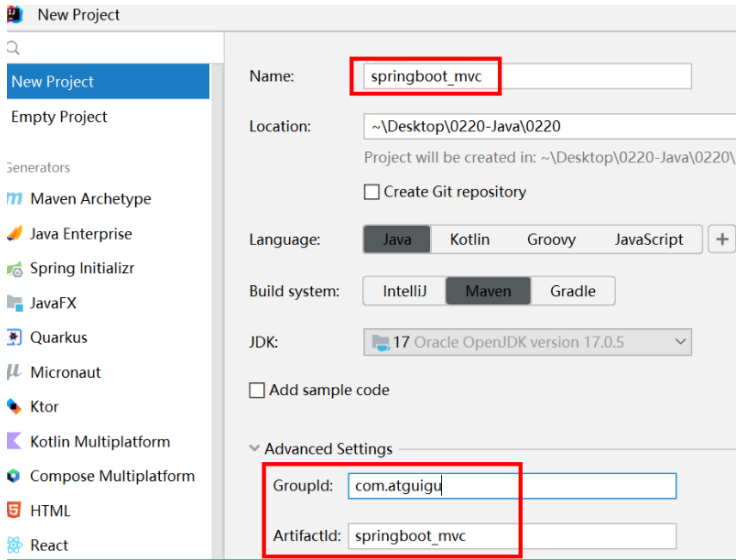


1、Springboot 快速入门

1) 创建 Maven 工程



2) 添加依赖(springboot 父工程依赖, web 启动器依赖)

1) 添加父工程坐标

SpringBoot 可以帮我们方便的管理项目依赖, 在 Spring Boot 提供了一个名为 **spring-boot-starter-parent** 的工程, 里面已经对各种常用依赖的版本进行了管理, 我们的项目需要以这个项目为父工程, 这样我们就不用操心依赖的版本问题了, 需要什么依赖, 直接引入坐标(不需要添加版本)即可。

```
<!--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) 添加 web 启动器

为了让 Spring Boot 帮我们完成各种自动配置, 我们必须引入 Spring Boot 提供的 **自动配置依赖**, 我们称为 **启动器**。因为我们是 web 项目, 这里我们引入 web 启动器 **spring-boot-starter-web**, 在 pom.xml 文件中加入如下依赖:

```
<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) 编写启动类(springboot 项目运行的入口)

创建 package: com.atguigu

创建启动类: MainApplication

```
package com.atguigu;

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

@SpringBootApplication
public class MainApplication {
    //SpringApplication.run() 方法是启动 Spring Boot 应用程序的关键步骤。它创建应用程序上下文、
    // 自动配置应用程序、启动应用程序, 并处理命令行参数, 使应用程序能够运行和提供所需的功能
    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class,args);
    }
}
```

```
@SpringBootApplication
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication {}
```

4) 编写处理器 Controller

创建 package: com.atguigu.controller

创建类: HelloController

注意: IoC 和 DI 注解需要在启动类的同包或者子包下方可生效, 无需指定, 约束俗称。

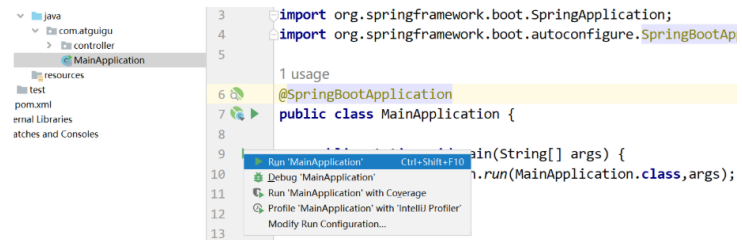
```
package com.atguigu.controller;

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

@RestController

@GetMapping("/hello")
public String hello(){
    return "Hello, Spring Boot 3!";
}
}
```

5) 启动项目



2、入门总结

1) 为什么依赖不需要写版本?

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。

2) 启动器 (Starter)

Spring Boot 提供了一种叫做 Starter 的概念, 它是一组预定义的依赖项集合, 旨在简化 Spring 应用程序的配置和构建过程。Starter 包含了一组相关的依赖项, 以便在启动应用程序时自动引入所需的库、配置和功能。

主要作用如下:

1) 简化依赖管理: Spring Boot Starter 通过捆绑和管理一组相关的依赖项, 减少了手动解析和配置依赖项的工作。只需引入一个相关的 Starter 依赖, 即可获取应用程序所需的全部依赖。

2) 自动配置: Spring Boot Starter 在应用程序启动时自动配置所需的组件和功能。通过根据类路径和其他设置的自动检测, Starter 可以自动配置 Spring Bean、数据源、消息传递等常见组件, 从而使应用程序的配置变得简单和维护成本降低。

3) 提供约定优于配置: Spring Boot Starter 遵循“约定优于配置”的原则, 通过提供一组默认设置和约定, 减少了手动配置的需要。它定义了标准的配置文件命名约定、默认属性值、日志配置等, 使得开发者可以更专注于业务逻辑而不是繁琐的配置细节。

4) 快速启动和开发应用程序: Spring Boot Starter 使得从零开始构建一个完整的 Spring Boot 应用程序变得容易。它提供了主要领域(如 Web 开发、数据访问、安全性、消息传递等)的 Starter, 帮助开发者快速搭建一个具备特定功能的应用程序原型。

5) 模块化和可扩展性: Spring Boot Starter 的组织结构使得应用程序的不同模块可以进行分离和解耦。每个模块可以有自己的 Starter 和依赖项, 使得应用程序的不同部分可以按需进行开发和扩展。

3) @SpringBootApplication 注解的能效

@SpringBootApplication 添加到启动类上, 是一个组合注解, 也是 Spring Boot 框架中的核心注解, 主要作用是简化 Spring Boot 应用程序的配置和启动过程。它自动配置应用程序、扫描并加载组件, 并将配置和启动类合二为一, 简化了开发者的工作量, 提高了开发效率。

具体而言, @SpringBootApplication 注解起到以下几个主要作用:

1) 自动配置: @EnableAutoConfiguration 注解, 用于启用 Spring Boot 的自动配置机制。自动配置会根据应用程序的依赖项和类路径, 自动配置各种常见的 Spring 配置和功能, 减少开发者的手动配置工作。它通过智能地分析类路径、加载配置和条件判断, 为应用程序提供适当的默认配置。

2) 组件扫描: @ComponentScan 注解, 用于自动扫描并加载应用程序中的组件, 例如控制器 (Controllers)、服务 (Services)、存储库 (Repositories) 等。它默认会扫描 @SpringBootApplication 注解所在类的包及其子包中的组件, 并将它们纳入 Spring Boot 应用程序的上下文中, 使它们可被自动注入和使用。

3) 声明配置类: @Configuration 注解, 将被标注的类声明为配置类。配置类可以包含 Spring 框架相关的配置、Bean 定义, 以及其他的自定义配置。通过 @SpringBootApplication 注解, 开发者可以将配置类与启动类合并在一起, 使得配置和启动可以同时发生。

3、SpringBoot3 配置文件

SpringBoot 工程下，进行统一的配置管理，你想设置的任何参数（端口号、项目根路径、数据库连接信息等等）都集中到一个固定位置和命名的配置文件（**application.properties** 或 **.yml** 或 **.yaml**）中。
注：1> 配置文件应该放置在 Spring Boot 工程的 **src/main/resources** 目录下。这是因为 **src/main/resources** 目录是 Spring Boot 默认的路径（classpath），配置文件会被自动加载并可供应用程序访问。
2> 如果 **application.properties** 和 **application.yml**（.yaml）同时存在，**properties** 的优先级更高。
3> 配置基本都有默认值。
4、属性配置文件使用
application.properties 为统一配置文件，内部包含：固定功能的key，自定义的key；此处的配置信息，我们可以在程序中**@Value**等注解读取。
1）在 **resource** 文件夹下面新建 **application.properties** 配置文件

```
# 固定的key
# 启动端口号
server.port=80

# 自定义
spring.jdbc.datasource.driverClassName=com.mysql.cj.jdbc.driver
spring.jdbc.datasource.url=jdbc:mysql:///springboot_01
spring.jdbc.datasource.username=root
spring.jdbc.datasource.password=root
```

2）读取配置文件：@Value("\${key}")

```
package com.atguigu.properties;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class DataSourceProperties {

    @Value("${spring.jdbc.datasource.driverClassName}")
    private String driverClassName;

    @Value("${spring.jdbc.datasource.url}")
    private String url;

    @Value("${spring.jdbc.datasource.username}")
    private String username;
```

5、YAML 配置文件使用

1）yaml 格式介绍
yaml 是一种基于层次结构的数据序列化格式，与 **.properties** 文件相比，yaml 格式有以下优势：
1> 层次结构：yaml 文件中字段和值之间使用**冒号分隔**，并使用**缩进表示层级关系**，使得数据之间的关系更加清晰和直观。这样可以更容易理解和维护复杂的配置，特别适用于深层次嵌套的配置情况。
3> 注释支持：yaml 格式支持注释，可以在配置文件中添加说明性的注释，使配置更具可读性和可维护性。相比之下，**.properties** 文件不支持注释，无法提供类似的解释和说明。
4> 多行文本：yaml 格式支持多行文本的表示，可以更方便地表示长文本或数据块。相比之下，**.properties** 文件需要使用转义符或将长文本拆分为多行。
5> 类型支持：yaml 格式天然支持复杂的数据类型，如列表、映射等。这使得在配置文件中表示嵌套结构或数据集更加容易，而不需要进行额外的解析或转换。
2）yaml 语法说明
1> 数据结构用树形结构呈现，通过缩进来表示层级；
2> 连续的项目（集合）通过减号“-”来表示；
3> 键值结构里面的 key/value 对用冒号“:”来分隔；
4> yaml 配置文件的扩展名是 **yaml** 或 **yml**。

# YAML配置文件示例	database:	features:
app_name: 我的应用程序	host: localhost	- 登录
version: 1.0.0	port: 5432	- 注册
author: 张三	username: admin	- 仪表盘
	password: password123	

3）读取配置文件中单个属性值：@Value("\${key}")

6、批量配置文件注入

@ConfigurationProperties 可以将一些配置属性批量注入 bean 对象。
1> 创建类，添加属性和注解
在类上通过**@ConfigurationProperties** 注解声明该类要读取属性配置，**prefix="spring.jdbc.datasource"** 读取属性文件中前缀为 **spring.jdbc.datasource** 的值。前缀和属性名称和配置文件中的 key

必须要保持一致才可以注入成功。

```
spring:
  jdbc:
    datasource:
      driverClassName: com.mysql.jdbc.Driver
      url: jdbc:mysql:///springboot_02
      username: root
      password: root

server:
  port: 80
```

```
package com.atguigu.properties;

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

@Component
@ConfigurationProperties(prefix = "spring.jdbc.datasource")
public class DataSourceConfigurationProperties {

    private String driverClassName;
    private String url;
    private String username;
    private String password;
```

```
@RestController
public class HelloController {

    @Autowired
    private DataSourceConfigurationProperties dataSourceConfigurationProperties;

    @GetMapping("/hello")
    public String hello(){
        System.out.println("dataSourceConfigurationProperties = " +
        dataSourceConfigurationProperties);
        return "Hello, Spring Boot 3!";
    }
}
```

7、多环境配置和使用（spring.profiles.active）

在 Spring Boot 中，可以使用多环境配置来根据不同的运行环境（如开发、测试、生产）加载不同的配置。SpringBoot 支持多环境配置让应用程序在不同的环境中使用不同的配置参数，例如数据库连接信息、日志级别、缓存配置等。
以下是实现 Spring Boot 多环境配置的常见方法：
1）属性文件分离：将应用程序的配置参数分离到不同的属性文件中，每个环境对应一个属性文件。例如，可以创建 **application-dev.properties**、**application-prod.properties** 和 **application-test.properties** 等文件。在这些文件中，可以定义各自环境的配置参数，如数据库连接信息、端口号等。然后，在 **application.properties** 中通过 **spring.profiles.active** 属性指定当前使用的环境。Spring Boot 会根据该属性来加载对应环境的属性文件，覆盖默认的配置。
2）yaml 配置文件：与属性文件类似，可以将配置参数分离到不同的 yaml 文件中，每个环境对应一个文件。例如，可以创建 **application-dev.yml**、**application-prod.yml** 和 **application-test.yml** 等文件。在这些文件中，可以使用 **yaml** 各自环境的配置参数。同样，在 **application.yaml** 中通过 **spring.profiles.active** 属性指定当前的环境，Spring Boot 会加载相应的 **yaml** 文件。
3）命令行参数（动态）：可以通过命令行参数来指定当前的环境。例如，可以使用 **--spring.profiles.active=dev** 来指定使用开发环境的配置。
【注意】如果设置了 **spring.profiles.active**，并且和 **application** 有重叠属性，以 **active** 设置优先；如果设置 **spring.profiles.active**，和 **application** 无重叠属性，**application** 设置依然生效。
8、web 相关配置
位置：application.yml
1> **server.port**：指定应用程序的 HTTP 服务器端口号。默认情况下，Spring Boot 使用 8080 作为默认端口。
2> **server.servlet.context-path**：设置应用程序的项目根路径。这是应用程序在 URL 中的基本路径。默认情况下，项目根路径为空即/。
3> **spring.mvc.view.prefix** 和 **spring.mvc.view.suffix**：这两个属性用于配置视图解析器的前缀和后缀。视图解析器用于解析控制器返回的视图名称，并将其映射到实际的视图页面。
4> **spring.resources.static-locations**：配置静态资源的位置。静态资源可以是 CSS、JavaScript、图像等。默认情况下，Spring Boot 会将静态资源放在 **classpath:/static** 目录下。
5> **spring.http.encoding.charset** 和 **spring.http.encoding.enabled**：这两个属性用于配置 HTTP 请求和响应的字符编码。
spring.http.encoding.charset 定义字符编码的名称（例如 UTF-8），**spring.http.encoding.enabled** 用于启用或禁用字符编码的自动配置。

9、静态资源处理

在 WEB 开发中我们需要引入一些静态资源,例如 : HTML,CSS,JS,图片等,如果是普通的项目静态资源可以放在项目的 webapp 目录下。现在使用 Spring Boot 做开发,项目中没有 webapp 目录,我们的项目是一个 jar 工程,那么就没有 webapp,一般默认的静态资源路径有:

- classpath:/META-INF/resources/
 - classpath:/resources/
 - classpath:/static/
 - classpath:/public/
- 我们只要静态资源放在这些目录中任何一个,SpringMVC 都会帮我们处理,我们习惯会把静态资源放在 classpath:/static/目录下。

【注意】可以设置 static-locations 属性来配置静态资源的位置

```
spring:
  web:
    resources:
      # 配置静态资源地址,如果设置,会覆盖默认值
      static-locations: classpath:/webapp
```

10、自定义拦截器(SpringMVC 配置)

1) 拦截器声明

```
package com.atguigu.interceptor;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

@Component
public class MyInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        System.out.println("MyInterceptor拦截器的preHandle方法执行...");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("MyInterceptor拦截器的postHandle方法执行...");
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        System.out.println("MyInterceptor拦截器的afterCompletion方法执行...");
    }
}
```

2) 拦截器配置

正常使用配置类,要保证配置类要在启动类的同包或者子包方可生效。

```
package com.atguigu.config;

import com.atguigu.interceptor.MyInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class MvcConfig implements WebMvcConfigurer {

    @Autowired
    private MyInterceptor myInterceptor ;

    /**
     * /** 拦截当前目录及子目录下的所有路径 /user/** /user/findAll /user/order/findAll
     * /** 拦截当前目录下的以及子路径 /user/* /user/findAll
     * @param registry
     */
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(myInterceptor).addPathPatterns("/**");
    }
}
```

11、SpringBoot3 整合 Druid 数据源

【注意】通过源码分析,druid-spring-boot-3-starter 目前最新版本是 1.2.18,虽然适配了 SpringBoot3,但缺少自动装配的配置文件,需要手动在 resources 目录下创建 META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports 指定 Druid 的指定配置类,文件内容如下:

```
com.alibaba.druid.spring.boot3.autoconfigure.DruidDataSourceAutoConfigure
```

1) 引入依赖

```
<!-- 数据库相关配置启动器 jdbc-template 事务相关-->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>

<!-- druid启动器的依赖 -->
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid-spring-boot-3-starter</artifactId>
  <version>1.2.18</version>
</dependency>

<!-- 驱动类-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.28</version>
</dependency>

<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.28</version>
</dependency>
```

2) 启动类

```
@SpringBootApplication
public class MainApplication {

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

3) 配置文件编写: 添加 druid 连接池的基本配置

```
spring:
  datasource:
    # 连接池类型
    type: com.alibaba.druid.pool.DruidDataSource

    # Druid的其他属性配置 springboot3整合情况下,数据库连接信息必须在Druid属性下!
  druid:
    url: jdbc:mysql://localhost:3306/day01
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
    # 初始化时建立物理连接的个数
    initial-size: 5
    # 连接池的最小空闲数量
    min-idle: 5
    # 连接池最大连接数量
```

4) 编写 Controller

```
@Slf4j
@Controller
@RequestMapping("/user")
public class UserController {
    //导入了jdbc的启动器, springboot就会自动把JdbcTemplate注入IOC容器中
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @GetMapping("/getUser")
    @ResponseBody
    public User getUser(){
        String sql = "select * from users where id = ? ";
        User user = jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<>
(User.class), 1);
        log.info("查询的用户数据为:{}",user.toString());
        return user;
    }
}
```


12、SpringBoot3 整合 Mybatis

MyBatis 整合步骤

1) 导入依赖：在您的 Spring Boot 项目的构建文件（如 pom.xml）中添加 MyBatis 和数据库驱动的相关依赖。例如，如果使用 MySQL 数据库，您需要添加 MyBatis 和 MySQL 驱动的依赖。

```
<!-- 数据库相关配置启动器 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>

<!-- druid启动器的依赖 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-3-starter</artifactId>
    <version>1.2.18</version>
</dependency>

<!-- 驱动类-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.28</version>
</dependency>

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.28</version>
</dependency>
```

2) 配置数据源：在 application.properties 或 application.yml 中配置数据库连接信息，包括数据库 URL、用户名、密码、mybatis 的功能配置等。


```
server:
  port: 80
  servlet:
    context-path: /
spring:
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    druid:
      url: jdbc:mysql:///day01
      username: root
      password: root
      driver-class-name: com.mysql.cj.jdbc.Driver











mybatis:
  configuration: # setting配置
    auto-mapping-behavior: full
    map-underscore-to-camel-case: true
    log-impl: org.apache.ibatis.logging.slf4j.Slf4jImpl
  type-aliases-package: com.atguigu.pojo # 配置别名
  mapper-locations: classpath:/mapper/*.xml # mapper.xml位置
```

3) 创建实体类：创建与数据库表对应的实体类。

4) 创建 Mapper 接口：创建与数据库表交互的 Mapper 接口。

5) 创建 Mapper 接口 SQL 实现：可以使用 mapper.xml 文件或注解方式；位置：resources/mapper/UserMapper.xml

▼  springboot-base-mybatis-05

- ▼  src
 - ▼  main
 - ▼  java
 - ▼  com.atguigu
 - ▼  mapper
 -  UserMapper
 -  Main
 - ▼  resources
 - ▼  mappers
 -  UserMapper.xml

6) 创建程序启动类

7) 注解扫描：在 Spring Boot 的主应用类上添加 @MapperScan 注解，用于扫描和注册 Mapper 接口。

```
@MapperScan("com.atguigu.mapper") //mapper接口扫描配置
@SpringBootApplication
public class MainApplication {

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

8) 使用 Mapper 接口：在需要使用数据库操作的地方，通过依赖注入或直接实例化 Mapper 接口，并调用其中的方法进行数据库操作。

13、声明式事务整合配置

依赖导入：

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

SpringBoot 项目会自动配置一个 DataSourceTransactionManager，所以我们只需在方法（或者类）加上 @Transactional 注解，就自动纳入 Spring 的事务管理。

```
@Transactional
public void update(){
    User user = new User();
    user.setId(1);
    user.setPassword("test2");
    user.setAccount("test2");
    userMapper.update(user);
}
```

14、AOP 整合配置

依赖导入：

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

直接使用 aop 注解即可：

```
@Component
@Aspect
public class LogAdvice {

    @Before("execution(* com..service.*.*(..))")
    public void before(JoinPoint joinPoint){
        System.out.println("LogAdvice.before");
        System.out.println("joinPoint = " + joinPoint);
    }

}
```