嵌套测试

JUnit 5 可以通过 Java 中的内部类和 @Nested 注解实现嵌套测试，从而可以更好的把相关的测试方法组织在一起。在内部类中可以使用 @BeforeEach 和 @AfterEach 注解，而且嵌套的层次没有限制。

```
package com.myxh.springboot.features;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.EmptyStackException;
import java.util.Stack;

import static org.junit.jupiter.api.Assertions.*;

/**
 * @author MYXH
 * @date 2023/9/23
 */
@SpringBootTest
@DisplayName("一个堆栈")
public class HelloTest
{
    Stack<Object> stack;

    @Test
    @DisplayName("使用 new Stack() 实例化堆栈")
    void isInstantiatedWithNew()
    {
        new Stack<>();
    }

    @Nested
    @DisplayName("当新建时")
    class WhenNew
    {
        @BeforeEach
        void createNewStack()
        {
            stack = new Stack<>();
        }

        @Test
        @DisplayName("为空")
        void isEmpty()
        {

```

```
        assertTrue(stack.isEmpty());
    }

    @Test
    @DisplayName("在弹出时抛出 EmptyStackException 异常")
    void throwsExceptionWhenPopped()
    {
        assertThrows(EmptyStackException.class, stack::pop);
    }

    @Test
    @DisplayName("在查看时抛出 EmptyStackException 异常")
    void throwsExceptionWhenPeeked()
    {
        assertThrows(EmptyStackException.class, stack::peek);
    }

    @Nested
    @DisplayName("在推入元素后")
    class AfterPushing
    {
        String anElement = "一个元素";

        @BeforeEach
        void pushAnElement()
        {
            stack.push(anElement);
        }

        @Test
        @DisplayName("它不再为空")
        void isEmpty()
        {
            assertFalse(stack.isEmpty());
        }

        @Test
        @DisplayName("当弹出元素并且为空时返回该元素")
        void returnElementWhenPopped()
        {
            assertEquals(anElement, stack.pop());
            assertTrue(stack.isEmpty());
        }
    }
}
```

```

@Test
@DisplayName("当查看元素时返回该元素，但仍然保持非空")
void returnElementWhenPeeked()
{
    assertEquals(anElement, stack.peek());
    assertFalse(stack.isEmpty());
}
}
}
}

```

4.4.2.5 参数化测试

参数化测试是 JUnit5 很重要的一个新特性，它使得用不同的参数多次运行测试成为了可能，也为单元测试带来许多便利。

利用 **@ValueSource** 等注解，指定入参，将可以使用不同的参数进行多次单元测试，而不需要每新增一个参数就新增一个单元测试，省去了很多冗余代码。

@ValueSource：为参数化测试指定入参来源，支持八大基础类以及 String 类型，Class 类型。

@NullSource：表示为参数化测试提供一个 null 的入参。

@EnumSource：表示为参数化测试提供一个枚举入参。

@CsvFileSource：表示读取指定 CSV 文件内容作为参数化测试入参。

@MethodSource：表示读取指定方法的返回值作为参数化测试入参（注意方法返回需要是一个流）。

```

package com.myxh.springboot.features;

import com.myxh.springboot.features.service.HelloService;
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;
import org.junit.jupiter.params.provider.ValueSource;
import org.junit.platform.commons.util.StringUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.stream.Stream;

// 具备测试 SpringBoot 应用容器中所有组件的功能，测试类必须在主程序所在的包及其子包
@SpringBootTest
class Boot306FeaturesApplicationTests
{
    // 自动注入任意组件即可测试
    @Autowired
    HelloService helloService;

    @DisplayName("测试 sum() 方法 ")
    @Test
    void contextLoads()
    {
        Integer sum = helloService.sum(1, 2);
        Assertions.assertEquals(3, sum);
    }

    @DisplayName("测试1")
    @Test
    public void test1()
    {
        System.out.println("test1");
    }

    // 参数化测试
    @ParameterizedTest
    @ValueSource(strings = {"one", "two", "three"})
    @DisplayName("参数化测试1")
    public void parameterizedTest1(String string)
    {
        System.out.println(string);
        Assertions.assertTrue(StringUtils.isNotBlank(string));
    }
}

```

```

    }

    @ParameterizedTest
    // 指定方法名，返回值就是测试用的参数
    @MethodSource("method")
    @DisplayName("方法来源参数")
    public void testWithExplicitLocalMethodSource(String name)
    {
        System.out.println(name);
        Assertions.assertNotNull(name);
    }

    static Stream<String> method()
    {
        // 返回 Stream 则可
        return Stream.of("apple", "banana");
    }

    // 所有测试方法运行之前先运行这个，只打印一次
    @BeforeAll
    static void initAll()
    {
        System.out.println("hello");
    }

    // 每个测试方法运行之前先运行这个，每个方法运行打印一次
    @BeforeEach
    void init()
    {
        System.out.println("world");
    }
}

```

第 5 章 SpringBoot3-核心原理

5.1 事件和监听器

5.1.1 生命周期监听

场景：监听应用的生命周期。

5.1.1.1 监听器 SpringApplicationRunListener

1、自定义 `SpringApplicationRunListener` 来监听事件。

- ① 编写 `SpringApplicationRunListener` 实现类。

```

package com.myxh.springboot.core.listener;

import org.springframework.boot.ConfigurableBootstrapContext;
import org.springframework.boot.SpringApplicationRunListener;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.core.env.ConfigurableEnvironment;

import java.time.Duration;

/**
 * @author MYXH
 * @date 2023/9/24
 * @description
 * SpringBoot 应用生命周期监听
 * <br>
 * Listener先要从 META-INF/spring.factories 读到
 * <br>
 * 1、引导: 利用 BootstrapContext 引导整个项目启动
 *     starting:          应用开始, SpringApplication 的 run 方法一调用, 只要有
 *     environmentPrepared: 环境准备好 (把启动参数等绑定到环境变量中), 但是 ioc 还
 * 2、启动:
 *     contextPrepared:    ioc 容器创建并准备好, 但是 sources (主配置类) 没加载, :
 *     contextLoaded:      ioc 容器加载, 主配置类加载进去了, 但是 ioc 容器还没刷新
 *     ----- 截止现在, ioc 容器里面还没造 bean -----
 *     started:            ioc 容器刷新了 (所有 bean 造好了), 但是 runner 没调用
 *     ready:              ioc 容器刷新了 (所有 bean 造好了), 所有 runner 调用完
 * 3、运行:
 *     以上步骤都正确执行, 代表容器 running。
 */
public class MyAppListener implements SpringApplicationRunListener
{
    @Override
    public void starting(ConfigurableBootstrapContext bootstrapContext)
    {
        System.out.println("----- starting() 正在启动 -----");
    }

    @Override
    public void environmentPrepared(ConfigurableBootstrapContext bootstrapContext, ConfigurableEnvironment environment)
    {
        System.out.println("----- environmentPrepared() 环境准备完成 -----");
    }

    @Override

```

```
public void contextPrepared(ConfigurableApplicationContext context)
{
    System.out.println("----- contextPrepared() ioc 容器准备完成 -----");
}

@Override
public void contextLoaded(ConfigurableApplicationContext context)
{
    System.out.println("----- contextLoaded() ioc 容器加载完成 -----");
}

@Override
public void started(ConfigurableApplicationContext context, Duration timeTaken)
{
    System.out.println("----- started() 启动完成 -----");
}

@Override
public void ready(ConfigurableApplicationContext context, Duration timeTaken)
{
    System.out.println("----- ready() 准备就绪 -----");
}

@Override
public void failed(ConfigurableApplicationContext context, Throwable exception)
{
    System.out.println("----- failed() 应用启动失败 -----");
}
}
```

```

package com.myxh.springboot.core.listener;

import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;

/**
 * @author MYXH
 * @date 2023/9/24
 */
public class MyListener implements ApplicationListener<ApplicationEvent>
{
    @Override
    public void onApplicationEvent(ApplicationEvent event)
    {
        System.out.println("----- 事件到达 -----");
        System.out.println("event = " + event);
        System.out.println("-----");
    }
}

```

- ② 在 META-INF/spring.factories 中配置

org.springframework.boot.SpringApplicationRunListener=自己的 Listener，还可以指定一个有参构造器，接受两个参数 (SpringApplication application, String[] args)。

```

org.springframework.boot.SpringApplicationRunListener=\
com.myxh.springboot.core.listener.MyAppListener

org.springframework.context.ApplicationListener=\
com.myxh.springboot.core.listener.MyListener

```

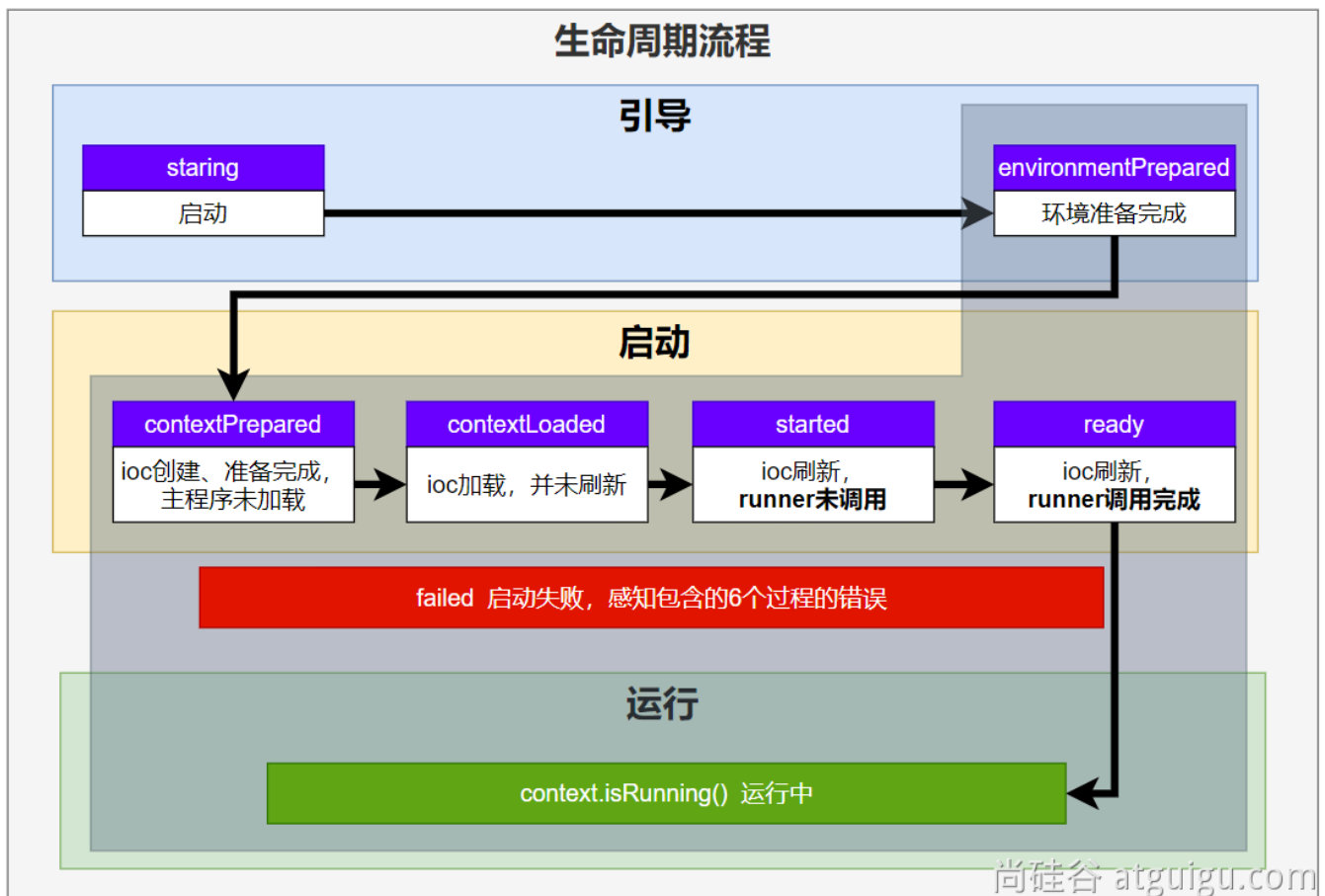
- ③ springboot 在 spring-boot.jar 中配置了默认的 Listener，如下。

```

# Run Listeners
org.springframework.boot.SpringApplicationRunListener=\
org.springframework.boot.context.event.EventPublishingRunListener

```

5.1.1.2 生命周期全流程



5.1.2 事件触发时机

5.1.2.1 各种回调监听器

- **BootstrapRegistryInitializer** : 感知特定阶段, 感知引导初始化。
 - META-INF/spring.factories。
 - 创建引导上下文 `bootstrapContext` 的时候触发。
 - `application.addBootstrapRegistryInitializer();`
 - 场景：进行密钥校对授权。
- **ApplicationContextInitializer** : 感知特定阶段, 感知 ioc 容器初始化。
 - META-INF/spring.factories。
 - `application.addInitializers();`
- **ApplicationListener** : 感知全阶段, 基于事件机制, 感知事件, 一旦到了哪个阶段可以做别的事。
 - `@Bean` 或 `@EventListener` : 事件驱动。
 - `SpringApplication.addListeners(...)` 或

- `SpringApplicationBuilder.listeners(...)`。
 - `META-INF/spring.factories`。
- **SpringApplicationRunListener**：感知全阶段生命周期 + 各种阶段都能自定义操作，功能更完善。
 - `META-INF/spring.factories`。
- **ApplicationRunner**：感知特定阶段，感知应用就绪 Ready，卡死应用，就不会就绪。
 - `@Bean`。
- **CommandLineRunner**：感知特定阶段，感知应用就绪 Ready，卡死应用，就不会就绪。
 - `@Bean`。

最佳实战：

- 如果要在项目启动前操作：`BootstrapRegistryInitializer` 和 `ApplicationContextInitializer`。
- 如果要在项目启动完成后操作：**ApplicationRunner** 和 **CommandLineRunner**。
- 如果要干涉生命周期做事：**SpringApplicationRunListener**。
- 如果想要用事件机制：**ApplicationListener**。

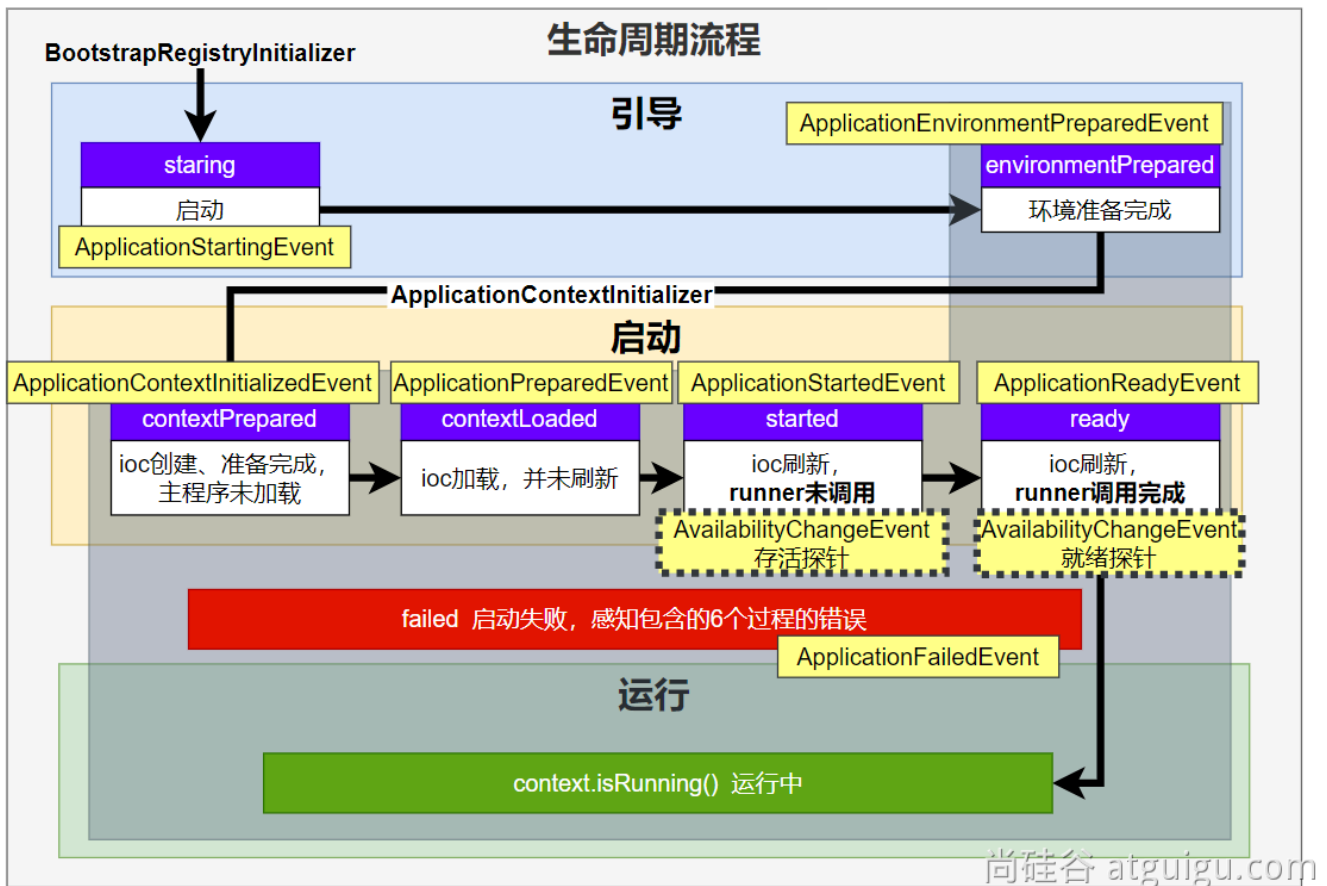
5.1.2.2 完整触发流程

9 大事件触发顺序和时机：

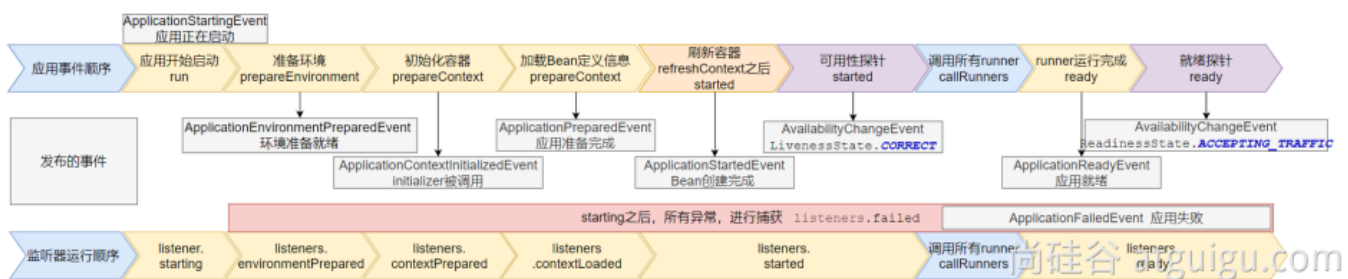
- 1、`ApplicationStartingEvent`：应用启动但未做任何事情，除过注册 `listeners` 和 `initializers`。
- 2、`ApplicationEnvironmentPreparedEvent`：Environment 准备好，但 context 未创建。
- 3、`ApplicationContextInitializedEvent`：ApplicationContext 准备好，`ApplicationContextInitializers` 调用，但是任何 bean 未加载。
- 4、`ApplicationPreparedEvent`：容器刷新之前，bean 定义信息加载。
- 5、`ApplicationStartedEvent`：容器刷新完成，runner 未调用。

以下就开始插入了探针机制。

- 6、AvailabilityChangeEvent : LivenessState.CORRECT 应用存活，存活探针。
- 7、ApplicationReadyEvent : 任何 runner 被调用。
- 8、AvailabilityChangeEvent : ReadinessState.ACCEPTING_TRAFFIC 就绪探针，可以接请求。
- 9、ApplicationFailedEvent : 启动出错。



应用事件发送顺序如下：



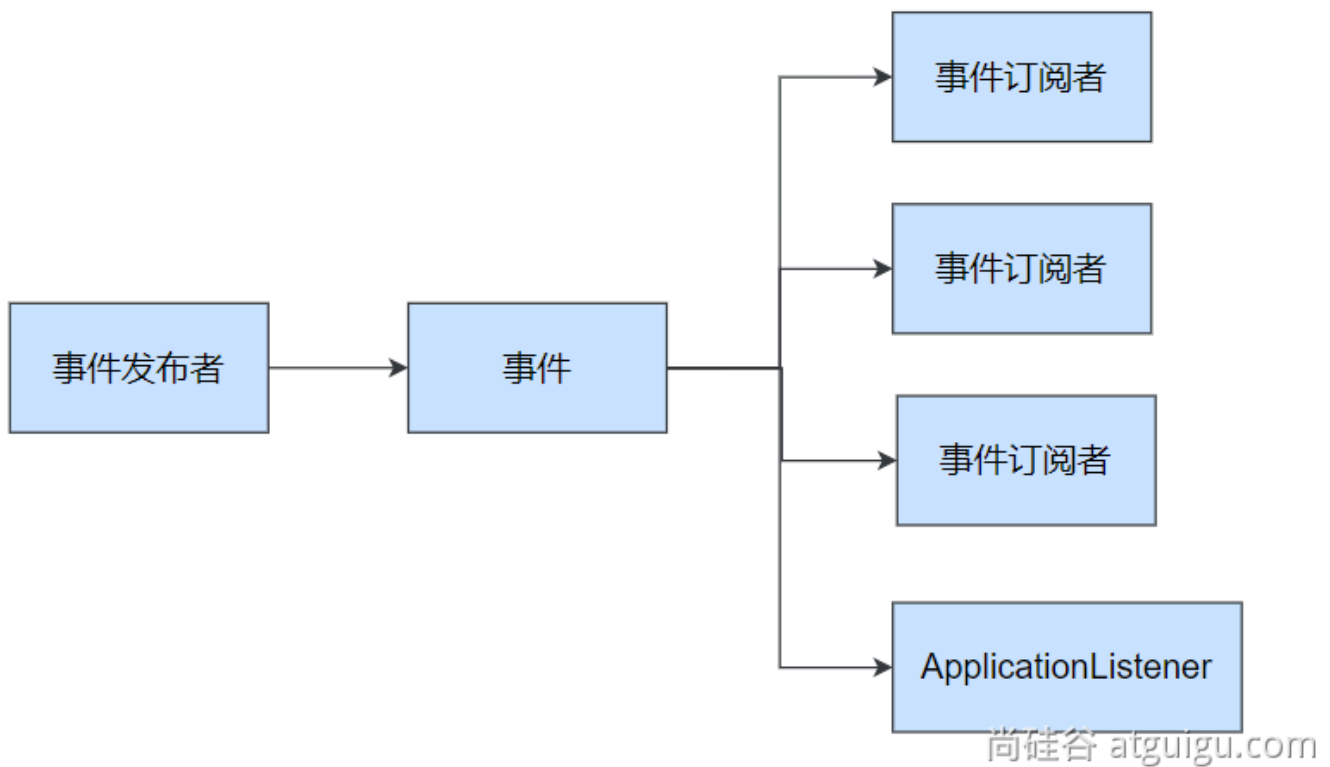
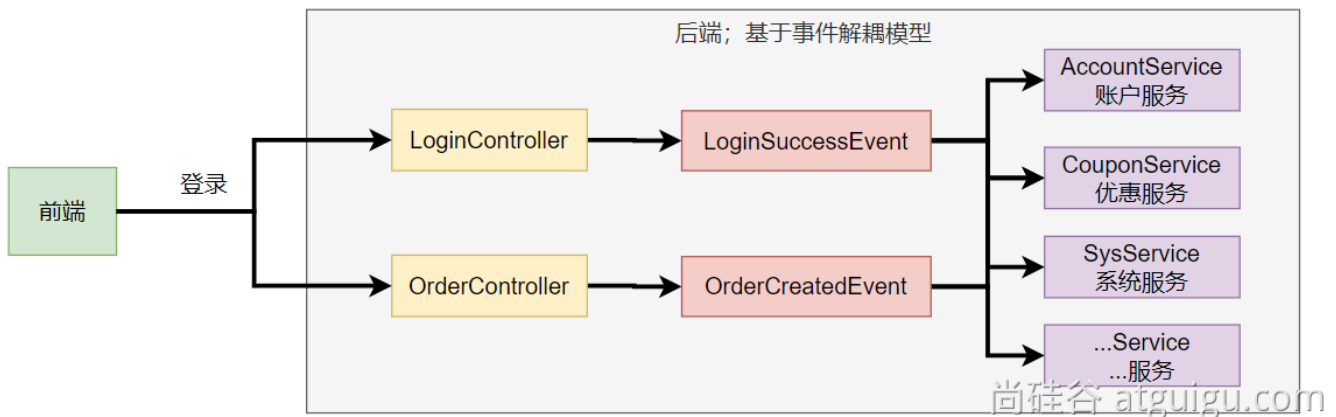
感知应用是否存活了：可能植物状态，虽然活着但是不能处理请求。

应用是否就绪了：能响应请求，说明确实活的比较好。

5.1.2.3 SpringBoot 事件驱动开发

应用启动过程生命周期事件感知（9 大事件）、应用运行中事件感知（无数种）。

- **事件发布**： `ApplicationEventPublisherAware`
或 注入 `ApplicationEventMulticaster` 。
- **事件监听**： 组件 + `@EventListener` 。



事件发布者。


```

package com.myxh.springboot.core.event;

import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;
import org.springframework.stereotype.Service;

/**
 * @author MYXH
 * @date 2023/9/24
 * @description 事件是广播出去的，所有监听这个事件的监听器都可以收到
 */
@Service
public class EventPublisher implements ApplicationEventPublisherAware
{
    // 底层发送事件用的组件，SpringBoot 会通过 ApplicationEventPublisherAware 接口自动注入
    ApplicationEventPublisher applicationEventPublisher;

    /**
     * 所有事件都可以发
     *
     * @param event 事件
     */
    public void sendEvent(ApplicationEvent event)
    {
        // 调用底层 API 发送事件
        applicationEventPublisher.publishEvent(event);
    }

    /**
     * 会被自动调用，把真正发事件的底层组组件给注入进来
     *
     * @param applicationEventPublisher 此对象要使用的事件发布者
     */
    @Override
    public void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher)
    {
        this.applicationEventPublisher = applicationEventPublisher;
    }
}

```

```
package com.myxh.springboot.core.event;

import com.myxh.springboot.core.entity.UserEntity;
import org.springframework.context.ApplicationEvent;

/**
 * @author MYXH
 * @date 2023/9/24
 * @description 登录成功事件，所有事件都推荐继承 ApplicationEvent
 */
public class LoginSuccessEvent extends ApplicationEvent
{
    /**
     * 登录成功事件
     *
     * @param source 事件来源，代表是谁登录成了
     */
    public LoginSuccessEvent(UserEntity source)
    {
        super(source);
    }
}
```

事件订阅者。

```
package com.myxh.springboot.core.service;

import com.myxh.springboot.core.entity.UserEntity;
import com.myxh.springboot.core.event.LoginSuccessEvent;
import org.springframework.context.event.EventListener;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Service;

/**
 * @author MYXH
 * @date 2023/9/24
 */
@Service
public class SystemService
{
    @Order(1)
    @EventListener
    public void onEvent(LoginSuccessEvent event)
    {
        System.out.println("----- SystemService 感知到事件 -----");
        System.out.println("event = " + event);
        System.out.println("-----");
        UserEntity source = (UserEntity) event.getSource();
        recordLog(source.getUsername());
    }

    public void recordLog(String username)
    {
        System.out.println(username + "登录信息已被记录");
    }
}
```

```
package com.myxh.springboot.core.service;

import com.myxh.springboot.core.entity.UserEntity;
import com.myxh.springboot.core.event.LoginSuccessEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Service;

/**
 * @author MYXH
 * @date 2023/9/24
 */
@Order(2)
@Service
public class AccountService implements ApplicationListener<LoginSuccessEvent>
{
    public void addAccountScore(String username)
    {
        System.out.println(username + " 加了 1 分");
    }

    @Override
    public void onApplicationEvent(LoginSuccessEvent event)
    {
        System.out.println("----- AccountService 收到事件 -----");
        UserEntity source = (UserEntity) event.getSource();
        addAccountScore(source.getUsername());
    }
}
```

```
package com.myxh.springboot.core.service;

import com.myxh.springboot.core.entity.UserEntity;
import com.myxh.springboot.core.event.LoginSuccessEvent;
import org.springframework.context.event.EventListener;
import org.springframework.core.annotation.Order;
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;

/**
 * @author MYXH
 * @date 2023/9/24
 */
@Service
public class CouponService
{
    public CouponService()
    {
        System.out.println("构造器调用");
    }

    @Async
    @Order(3)
    @EventListener
    public void onEvent(LoginSuccessEvent loginSuccessEvent)
    {
        System.out.println("----- CouponService 感知到事件 -----");
        System.out.println("loginSuccessEvent = " + loginSuccessEvent);
        System.out.println("-----");
        UserEntity source = (UserEntity) loginSuccessEvent.getSource();
        sendCoupon(source.getUsername());
    }

    public void sendCoupon(String username)
    {
        System.out.println(username + " 随机得到了一张优惠券");
    }
}
```

```
package com.myxh.springboot.core.service;

import com.myxh.springboot.core.event.LoginSuccessEvent;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Service;

/**
 * @author MYXH
 * @date 2023/9/24
 */
@Service
public class HelloService
{
    @EventListener
    public void onEvent(LoginSuccessEvent event){
        System.out.println("----- HelloService 感知到事件 -----");
        System.out.println("event = " + event);
        System.out.println("-----");

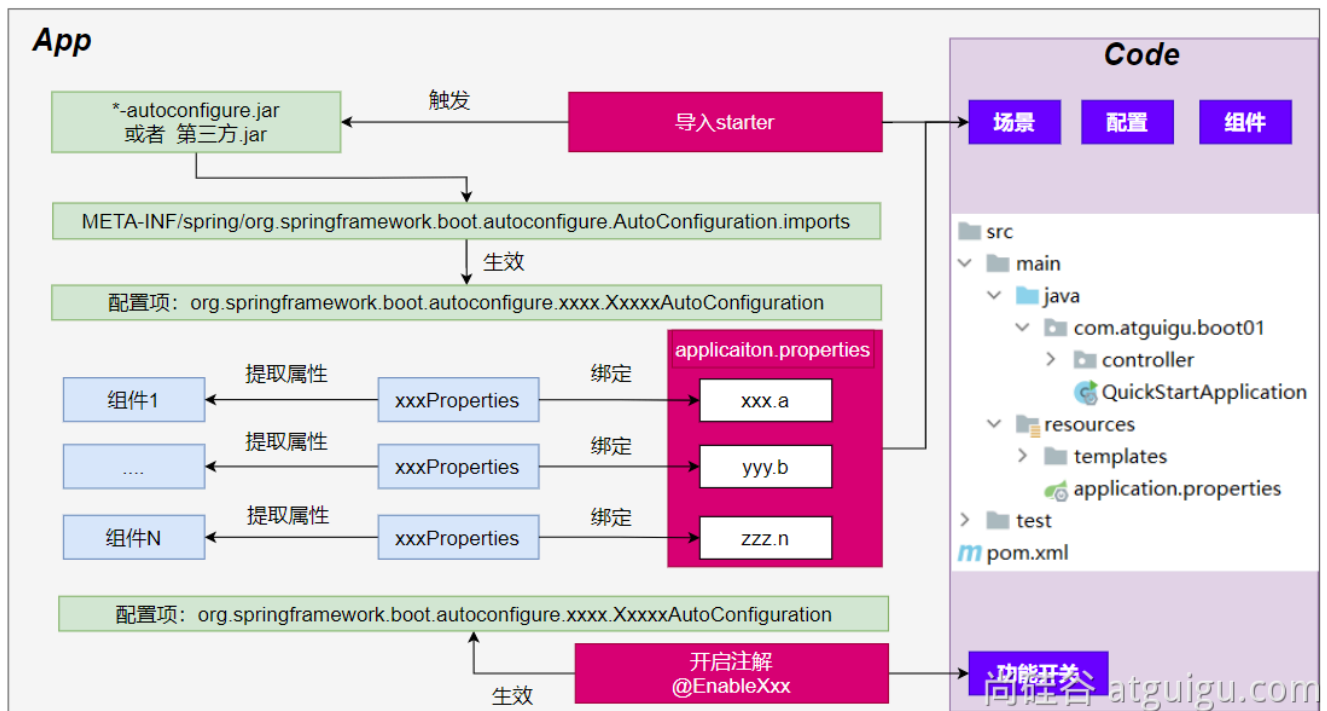
        // 调用业务
    }
}
```

5.2 自动配置原理

5.2.1 入门理解

应用关注的三大核心：场景、配置、组件。

5.2.1.1 自动配置流程



1、导入 starter 。

2、依赖导入 autoconfigure 。

3、寻找类路径下

META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports 文件。

4、启动，加载所有自动配置类 xxxAutoConfiguration 。

- ① 给容器中配置功能组件。
- ② 组件参数 绑定到 属性类 中， xxxProperties 。
- ③ 属性类 和 配置文件 前缀项绑定。
- ④ @Contional 派生的条件注解进 行判断是否组件生效。

5、效果：

- ① 修改配置文件，修改底层参数。
- ② 所有场景自动配置好直接使用。
- ③ 可以注入 SpringBoot 配置好的组件随时使用。

5.2.1.2 SPI 机制

- **Java 中的 SPI (Service Provider Interface) 是一种软件设计模式，用于在应用程序中动态地发现和加载组件。** SPI 的思想是，定义一个接口或抽象类，然后通过 `classpath` 中定义实现该接口的类来实现对组件的动态发现和加载。
- **SPI 的主要目的是解决在应用程序中使用可插拔组件的问题。** 例如，一个应用程序可能需要使用不同的日志框架或数据库连接池，但是这些组件的选择可能取决于运行时的条件。通过使用 **SPI**，应用程序可以在运行时发现并加载适当的组件，而无需在代码中硬编码这些组件的实现类。
- 在 Java 中，**SPI 的实现方式是通过在 `META-INF/services` 目录下创建一个以服务接口全限定名为名字的文件，文件中包含实现该服务接口的类的全限定名。** 当应用程序启动时，Java 的 **SPI 机制** 会自动扫描 `classpath` 中的这些文件，并根据文件中指定的类名来加载实现类。
- 通过使用 **SPI**，应用程序可以实现更灵活、可扩展的架构，同时也可以避免硬编码依赖关系和增加代码的可维护性。

在 SpringBoot

中，`META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`。

写一段 Java 的 SPI 机制代码。

下面是一个简单的 Java SPI (Service Provider Interface) 机制的示例代码，代码中包含了详细的注释说明：

```
package com.example.spi;

// 定义服务接口
public interface MyService
{
    void execute();
}
```



```
package com.example.spi;

// 定义服务提供者 1
public class MyService1 implements MyService
{
    @Override
    public void execute()
    {
        System.out.println("MyService1 正在执行。");
    }
}
```

```
package com.example.spi;

// 定义服务提供者 2
public class MyService2 implements MyService
{
    @Override
    public void execute()
    {
        System.out.println("MyService2 正在执行。");
    }
}
```

```

package com.example.spi;

import java.util.ServiceLoader;

// 定义服务加载器
public class ServiceLoader
{
    // 加载并执行服务
    public void loadAndExecuteServices()
    {
        // 使用Java SPI机制加载所有的服务提供者
        java.util.ServiceLoader<MyService> serviceLoader = java.util.ServiceLoader.load(MyService.class);

        // 遍历所有的服务提供者，并执行服务
        for (MyService service : serviceLoader)
        {
            service.execute();
        }
    }
}

```

```

package com.example.spi;
// 测试代码
public class Main
{
    public static void main(String[] args)
    {
        ServiceLoader loader = new ServiceLoader();
        loader.loadAndExecuteServices();
    }
}

```

上面的代码展示了如何使用 Java SPI 机制来加载并执行服务提供者。其中，`MyService` 是服务接口，`MyService1` 和 `MyService2` 是两个具体的服务提供者实现。`ServiceLoader` 是服务加载器，它通过 `java.util.ServiceLoader` 来加载所有的服务提供者，并执行服务的 `execute()` 方法。最后，在 `Main` 类的 `main()` 方法中调用 `ServiceLoader` 的 `loadAndExecuteServices()` 方法即可加载并执行服务提供者。

注意：在实际使用 Java SPI 机制时，需要在 `src/main/resources/META-INF/services` 目录下创建一个以服务接口全限定名命名的文件，文件内容为具体的服务提供者实现类的全限定名。在本示例中，需要创建 `src/main/resources/META-INF/services/com.example.MyService` 文件，并在

其中分别写入 `com.example.MyService1` 和 `com.example.MyService2`。这样，在调用 `java.util.ServiceLoader.load()` 方法时，就能正确加载到所有的服务提供者实现类。

5.2.1.3 功能开关

- 自动配置：全部都配置好，什么都不用管，自动批量导入。
 - 项目一启动，SPI 文件中指定的所有都加载。
- `@EnableXxx`：手动控制哪些功能的开启，手动导入。
 - 开启 xxx 功能。
 - 都是利用 `@Import` 把此功能要用的组件导入进去。

5.2.2 进阶理解

5.2.2.1 @SpringBootApplication

@SpringBootConfiguration。

就是 `@Configuration`，容器中的组件，配置类，Spring IOC 启动就会加载创建这个类对象。

@EnableAutoConfiguration：开启自动配置。

开启自动配置。

@AutoConfigurationPackage：扫描主程序包，加载自己的组件。

- 利用 `@Import(AutoConfigurationPackages.Registrar.class)` 想要给容器中导入组件。
- 把主程序所在的包的所有组件导入进来。
- 为什么 **SpringBoot** 默认只扫描主程序所在的包及其子包。

@Import(AutoConfigurationImportSelector.class)：加载所有自动配置类，加载 starter 导入的组件。

```
List<String> configurations = ImportCandidates.load(AutoConfiguration.class, getBeanClassLoader())
    .getCandidates();
```

扫描 SPI 文件：**META-INF/spring/**
org.springframework.boot.autoconfigure.AutoConfiguration.imports。

@ComponentScan。

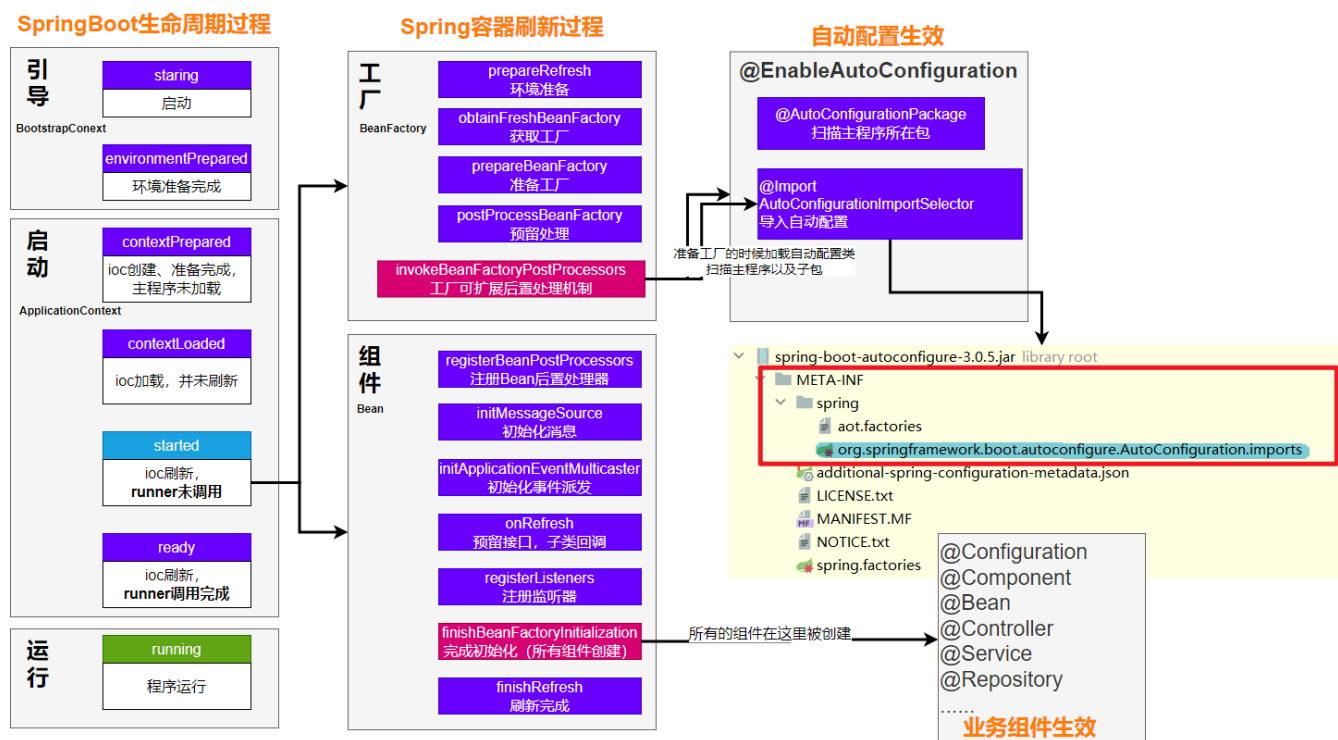
组件扫描：排除一些组件（哪些不要）。

排除前面已经扫描进来的配置类、和自动配置类。

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

5.2.2.2 完整启动加载流程

生命周期启动加载流程。



5.3 自定义 starter

场景：抽取聊天机器人场景，它可以打招呼。

效果：任何项目导入此 **starter** 都具有打招呼功能，并且问候语中的人名需要可以在配置文件中修改。

- 1、创建 自定义 starter 项目，引入 spring-boot-starter 基础依赖。
- 2、编写模块功能，引入模块所有需要的依赖。
- 3、编写 xxxAutoConfiguration 自动配置类，帮其他项目导入这个模块需要的所有组件。

- 4、编写配置文件

META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports

指定启动需要加载的自动配置。

- 5、其他项目引入即可使用。

5.3.1 业务代码

自定义配置有提示，导入以下依赖重启项目，再写配置文件就有提示。

```
<!-- 导入配置处理器，配置文件自定义的 properties 配置都会有提示 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
  <optional>true</optional>
</dependency>
```

```
package com.myxh.springboot.starter.robot.properties;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

/**
 * @author MYXH
 * @date 2023/9/24
 */
// 此属性类和配置文件指定前缀绑定
@ConfigurationProperties(prefix = "robot")
@Component
@Data
public class RobotProperties
{
    private String name;
    private Integer age;
    private String email;
}
```

```
robot.name=MYXH
robot.age=21
robot.email=1735350920@qq.com
```

5.3.2 基本抽取

- 创建 starter 项目，把公共代码需要的所有依赖导入。
- 把公共代码复制进来。
- 自己写一个 `RobotAutoConfiguration`，给容器中导入这个场景需要的所有组件。
 - 为什么这些组件默认不会扫描进去？
 - **starter 所在的包和引入它的项目的主程序所在的包不是父子层级。**

```
package com.myxh.springboot.starter.robot;

import com.myxh.springboot.starter.robot.controller.RobotController;
import com.myxh.springboot.starter.robot.properties.RobotProperties;
import com.myxh.springboot.starter.robot.service.RobotService;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

/**
 * @author MYXH
 * @date 2023/9/24
 */
// 给容器中导入 Robot 功能要用的所有组件
@Import({RobotProperties.class, RobotController.class, RobotService.class})
@Configuration
public class RobotAutoConfiguration
{
    /**
     * // 把组件导入到容器中
     * @Bean
     * public RobotController robotController()
     * {
     *     return new RobotController();
     * }
     */
}
```

- 别人引用这个 `starter`，直接导入这个 `RobotAutoConfiguration`，就能把这个场景的组件导入进来。
- 功能生效。
- 测试编写配置文件。

5.3.3 使用@EnableXxx 机制

```
package com.myxh.springboot.starter.robot.annotation;

import com.myxh.springboot.starter.robot.RobotAutoConfiguration;
import org.springframework.context.annotation.Import;

import java.lang.annotation.*;

/**
 * @author MYXH
 * @date 2023/9/24
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Documented
@Import(RobotAutoConfiguration.class)
public @interface EnableRobot
{
}
```

别人引入 `starter` 需要使用 `@EnableRobot` 开启功能。

5.3.4 完全自动配置

- 依赖 SpringBoot 的 SPI 机制。
- `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 文件中编写好自动配置类的全类名即可。
- 项目启动，自动加载自动配置类。