

SSM

SSM

- Spring framework
- SpringMVC
- Mybatis

SSM



SpringMVC

MVC

- Model
- View
- Controller



SpringMVC

SpringMVC

WebMvcConfigurationSupport

- Spring MVC

WebMvcConfigurer

- Spring Boot MVC
- WebMvcConfigurer

Spring

- Spring
- SpringMVC
- SpringBoot
- SpringCloud

Maven

Maven

Maven

- clean
- compile
- test
- package

- install命令

通过命令行安装到本地仓库

通过命令行发布到私服

- 通过命令行发布到私服
- 通过命令行发布到私服
- 通过命令行发布到私服



二、

通过命令行发布到私服

通过命令行

通过命令行

二、

- 通过Maven命令
- 通过pom.xml命令

```
<!-- 需要添加Apache Commons IO 依赖 -->
<dependency>
    <!-- 1. 组ID: 
        com/公司/项目名/模块名/版本号
        com Apache 项目名 -->
    <groupId>commons-io</groupId>

    <!-- 2. artifactID: 
        com/公司/项目名 Commons IO 项目名 -->
    <artifactId>commons-io</artifactId>

    <!-- 3. Version: 
        版本号2.11.0 项目名 -->
    <version>2.11.0</version>

    <!-- 4. Scope: 
        compile测试用例
        test测试providedJDK -->
    <scope>compile</scope> -->

    <!-- 5. Optional: 
        <exclusions> 项目名 </exclusions>
    <!-- <exclusions>
        <exclusion>
            <groupId>项目名</groupId>
            <artifactId>项目名</artifactId>
        </exclusion>
    </exclusions>
-->
```

```
</exclusions> -->
</dependency>
<! -- XML --> </dependency>-->
```

依赖管理

通过 pom.xml 文件实现对项目的依赖管理

- 项目+模块
- 通过父 pom.xml 实现
- 通过 web.xml 实现

父子

通过父 pom.xml 实现 java 项目的依赖管理



通过父 pom.xml 实现

父子 <parent> ... </parent>

父子

- jar 项目
- pom 项目
- war 项目 web Tomcat 项目
- 项目 <parent> pom </parent>

父子

1. 项目 pom
2. pom pom.xml 项目
◦ <relativePath> </relativePath> 项目
3. 项目

通过父 pom.xml 实现

父子

通过 pom.xml 项目 <dependencyManagement> 实现

- 项目
- 项目
- 项目

通过 properties

- <properties> </properties> 项目
- \${} 项目

```
<properties>
    <lombok.verison> 1.9.3 </lombok.verison>
</properties>
```

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <version> ${lombok.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- <dependencies> 旣存する依存関係を定義
- <dependencyManagement> 依存関係のバージョンを定義

□□

既存の依存関係を定義

- 依存関係を定義
- 依存関係を定義した pom ファイルを読み込む
- 例: <modules> 定義

```
<modules>
    <!-- 宗旨 -->
    <module> 旣1 </module>
    <module> 旣2 </module>
</modules>
```

- 依存関係を定義
- 依存関係を定義

□□

既存の依存関係

- 既存の依存関係を定義
- 既存の依存関係-->既->既存

□□□

- 既存の依存関係 1.0-SNAPSHOT
- RELEASE 既存の依存関係を既存の依存関係 RELEASE 旣
- SNAPSHOT 既存の依存関係を既存の依存関係 SNAPSHOT 旣

□□

既存の依存関係 settings.xml 旣maven既存の依存関係

- settings旣存の依存関係

```
<!-- settings.xml -->
<servers>
    <!-- releases 旣存の依存関係 -->
    <server>
        <id>maven-releases</id>
```

```

<username>admin</username>
<password>admin</password>
</server>

<!--snapshots-->
<server>
    <id>maven-snapshots</id>
    <username>admin</username>
    <password>admin</password>
</server>
</servers>

```

- pom.xml配置文件

```

<distributionManagement>
    <repository>
        <id>maven-releases</id>
        <url>.....</url>
    </repository>
    <snapshotRepository>
        <id>maven-snapshots</id>
        <url>.....</url>
    </snapshotRepository>
</distributionManagement>

```

- settings.xml配置文件

```

<!--settings.xml-->
<!--设置仓库地址，镜像地址等-->
<mirrors>
    <mirror>
        <id>maven-public</id>
        <mirrorOf>*</mirrorOf>
        <url>.....</url>
    </mirror>
</mirrors>

```

```

<profile>
    <id>allow-snapshots</id>
    <activation>
        <activeByDefault>true</activeByDefault> <!-- 激活Profile -->
    </activation>
    <repositories>
        <repository> <!-- 仓库 -->
            <id>maven-public</id> <!-- ID -->
            <url>http://192.168.150.101:8081/repository/maven-public/</url> <!-- 地址 -->
        </repository>
        <releases> <!-- RELEASE -->
            <enabled>true</enabled> <!-- 状态 -->
        </releases>
    </repositories>
</profile>

```

```

        </releases>
        <snapshots>          <!-- 禁用SNAPSHOT仓库 -->
            <enabled>true</enabled> <!-- 禁用仓库 -->
        </snapshots>
    </repository>
</repositories>
</profile>

```

SpringBoot

- SpringBoot和Spring Framework

依赖lombok

- @Data 生成get/set方法
- @AllArgsConstructor 生成所有参数构造器
- @NoArgsConstructor 生成无参数构造器

Http-注入

- SpringBoot和Tomcat的web容器
- Tomcat的HTTP请求解析器HttpServletRequest

RestController

- @RestController 生成Controller
- 自定义Bean @RestController("adminShopController") 生成Bean
- @RequestMapping 映射请求
 - value 和 method 属性
 - @PostMapping 和@PostMapping注解
- @PathVariable 路径变量 /depts/{id}
- @RequestParam 请求参数 /depts?id=123
 - 日期格式化 @DateTimeFormat(pattern="yyyy-MM-dd")
 - MultipartFile 多文件
- 线程池线程管理

```

package com.example.demo;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class RequestController {
    @RequestMapping("/request")
    public String request(HttpServletRequest request) {
        // 1.处理逻辑
    }
}

```

```

        String method = request.getMethod();
        System.out.println("方法名：" + method);
        // 2.请求的url
        String url = request.getRequestURL().toString(); //将URL放入StringBuffer中
        System.out.println("请求的url：" + url);

        String uri = request.getRequestURI(); // 请求的URI
        System.out.println("请求的uri：" + uri);
        // 3.请求的协议
        String protocol = request.getProtocol();
        System.out.println("请求的协议：" + protocol);
        // 4.请求的参数
        String name = request.getParameter("name");
        String age = request.getParameter("age");
        System.out.println("name：" + name + ", age：" + age);
        // 5.请求的头
        String accept = request.getHeader("Accept");
        System.out.println("Accept：" + accept);

        return "OK";
    }
}

```

- @Requestbody 将请求体放入json对象中

```

@RestController
public class UserController {

    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestBody User user) {
        // Spring 将 JSON 转为 User 对象
        System.out.println("用户名：" + user.getUsername());
        System.out.println("密码：" + user.getPassword());
        System.out.println("邮箱：" + user.getEmail());
        System.out.println("年龄：" + user.getAge());

        return ResponseEntity.ok("登录成功");
    }
}

```

Http-状态

- 200-299 3xx 重定向 4xx 错误 5xx 错误
- 200
- 500
- 404
- 通过 **HTTPServletResponse** 响应

- `jakarta.servlet.http.HttpServletResponse` ဗုဒ္ဓနအတွက်**body**၏

```
package com.example.demo;

import jakarta.servlet.http.HttpServletResponse;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.io.IOException;

@RestController
public class ResponseController {
    @RequestMapping("/response")
    public void response(HttpServletResponse response) throws IOException {
        // 1. အဆင့်
        response.setStatus(401);
        // 2. အဆင်
        response.setHeader("name", "whs");
        // 3. အဆင်
        response.getWriter().write("<h1>whs</h1>");
    }
    // အဆင်
    @RequestMapping("/response2")
    public ResponseEntity<String> response2(){
        return ResponseEntity.status(301).header("name", "whs").body("<h1>whs</h1>");
    }
}
```

ဗုဒ္ဓန

- Controller ဗုဒ္ဓနအတွက်
- Service ဗုဒ္ဓနအတွက်
- Dao ဗုဒ္ဓနအတွက်

ဗုဒ္ဓနအတွက်

ဗုဒ္ဓနအတွက်**Springboot** ဗုဒ္ဓနအတွက်**IOC**၊ **DI**

ဗုဒ္ဓန**IOC**

- ဗုဒ္ဓနအတွက်**Bean**၏
- ဗုဒ္ဓနအတွက်

```
public interface EmailSender {
    void sendEmail(String to, String message);
}
```

`@Component`

```
public class SmtpEmailSender implements EmailSender {  
    @Override  
    public void sendEmail(String to, String message) {  
        // ...SMTP...  
    }  
}
```

www.IOC.org

- `@Component` 
 - `@Service`

```
public interface EmailSender {  
    void sendEmail(String to, String message);  
}  
  
@Service  
public class XX implements EmailSender {  
    @Override  
    public void sendEmail(String to, String message) {  
    }  
}
```

- `@Repository` 

```
public interface DataLoader {  
}  
  
@Repository  
public class XX implements DataLoader {  
}
```

- `@RestController`

```
// ��ָ��  
@RestController  
public class XX {  
}
```

10

-  Service 

6

- 

```
@RestController
public class UserController(){
    private UserService userService;

    @Autowired
    public UserController(UserService userservice){
        this.userService = userservice;
    }
}
```

- `UserService`

```
@RestController
public class UserController(){
    @Autowired
    private UserService userService;
}
```

- setting

```
@RestController
public class UserController(){
    private UserService userService;

    @Autowired
    public void setUserService(UserService userservice){
        this.userService = userservice;
    }
}
```

□□□□□

- `@Autowired` □□□□□□□Bean
 - `UserService` `@Component`
- `@Qualifier` □□□□□ Bean □□□ `@Qualifier(bean_name)`

```
@RestController
public class UserController(){
    @Autowired
    @Qualifier(bean_name)
    private UserService userService;
}
```

- `@Primary` □□□□□□□□□□□□□

```
// □□□□□
@Primary
```

```
@Service
public class UserServiceImpl2 implements UserService{

}
```

- `@Resource` ကြည့်မှု Bean မှာ `@Resource(name=bean_name)`
 - `@Resource` မှ `@Autowired` မှုပါ
 - `@Resource` ကြည့်မှု/အသိပေးမှု Bean မှာ အသိပေးမှုပါ
 - `@Autowired` ကြည့်မှု/အသိပေးမှု **Bean**

1

- `@ControllerAdvice` @RestControllerAdvice 之間
 - `@ControllerAdvice` ExceptionHandler 之間
 - `ExceptionHandler`springboot 之間

```
// 用于全局异常处理
@Slf4j
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler
    public Result handleException(Exception e) {
        log.error(e.getMessage());
        return Result.error(e.getMessage());
    }
}
```

JDBC

API

1

1. ००००
 2. ००००
 3. ००**SQL**००००००
 4. ००SQL
 5. ००००

```
import java.sql.DriverManager  
public class Jdbc{  
    public void test(){  
        // 1. ȿȿȿȿ  
        Class.forName("ȿȿȿ");  
        // 2. ȿȿȿȿ  
        Connection conection = DriverManager.getConnection(URL,username,password);  
        // 3. ȿȿȿȿȿȿ  
        Statement statement = conection.createStatement();  
        // 4. ȿȿSQLȿȿȿȿ  
        statement.executeUpdate(sqlȿ);
```

```
// 5. 關閉  
statement.close();  
connection.close();  
}  
}
```

二、DQL

- `ResultSet` 宣傳語句: `ResultSet rs = statement.executeQuery()`
 - `next()` 依序回傳結果集的每一列資料
 - `getXXX()` 取得欄位值

```
String sql = "SELECT * FROM user";  
  
stmt = conn.prepareStatement(sql);  
  
ResultSet rs = stmt.executeQuery();  
while(rs.next()) {  
    User user = new User(  
        rs.getInt("id"),  
        rs.getString("username"),  
        rs.getString("password"),  
        rs.getString("name"),  
        rs.getInt('age')  
    );  
    System.out.println(user);  
}  
}
```

三、SQL

- 常見的SQL語句
- 常見SQL語句
- 常見SQL語句

Mybatis

Mybatis是用於DAO層的JDBC工具

特點

- 在 `application.properties` 定義資料庫
 - 資料庫名稱、連接池、方言等

特點

- 在 `mybatis-config.xml` 定義 `@Mapper` 介面
 - `Mapper` 是 `Spring` 容器管理的 `Bean`
 - `Mapper` 定義方法
- 在 `Mapper` 定義方法中，使用 `sql` 定義SQL語句，由 `mybatis` 完成SQL語句的執行

```
// 1
@Mapper
public interface UserMapper{
    @Select("select * from user")
    public List<User> findAll();
}

// 2
public class Test{
    @Autowired
    private UserMapper usermapper;
    public void testQuery(){
        List<User> list = usermapper.findAll();
    }
}
```

- application.properties 1
- MySQL IDEA Mysql 2

3

- @Results @Result 3
- ④Mapper 4

```
@Results({
    @Result(column="update_time",properties="updateTime")
})
```

5

```
@Select("select id,name,create_time createTime,update_time updateTime from
dept order by update_time desc")
```

6

- ④驼峰命名 create_time 6 createTime

```
map-underscore-to-camel-case: true
```

7

- ④ JDBC Connection 7
- ④ mybatis 8
 - mybatis 8

9

- ④ Datasource 9

- `getConnection()` 用于
建立连接
 - Spring `Hikari` 用于
连接池
- `#{id}` 用于
插入语句
- `Integer` 用于
DML 语句

```
@Mapper
public interface UserMapper{
    @Delete("delete from user where id = #{id}")
    public void deleteById(Integer id);
}
```

- `insert` 语句
- `values` 语句

```
@Mapper
public interface UserMapper{
    @Insert("insert into user(username,password,name) values(#{username},#{password},#{name})")
    public void insert(User user);
}

public class Test{
    @Autowired
    private UserMapper userMapper;

    public testInsert(){
        User user = new User("1",123,"2");
        userMapper.insert(user);
    }
}
```

- `update` 语句
- `SQL` 语句

```
@Mapper
public interface UserMapper{
    @Update("update user set username = #{username},password = #{password},name = #{name} where id = #{id}")
    public void update(User user,Integer id);
}

public class Test{
```

```
@Autowired  
private UserMapper usermapper;  
  
public testUpdate(){  
    Integer id = 1;  
    User user = new User("1",123,"2");  
    usermapper.update(user,id);  
}  
}
```

1

- `④ @Param` ⑤SQL⑥
 - ⑦springboot⑧SQL⑨

```
// ��
@Mapper
public interface UserMapper{
    @Select("select * from user where username=#{username} and password=#{password}")
    public User find(@Param("username")String username,@Param("password")String
password);
}

// ��
public class Test{
    @Autowired
    private UserMapper usermapper;
    public void testQuery(){
        User user = usermapper.find("��", "123456");
    }
}
```

[] concat []

```
select e.* ,d.name deptName from emp e left join dept d on d.id = e.dept_id  
where  
    e.name like concat('%',#{name},'%')  
    and e.gender = #{gender}  
    and e.entry_date between #{begin} and #{end}  
order by update_time desc
```

XML

- mybatis 通过注解实现SQL语句的 XML映射器 SQL语句的自动注入
 - 三种方式
 - XML映射器 Mapper XML映射器 XMLMapper
 - XML映射器 namespace Mapper XMLMapper
 - XML映射器 sql id Mapper XMLMapper

□□□□□□□ / □□□□□□□

```
// 代码
@Mapper
public interface UserMapper{
    public List<User> findAll();
}
```

- `id`
- `resultType`

```
<mapper namespace="com.zxb.mapper.UserMapper">
    <select id = "findAll" resultType="com.zxb.pojo.User">
        select id, username, password, name, age from user
    </select>
</mapper>
```

配置

MyBatis XML 配置文件

- 在 `application.properties` 中配置 XML 配置文件位置
 - `mybatis.mapper-locations=classpath:mapper/*.xml` 表示从 mapper 目录下读取所有 XML 文件
 - `src/main/resources` 和 `src/main/java` 目录下的 Java 文件
- MyBatis 官方推荐的 Java 配置方式
 - `type-aliases-package:com.sky.entity` 表示从 com.sky.entity 包下读取别名

通过 mybatisx 配置文件

SQL

MyBatis SQL 语句

- `<if>` 标签表示当 true 时执行 SQL
 - `test` 表达式
- `<where>` 标签表示 where 子句，可以使用 `and` 或 `or`
 - `>`
 - `<`
 - `xiao'yu`
- `<foreach>` 标签
 - `collection` 表示集合
 - `item` 表示元素
 - `separator` 表示分隔符
 - `open` 表示打开括号
 - `close` 表示关闭括号
- `<set>` 标签表示 set 子句

4

□□□□□□□□□□□□

- ☰ Options ☱

- `useGeneratedKeys` `Boolean` `true`
 - `keyProperty` `String` `id`

```
@Options(useGeneratedKeys = true, keyProperty = "id")
// 通过emp.id插入
void insert(Emp emp);
```

4

□□□□□ user □□□ department □□

1

- 二〇二〇年九月三十日
 - 二〇二〇年十一月三十日
 - 二〇二〇年十二月三十日
 - 二〇二一年一月三十日
 - 二〇二一年三月三十日
 - 二〇二一年五月三十日
 - 二〇二一年七月三十日
 - 二〇二一年九月三十日

```
-- 临时表
-- ALTER TABLE `user` ADD CONSTRAINT `tmp_dept_id` FOREIGN KEY (`dept_id`) REFERENCES `department`(`id`);

-- constraint tmp_dept_id foreign key (dept_id) references department(id)
alter table user add constraint tmp_dept_id foreign key (dept_id) references department(id);

-- 临时表
-- 临时表的外键约束
alter table user drop foreign key tmp_dept_id;
```

三

-                                
 -                                                                              <img alt="Icon representing a

```
-- ①
-- ②card③
create table card(
    id int primary key,
    card varchar(18)
```

```
);

-- 用户表
create table user(
    id int primary key,
    -- 姓名
    -- 性别
    card_id int not null unique,
    constraint user_card_id foreign key (card_id) references card(id)
);
```

三

- 亂世の政治家
 - 亂世の政治家
 - 亂世の政治家
 - 亂世の政治家



1

- 1 1 2

```
-- ①②③④⑤⑥  
select * from user,department;
```

-

1



- 
 -

```
select * from user as a inner join department as b on a.dept_id = b.id;
```



```
select * from user as a,department as b where a.dept_id = b.id;
```

- AとBの結合

- AとBの左結合

```
select * from user a left outer join department b on a.dept_id = b.id;
```

- AとBの右結合

```
select * from user a right outer join department b on a.dept_id = b.id;
```

- on クリックして表示

- where クリックして表示

…

SQL文 select クリックして表示

- SQL文

```
-- データ
```

```
-- データ
```

```
select salary from user where name = "田中";
```

```
-- データ
```

```
-- データ
```

```
select * from user where salary = ?;
```

```
-- データ
```

```
select * from user where salary = (select salary from user where name = "田中");
```

- SQL文

```
-- データ
```

```
-- データ
```

```
-- データ
```

```
select id from dept where name = "人事" or name = "開発";
```

```
-- データ
```

```
-- データ
```

```
select * from user where dept_id = ?;
```

```
-- データ
```

```
select * from user where dept_id in (select id from dept where name = "人事" or name = "開発");
```

```
-- 传统SQL  
-- 语句  
select salary,job from user where name = "王五";  
  
-- 语句  
-- 语句  
-- 语句  
-- 语句  
select * from user where salary = ? and job = ?;  
  
-- 语句  
select * from user where salary = (select salary from user where name = "王五")  
and job = (select job from user where name = "王五");  
  
select * from user where (salary,job) = (select salary,job from user where  
name = "王五");
```

- 

-

```
-- 语句1  
  
-- 语句2  
select dept_id ,max(salary) from user group by dept_id;  
  
-- 语句3  
  
-- 语句4  
  
-- 语句5  
select * from user a, (select dept_id ,max(salary) max_salary from user group  
by dept_id) b where a.dept_id = b.dept_id and a.salary = b.salary;  
  
-- 语句6  
select * from user a inner join (select dept_id ,max(salary) max_salary from  
user group by dept_id) b on a.dept_id = b.dept_id where a.salary = b.salary;
```

11

- `开始事务` start transaction `开始` begin
 - `提交` commit
 - `回滚` rollback

begin

```
-- 事务sql语句
```

```
-- 提交  
commit;
```

```
-- 回滚  
rollback;
```

Spring事务管理

- 标注事务的注解 @Transactional 包含
 - rollbackFor 指定回滚的异常类型
 - RuntimeException 以及它的子类 Error 会回滚

```
public class Test{  
    @Transactional(rollbackFor = {Exception.class})  
    public void save(){  
  
    }  
}
```

- propagation 属性
 - 表示传播行为，如果当前有事务，则执行在当前事务中；如果没有事务，则自动创建一个新事务
 - REQUIRED 表示如果当前有事务，则执行在当前事务中；如果没有事务，则自动创建一个新事务
 - REQUIRES_NEW 表示如果当前有事务，则执行在新事务中；如果没有事务，则自动创建一个新事务

```
@Service  
public class WorkParent{  
  
    @Autowired  
    private WorkChild workchild;  
  
    @Transactional  
    public void save(){  
        try{  
            /**  
             *  
             */  
        }  
        finally{  
            // 提交  
            workchild.save();  
        }  
    }  
}  
  
@Service  
// WorkChild  
public class WorkChild{
```

```

    // 事务传播
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void save(){
        /**
         * ...
         */
    }
}

```

- 事务传播

```

logging:
level:
org.springframework.jdbc.support.JdbcTransactionManager: debug

```

日志

- Spring 日志输出

配置

通过 PageHelper 配置

配置

```

<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper-spring-boot-starter</artifactId>
    <version>2.1.0</version>
</dependency>

```

- Mapper 配置

```

@Select("select e.*,d.name deptName from emp e left join dept d on d.id = e.dept_id
order by e.update_time desc ")
List<Emp> list();

```

- Service 通过 PageHelper 实现
 - 定义方法
 - .getTotal() 实现
 - .getResult() 实现
 - 定义自定义的 Select 实现

```

@Override
public PageResult<Emp> page(Integer page, Integer pageSize){
    // 1. 定义方法
    PageHelper.startPage(page,pageSize);

    // 2. 执行
}

```

```
List<Emp> rows = empMapper.list();

// 3.分页查询
Page<Emp> p = (Page<Emp>) rows;
return new PageResult<Emp>(p.getTotal(),p.getResult());
```

1

Mybatis

Mapper.xml

- CRUD에서 resultMap 설정
 - resultMap 설정
 - id 설정 type pojo 설정
 - id 설정
 - result 설정
 - column 설정
 - property 설정

三

```
<resultMap id="empResultMap" type="com.zxb.pojo.Emp">
    <id column="id" property="id"/>
    <result column="username" property="username"/>
    <result column="password" property="password"/>
    <result column="name" property="name"/>
    <result column="gender" property="gender"/>
    <result column="image" property="image"/>
    <result column="entry_date" property="entryDate"/>
    <result column="dept_id" property="deptId"/>
    <result column="create_time" property="createTime"/>
    <result column="update_time" property="updateTime"/>

<!--exprList-->
<collection property="exprList" ofType="com.zxb.pojo.EmpExpr">
    <id column="ee_id" property="id"/>
    <result column="ee_company" property="company"/>
    <result column="ee_job" property="job"/>
    <result column="ee_begin" property="begin"/>
    <result column="ee_end" property="end"/>
    <result column="ee.empid" property="empId"/>
</collection>
</resultMap>

<select id="findById" resultMap="empResultMap">
    select
        e.*,

```

```
ee.id ee_id,
ee.company ee_company,
ee.job ee_job,
ee.begin ee_begin,
ee.end ee_end,
ee.emp_id ee.empid
from emp e left join emp_expr ee on e.id = ee.emp_id
where e.id = #{id}
</select>
```

SQL

SQL

SQL case 表达式语句

- when 表达式语句
- then 表达式语句

select

```
(case when job=1 then '经理'
      when job=2 then '副经理'
      when job=3 then '总监'
      when job=4 then '项目经理'
      when job=5 then '高级工程师'
      else '普通员工' end) pos,
count(*) num
from emp group by job order by num
```

SQL

SQL if 表达式语句

select

```
if(gender=1,'男','女') name,
count(*) value
from emp group by gender
```

SpringBoot配置

application.properties , yml , yaml

- properties 属性 key value 配置 . 配置
- yml 配置文件
 - 属性文件
 - 属性文件
 - 属性文件 0 属性文件 '' 属性文件0 属性文件 yml 配置文件
 - yml 配置文件 属性文件 @Value 属性文件

```
@Value("${user.name}")
private String name;
```

- **Java** `@Value("${user.name}")` `private String name;` @ConfigurationProperties `prefix` `user` `yml` `yaml` Bean 为

```
@Data
@Component
@ConfigurationProperties(prefix = "aliyun.oss")
public class AliyunOSSProperties {
    private String endpoint;
    private String bucketName;
    private String region;
}
```

```
## list/set
```

```
hobby:
```

- java
- game
- sport

```
## map
```

```
user:
```

```
name: 陈
age: 18
password: 123456
```

SpringBoot配置

SpringBoot配置文件有三个层次：Java配置类、properties/yml/yaml配置文件、application.properties/yml/yaml配置文件。

Java配置类

properties/yml/yaml配置文件

- `application-{profile}.yml` 和 `application-{profile}.properties` 为
- `application.yml` 和 `application.properties`

```
spring:
profiles:
active: dev # 为 dev 环境
```

application.properties/yml/yaml

application

- **Nginx**
- **Tomcat**

- RESTful API

Restful

REST API は、RESTful API と呼ばれます。

RESTful API は、以下の特徴を持っています。

RestfulUrl	メソッド	説明
http://localhost:8080/users/1	GET	id=1のデータ
http://localhost:8080/users/1	DELETE	id=1のデータ
http://localhost:8080/users	POST	新規登録
http://localhost:8080/users	PUT	更新

Apifox/Postman

API マネージャーとして Mock API を実装する

API フレームワーク

API フレームワーク LogBack

- ログ出力用 XML ファイル logback.xml
- ログ出力用のアノテーション
- ログ出力用の属性
 - @Slf4j と Lombok を用いる

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class UserService {
    public void processUser(int age) {
        log.info("{}", age);
    }
}
```

- final 属性
 - getLogger メソッドを用いる

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
public class UserService{
    private static final Logger logger =
LoggerFactory.getLogger(UserService.class);
    public void processUser(int age){
        logger.info("{}", age);
```

```
    }
}
```

- `logger.info("{}")`

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class UserService {
    public void processUser(int age) {
        log.info("{}", age);
    }
}
```

输出

- `logger.level`
 - off
 - all
 - trace, debug, info, warn, error
- `logger STDOUT FILE`
- `logger appender`
- `logger`

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!-- 根节点 -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <!-- 文件 -->
    <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>logs/application.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>logs/application.%d{yyyy-MM-dd}.log</fileNamePattern>
            <maxHistory>30</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <!-- 根节点 -->
    <root level="INFO">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

```

<appender-ref ref="FILE" />
</root>

<!-- 例のロギング設定 -->
<logger name="com.example.service" level="DEBUG" />
<logger name="org.springframework" level="WARN" />
</configuration>

```

日志

SpringBootでweb開発を行う際に重要な機能

ログ出力機能

cookie



- クッキーはHTTPリクエストとレスポンスの間に渡るデータ
- クッキーを設定する際はcookie名をSet-Cookie:cookieとしてレスポンスに付けてcookieを設定する
- クッキーを取得する
 - Cookie名 cookie 値
- cookieを削除

クッキーをURLに付けて渡す

```

@RestController
public class AuthController {

    @GetMapping("/login")
    public ResponseEntity<String> login(HttpServletRequest response) {
        // クッキー
        Cookie cookie = new Cookie("auth_token", "user123");
        cookie.setMaxAge(3600); // 1時間
        cookie.setHttpOnly(true); // XSS対策
        cookie.setPath("/"); // ルート

        response.addCookie(cookie);
        return ResponseEntity.ok("ログイン");
    }

    @GetMapping("/profile")
    public String profile(@CookieValue("auth_token") String token) {
        return "Token: " + token;
    }
}

```

session



- **Cookie**
- **Session** Set-Cookie session——sessionId
 - sessionId
- **Nginx**
- HttpSession 的 setAttribute 和 getAttribute

```
@RestController
@SessionAttributes("user")
public class SessionController {

    @GetMapping("/setSession")
    public String setSession(HttpSession session) {
        session.setAttribute("username", "张三");
        session.setAttribute("loginTime", new Date());
        return "Session设置";
    }

    @GetMapping("/getSession")
    public String getSession(HttpSession session) {
        String username = (String) session.getAttribute("username");
        Date loginTime = (Date) session.getAttribute("loginTime");
        return "用户名: " + username + ", 登录时间: " + loginTime;
    }
}
```

二



JWT

一

- Header
- json

二

- Header
- Payload
- Signature token

三

- Header Payload Base64URL

- Signature \square **Header** \square **Payload** \square

.....

- **Header** \square **Payload** \square
- **Header** \square **Header** \square **Payload** \square

.....

- jjwt
- **jjwt**
 - Jwts builder JWT.....
 - signWith
 - setClaims
 - setExpirationDate.....
 - compact
- **jjwt**
 - parser
 - setSigningKey
 - parseClaimsJws
 -
 - getBody()payload

```
public class JwtUtils{
    // ...
    private static final SecretKey SECRET_KEY =
    Keys.secretKeyFor(SignatureAlgorithm.HS256);
    // ...
    private static final long EXPIRATION_TIME = 3600 * 1000;

    public static String generateToken(Map<String, Object> claims){
        return Jwts.builder()
            .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
            .setClaims(claims)
            .setExpiration(new Date(System.currentTimeMillis() +
EXPIRATION_TIME))
            .compact();
    }
    public static Claims parseToken(String token){
        return Jwts.parser()
            .setSigningKey(SECRET_KEY)
            .parseClaimsJws(token)
            .getBody();

    }
}
```

Filter.....

JavaWeb.....

.....

- Filter Filter init doFilter destroy
 - HttpServletRequest HttpServletRequest
 - filterChain.doFilter() FilterChain
 - filterChain FilterChain
- @WebFilter
 - urlPatterns
 - /login
 - /*
 - exp/*
- @ServletComponentScan @WebFilter
-

```

import javax.servlet.*;

@.Slf4j // ०००००
@WebFilter("/*") // ०००००
public class LoginFilter implements Filter{
    @Override
    public void init(FilterConfig filterConfig){
        log.info("०००.....");
    }

    @Override
    public void doFilter(ServletRequest httprequest, ServletResponse httpresponse,
FilterChain filterChain){
        log.info("०००००");
        HttpServletRequest request = (HttpServletRequest) httprequest;
        HttpServletResponse response = (HttpServletResponse) httpresponse;

        // ०००००००
        String requestURI = request.getRequestURI();

        // ०००००००token
        String token = request.getHeader("token");

        // ००token०००
        if(token == null){
            log.info("०००००");
            // ०००००
            response.setStatus(401);
            return;
        }

        // ०००००
        try{
            JwtUtils.parseToken(token); // ०००००००००
        }
        catch(Exception e){
            log.info("०००००");
        }
    }
}

```

```

        // ...
        response.setStatus(401);
        return;
    }

    // ...
    filterChain.doFilter(request, response);
}

@Override
public void destroy(){
    log.info("....");
}
}

```

.....

- `init()`
- `doFilter()`
- `destroy()`

Interceptor

Spring `HandlerInterceptor`

.....



.....

- `@Component` `HandlerBean`
- `HandlerInterceptor` `HandlerInterceptor`
 - `preHandle` `True` `False`
 - `postHandle`
 - `afterCompletion`
- `HandlerInterceptorAdapter`
 - `HandlerInterceptor`
 - `@Configuration`
 - `WebMvcConfigurer` `addInterceptors` `registry.addInterceptor`
 - `addPathPatterns` `excludePathPatterns`
 - `/*`
 - `/*`
 - `.order()`

.....

```

@Slf4j
@Component
public class LoginInterceptor implements HandlerInterceptor{
    @Override

```

```

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler){
    /*
     * ...
     */
    return true; // ...
}

@Override
public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView){
    log.info("Controller调用");
}

/**
 * ...
 */
@Override
public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {
    log.info("方法完成");
}
}

```

方法

```

@Configuration
public class Webconfig implements WebMvcConfigurer{
    @Autowired
    private LoginInterceptor interceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry){
        // ...
        registry.addInterceptor(interceptor).addPathPatterns("/**").excludePathPatterns("/login")
    }
}

```

SpringAOP

AOP：面向切面编程

面向对象编程+AOP编程

- 面向对象
- 面向接口
- 面向属性
- 面向事件

SpringAOP

- **SpringAOP**
- AOP
Aspect Oriented Programming
- 横切关注点的处理，通过增强类实现

二、AOP

- **@Component** 或 **@Aspect** 标注的类
- **ProceedingJoinPoint** 接口的 **proceed** 方法返回值为 **Object**
- **@Around** 注解标注的 AOP 处理方法

```
// @Around
@Aspect
@Component
public class TestAop {
    // com.zxb.project.*.*.*(*)
    @Around("execution(* com.zxb.project.*.*(..))")
    public Object test(ProceedingJoinPoint pj) throws Throwable{
        System.out.println("环绕");
        // 代理对象
        Object result = pj.proceed();
        return result;
    }
}
```

三、AOP

- 基本的 AOP 框架
- 代理模式
- 通过切面将横切关注点从业务逻辑中分离出来 AOP 框架
- 容器管理
- 配置管理

Spring AOP

四、AOP

注解	说明
@Around	环绕通知
@After	方法执行后通知
@Before	方法执行前通知
@AfterReturning	方法正常返回后通知
@AfterThrowing	方法异常抛出后通知

- **@Around** 环绕通知

```
@Component
@Aspect
public class TestAop{
    // Around...
```

```

@Around("execution(* com.zxb.project0.*(..))")
public Object testAround(ProceedingJoinPoint pj) throws Throwable{
    System.out.println("环绕");
    Object result = pj.proceed();
    return result;
}

// @Before
// 用于前置通知
@Before("execution(* com.zxb.project1.*(..))")
public void testBefore(){
    System.out.println("前置");
}

// @After
// 用于后置通知
@After("execution(* com.zxb.project2.*(..))")
public void testAfter(){
    System.out.println("后置");
}
}

```

测试类

com.zxb.project0

- 用于前置通知
 - @Before
 - @BeforeOrder
- 用于后置通知
 - @After @Order
 - @Before
 - @After

```

@Component
@Aspect
@Order(3)
public class AopOrder0{
    @Before("execution(* com.zxb.project.*(..))")
    public void testBefore() {
        System.out.println("AopOrder0-前置");
    }
    @After("execution(* com.zxb.project.*(..))")
    public void testAfter() {
        System.out.println("AopOrder0-后置");
    }
}

@Component
@Aspect
@Order(5)
public class AopOrder1{

```

```

    @Before("execution(* com.zxb.project.*(..))")
    public void testBefore() {
        System.out.println("AopOrder1-00");
    }
    @After("execution(* com.zxb.project.*(..))")
    public void testAfter() {
        System.out.println("AopOrder1-00");
    }
}

```

测试结果

- AopOrder0.testBefore
- AopOrder1.testBefore
- AopOrder1.testAfter
- AopOrder0.testAfter

注解

execution通知

- execution 通知
 - 通知类型：环绕通知
 - 通知表达式：execution(方法名? 包名.类名.?方法(参数)) throws 异常
 - 通知点：切入点
 - 通知类型
 - 通知表达式
 - throws 异常
 - * 表示所有方法
 - .. 表示所有包

```

// 环绕通知
@Aspect
@Component
public class TestAop {
    // com.zxb.project包下所有方法的环绕通知
    @Around("execution(* com.zxb.project.*.*(..))")
    public Object test(ProceedingJoinPoint pj) throws Throwable{
        System.out.println("00");
        // 执行通知体
        Object result = pj.proceed();
        return result;
    }
}

```

- @annotation 通知
 - 通知类型：基于注解通知

```

// 用于注解
// 用于方法
@Target(ElementType.METHOD)
public @interface LogOperation{
}

// 用于类
public class Test{
    @LogOperation
    public void test(){
        System.out.println("hello");
    }
}

// 用于类
@Aspect
@Component
public class TestAop {
    // 用于LogOperation注解的方法
    @Before("@annotation(com.zxb.project.LogOperation)")
    public void test(){
        System.out.println("AOP");
    }
}

```

注意

- 通过 com.zxb.project 包名匹配 * com.zxb.project.*(..)
- 通过 com.zxb.project 包名匹配 * com.zxb.project..*.*(..)
 - ..* 匹配 com.zxb.project 包下的所有类

匹配点切面

通过 @Pointcut 定义匹配点

- 通过包名匹配 @Pointcut 定义的匹配点
- 通过 private 匹配的匹配点
- 通过空字符串 "" 匹配的匹配点

```

@Component
@Aspect
public class AopOrder{

    // 用于方法
    @Pointcut("execution(* com.zxb.project.*(..))")
    private void all(){}
}

```

```

    @Before("all()")
    public void testBefore() {
        System.out.println("AopOrder-前置");
    }

    @After("all()")
    public void testAfter() {
        System.out.println("AopOrder-后置");
    }
}

```

二、Spring AOP

Spring 中 JointPoint 表示切入点

- 前置 @Around 切入点 ProceedingJoinPoint
- 后置 @After 切入点 JoinPoint

JointPoint 接口方法

- getTarget() 返回目标对象
- getTarget().getClass().getName() 返回目标对象的类名
- getSignature() 返回方法签名
- getArgs() 返回参数

```

@Component
@Aspect
public class AopTest{
    @Before("execution(* com.zxb.project..*.*(..))")
    public void test(JoinPoint jp){
        // 目标对象
        Object target = jp.getTarget();

        // 方法名
        String name = jp.getTarget().getClass().getName();

        // 方法名
        String methodName = jp.getSignature().getName();

        // 参数
        Object[] args = jp.getArgs();
    }
}

```

Bean

容器 IOC 容器管理 Bean

容器 IOC 容器管理 Bean

1. IOC 容器

- ApplicationContext Spring Framework 容器 Springboot 容器

- 例

- `getBean()` 通过 BeanName 调用 Bean

```
@Component
public class SimpleService {
    public void sayHello() {
        System.out.println("Hello from SimpleService!");
    }
}

@Component
public class Test{
    // ApplicationContext
    @Autowired
    private ApplicationContext applicationContext;
    public void test(){
        System.out.println(applicationContext.getBean("simpleService"));
    }
}
```

- Bean

- `singleton` IOC 容器管理 bean
- `prototype` IOC 容器管理 bean

Bean 容器管理

- `singleton` IOC 容器管理 bean
- `prototype` IOC 容器管理 bean

例 `@Scope` IOC 容器管理

```
@Scope("prototype")
@Component
public class TestScope{
    public void test(){
        System.out.println("测试");
    }
}

// 例
public class Test{
    // ...
    @Autowired
    private TestScope testScope;
}
```

IOC 容器管理 bean

- `singleton` IOC 容器管理 singleton

- **构造函数注入**
- **通过bean属性注入 prototype**
 - **通过构造器注入bean属性注入**

三、@Bean 定义方式

- **通过@Configuration @Configuration 定义**
- **通过@Component @Component 定义**
 - **通过构造器注入**
 - **通过方法注入Bean属性**
 - **通过new关键字注入 Bean属性**
 - **通过 @Bean 定义**

```
public class ThirdWay{
    /*
    三种
    */
}

@Configuration
public class Config{
    @Bean
    public ThirdWay thirdWay(){
        return new ThirdWay();
    }
}

// 第二种
@Component
public class MainProcess{
    @Autowired
    private ThirdWay thirdWay;
}
```

SpringBoot 定义

1. 定义

SpringBoot 定义：通过 maven 插件自动完成对 Spring 的配置

2. 定义

通过 spring 容器对 bean 进行 IOC 定义

- **通过 ApplicationContext 容器对 bean 进行 IOC 定义**
- **SpringBoot 容器对 bean 进行 IOC 定义**

通过 Springboot 容器对 @ComponentScan 容器对 basePackages 容器对 IOC 定义

- **通过@ComponentScan 容器对 basePackages 容器对 IOC 定义**

```
@ComponentScan(basePackages={"com.example"})
@SpringBootApplication
public class Begin{

}
```

ImportSelector @Import 选择器
ImportSelector
ImportSelector ImportSelector 选择器 @Import 选择器

```
public class ImportConfig implements ImportSelector{
    public String[] selectImports(AnnotationMetadata importClassMetadata){
        return new String[]{"com.example.Hello","com.example.Goodbye"};
    }
}

@Import(ImportConfig.class)
@SpringBootApplication
public class Begin{
    @Autowired
    private Hello hello; // 选择器
}
```

EnableXXX @EnableXXX 启动器
启动器

Springboot启动器 @SpringBootApplication 启动器

- @SpringBootConfiguration 启动器 @Configuration 启动器
- @ComponentScan 启动器
- @EnableAutoConfiguration 启动器
 - 选择器 @Import 选择器 ImportSelector 选择器 SelectImports 选择器
选择器
 - 通过注解 @Import 选择器 ImportSelector 选择器 SelectImports 选择器
 - 通过注解 @Configuration Springboot启动器 @Configuration 启动器
启动器 Bean IOC
 - 启动器 @Bean 启动器 IOC

```
@Configuration
public class Config{
    @Bean
    public ThirdWay thirdWay(){
        return new ThirdWay();
    }
}
```

```

    }

    // ...
    @Component
    public class MainProcess{
        @Autowired
        private ThirdWay thirdWay;
    }

```

三、@Conditional 介绍

- 通过注解对bean的生命周期进行控制，bean是否被SpringIOC管理
- 用途广泛
 - @ConditionalOnClass 某些类存在时，某些bean被IOC管理
 - name：类名
 - @ConditionalOnMissingBean 某些类不存在时，某些bean被IOC管理
 - ConditionalOnProperty 某些属性存在时，某些bean被IOC管理
 - name：属性名
 - havingValue：值

四、starter

- 启动器（Starter）
- 通过 AutoConfiguraion 容器自动加载 XXXAutoConfiguration
- 在META-INF/spring.factories文件中 META-INF/spring/.....
- 在META-INF/starter文件夹中

五、POJO

- POJO 包含java
 - Entity 实体类
 - DTO 数据传输对象
 - VO 视图对象

调用 Bean.utils.copyProperties(DTO,Entity) 完成转换

使用 @Builder 容器生成构造方法

```

Employee employee = Employee.builder().
    id(id)
    .status(status)
    .build();

```

Swagger

Knife4j 客户端

1. 在pom.xml中添加knife4j-maven依赖
2. 在项目中引入knife4j

3. Swagger API 文档生成

1 Bean 定义

- docket 定义
- 文档信息
- Swagger API 定义

```
@Bean
public Docket docket() {
    ApiInfo apiInfo = new ApiInfoBuilder()
        .title("Swagger API")
        .version("2.0")
        .description("Swagger API")
        .build();
    Docket docket = new Docket(DocumentationType.SWAGGER_2)
        .apiInfo(apiInfo)
        .select()
        // ...
        .apis(RequestHandlerSelectors.basePackage("com.sky.controller"))
        .paths(PathSelectors.any())
        .build();
    return docket;
}
```

2 配置

```
protected void addResourceHandlers(ResourceHandlerRegistry registry) {

    registry.addResourceHandler("/doc.html").addResourceLocations("classpath:/META-INF/resources/");

    registry.addResourceHandler("/webjars/**").addResourceLocations("classpath:/META-INF/resources/webjars/");
}
```

3 Annotations

- `@Api` 定义 Controller 方法
◦ tag 属性
- `@ApiModelProperty` 定义 entity / DTO / VO
◦ description 属性
- `@ApiModelProperty` 定义 Controller 方法属性
- `@ApiOperation` 定义 Controller 方法
◦ value 属性

4 Swagger API 文档生成

ThreadLocal

ThreadLocal 用于线程 Thread 中的 ThreadLocal

-
 -

4

- `public void set(T value)` 旣存之資料修改
 - `public T get()` 旣存之資料查詢
 - `public void remove()` 旣存之資料刪除

SpringCache

- EHCACHE
 - Caffeine
 - Redis

1

SpringTask

Spring

1

1. `mvn maven-compiler-plugin:3.8.1:compile -Djava.source=1.8 -Djava.target=1.8`
 2. `@Configuration @EnableScheduling`
 3. `@Component`
 - `@Component @Component`

- 定时任务 @Scheduled 定时任务

cron表达式

cron表达式cron表达式

时间间隔6-7分钟间隔

- 表达式由6个部分组成，分隔符为空格
- 表达式由6个部分组成，分隔符为空格

表达式部分

- * 表示所有
- ? 表示所有
- - 表示一个范围
- , 表示两个以上
- / 表示步长
- L 表示列表
- W 表示星期几
- # 表示N

WebSocket

TCP协议的升级版

- 协议
- 容器化
- 容器



容器化

- **WebSocketServer** 容器化
- **WebSocketConfiguration** 容器化 WebSocketServer
 - @Configuration 容器化
 - 容器化 Bean 容器化 ServerEndpointExporter 容器化
 - **ServerEndpointExporter** 容器化 @ServerEndpoint 容器化 **Websocket**

容器化

- @ServerEndpoint 容器化 WebSocket 容器化 URL
- @OnOpen 容器化
- @OnMessage 容器化

Apache POI

Miscrosoft Office

Java Microsoft Office

5

- maven

```
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi</artifactId>
    <version>3.16</version>
</dependency>
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>3.16</version>
</dependency>
```

- `com.spreadsheet.usermodel.excel.usermodel.IO` 類別
 - `com.spreadsheet.usermodel`
 - `com.spreadsheet.usermodel.usermodel XSSFWorkbook` 類別

1

CORS

ASP.NET MVC CORS Configuration

步骤

- addCorsMappings 方法接收 CorsRegistry registry
- registry 包含以下方法
 - .addMapping() 添加映射
 - .allowedOrigins() 允许源
 - .allowedMethods() 允许方法
 - .allowedHeaders() 允许头部

```
@Configuration
public class MvcConfiguration implements WebMvcConfigurer{
    @Override
    public void addCorsMappings(CorsRegistry registry){
        registry.addMapping("/**")
            .allowedOrigins("*")
            .allowedMethods("GET")
            .allowedHeaders("*");
    }
}
```