SSM

□三部分组成：

- `Spring framework`
- `SpringMVC`
- `Mybatis`

在企业开发web应用的时候需要基于SSM进行开发，但是我们使用的是 `SpringBoot` 进行快速开发， `SpringBoot` 是为了□化**SSM**这类**Spring**应用开发的一个框架

image-20250908000841239

SpringMVC

MVC全称为**Model View Controller**，是一种设计创建**web**应用程序的模式

- `Model` （模型）：指数据模型，用于存储数据以及处理用户请求的业务逻辑， `JavaBean` 对象、业务模型等都属于 `Model`
- `View` （视图）：用于展示模型中的数据的，一般为 `html` 文件
- `Controller` （控制器）：用于处理用户交互的部分，接收视图给出的请求，将数据交给模型处理，并把处理后的结果交给视图显示

image-20250908123944707

而 **`SpringMVC`** 时一个基于**MVC**模式的轻量级**Web**框架，是 **`Spring`** 框架的一个模块

自定义配置

方式一：继承 `WebMvcConfigurationSupport` 实现自定义配置类

- 继承它并重写方法可以完全接管**Spring MVC**的默认配置，但需要手动配置所有相关组件

方式二：实现 `WebMvcConfigurer` 接口来实现自定义配置类

- 提供了一系列默认方法，允许在不破坏Spring Boot自动配置的情况下扩展或修改**MVC**配置
- **实现 `WebMvcConfigurer` 不会影响 `WebMvcAutoConfiguration` **，Spring Boot仍然会提供默认配置，我们只需覆盖需要自定义的部分

功能

- 实现跨域配置
- 实现消息转换器
- 实现拦截器注册
- 实现接口文档生成

Maven

特性：依赖传递

生命周期：

- clean：清理
- compile：编译
- test：测试
- package：打包

- install：安装

在同一套生命周期中，当运行后面的阶段时，前面的阶段都会运行

项目对象模型：把每个项目看成一个对象

- 本地仓库：自己计算机的一个目录
- 中央仓库：全球统一的仓库
- 私服仓库：一般□公司团队搭建的私有仓库

image-20250601214430146

作用

依赖管理：方便快捷的管理项目依赖的资源

跨平台项目构建

统一项目结构

使用

- 新建一个**Maven**项目
- 在**pom.xml**文件进行依赖的配置（就是坐标）

```xml
<!-- 依赖配置示例：Apache Commons IO 工具库 -->
<dependency>
    <!-- 1. 组织标识（Group ID）：
         通常用公司/组织域名的反向形式，表示该依赖的发布者。
         这里是 Apache 基金会提供的公共库 -->
    <groupId>commons-io</groupId>

    <!-- 2. 项目标识（Artifact ID）：
         当前依赖的具体模块名称，这里是 Commons IO 库的核心模块 -->
    <artifactId>commons-io</artifactId>

    <!-- 3. 版本号（Version）：
         指定需要的库版本，2.11.0 是一个稳定版本 -->
    <version>2.11.0</version>

    <!-- 4. 依赖范围（Scope，可选）：
         默认是 compile（编译和运行都生效）。
         其他常见值：test（仅测试）、provided（由JDK或容器提供） -->
    <!-- <scope>compile</scope> -->

    <!-- 5. 排除传递性依赖（Optional，可选）：
         如果此依赖会引入不需要的间接依赖，可以用 <exclusions> 排除 -->
    <!-- <exclusions>
           <exclusion>
             <groupId>不需要的组</groupId>
             <artifactId>不需要的模块</artifactId>
           </exclusion>
```

```
        </exclusions> -->
</dependency>
<!-- 注意：XML 标签必须正确闭合，例如 </dependency>-->
```

分模块设计与开发

核心含义：把一个大的项目拆分成若干个子模块，也方便模块之间的相互引用

- 按功能模块+层进行拆分
- 将模块进行拆分后，用 `pom.xml` 将其它模块导入即可
- 最终用web模块打包，其 `xml` 文件会引入其它模块

继承

描述的是两个工程的关系，与java中的继承类似，子工程可以继承父工程的配置信息，不支持多继承

image-20250826164719979

作用：简化依赖配置，统一管理依赖

实现：`<parent> ... </parent>`

打包方式：

- `jar`：普通模块打包
- `pom`：父工程或依赖工程，该模块不写代码
- `war`：普通web程序打包，需要部署在外部的Tomcat服务器运行
- 设置打包方式：`<packing> pom </packing>`

实现

1. 创建父工程，打包方式设置为 `pom`
2. 在子工程的 `pom.xml` 文件中，配置继承关系
    - `<relativePath> </relativePath>` 用来配置父工程的相对路径
3. 在父工程中配置各个子工程共有依赖

如果不是各个模块公共的依赖，不要直接引入进父工程

版本锁定

在父工程的 `pom.xml` 中，用 `<dependencyManagement>` 来统一管理依赖版本

- 主要是为了解决非公共依赖的版本变更问题
- 非公共依赖在子工程定义，要修改版本的时候非常麻烦，但可以通过父工程来统一管理依赖版本
- 在父工程指定版本，子工程不指定

自定义属性和引用属性

- 在 `<properties> </properties>` 标签中自定义属性，可以指定版本号等信息，然后在引入依赖的时候把属性传入即可
- 引入依赖的时候用 `${}` 包裹属性

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

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

- `<dependencies>` 是直接依赖，在父工程配置了依赖，子工程直接继承
- `<dependencyManagement>` 是统一管理依赖版本，不会直接依赖，还需要在子工程中引入所需依赖

聚合

含义：将多个模块组织成一个整体，同时进行项目的构建

- 分模块之后，模块和模块之间是相互依赖的
- 需要有一个聚合工程（有且仅有一个 `pom` 文件），一般来说是父工程
- 需要通过 `<modules>` 设置当前聚合工程所包含的子模块名称

```
<modules>
    <!--需要退到和子模块同级-->
    <modeule> 模块1 </modeule>
    <modeule> 模块2 </modeule>
</modules>
```

- 指定了聚合工程，可以直接在聚合工程打包
- 直接一键打包，方便快捷

私服

私有服务器

- 是一种特殊的远程仓库，是架设在局域网内的仓库服务
- 依赖的查找顺序：本地仓库-->私服->中央仓库

项目版本

- 版本后跟的，如 `1.0-SNAPSHOT`
- `RELEASE` （发行版本）：功能稳定，当前更新停止，可以用于发行，存储在私服中的 `RELEASE` 版本
- `SNAPSHOT` （快照版本）：功能不稳定，尚处于开发，存储在私服的 `SNAPSHOT` 仓库中

配置

下面都有例子，但是不完善，`settings.xml` 是自己**maven**安装目录下的文件

- settings文件配置用户名，密码

```
<!--在settings.xml中配置-->
<servers>
    <!--releases仓库的用户名和密码-->
    <server>
        <id>maven-releases</id>
```

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

    <!--snapshots仓库的用户名和密码-->
    <server>
        <id>maven-snapshots</id>
        <username>admin</username>
        <password>admin</password>
    </server>
</servers>
```

- pom文件配置上传（发布）地址

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

- settings文件中的mirrors、profiles配置私服依赖下载的仓库地址

```
<!--在settings.xml中配置-->
<!--强制将所有对于远程仓库的□赖下载请求，重定向到公司内部的私有服务器上，如果私服没有这个包，再去远程仓库下载-->
<mirrors>
    <mirror>
        <id>maven-public</id>
        <mirrorOf>*</mirrorOf>
        <url>......</url>
    </mirror>
</mirrors>
```

```
<profile>
    <id>allow-snapshots</id>
    <activation>
        <activeByDefault>true</activeByDefault> <!-- 最关键的配置：默认就激活此Profile -->
    </activation>
    <repositories>
        <repository>                <!-- 定义一个具体的仓库 -->
            <id>maven-public</id>   <!-- 仓库的ID -->
            <url>http://192.168.150.101:8081/repository/maven-public/</url> <!-- 仓库地址，私服 -->
            <releases>              <!-- 关于发行版（RELEASE）包的配置 -->
                <enabled>true</enabled> <!-- 允许从该仓库下载发行版□赖 -->
```

```
            </releases>
            <snapshots>             <!-- 关于快照版（SNAPSHOT）包的配置 -->
                <enabled>true</enabled> <!-- 允许从该仓库下载快照版□赖 -->
            </snapshots>
        </repository>
    </repositories>
</profile>
```

SpringBoot

- **Spring**家族下的一个子框架，底层框架是**Spring Famework**

有用的依赖：**lombok**

- 可以用 `@Data` 注解为类的属性增加**get**，**set**方法
- 可以用 `@AllArgsConstructor` 为类增加全参构造
- 可以用 `@NoArgsConstructor` 为类增加无参构造

Http-请求

- **SpringBoot**依赖**Tomocat**（一个**web**服务器）运行。
- **Tomocat**会对**HTTP**协议的请求数据进行解析，并进行封装（**HttpServletRequest**），便于开发

获取请求数据

- 使用 `@RestController` 注解要处理请求的类
    - 可以在注解里面加上自定义Bean名称，如 **`@RestController("adminShopController")`**，解决**Bean**名称冲突
- 用 `@RequestMapping` 映射请求路径，是通用注解
    - 要制定请求方式的话需要填入 `value` 和 `method` 参数
    - `@PostMapping` 是post请求方式的注解，其它类似
- 使用 `@PathVarible` 获取路径变量，如 `/depts/{id}`
- 使用 `@ResuqestParam` 获取查询参数，如 `/depts?id=123`
    - 如果参数名和前端传入的一致，则可以省略
    - 使用 `MultipartFile` 接收文件数据
- `@DateTimeFormat(pattern="yyyy-MM-dd")` 可以指定传入的日期格式
- 传入的参数如果和自定义类的字段名相同，传入的参数会自动变成类

```
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、获取请求方式
```

```
        String method =  request.getMethod();
        System.out.println("请求方式: "+method);
        // 2、获取请求url地址
        String url = request.getRequestURL().toString(); //拿到的是StringBuffer对象
        System.out.println("请求url地址: "+url);

        String uri = request.getRequestURI(); // 获取的是资源访问路径
        System.out.println("请求uri地址: "+uri);
        // 3、获取请求协议
        String protocol = request.getProtocol();
        System.out.println("请求协议: "+protocol);
        // 4、获取请求参数
        String name = request.getParameter("name");
        String age = request.getParameter("age");
        System.out.println("name:"+name+", age:"+age);
        // 5、获取请求头
        String accept = request.getHeader("Accept");
        System.out.println("Accept:"+accept);

        return "OK";

    }
}
```

- `@Requestbody` 可以把前端发送的json格式数据转换成类

```
@RestController
public class UserController {

    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestBody User user) {
        // Spring 会自动将 JSON 转换为 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-响应

- 重定向状态码: 3xx, 让客户端再发起一次请求
- 响应成功: 200，
- 服务器内部错误: 500
- 请求资源不存在: 404
- 服务器会对**HTTP**协议的响应数据进行封装（`HttpServletResponse`），便于开发

- 也可以用 `ResponseEntity` 进行响应，需要用泛型，泛型针对的是其**body**部分

```
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、设置响应状态码
        response.setStatus(401);
        // 2、设置响应头
        response.setHeader("name","whs");
        // 3、设置响应体
        response.getWriter().write("<h1>whs</h1>");

    }
    // 基于泛型设置
    @RequestMapping("/response2")
    public ResponseEntity<String> response2(){
        return ResponseEntity.status(301).header("name","whs").body("<h1>whs</h1>");
    }

}
```

三层架构

- `Controller`：控制层，接受前端发送的请求，对请求进行处理并响应数据
- `Service`：业务逻辑层，处理具体业务逻辑
- `Dao`：数据访问层，负责数据访问操作

控制反转与依赖注入

注解生效的前提：**Springboot**启动文件会扫描他所在的包的所有子包，没被扫描到也意味着无法使用**IOC**和**DI**

控制反转（**IOC**）

- 用于类上，把实现了接口的类（也可以是具体类，不实现接口）的对象放入一个中间件存储为**Bean**对象
- 有利于代码之间的解耦

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

@Component
```

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

要把某个对象交给IOC容器管理，需要在对应类加上常见注解：

- `@Component`：通用注解
- `@Service`：用于业务逻辑层

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

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

- `@Repository`：用于数据访问层

```
public interface DataLoader {

}

@Repository
public class XX implements DataLoader {

}
```

- `@RestController`：用于控制层

```
// 做请求处理
@RestController
public class XX {

}
```

依赖注入（**DI**）

- 被注入的类也要添加组件注解
- 从 `Service` 声明的依赖注入的类型可以是类或者接口

常见方法有：

- 构造器注入（推荐写法，只有一个构造方法的时候可以不用加注解）

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

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

- 属性注入

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

- setting注入

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

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

常见注解:

- `@Autowired`：自动按类型注入Bean
  - 类上要加`@Component`
- `@Qualifier`：指定注入的 Bean 名称，`@Qualifier(bean_name)`

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

- `@Primary`：在想要被优先注入的类上加注解

```
// 进行□赖注入的
@Primary
```

```
@Service
public class UserServiceImpl2 implements UserService{

}
```

- `@Resource`：指定注入的 Bean 名称，`@Resource(name=bean_name)`

`@Resource` 和 `@Autowired` 的区别：

- `@Resource`：默认按照名称注入，根据字段名/方法名匹配 `Bean` 名称，若未找到则回退到按类型匹配
- `@Autowired`：根据字段/方法的参数类型在容器中查找匹配的 **Bean**

全局异常处理器

- 定义一个类，用 `@RestControllerAdvice` 进行注解
- 里面定义异常处理方法，用 `ExceptionHandler` 注解
    - 可以定义多个异常处理方法，springboot会以从下到上的继承关系去匹配异常

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

JDBC

是操作关系型数据库的一套**API**

基本步骤：

1. 注册驱动
2. 获取连接
3. 获取**SQL**语句执行对象
4. 执行SQL
5. 关闭连接

```
import java.sql.DriveManager
public class Jdbc{
    public void test(){
        // 1. 注册驱动
        Class.forName("全类名");
        // 2. 获取连接
        Connection coneection = DriveManager.getConnection(URL,username,password);
        // 3. 获取执行对象
        Statement statement = coneection.createStatement();
        // 4. 执行SQL，更新操作
        statement.executeUpdate(sql语句);
```

```
        // 5. 释放资源
        statement.close();
        connection.close();
    }
}
```

执行**DQL**语句

- `ResultSet`（结果集对象）：`ResultSet rs = statement.executeQuery()`
    - `next()`：将光标从当前位置向前移动一行，并判断当前行是否为有效行，返回值为boolean
    - `getXxx()`：获取数据，根据列名获取

```
String sql = "查询语句";

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);

}
```

预编译**SQL**

- 能够进行参数的动态传递
- 可以防止SQL注入
- 使用问号作为占位符，并没有把参数写死

Mybatis

是一个用于持久层（**DAO**）的框架，是对**JDBC**的封装

参数配置

- 需要在 `application.properties` 编写数据库配置信息
    - 包括用户名，主机名，端口，数据库驱动的全类名，使用的数据库类型等等

具体操作

- 需要创建一个接口，作为mybatis的持久层接口，并且用 `@Mapper` 注解做标识
    - 这个接口用Mapper注解后**Spring**会自动为其创建一个代理对象并进行控制反转存入中间件变成一个**Bean**对象
    - 使用的时候用依赖注入即可
- 在接口里面创建一个方法，注解对应的操作类型（增删改查）并写入对应的sql语句，当调用了这个接口的方法，mybatis会自动对数据库进行操作并把结果返回给这个方法

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

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

}
```

- 要看数据库操作日志可以去配置文件 `application.properties` 加对应的配置信息
- 要想对sql语句进行提示，需要在IDEA上填写对应的数据库配置信息和数据库名，并且把**Mysql**标记为项目语言

对查询结果进行映射

- 使用 `@Results` 和 `@Result` 进行结果映射
    - 在Mapper接口方法前添加

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

- 也可以起别名

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

- 也可以在配置文件开启驼峰命名规则
    - 会将字段名的 `create_time` 转换成 `createTime`

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

数据库连接池

- 存放着多个数据库连接对象，也就是上面的JDBC的Connection对象
- 项目初始化时会分配一系列连接对象给数据库连接池，mybatis执行数据库操作的时候会从数据库连接池获取连接对象，执行完操作后再把连接对象还给连接池
    - mybatis有设置数据库连接对象的最大空闲时间，因为有些客户端拿到连接对象之后不执行数据库操作，导致连接对象处于空闲状态，客户端一直占用，占用时间超过最大空闲时间数据库连接对象就会被归还

数据库连接池如何实现的

- 官方提供了 `Datasource` 接口，连接池需要去实现这个接口

- 要想获得连接池对象用 `getConnection()` 方法

数据库连接池的类型

- Spring默认的连接池是**Hikari**，也可以到配置文件配置自己需要的连接池

增删改查操作

删除

- 使用 `#{id}` 进行占位，动态接受向接口方法传入的参数
- 定义接口方法可以用 **`Integer`** 声明，返回值表示DML语句执行后影响的行数

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

增加

- 当要传递的参数太多，可以传递一个对象，把多个要传递的参数封装到一个对象当中

```java
@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);
    }
}
```

修改

- 接口内方法有多个参数时，要按照**SQL**语句使用参数的顺序传递参数

```java
@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);
    }
}
```

查询

- 使用 `@Param` 为接口的方法形参起名字，与SQL语句的参数对应
- 基于官方骨架创建的springboot项目中，接口编译时会保留方法形参名，传给SQL语句

```
// 声明接口
@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);
}

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

注意：可以使用 `concat` 进行字符串的拼接

```
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映射配置

- 在mybatis中，可以通过注解配置SQL语句，也可以通过**XML**配置文件配置**SQL**语句，这样的话接口方法就不用写注解了
- 默认规则：
    - XML映射文件的名称和**Mapper**接口名称一致，并且XML映射文件和Mapper接口放置在相同包下（包名要相同）
    - XML映射文件的 `namespace` 属性与Mapper接口的全限定名（也就是路径信息）一致
    - XML映射文件中的sql语句的id与Mapper接口中的方法名一致，且返回类型保持一致

创建目包的时候用 `/` 进行分隔！！！

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

- 需要什么操作类型就用什么标签包裹
- `id` 为SQL语句的唯一标识，也就是方法名
- `resultType` 是返回类型，也要用全限定名，表示的是查询的单条记录所封装的类型

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

辅助配置

可以自己配置XML映射文件的存放目录

- 在 `application.properties` 中配置XML映射文件的存放目录
    - `mybatis.mapper-locations=classpath:mapper/*.xml` ：表示在类路径下的mapper目录下去查找**XML**映射文件
        - 类路径包含： `src/main/resources` 下的所有文件和 `src/main/java` 下的所有 `.java` 文件
- 可以设置类型别名扫描的包路径，让**MyBatis**自动扫描指定包下的**Java**类，并为它们注册别名（首字母小写）
    - `type-aliases-package:com.sky.entit` 表示 `com.sky.entity` 包下的所有类会自动注册为别名

可以用 `mybatisx` 插件提高开发效率

动态**SQL**

根据用户的输入或者外部条件变化而变化的**SQL**语句

- <if>：判断条件是否成立，为true则拼接SQL
    - `test` 字段后跟条件
- <where>：根据查询条件来生成 `where` 关键字，并且自动去除条件前面多余的 `and` 或者 `or`
    - `>` ：大于
    - `<` ：xiao'yu
- <foreach>：动态遍历数据
    - `collection` ：集合名称，要遍历的变量名
    - `item` ：集合遍历出来的元素
    - `separator` ：每一次遍历使用的分隔符
    - `open` ：遍历开始前拼接的字段
    - `close` ：遍历结束后拼接的字段
- <set>：自动生成set关键字，并且自动删除掉更新的字段后多余的逗号

主键返回

插入数据之后怎么获取该数据的主键?

- 使用 `Options` 注解
    - 将 `useGeneratedKeys` 设置为 `true`
    - `keyProperty` 表示的是要把主键传回给入参的哪个属性

```
@Options(useGeneratedKeys = true,keyProperty = "id")
// 把主键返回给emp的id属性
void insert(Emp emp);
```

多表关系

假设有两个表: `user` 用户表和 `department` 部门表

一对多

- 一的为父表，多的为子表
- 给子表增加字段，然后通过外键约束保证数据一致性和完整性
    - 外键字段名为子表的字段名
    - 外键名称为自定义的名称
    - 主表一般为父表
    - 字段名为父表的字段名（主键）
    - 业务实现一般用逻辑外键，在代码中实现

```
-- 添加外键约束
-- ALTER TABLE 子表名 ADD CONSTRAINT 外键名称 FOREIGN KEY (子表字段名) REFERENCES 主表名(主表字段名);

-- constraint 外键名称 foreign key (外键字段名) references 主表(字段名)
alter table user add constraint tmp_dept_id foreign key (dept_id) references department(id);

-- 删除外键约束
-- 删除的外键的字段名称（自定义的）
alter table user drop foreign key tmp_dept_id;
```

一对一

- 可以看作是用户和身份证关系
- 在任意一张表通过外键约束建立连接，关联另一张表主键即可，但是这个外键不能重复，要加 `unique` 约束

```
-- 添加外键约束

-- 创建card表
create table card(
    id int primary key,
    card varchar(18)
```

```
);

-- 创建user表
create table user(
    id int primary key,
    -- 其它字段
    -- 外键，非空且唯一
    card_id int not null unique,
    constraint user_card_id foreign key (card_id) references card(id)
);
```

多对多

- 可以看作是学生与课程关系
- 需要定义第三张表（中间表），并且在中间表定义两个外键分别和两张表的主键做关联
    - 查询的时候使用第三张表
- 示例：

image-20250825145449821

多表查询

- 直接进行多表查询会出现笛卡尔积，也就是会出现很多无用数据，比如表1的1行数据会和表2的所有数据进行拼接再输出

```
-- 产生笛卡尔积
select * from user,department;
```

- 因此查询的时候要用一定条件消除无效的笛卡尔积

连接查询

image-20250825135942870

- 内连接：查询交集部分数据
    - 显式内连接：

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

        - 用 `inner join` 连接两张表，`on` 后面跟查询条件（指定连接条件）
        - 还有条件要加就用 `where` （指定过滤条件），对查询后的数据进行过滤
        - 可以用 `as` 给表起别名
    - 隐式内连接

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

- 外连接：查询A或者B的全部数据
    - 左外连接：查A的所有数据

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

    - 右外连接：查B的所有数据

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

    - `on` 后面跟的是交集数据的查询条件（指定连接条件）
    - 还有条件要加就用 `where` （指定过滤条件），对查询后的数据进行过滤

子查询

含义：在SQL语句中嵌套 `select` 语句，子查询的外部语句可以是任意类型的操作

- 标量子查询：返回一个值

  ```
  -- 子查询语句

  -- 获取李忠的工资
  select salary from user where name = "李忠";

  -- 外部语句

  -- 查找和李忠工资相同的员工的信息
  select * from user where salary = ?;

  -- 总语句
  select * from user where salary = (select salary from user where name = "李忠");
  ```

- 列子查询：查询返回一列

  ```
  -- 子查询语句

  -- 返回1列
  -- 获取教研部或者研发部的部门id
  select id from dept where name = "教研部" or name = "研发部";

  -- 外部语句

  -- 获取部门为教研部或者研发部的员工的信息
  select * from user where dept_id = ?;

  -- 总语句
  select * from user where dept_id in (select id from dept where name = "教研部" or name = "研发部");
  ```

- 行子查询：查询返回一行

```
-- 子查询语句

-- 获取李忠的工资和职位
select salary,job from user where name = "李忠";

-- 外部语句

-- 获取和李忠的工资和职位相同的员工的信息
select * from user where salary = ? and job = ?;

-- 总语句
select * from user where salary = (select salary from user where name = "李忠")
and job = (select job from user where name = "李忠");

select * from user where (salary,job) = (select salary,job from user where
name = "李忠");
```

- 表子查询：查询返回多行多列
    - 就是把查询回来的结果当作一个表，然后用内连接

```
-- 子查询语句

-- 获取每个部门中的最高薪资
select dept_id ,max(salary) from user group by dept_id;

-- 总语句

-- 获取每个部门中薪资最高的员工信息

-- 隐式内连接
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;

-- 显式内连接
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;
```

事务

核心：一组操作的集合，这些操作要么同时成功，要么同时失败

- 开启事务：`start transaction` 或者 `begin`
- 提交事务：`commit`
- 回滚事务：`rollback`

```
begin
-- 一系列sql语句
```

```
-- 一系列sql语句

-- 无错误
commit;

-- 出现错误，回滚
rollback;
```

Spring中如何使用?

- 在类、方法、接口上都可以加，加上 `@Transactional` 注解
    - 有 `rollbackFor` 属性，里面传异常类的字节码文件，用于控制出现何种异常才会回滚事务
        - 默认出现运行时异常 `RuntimeException` 或者错误异常 `Error` 才会回滚

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

    }
}
```

-
    - 有 `propagation` 属性来控制
        - 事务传播行为：当一个事务方法被另一个事务方法调用的时候，这个事务方法如何进行事务控制
        - `REQUIRED`：需要事务，有的话则加入，没有则创建新事务
        - `REQUIRES_NEW`：需要新事务，无论有无总创建新事务

```
@Service
public class WorkParent{

    @Autowired
    private WorkChild workchild;

    @Transactional
       public void save(){
     try{
        /**
        方法体
        */
        }
        finally{
            // 新事务
            workchild.save();
        }
    }
}

@Service
// WorkChild类
public class WorkChild{
```

```
    // 设置propagation
    @Transactional(propagation = Progation.REQUIRES_NEW)
    public void save(){
        /**
        方法体
        */
    }
}
```

- 可以在日志配置文件添加事务管理的日志输出

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

注意：

- **Spring** 事务是基于代理的，只有通过代理对象调用的方法才会被事务拦截器增强

分页查询

使用 `PageHelper` 进行分页查询

导入依赖

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

- `Mapper` 层正常查询

```
@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` 层调用 `PageHelper` 类进行总记录数的计算
  - 要进行类型强转
  - `.getTotal()` 获得总记录数
  - `.getResult()` 获得总的记录
  - 仅能对紧跟在后面的第一个 `Select` 语句进行分页查询

```
 @Override
    public PageResult<Emp> page(Integer page, Integer pageSize){
        // 1. 设置分页参数
        PageHelper.startPage(page,pageSize);

        // 2. 执行查询
```

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

        // 3.整合结果进行返回
        Page<Emp> p = (Page<Emp>) rows;
        return new PageResult<Emp>(p.getTotal(),p.getResult());

    }
```

手动结果封装

Mybatis无法帮助我们对复杂查询结果进行封装，此时需要我们自己来进行封装

在Mapper文件对应的xml配置文件中手动指定封装逻辑

- 在CRUD语句的 `resultMap` 填入我们自定义的映射逻辑
- 指定映射逻辑，用 `resultMap` 标签包裹
    - `id` 是这个映射文件的唯一标识符， `type` 为自己指定的 `pojo` 实体类，告诉他我们想要这样封装
        - `id` 是这条信息的主键
        - `result` 是查询结果
            - `column` 是数据库中的字段或者是自己查询时定义的别名
            - `property` 是这个信息在实体类中对应的名称

示例:

```
<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"/>

<!--封装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>
```

常用的方法

字段映射

可以使用 `case` 把获取到的信息进行映射

- `when` 后面填不同类别的信息
- `then` 后面填要映射成什么

```
select
            (case when job=1 then '班主任'
                  when job=2 then '讲师'
                  when job=3 then '学工主管'
                  when job=4 then '教研主管'
                  when job=5 then '咨询师'
                  else '其它' end) pos,
            count(*) num
        from emp group by job order by num
```

条件判断

可以用 `if` 来进行条件判断映射，如果为true则映射为第一个值，否则映射为第二个值

```
 select
            if(gender=1,'男性员工','女性员工') name,
            count(*) value
        from emp group by gender
```

SpringBoot项目配置

支持的配置文件格式：`properties`，`yml`，`yaml`

- `properties`：是key value格式，多级之间使用 `.` 分隔
- `yml`：用冒号来分隔（之前跑算法学过）
    - 数值前边必须有空格
    - 使用缩进表达层级关系
    - 如果配置项是以 `0` 开头，需要用 `''` 引起来，因为以**0**开头的配置项在**yml**是八进制数据
    - 在 `yml` 文件配置好信息后，可以在变量上加上 `@Value` 注解获取信息

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

  - 也可以维护一个类存储配置信息，然后在这个类上加上 `@ConfigurationProperties` ，注解后的 `prefix` 加上 `yml` 前缀名，要使用的时候注入 `Bean` 即可

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

```yaml
## 定义list或者set
hobby:
 - java
 - game
 - sport

## 定义对象/map
user:
 name: 张三
 age: 18
 password: 123456
```

SpringBoot配置优先级

命令行参数的优先级最高，其次是Java系统属性，接着是 `properties` 配置文件，然后是 `yml` 配置文件，最后是 `yaml` 配置文件

多环境配置

可以使用多环境配置，即创建多个配置文件，在不同的场景下进行切换

- 创建 `application-{profile}.yml` 或者 `application-{profile}.properties` 文件
- 在 `application.yml` 中指定要激活哪个配置文件

```yaml
spring:
  profiles:
    active: dev  # 默认激活 dev 环境
```

开发范式

前后端分离

- 前端写的代码部署在**Nginx**上
- 后端写的代码部署在**Tomcat**上

- 最后要进行前后端联调测试

Restful

REST：表述性状态转换，是一种软件架构风格

通过请求方式决定增删改查的哪一项操作

Rest风格Url	请求方式	含义
`http://localhost:8080/users/1`	GET	查找id为1的用户
`http://localhost:8080/users/1`	DELETE	删除id为1的用户
`http://localhost:8080/users`	POST	新增用户
`http://localhost:8080/users`	PUT	修改用户

Apifox/Postman

作用：接口文档管理、接口请求测试、**Mock**服务（给虚拟**API**测试）

日志框架

现在常用的日志框架为 `LogBack`

- 需要引入依赖，定义配置文件 `logback.xml`
- 记录日志的时候要定义日志记录对象
- 有两种方法定义日志
    - 使用注解 `@Slf4j` （ `Lombok` 注解）

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

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

    - 创建final静态对象
        - `getLogger` 方法里面传要记录日志的类的字节码

```java
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("用户的年龄是{}",age);
```

```
        }
    }
```

- 输出的时候可以用 `{}` 充当占位符

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

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

配置文件

- 能够去配置文件修改level从而设置可看到的日志等级
  - `off` ：关闭日志输出
  - `all` ：输出所有级别的日志
  - 日志等级：`trace` 、`debug` 、`info` 、`warn` 、`error`
- 控制台输出：`STDOUT` 、文件输出：`FILE`
- 在 `logger` 标签设置要输出的日志级别，在 `appender` 标签配置输出的日志格式
- 示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!-- 控制台输出 -->
    <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>

    <!-- 文件输出 -->
    <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>

    <!-- 根日志级别设置 -->
    <root level="INFO">
        <appender-ref ref="STDOUT" />
```

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

    <!-- 特定包或类的日志级别设置 -->
    <logger name="com.example.service" level="DEBUG" />
    <logger name="org.springframework" level="WARN" />
</configuration>
```

会话技术

含义：用户访问web浏览器，会话建立，一次会话可以包含多次请求和响应

会话跟踪：同一次会话中多次请求间共享数据

cookie

image-20250825143543584

- 一般是键值对，并且移动端无法使用，存储在客户端，不安全
- 第一次发送请求的时候服务端生成**cookie**并且在响应头加上 `Set-Cookie:cookie` 返回给客户端，客户端拿到cookie之后进行存储，后续请求的时候携带这个cookie给服务器进行认证
    - 用户可以拿到 `cookie` 的键值对
- cookie不支持跨域

跨域：当网页请求的**URL**与当前页面的协议、域名、端口任一不同，就是跨域请求

```
@RestController
public class AuthController {

    @GetMapping("/login")
    public ResponseEntity<String> login(HttpServletResponse response) {
        // 创建Cookie
        Cookie cookie = new Cookie("auth_token", "user123");
        cookie.setMaxAge(3600); // 1小时过期
        cookie.setHttpOnly(true); // 防止XSS攻击
        cookie.setPath("/"); // 作用路径

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

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

session

image-20250825143525273

- 存储在服务器中，底层是基于cookie的，因此不支持跨域
- 客户端第一次请求之后，服务端会在服务器生成一个**session**对象存储一些信息，在响应头加上 `Set-Cookie` 字段（存储session对象的唯一标识符——`sessionId`）
    - 用户能拿到 `sessionId`
- 集群环境下无法发挥作用，因为会使用**Nginx**作负载均衡，服务器间内存不共享
- 参数类型为 `HttpSession`，用 `setAttribute` 方法进行字段设置，用 `getAttribute` 方法进行字段值获取

```
@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设置成功";
    }

    @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;
    }
}
```

令牌

image-20250825143500995

一般使用**JWT**技术进行令牌生成

流程：

- 客户端请求，服务器端生成令牌并返回给客户端，客户端后续请求携带令牌，服务端进行校验
- 跨平台可用，以json格式进行传输

令牌组成部分：

- 第一部分：Header，用于记录令牌类型，签名算法
- 第二部分：Payload，携带一些自定义信息
- 第三部分：Signature，签名，防止token被篡改，保证安全

计算方法：

- Header和Payload：直接用 `Base64URL` 编码

- Signature：将**Header**、**Payload**和密钥融合在一起，通过签名算法进行计算

验证：

- 先对Header和Payload进行解码
- 服务端接受客户端提供的签名，用密钥、解码后的Header和Payload重新计算签名，然后和传入的令牌的签名进行比对

使用：

- 引入 `jjwt` 依赖
- 采用链式编程构建令牌
    - 调用工具类 `Jwts` 的 `builder` 方法创建JWT令牌的构建器
    - 用 `signWith` 方法指定签名算法和密钥
    - 用 `setClaims` 方法添加自定义信息，键值对形式
    - 用 `setExpiration` 方法指定过期时间，单位为毫秒，要包装为Date对象
    - 用 `compact` 方法构建令牌
- 采用链式编程解析令牌
    - 调用 `parser` 方法创建一个解析器的构建器
    - 用 `setSigningKey` 方法设置密钥
    - 用 `parseclaimjws` 方法接受令牌进行解析、验证
        - 验证不通过直接抛异常
    - `getBody()` 方法用于获取payload数据

```
public class JwtUtils{
     // 密钥（实际项目中应该从配置文件中读取）
    private static final SecretKey SECRET_KEY =
Keys.secretKeyFor(SignatureAlgorithm.HS256);
    // 过期时间：1小时（单位：毫秒）
    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（过滤器）

JavaWeb中的组件，能拦截所有资源请求

实现：

- 需要定义一个类实现 `Filter` 接口，并实现其所有方法，`init`、`doFilter`、`destory`
    - 需要对请求的参数进行强转，强转为 `HttpServletRequest` 和 `HttpServletResponse`
    - 放行用 `filterChain.doFilter()` 方法，要传入请求和响应参数
        - `filterChain` 是 `FilterChain` 的实例对象
- 对这个类加上 `@WebFilter` 注解
    - 注解参数用 `urlPatterns` 注册拦截路径
        - 拦截具体路径：`/login`
        - 拦截所有路径：`/*`
        - 拦截目录：`exp/*`
- 项目启动类上加上 `@ServletComponentScan` 注解，会自动扫描 `@WebFilter` 注解
- 多个过滤器可以形成一个过滤器链，优先级按照过滤器类名字的字符串自然排序

```java
import javax.servlet.*;

@Slf4j // 配置目录框架
@WebFilter("/*") // 配置拦截路径
public class LoginFilter implements Filter{
    @Override
    public void init(FilterConfig filterConfig){
        log.info("初始化中.....");
    }


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

        // 获取到请求路径
        String requestURI = request.getRequestURI();

        // 获取请求头中的token
        String token = request.getHeader("token");

        // 判断token是否为空
        if(token == null){
            log.info("用户未登录");
            // 设置状态码
            response.setStatus(401);
            return;
        }

        // 校验令牌
        try{
            JwtUtils.parseToken(token); // 这个是自己实现的类
        }
        catch(Exception e){
            log.info("令牌无效");
```

```
            // 设置状态码
            response.setStatus(401);
            return;
        }

        // 令牌合法
        filterChain.doFilter(request,response);
    }


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

工作流程：

- 启动项目时运行 `init` 方法
- 客户端请求进行拦截处理的时候运行 `doFilter` 方法
- 项目结束时使用 `destroy` 方法

Interceptor（拦截器）

Spring中的组件，只能对Spring的资源进行拦截

如果同时启用拦截器和过滤器，过滤器先执行，拦截器后执行

image-20250825140013326

使用：

- 定义一个类，需要加 `@Component` 注解，存为Bean对象
- 实现 `HandlerInterceptor` 接口，实现其所有方法
    - `preHandle`：进行拦截处理的地方，返回 `True` 放行，否则不放行
    - `postHandle`：拦截处理完后，视图渲染前执行的函数
    - `afterCompletion`：视图渲染完毕后执行（不懂）
- 实现配置类，用于注册拦截器
    - 需要注入拦截器
    - 使用 `@Configuration` 注解
    - 要实现 `WebMvcConfigurer` 接口，重写里面的 `addInterceptors` 方法，用 `registry.addInterceptor` 方法注册拦截器，用 `addPathPatterns` 方法配置拦截路径，用 `excludePathPatterns` 方法配置不需要拦截的路径
        - `/**`：代表拦截所有请求路径
        - `/*`：只能匹配一级路径
        - 可以使用 `.order()` 方法进行拦截器拦截顺序的配置，数字越小优先级越高

拦截器类

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

```
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler){
        /*
        令牌校验代码
        */
        return true; // 放行
    }

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

    /**
     * 最终完成方法（视图渲染完成后调用，常用于资源清理）
     */
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) {
       log.info("整个请求处理完成");
    }
}
```

配置类

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

    @Override
    public void addInterceptors(InterceptorRegistry registry){
        // 注册拦截器，不对/login进行拦截

registry.addInterceptor(interceptor).addPathPatterns("/**").excludePathPatterns("/logi
    }
}
```

SpringAOP

AOP：面向切面编程，也就是面向特定的方法编程

将重复性的代码统一到AOP程序当中

- 减少重复代码
- 原始代码无侵入
- 提高开发效率
- 维护方便

SpringAOP

- 导入SpringAOP依赖
- AOP程序中间执行原始业务逻辑
- 基于动态代理实现，为目标对象实现代理对象，依赖注入使用的时候是代理对象

创建**AOP**类

- 加上 `@Component` 和 `@Aspect` 注解，声明这是一个切面类
- 用 `ProceedingJoinPoint` 类的实例调用 `proceed` 方法执行原始业务逻辑，返回值为 `Object`
- 在方法上加上 `@Around` 注解，写入切入点表达式，指明当前AOP程序对哪些类中的哪些方法有效

```
// 基于Around
@Aspect
@Component
public class TestAop {
    // 匹配com.zxb.project下的所有类的方法(不管有参无参)
    @Around("execution(* com.zxb.project.*.*(..))")
    public Object test(ProceedingJoinPoint pj) throws Throwable{
        System.out.println("你好");
        // 执行原始类的业务逻辑
        Object result = pj.proceed();
        return result;
    }
}
```

核心概念

- 连接点：可以被AOP控制的方法
- 通知：指那些重复的逻辑，共性功能
- 切入点：匹配连接点的条件，通知仅会在切入点条件满足时被应用，也就是实际被AOP控制的方法
- 切面：描述通知与切入点的对应关系
- 目标对象：通知所应用的对象，也就是要实现重复逻辑的类的对象

切入点一定是连接点，连接点不一定是切入点

通知类型

通知类型	作用
@Around	环绕通知，此注解标注的方法在目标方法前后都被执行
@After	后置通知，此注解标注的方法在目标方法后被执行，无论是否有异常都会执行
@Before	前置通知，此注解标注的方法在目标方法前被执行
@AfterReturning	返回后通知，此注解标注的方法在目标方法后被执行，有异常不会执行
@AfterThrowing	异常后通知，此注解标注的方法发生异常后被执行

- 除了 `@Around` 注解的通知方法都不用手动指定目标方法执行

```
@Component
@Aspect
public class TestAop{
    // 用Around注解
```

```
     @Around("execution(* com.zxb.project0.*(..))")
    public Object testAround(ProceedingJoinPoint pj) throws Throwable{
        System.out.println("你好");
        Object result = pj.proceed();
        return result;
    }
    // 用Before注解
    // 在目标方法执行前下面这个方法就执行了
    @Before("execution(* com.zxb.project1.*(..))")
    public void testBefore(){
        System.out.println("你好");
    }

     // 用After注解
    // 在目标方法执行后下面这个方法才执行
    @After("execution(* com.zxb.project2.*(..))")
    public void testAfter(){
        System.out.println("你好");
    }
}
```

通知顺序

当项目中多个通知方法匹配到同一个目标方法的时候，通知方法的执行顺序是什么？

- 不同切面类中，默认按照切面类的类名字母排序
  - 目标方法前的通知方法：字母排名靠前先执行
  - 目标方法后的通知方法：字母排名靠前后执行
- 可以手动指定执行顺序
  - 在切面类上加 `@Order` 注解，里面传入一个数字
  - `@Before` 是数字越小越先执行
  - `@After` 是数字越小越后执行

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

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

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

最后执行的顺序是

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

切入点表达式

切入点表达式有两种书写方式

- `execution`：基于方法签名匹配
    - 主要根据方法的返回值、包名、类名、方法名、方法参数等信息匹配
    - 语法：`execution(访问修饰符? 返回值 包名.类名.?方法名(方法参数) throws 异常)`
    - 其中带 `?` 是表示可以省略的部分
        1. 访问权限修饰符
        2. 包名.类名
        3. throws 异常
    - 可以使用通配符描述切入点
        - `*`：可以匹配任意返回值、包名、类名、方法名、任意类型的一个参数，也可以通配包、类、方法名的一部分
        - `..`：多个连续的任意符号，可以通配任意层级的包，或者任意类型、任意个数的参数

```
// 基于Around
@Aspect
@Component
public class TestAop {
    // 匹配com.zxb.project下的所有一级子包的所有类的方法(不管有参无参)
     @Around("execution(* com.zxb.project.*.*(..))")
    public Object test(ProceedingJoinPoint pj) throws Throwable{
        System.out.println("你好");
        // 执行原始类的业务逻辑
        Object result = pj.proceed();
        return result;
    }
}
```

- `@annotation`：根据注解匹配，根据注解匹配带有某某注解的方法
    - 可以自定义注解，然后使用 `@annotation` 去匹配

```
// 自定义注解

// 元注解
@Target(ElementType.METHOD)
public @interface LogOperation{
}



// 真实业务逻辑类
public class Test{
    @LogOperation
    public void test(){
        System.out.println("hello");
    }
}



// 切面类
@Aspect
@Component
public class TestAop {
    // 根据LogOperatio注解匹配到Test类下的test方法
    @Before("@annotation(com.zxb.project.LogOperation)")
    public void test(){
        System.out.println("你好");
    }
}
```

示例：

- 只匹配 `com.zxb.project` 包直接下属的类： `* com.zxb.project.*(..)`
- 匹配 `com.zxb.project` 包及其所有子包下的类： `* com.zxb.project..*.*(..)`
    - `..*` 只匹配 `com.zxb.project` 包及其下面的所有子包（不包括类）

公共切入点表达式

使用 `@Pointcut` 定义公共切入点表达式

- 实现一个方法，在这个方法上加上 `@Pointcut` 注解，方法可以空实现
- 方法为 `private` 则只能在当前切面类中使用
- 通知方法中用 `""` 包裹方法名即可

```
@Component
@Aspect
public class AopOrder{

    // 只能在当前切面类使用
    @Pointcut("execution(* com.zxb.project.*(..))")
    private void all(){}
```

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

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

连接点

Spring中用 `JointPoint` 抽象了连接点，可以用它获得目标方法的相关信息

- 对于 `@Around` 只能使用 `ProceedingJoinPoint`
- 对于其它四种通知类型，只能使用 `JoinPoint`

获取目标方法的相关信息

- `getTarget()`：获取目标对象
- `getTarget().getClass().getName()`：获取目标类名，先获取目标对象，再获取字节码文件，再获取类名
- `getSignature()`：获取目标签名，可以获得目标方法名
- `getArgs()`：获取目标方法参数

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

        // 获取目标类
        String name = jp.getTarget().getClass().getName();

        // 获取目标方法
        String methodName = jp.getSignature().getName();

        // 获取目标方法参数
        Object[] args = jp.getArgs();
    }
}
```

Bean

含义：IOC容器管理的对象叫做 `Bean`

类被注入到IOC容器之后，**Bean**对象名称是类名首字母小写

获取**IOC**容器

- 使用 `ApplicationContext`，SpringFramework里面的（直接注入即可，得益于**Springboot**的自动配置功能）

- 方法
    - `getBean()`：获取类的Bean对象，方法里传入Bean对象名称

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

- 懒加载：
    - 给要注入到**IOC**容器的类上加上 `@Lazy` 可以延迟加载
    - 延迟到第一次使用这个Bean对象的时候加载，否则就是项目启动的时候把类注入到IOC容器

Bean对象的常见作用域

- `singleton`（默认）：单例，容器内同名称的bean只有一个实例
- `prototype`：多例，每次使用该bean时都会创建新实例

可以用 `@Scope` 在要注入IOC容器的类上声明作用域

```
@Scope("prototype")
@Component
public class TestScope{
    public void test(){
        System.out.println("方法正在执行");
    }
}

// 使用
public class Test{
    // 多例注入
    @Autowired
    private TestScope testScope;

}
```

何时使用单例和多例bean?

- 无状态的bean（没存数据）用 `singleton`

    - 没有存数据不会有线程资源竞争问题，可以用单例
- 有状态的bean（存储了数据） `prototype`
    - 存储数据需要开多个bean，否则会有线程安全问题

可以用 **`@Bean`** 注解注入第三方类

- 需要创建一个配置类，用 `@Configuration` 注解
- 在这个配置类中，需要定义方法，在这个方法去做注入
    - 方法返回值为第三方类名
    - 方法名为第三方类的Bean名称
    - 方法体里要new一个第三方类的实例然后返回
    - 方法上要加 `@Bean` 注解

```
public class ThirdWay{
    /*
    第三方类
    */
}


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

// 正常依赖注入
@Component
public class MainProcess{
    @Autowired
    private ThirdWay thirdWay;
}
```

SpringBoot原理

起步依赖

`SpringBoot` 提供了许多起步依赖，引入起步依赖，可以利用**maven**的依赖传递特性，引入所需的所有依赖

自动配置

含义：当**spring**项目启动后，一些配置类、**bean**对象就自动存入了**IOC**容器中

- 比如前面的 `ApplicationContext` ，是第三方的类，但是我们可以直接注入，得益于自动配置
- **SpringBoot**的自动配置功能是跨模块的！！！

方式一：在Springboot的启动类上加上 `@ComponentScan` 注解，在里面的 `basePackages` 字段填写要扫描的包

- 因为引入的第三方工具不和启动类所在包在同一个模块下，默认启动的话第三方的类不会被扫描注入到IOC容器

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

}
```

方式二：启动类上加 `@Import` 注解，里面填写普通类！！的字节码文件

方式三：上面说过的第三方配置

方式四：定义一个类实现 `ImportSelector` 接口，然后把这个类的字节码文件放入 `@Import` 注解中，可以实现批量导入

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

@Import(ImportConfig.class)
@SpringBootApplication
public class Begin{
    @Autowired
    private Hello hello; // 可以正常使用
}
```

方式五：使用第三方工具开发者提供的注解 `@EnableXXX` （他们已经在这个注解配置好要导入什么依赖），在启动类上加上这个注解

源码跟踪

Springboot的启动类注解 `@SpringBootApplication` □三个部分组成

- `@SpringBootConfiguration`：该注解与 `@Configuration` 作用相同，声明当前也是一个配置类
- `@ComponentScan`：组件扫描，默认扫描当前类所在包及其子包
- `@EnableAutoConfiguration`：实现自动化配置的核心注解
    - 封装了一个 `@Import` 注解，导入了 `ImportSelector` 的实现类，这个实现类实现了 `SelectImports` 方法，返回值是自动配置类的全类名
        - 自动配置类存储在一个文件中，这些类是自动配置类，自动配置类里面声明了要注入的类
            - 要实现跨模块自动配置的话，实现自定义配置类之后要把配置类全类名写在这里面
            - 下面的代码也可以实现自动配置，不过是模块内自动配置，Springboot会扫描带有 `@Configuration` 注解的配置类，然后把配置类里面的Bean注入到IOC容器
        - 拿到自动配置类的全类名之后，去找到里面用 `@Bean` 注解的类，并把他们注入到**IOC**容器，如下所示

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

```
}

// 正常依赖注入
@Component
public class MainProcess{
    @Autowired
    private ThirdWay thirdWay;
}
```

剖析下 `@Conditional` 注解，条件注解

- 作用：按照一定条件进行判断，满足条件才会注册对应的bean对象到SpringIOC容器当中
- 派生出子注解
    - `@ConditionalOnClass`：判断环境中是否有对应的字节码文件，有才注册bean到IOC
        - 里面有个name字段，填写要判断的字节码文件的全类名
    - `@ConditionalOnMissingBean`：判断环境中有没有对应的**bean**，没有才注册bean到IOC
    - `ConditionalOnProperty`：判断配置文件中有没有对应的属性和值，有才注册
        - 里面有 `name` 和 `havingValue` 字段，填写对应的属性和值

自定义**starter**

- 创建starter模块作为父工程
- 创建 `AutoConfiguraion` 模块，里面要实现 `XXXAutoConfiguration` 类
- 把这个类的全类名写入到一个文件中 `META-INF/spring/.....`
- 其它模块直接导入starter即可实现自动配置

实体类

- `POJO`：普通java对象
    - `Enity`：实体，通常和数据库中的表对应
    - `DTO`：数据传输对象，用于程序中各层之间传递数据
    - `VO`：视图对象，为前端展示数据提供的对象

可以使用 `Bean.utils.copyProperties(DTO,Enity)` 进行对象属性拷贝

可以在类上 `@Builder` 注解，进行类属性赋值的时候获取构建器对象进行链式赋值

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

Swagger

使用 `Knife4j` 依赖进行接口测试

1. 导入 `knife4j` 的maven坐标
2. 在配置类加入 `knife4j` 相关配置

3. 设置静态资源映射，否则接口文档页面无法访问

创建**Bean**对象

- docket对象会被进行扫描
- 然后被执行自动注入
- 最后生成API文档

```
@Bean
    public Docket docket() {
        ApiInfo apiInfo = new ApiInfoBuilder()
                .title("苍穹外卖项目接口文档")
                .version("2.0")
                .description("苍穹外卖项目接口文档")
                .build();
        Docket docket = new Docket(DocumentationType.SWAGGER_2)
                .apiInfo(apiInfo)
                .select()
                //指定要扫描的包
                .apis(RequestHandlerSelectors.basePackage("com.sky.controller"))
                .paths(PathSelectors.any())
                .build();
        return docket;
    }
```

静态资源映射

```
protected void addResourceHandlers(ResourceHandlerRegistry registry) {

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

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

常用注解

- `@Api`：用在类上，例如 `Controller`，表示对类的说明
  - 用 `tag` 属性
- `@ApiModel`：用在类上，例如 `entity`、`DTO`、`VO`
  - 用 `description` 属性
- `@ApiModelProperty`：用在属性上，描述属性信息
- `@ApiOperation`：用在方法上，例如 `Controller` 的方法，说明方法的用途、作用
  - 用 `value` 属性

使用这些注解可以在生成API文档的时候更加直观，可读性高

ThreadLocal

`ThreadLocal` 并不是一个Thread，而是Thread的局部变量

- 能为每个线程提供单独一份存储空间，具有线程隔离的效果
- 只有在线程内才能获得到对应的值，线程外则不能访问

常用方法

- `public void set(T value)`：设置当前线程的线程局部变量的值
- `public T get()`：返回当前线程所对应的线程局部变量的值
- `public void remove()`：移除当前线程的线程局部变量

SpringCache

是一个框架，实现了基于注解的缓存功能，简单地添加一个注解，就能实现缓存

底层可以切换不同的缓存实现：

- `EHCache`
- `Caffeine`
- `Redis`

常用注解：

- `@EnableCaching`：开启缓存注解功能，通常加在启动类上
- `@Cacheable`：方法执行前先查询缓存中是否有数据，有数据则直接返回缓存数据，没有的话调用方法并将方法返回值放到缓存中
  - 通过`cacheNames`声明缓存名字
  - 通过`key`声明key，需要动态填入
    - 如果和方法的形参有关，需要和形参相同，并在前面加`#`
    - 也可以用`#result`获取到方法的返回值
  - 最后生成键的时候是`cacheNames::key`
- `@CachePut`：将方法的返回值放到缓存中
  - 通过`cacheNames`声明缓存名字
  - 通过`key`声明key，需要动态填入
    - 如果和方法的形参有关，需要和形参相同，并在前面加`#`
    - 也可以用`#result`获取到方法的返回值
  - 最后生成键的时候是`cacheNames::key`
- `@CacheEvict`：将一条或多条数据从缓存中删除
  - 通过`cacheNames`声明缓存名字
  - 通过`key`声明key，需要动态填入
    - 如果和方法的形参有关，需要和形参相同，并在前面加`#`
    - 也可以用`#result`获取到方法的返回值
  - 最后生成键的时候是`cacheNames::key`
  - 如果要实现批量删除，不使用**`key`**，使用**`allEntries=true`**

SpringTask

是Spring框架提供的任务调度工具，可以按照约定的时间自动执行某个代码逻辑

使用步骤：

1. 导入maven坐标
2. 启动类添加注解`@EnableScheduling`开启任务调度
3. 自定义定时任务类
   - 需要加上`@Component`注解，交给IOC容器管理

    - 自定义方法（无返回值），加上 `@Scheduled` 注解指定任务什么时候触发

cron表达式

就是一个字符串，通过cron表达式可以定义任务触发的时间

构成规则：分为**6**或**7**个域，□空格分隔开，每个域代表一个含义

- 每个域的含义分别为：秒、分钟、小时、日、月、周、年（年可以不选）
- 格式如下：秒 分钟 小时 日 月 周 年

特殊字符说明：

- `*`：每秒执行
- `?`：忽略字段，仅用于日或星期，避免冲突
- `-`：范围
- `,`：表示多个值
- `/`：步长（间隔触发）
- `L`：最后一天（仅日或星期）
- `W`：最近工作日（仅日）
- `#`：第N个星期几（仅星期）

WebSocket

基于**TCP**的一种新的网络协议，实现了浏览器和服务器的全双工通信

- 只需进行一次握手
- 二者可以创建持久性连接，并进行双向数据传输
- 长连接

image-20250913103324320

使用步骤：

1. 创建 **`WebSocketServer`** 类，编写连接调用的方法
2. 创建 **`WebSocketConfiguration`** 类，自定义配置类，注册 `WebSocketServer` 类
    - 需要用 `@Configuration` 注解，标记为配置类
    - 使用注册第三方 `Bean` 的方式编写 `ServerEndpointExporter` 类的构造方法
        - **`ServerEndpointExporter`** 会自动扫描并注册所有被 **`@ServerEndpoint`** 注解的类，将其暴露为**Websocket**端点

常用注解

- `@ServerEndpoint`：定义 `WebSocket` 服务端入口，指定客户端连接的URL路径
- `@OnOpen`：处理连接建立事件
- `@OnMessage`：处理收到的信息

- `@OnClose`：处理连接关闭事件
- `@PathParm`：接收路径参数，WebSocket专属

Apache POI

是一个处理Miscrosoft Office各种文件格式的开源项目

可以使用Java程序对Miscrosoft Office各种文件进行读写操作

使用步骤：

- 导入maven坐标

```
<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>
```

- 用下述接口创建好excel文件后需要调用IO流把文件从内存写入磁盘
- 最后需要进行资源关闭
- 读取excel文件的时候需要使用 `XSSFWorkBook` 类的有参构造方法，传入输入流

常用接口

- `XSSFWorkBook` 类：在内存中创建一个Excel文件
- `XSSFSheet` 类：sheet的类
- `.createSheet(sheetname)`：在Excel文件中创建sheet页
- `.createRow(rownum)`：在sheet页中创建行，`rownum` 从0开始，sheet的方法
- `.createCell(index)`：在行上面创建单元格，`index` 从0开始，row的方法
- `.setCellValue()`：写入文件内容
- `.write(IO流)`：将文件写入磁盘
- `.close()`：关闭资源
- `.getSheetAt(index)`：获取第 `index + 1` 页的sheet
- `.getLastRowNum()`：获取sheet页中有文字的最后一行的行号
- `getCell()`：获取单元格对象
- `getStringCellValue()`：获取单元格文本内容

跨域配置

需要自定义MVC配置类，具体如何配置在上文有

步骤：

- 重写 `addCorsMappings` 方法，参数里面传 `CorsRegistry registry`
- 使用 `registry` 进行跨域配置
    - `.addMapping()`：为哪些路径进行跨域配置
    - `.allowedOrigins()`：允许访问的域名
    - `.allowedMethods()`：允许的请求方式
    - `.allowedHeaders()`：允许的请求头

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