1、SpringBoot3-快速入门

1、简介

1. 前置知识

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

2. 环境要求

环境&工具	版本（or later）
SpringBoot	3.0.5+
IDEA	2021.2.1+
Java	17+
Maven	3.5+
Tomcat	10.0+
Servlet	5.0+
GraalVM Community	22.3+
Native Build Tools	0.9.19+

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、写配置、启动运行
- 直接嵌入Tomcat、Jetty or 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...
  - 导包一堆，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
  - 如果这些场景我要使用（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。

2、快速体验

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

1. 开发流程

1. 创建项目

maven 项目

XML | 复制代码

```xml
<!--    所有springboot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>3.0.5</version>
    </parent>
```

2. 导入场景

场景启动器

XML | 复制代码

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

```

3. 主程序

Java | 复制代码

```java
@SpringBootApplication //这是一个SpringBoot应用
public class MainApplication {

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

4. 业务

Java | 复制代码

```java
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello(){

        return "Hello,Spring Boot 3!";
    }

}
```

5. 测试

默认启动访问： localhost:8080

6. 打包

XML | 复制代码

```xml
<!--    SpringBoot应用打包插件-->
    <build>
        <plugins>
            <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 demo.jar` 启动项目

2. 特性小结

1. 简化整合

导入相关的场景，拥有相关的功能。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 <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

2. 简化开发

无需编写任何配置，直接开发业务

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 <https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties>

4. 简化部署

打包为可执行的jar包。

linux服务器上有java环境。

5. 简化运维

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

.....

3. Spring Initializr 创建向导

一键创建好整个项目结构

3、应用分析

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

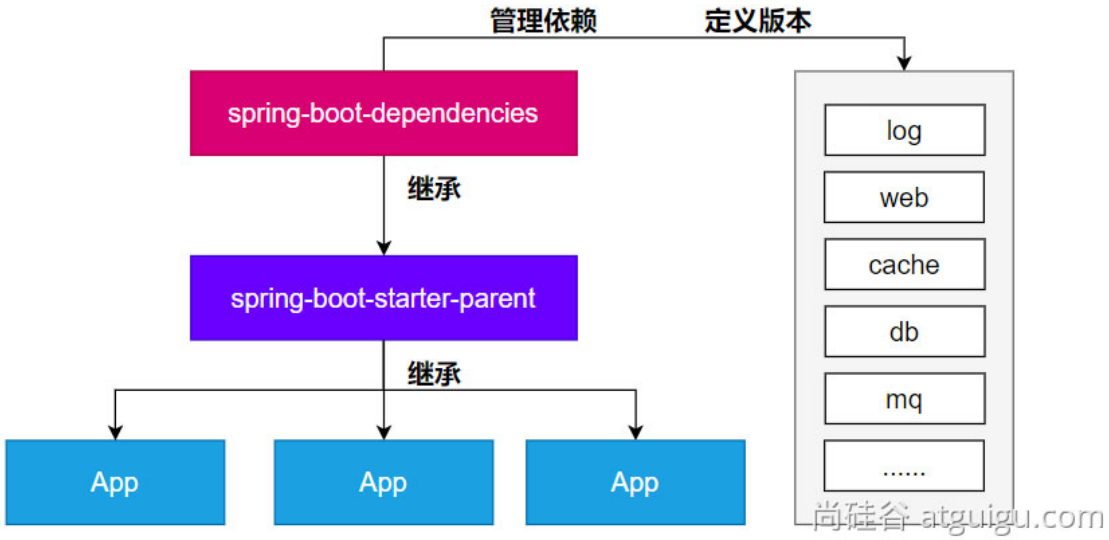
- 利用maven的就近原则
  - 直接在当前项目 `properties` 标签中声明父项目用的版本属性的key
  - 直接在**导入依赖的时候声明版本**

4、第三方的jar包

- boot父项目没有管理的需要自行声明好

```xml
<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.16</version>
</dependency>

```



2. 自动配置机制

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**

```java
    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[] names = ioc.getBeanDefinitionNames();
        //2、挨个遍历：
        // dispatcherServlet、beanNameViewResolver、characterEncoding
        // SpringBoot把以前配置的核心组件现在都给我们自动配置好了。
        for (String name : names) {
            System.out.println(name);
        }

    }
```

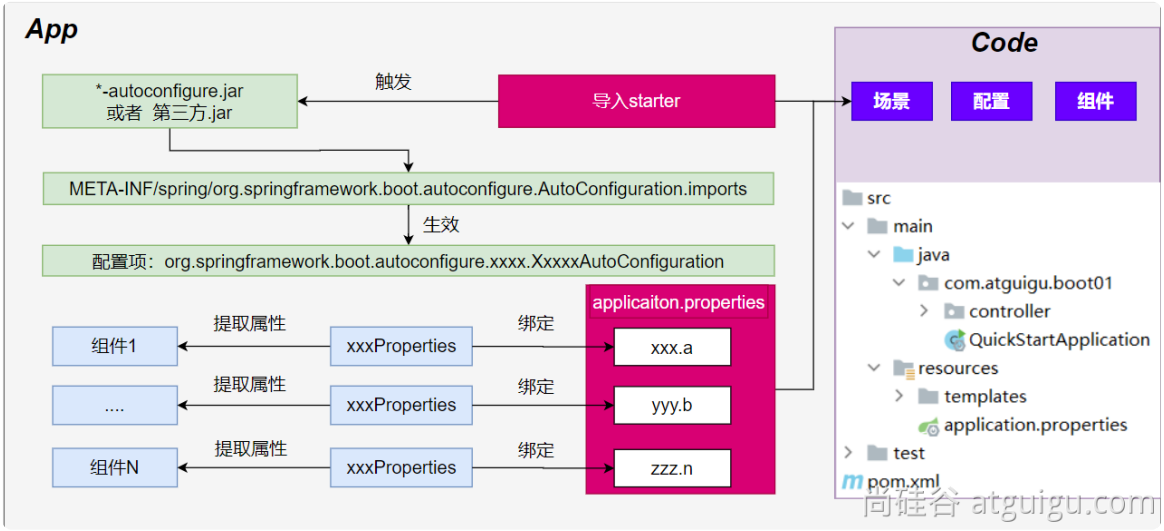
- **默认的包扫描规则**
  - `@SpringBootApplication` 标注的类就是主程序类
  - **SpringBoot只会扫描主程序所在的包及其下面的子包，自动的component-scan功能**
  - **自定义扫描路径**
    - @SpringBootApplication(scanBasePackages = "com.atguigu")
    - `@ComponentScan("com.atguigu")` 直接指定扫描的路径
- **配置默认值**
  - **配置文件**的所有配置项是和某个**类的对象**值进行一一绑定的。
  - 绑定了配置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

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，我们无需关心整合**
>
> 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**给我们做了整合操作），**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` 、 `@EnableAutoConfiguratio` 、 `@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 142个配置类。
  - 3. 这**142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
  - 项目启动的时候利用 @Import 批量导入组件机制把 `autoconfigure` 包下的142 `xxxxAutoConfiguration` 类导入进来（**自动配置类**）
  - 虽然导入了 `142` 个自动配置类
- 4、按需生效：
  - 并不是这 `142` 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、 `xxxxAutoConfiguration` 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。**
- **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。

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**
    a. **导入starter --> 生效xxxxAutoConfiguration --> 组件 --> xxxProperties --> 配置文件**
2. 理解**其他框架底层**
    a. 拦截器

3. 可以随时**定制化任何组件**

   a. **配置文件**

   b. **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，我们能不能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...）
- 分析这个场景给我们导入了**哪些能用的组件**
  - **自动装配**这些组件进行后续使用
  - 不满意boot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 <https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.build-systems.starters> ： `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`

4、核心技能

1. 常用注解

> SpringBoot摒弃XML配置方式，改为**全注解驱动**

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

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，名cat01，
> - 否则，就给容器中放一个 `Dog` 组件，名dog01
> - 如果系统中有 `dog01` 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)

@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)

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**

更多注解参照：Spring注解驱动开发
<https://www.bilibili.com/video/BV1gW411W7wy> 【1-26集】

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. 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。**换行
- 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，如：映射（map）/ 哈希（hash） / 字典（dictionary）
- **数组**：一组按次序排列的值，如：序列（sequence） / 列表（list）
- **纯量**：单个的、不可再分的值，如：字符串、数字、bool、日期

2. 示例

Java | 复制代码

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

@Data
public class Dog {
    private String name;
    private Integer age;
}

@Data
public class Child {
    private String name;
    private Integer age;
    private Date birthDay;
    private List<String> text; //数组
}

@Data
public class Cat {
    private String name;
    private Integer age;
}
```

properties表示法

Properties | 复制代码

```properties
person.name=张三
person.age=18
person.birthDay=2010/10/12 12:12:12
person.like=true
person.child.name=李四
person.child.age=12
person.child.birthDay=2018/10/12
person.child.text[0]=abc
person.child.text[1]=def
person.dogs[0].name=小黑
person.dogs[0].age=3
person.dogs[1].name=小白
person.dogs[1].age=2
person.cats.c1.name=小蓝
person.cats.c1.age=3
person.cats.c2.name=小灰
person.cats.c2.age=2
```

yaml表示法

YAML | 复制代码

```yaml
person:
  name: 张三
  age: 18
  birthDay: 2010/10/10 12:12:12
  like: true
  child:
    name: 李四
    age: 20
    birthDay: 2018/10/10
    text: ["abc","def"]
  dogs:
    - name: 小黑
      age: 3
    - name: 小白
      age: 2
  cats:
    c1:
      name: 小蓝
      age: 3
    c2: {name: 小绿,age: 2} #对象也可用{}表示
```

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

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

Java | 复制代码

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

使用 `@Data` 等注解

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 JUL（java.util.logging）
SLF4j（Simple Logging Facade for Java）	Log4j2 Logback
jboss-logging	

感兴趣日志框架关系与起源可参考:
https://www.bilibili.com/video/BV1gW411W76m
<https://www.bilibili.com/video/BV1gW411W76m> 视频 21~27集

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。
   a. spring5及以后 commons-logging被spring直接自己写了。
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**，但是我们不用担心，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。

2. 日志格式

Shell | 复制代码

```
2023-03-31T13:56:17.511+08:00  INFO 4944 --- [           main] o.apac
2023-03-31T13:56:17.511+08:00  INFO 4944 --- [           main] o.apac
```

默认输出格式:

- 时间和日期：毫秒级精度

- 日志级别：ERROR, WARN, INFO, DEBUG, or 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}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}`

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

3. 记录日志

Java | 复制代码

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

或者使用Lombok的@Slf4j注解
```

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, or 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

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

```
Java | 复制代码
1 logging.group.tomcat=org.apache.catalina,org.apache.coyote,org.apache
2 logging.level.tomcat=trace
```

SpringBoot 预定义两个组

Name	Loggers
web	org.springframework.core.codec, org.springframework.http, org.springframework.web, org.springframework.boot.actuate. org.springframework.boot.web.servlet.ServletContextInitialize
sql	org.springframework.jdbc.core, org.hibernate.SQL, org.jooq.to

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 or logging.file.path配置项。

logging.file.name	logging.file.path	示例	效果
未指定	未指定		仅控制台输出
指定	未指定	my.log	写入指定文件。可以
未指定	**指定**	/var/log	写入指定目录，文件
指定	**指定**		以logging.file.name

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
logging.logback.rollingpolicy.file-name-pattern	日志存档的文件名格式（默 ${LOG_FILE}.%d{yyyy-MM
logging.logback.rollingpolicy.clean-history-on-start	应用启动时是否清除以前存
logging.logback.rollingpolicy.max-file-size	存档前，每个日志文件的最 10MB）
logging.logback.rollingpolicy.total-size-cap	日志文件被删除之前，可以 值：0B）。设置1GB则磁盘 就会删除旧日志文件
logging.logback.rollingpolicy.max-history	日志文件保存的最大天数(默

8. 自定义配置

通常我们配置 application.properties 就够了。当然也可以自定义。比如：

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

如果可能，我们建议您在日志配置中使用 `-spring` 变量（例如， `logback-spring.xml` 而不是 `logback.xml` ）。如果您使用标准配置文件，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 `xx-spring.xml`

9. 切换日志组合

XML | 复制代码

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<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
JSON	com.fasterxml.jackson.core:jackson-databind	log4j2

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 sout 了**