



# 基础篇

## 1. 属性配置

SpringBoot的属性配置在resource下的application.properties，如下

```
1 # 修改服务器端口
2 server.port=80
3
4 # 修改banner
5 # 关闭
6 spring.main.banner-mode=off
7
8 # 日志
9 logging.level.root=info
```

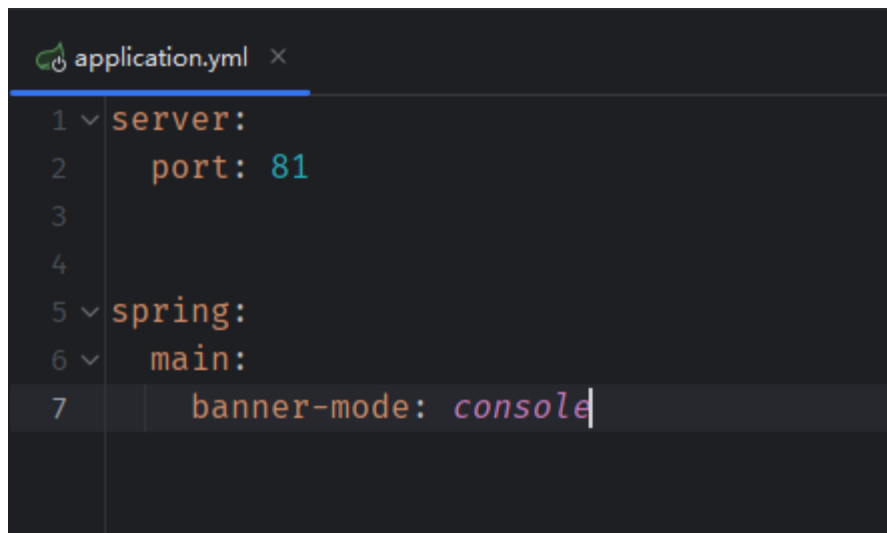
它可操作的属性有很多，在SpringBoot中导入对应的starter后，提供对应的属性配置。

参考[官方文档](#)。

## 2. 配置文件

SpringBoot提供了多种属性配置方式，如

- properties，传统格式（默认）
- yml，主流格式

A screenshot of a code editor showing the content of a file named 'application.yml'. The file contains the following YAML configuration:

```
1 server:
2   port: 81
3
4
5 spring:
6   main:
7     banner-mode: console
```

- yaml

A screenshot of a code editor showing the content of a file named 'application.yaml'. The file contains the following YAML configuration:

```
1 server:
2   port: 82
3 spring:
4   main:
5     banner-mode: console
```

实际上，yml和yaml文件没有任何区别，可以互换使用。

三种文件共存时，properties文件生效；yml和yaml文件共存时，yml生效。即加载优先级：**properties > yml > yaml**。

另外，多个配置文件共存时，相同属性会按照优先级覆盖，不同属性保留。

# 3. YAML

## yaml语法规则

- 大小写敏感
- 属性层级关系使用多行描述，每行结尾使用冒号结束
- 使用缩进表示层级关系，同层级左侧对齐，只允许使用空格（不允许使用Tab键）
- 属性值前面添加空格（属性名与属性值之间使用冒号+空格作为分隔）
- # 表示注释

```
enterprise:
  name: itcast
  age: 16
  tel: 4006184000
```

yaml

- 字面值表示方式

```
boolean: TRUE           #TRUE,true,True,FALSE,false,False均可
float: 3.14             #6.8523015e+5 #支持科学计数法
int: 123                #0b1010_0111_0100_1010_1110 #支持二进制、八进制、十六进制
null: ~                #使用~表示null
string: HelloWorld      #字符串可以直接书写
string2: "Hello World" #可以使用双引号包裹特殊字符
date: 2018-02-17        #日期必须使用yyyy-MM-dd格式
datetime: 2018-02-17T15:02:31+08:00 #时间和日期之间使用T连接，最后使用+代表时区
```

- 数组表示方式：在属性名书写位置的下方使用减号作为数据开始符号，每行书写一个数据，减号与数据间空格分隔

```
subject:
  - Java
  - 前端
  - 大数据
enterprise:
  name: itcast
  age: 16
  subject:
    - Java
    - 前端
    - 大数据
likes: [王者荣耀,刺激战场] #数组书写缩略格式
```

```
users: #对象数组格式
  - name: Tom
    age: 4
  - name: Jerry
    age: 5
users: #对象数组格式二
  -
    name: Tom
    age: 4
  -
    name: Jerry
    age: 5 #对象数组缩略格式
users2: [ { name:Tom , age:4 } , { name:Jerry , age:5 } ]
```

yaml中引用数据，

```
baseDir: C:\windows

tempDir: ${baseDir}\temp
```

yaml中要使用转义字符，需要使用双引号""包围。

可以使用Environment封装全部数据，

- 封装全部数据到Environment对象

```
lesson: SpringBoot

server:
  port: 82

enterprise:
  name: itcast
  age: 16
  tel: 4006184000
  subject:
    - Java
    - 前端
    - 大数据
```

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private Environment env;
    @GetMapping("/{id}")
    public String getById(@PathVariable Integer id){
        System.out.println(env.getProperty("lesson"));
        System.out.println(env.getProperty("enterprise.name"));
        System.out.println(env.getProperty("enterprise.subject[0]"));
        return "hello , spring boot!";
    }
}
```

用对象封装部分数据，并加载到SpringBoot中，

定义POJO对象，添加注解Component和ConfigurationProperties，在ConfigurationProperties中指定需要封装的配置数据对象名，如下

```
9  datasource:
10  url: jdbc:mysql://192.168.224.100/test|
11  username: study
12  password: 123456
13
```

```

6  @Component
7  @ConfigurationProperties("datasource")
8  public class MyDataSource {
9
10     3 usages
    private String url;
    3 usages
11     private String username;
    3 usages
12     private String password;
13

```

其他配置信息，也是通过这种方式使得SpringBoot读取的。

## 4. 整合第三方技术

整合第三方技术的一般步骤如下，

- 导入相应的starter
- 添加相应的配置，注意强大的默认配置
- 使用该技术进行开发

### 4.1 整合JUNIT

SpringBoot默认导入了测试依赖，spring-boot-starter-test，

```

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

```

执行测试，

```

7
8  @SpringBootTest
9  class Springboot03JUnitApplicationTests {
10
11      // SpringBoot测试步骤：
12      // 1. 注入你要测试的对象
13      // 2. 执行要测试对象对应的方法
14
15      @Autowired
16      private BookDao bookDao;
17
18      @Test
19      void contextLoads() {
20          bookDao.save();
21      }
22
23  }

```

注意，测试类需要有注解SpringBootTest

如果将测试类移出Application所在包及其子包下，则无法执行，报错内容如下，

```

java.lang.IllegalStateException: Unable to find a @SpringBootConfiguration, you need t
o use @ContextConfiguration or @SpringBootTest(classes=...) with your test

```

此时，需要在注解SpringBootTest中指定Application，即可正常执行。

```

8
9  @SpringBootTest(classes = Springboot03JunitApplication.class)
10 class Springboot03JunitApplicationTests {
11
12     // SpringBoot测试步骤：
13     // 1. 注入你要测试的对象
14     // 2. 执行要测试对象对应的方法
15
16     @Autowired
17     private BookDao bookDao;
18
19     @Test
20     void contextLoads() {
21         bookDao.save();
22     }
23
24 }

```

Application的注解SpringBpptApplication，包含注解SpringBootConfiguration，它包含了Configuration注解，即Application也是Spring容器的配置类。

注解SpringBootTest不指定classes属性时，自动查找当前包及其子包下的配置类，如果不存在，则报错，需要通过classes属性指定配置类。

## 4.2 整合MyBatis

MyBatis运行，需要有两部分，

- 核心配置，指定数据库连接信息
- 映射配置，SQL映射

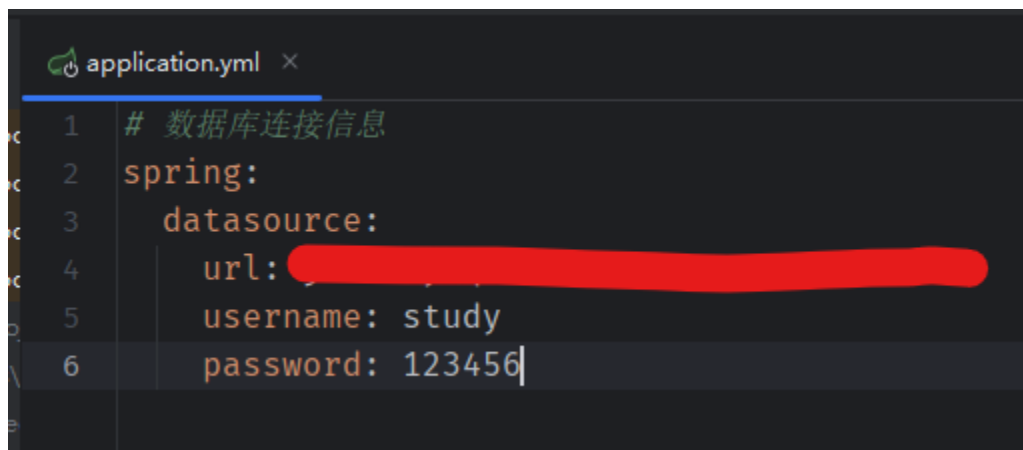
创建SpringBoot项目，添加**MyBatis Framework**和**MySQL driver**，坐标如下，

```
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>2.3.0</version>
</dependency>

<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <scope>runtime</scope>
</dependency>
```

SpringBoot的依赖命名规则为，spring-boot-starter-xxx，而第三方适配SpringBoot的依赖，命名规则为 xxx-spring-boot-starter。

添加数据配置信息，



```
application.yml x
1 # 数据库连接信息
2 spring:
3   datasource:
4     url: 
5     username: study
6     password: 123456
```

定义数据实体类，



```
public class Book {

    3 usages
    private Integer id;|
    3 usages
    private String type;
    3 usages
    private String name;
    3 usages
    private String description;
}
```

定义dao，这里也可以使用xml的形式定义映射文件，

```
@Mapper
public interface BookDao {

    1 usage
    @Select("select * from book where id = #{id}")
    Book getById(Integer id);

}
```

整合MyBatis完毕。

可以发现，与Spring整合MyBatis相比，不再需要自己定义SqlSessionFactoryBean和DataSource，简化了很多。

在低版本的SpringBoot中，整合MyBatis常见的一些问题，

- The server time zone value ...，在数据库连接配置的URL中，添加serverTimeZone参数，或者，修改MySQL数据库配置

## 4.3 整合MyBatis-Plus

由于MyBatis-Plus未被Spring收录，无法直接勾选其starter。

创建项目中，只勾选MySQL driver，完成后，手工导入MyBatis-Plus的starter，

```

<!-- https://mvnrepository.com/artifact/com.baomidou/mybatis-plus-boot
<dependency>
  <groupId>com.baomidou</groupId>
  <artifactId>mybatis-plus-boot-starter</artifactId>
  <version>3.5.3.1</version>
</dependency>

```

其他过程与整合MyBatis相同，不同的是，mapper的定义更简单了，

```

@Mapper
public interface BookDao extends BaseMapper<Book> {

}

```

只需要**继承BaseMapper**即可实现很多基础的CRUD。

如果存在表名和实体类名称不同，可以通过设置修改，如下，

```

9  #设置Mp相关的配置
10 mybatis-plus:
11   global-config:
12     db-config:
13       table-prefix: tbl_

```

也可以在mapper中使用注解TableName指定表名。

## 4.4 整合Druid

导入坐标，

```

32  <!-- https://mvnrepository.com/artifact/com.alibaba/druid
33  <dependency>
34    <groupId>com.alibaba</groupId>
35    <artifactId>druid-spring-boot-starter</artifactId>
36    <version>1.2.17</version>
37  </dependency>
38

```

添加配置有两种方式,

方式一, 通用型,

```
1 # 数据库连接信息
2 spring:
3   datasource:
4     url: jdbc:mysql://192.168.224.100/test
5     username: study
6     password: 123456
7     type: com.alibaba.druid.pool.DruidDataSource
```

方式二, Druid专用型,

```
9 spring:
10   datasource:
11     druid:
12       url: jdbc:mysql://192.168.224.100/test
13       username: study
14       password: 123456
```

配置完成。

## 5. SSMP整合案例

一个书籍信息的增删改查,

- 案例效果演示
- 案例实现方案分析
  - ◆ 实体类开发——使用Lombok快速制作实体类
  - ◆ Dao开发——整合MyBatisPlus，制作数据层测试类
  - ◆ Service开发——基于MyBatisPlus进行增量开发，制作业务层测试类
  - ◆ Controller开发——基于Restful开发，使用PostMan测试接口功能
  - ◆ Controller开发——前后端开发协议制作
  - ◆ 页面开发——基于VUE+ElementUI制作，前后端联调，页面数据处理，页面消息处理
    - 列表、新增、修改、删除、分页、查询
  - ◆ 项目异常处理
  - ◆ 按条件查询——页面功能调整、Controller修正功能、Service修正功能

## 5.1 使用lombok开发实体类

导入坐标,

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

在实体类上使用注解Data，即可自动生成Getter和Setter，以及hashCode、toString、equals等方法。

## 5.2 数据层开发

技术实现方案,

- MyBatisPlus (MP)
- Druid

MyBatisPlus新增数据时，对于主键有自动生成策略，如果要使用数据库的主键自增策略，需要添加配置如下，

```

14 mybatis-plus:
15   global-config:
16     db-config:
17       id-type: auto

```

MySQL重置主键自增值， `alter table book auto_increment=1;` 。

开启MP调试日志，在配置中添加如下设置，可以看出，这里是添加了日志的实现，下面添加的是控制台输出，

```

14 mybatis-plus:
15   global-config:
16     db-config:
17       id-type: auto
18   configuration:
19     log-impl: org.apache.ibatis.logging.stdout.StdOutImpl

```

分页查询功能，MP需要添加分页拦截器实现，

```

3 @Configuration
9 public class MpConfig {
10
11   @Bean
12   public MybatisPlusInterceptor mybatisPlusInterceptor() {
13     MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
14     interceptor.addInnerInterceptor(new PaginationInnerInterceptor());
15     return interceptor;
16   }
17
18 }

```

## 5.3 业务层开发

service层接口定义与dao层的接口的设计区别较大，需要注意，例如，在dao层定义的方法名 `getUserByNameAndPassword`，而在service层，对应的是登录功能，需要定义方法为 `login`。

可以使用MP简化业务层的开发，

在业务层接口继承IService，

```

1 implementation
public interface BookService extends IService<Book> {

    1 usage
    List<Book> getPage(Integer number, Integer size);

}

```

业务层实现类上，实现业务层接口，并继承ServiceImpl，

```

@Service
public class BookServiceImpl extends ServiceImpl<BookDao, Book> implements BookService {

    1 usage
    @Override
    public List<Book> getPage(Integer number, Integer size) {
        IPage<Book> p = new Page<>(number, size);
        return super.page(p).getRecords();
    }
}

```

由此，可以自动提供许多基础的service功能，自定义功能可以通过重载和新增方法。

## 5.4 表现层开发

技术实现，

- 使用Restful风格进行表现层开发
- 使用Postman测试表现层

Controller层开发如下，

```

@RestController
@RequestMapping(🌐"/books")
public class BookController {

    @Autowired
    private BookService bookService;

    @GetMapping(🌐)
    public List<Book> getAll() {
        return bookService.list();
    }

    @PostMapping(🌐)
    public Boolean save(@RequestBody Book book) {
        return bookService.save(book);
    }

    @PutMapping(🌐)
    public Boolean update(@RequestBody Book book) {
        return bookService.updateById(book);
    }

    @DeleteMapping(🌐"/{id}")
    public Boolean delete(@PathVariable Integer id) {
        return bookService.removeById(id);
    }

    @GetMapping(🌐"/{id}")
    public Book getById(@PathVariable Integer id) {
        return bookService.getById(id);
    }
}

```

## 5.5 表现层消息的一致性处理

上一小节的表现层开发中，给到前端的返回值类型不一致，有时候是JSON格式，有时候不是

JSON格式，需要做一致性处理，称为**前后端数据协议**。

设计统一的返回对象，

```
5  @Data
6  public class R {
7      private Boolean flag;
8      private Object data;
9
10     no usages
11     public R() {
12     }
13
14     3 usages
15     public R(Boolean flag) {
16         this.flag = flag;
17     }
18
19     3 usages
20     public R(Boolean flag, Object data) {
21         this.flag = flag;
22         this.data = data;
23     }
24 }
```

多个构造方法是为了方便使用。

controller改为，



```

public class BookController {

    @Autowired
    private BookService bookService;

    @GetMapping
    public R getAll() {
        return new R(flag: true, bookService.list());
    }

    @PostMapping
    public R save(@RequestBody Book book) {
        return new R(bookService.save(book));
    }

    @PutMapping
    public R update(@RequestBody Book book) {
        return new R(bookService.updateById(book));
    }

    @DeleteMapping("/{id}")
    public R delete(@PathVariable Integer id) {
        return new R(bookService.removeById(id));
    }

    @GetMapping("/{id}")
    public R getById(@PathVariable Integer id) {
        return new R(flag: true, bookService.getById(id));
    }

    @GetMapping("/{pageNum}/{pageSize}")
    public R getPage(@PathVariable Integer pageNum, @PathVariable Integer pageSize) {
        return new R(flag: true, bookService.page(pageNum, pageSize));
    }
}

```

完成。

## 5.6 对异常的统一处理

添加异常通知，捕获异常，统一返回响应对象R。

```
@RestControllerAdvice
public class ControllerExceptionHandler {

    @ExceptionHandler
    public R doException(Exception e) {
        e.printStackTrace();
        return new R( flag: false, msg: "服务器发生异常，请稍后重试");
    }
}
```

## 5.7 前端翻译的一些问题

在翻页中，有一些问题需要处理：

- 当前页只有一条数据时，删除数据，当前页为空
- 当前页数据条数与pageSize相同时，新增数据，需要手动翻页才能看到新增数据

参考：[翻页问题参考](#)