简答题

1.Spring IOC 的原理和作用：

解释什么是控制反转（IOC）和依赖注入（DI）。

IOC 容器的作用和核心功能。

区别和比较构造注入和设值注入。

2.Spring AOP 的概念和应用：

AOP 是什么，它解决了什么问题？

切面、连接点、通知、切点在AOP中的角色和作用。

举例说明AOP在实际项目中的应用场景。

3.MyBatis 的核心组件和工作原理：

映射文件（Mapper）的作用和结构。

MyBatis 的动态 SQL 是如何实现的？举例说明动态 SQL 的应用场景。

SqlSessionFactory 和 SqlSession 的区别和用途。

4.Spring中的Bean的作用域：

详细解释Singleton、Prototype、Request、Session和Global Session五种Bean的作用域，并举例说明它们在不同场景下的应用。

5.Spring中的注解方式实现IOC和AOP：

说明如何使用注解（如@Component, @Autowired, @Aspect等）来实现Spring中的IOC和AOP，以及它们与XML配置方式的区别和优劣势。

6.MyBatis中的映射文件：

介绍MyBatis映射文件的主要内容和结构，以及如何定义映射关系、参数映射、结果映射等。

程序设计题

1. Spring IOC 的实现：

使用 XML 配置文件实现一个简单的 Spring IOC 容器，包括至少两个 Bean 的定义和它们的依赖注入（设值注入或构造注入）。

1. Spring AOP 的实现：

使用注解方式实现一个简单的日志切面，记录指定方法的执行时间，并将其应用到一个业务类中。

1. MyBatis 的基本使用：

编写一个基本的 MyBatis 映射文件，包括一个简单的查询语句和一个插入语句，展示如何通过 MyBatis 执行 SQL 操作。

1. Spring MVC 的数据绑定和请求响应：

实现一个简单的 Spring MVC 控制器，处理 GET 和 POST 请求，并演示如何进行数据绑定（包括表单提交）和请求参数的处理。

1. Spring MVC 的文件上传和下载：

编写一个 Spring MVC 控制器，实现文件上传和下载功能，包括处理文件上传表单和提供文件下载的接口。

1. Spring MVC 的异常处理和乱码过滤：

实现一个全局异常处理器，捕获所有 Controller 中抛出的异常并进行统一处理；同时，演示如何配置字符编码过滤器解决乱码问题。

1. SSM 整合流程：

详细描述 Spring、Spring MVC、MyBatis 三者如何整合，包括配置文件的相互关系和依赖注入的实现过程

**控制反转（IoC）：**

控制反转是一种设计原则，它反转了传统的程序设计流程。传统情况下，程序员编写主要代码并直接调用依赖的组件（类、对象等）。而控制反转则将组件的创建和管理交给外部容器或框架，由它来控制整个程序的流程和对象的生命周期。这种方式将程序员编写的代码解耦（解开）了与具体实现的绑定，使得系统更加灵活、可扩展和易于维护。

**依赖注入（DI）：**

依赖注入是控制反转的一种实现方式。它通过将依赖对象的创建或获取，以及它们之间的关系的管理，从组件的代码中转移到外部进行。依赖注入的核心思想是将依赖通过构造函数、方法参数或者在对象实例化后通过设值的方式注入到对象中，而不是由对象自己去创建或查找依赖的方式。这样做的好处是提高了代码的可测试性、可维护性，并支持更好的模块化和复用。

**IoC 容器的作用和核心功能**：

对象的创建和管理：IoC 容器负责根据配置信息实例化对象，并管理它们的生命周期。

依赖注入：IoC 容器负责将依赖注入到对象中，可以通过构造函数注入（构造注入）或者在对象实例化后通过设值的方式注入（设值注入）。

配置管理：IoC 容器通常需要配置来告知它如何实例化对象、如何管理依赖关系，这些配置可以是 XML、注解或者其他形式的元数据。

生命周期管理：IoC 容器负责管理对象的生命周期，包括对象的创建、初始化、依赖的注入、依赖的解析等。

**构造注入：**

特点：通过对象的构造函数来注入依赖。

优点：对象在实例化时就能够获得所有必要的依赖，一旦实例化完成，它们的依赖关系就变得明确和不可变。

缺点：对于依赖关系的可选性处理不够灵活，不适合有大量可选依赖的情况。

设值注入：

特点：通过设值方法（setter方法）来注入依赖。

优点：可以更灵活地处理依赖的可选性，因为可以在对象实例化后随时调用设值方法来设置依赖。

缺点：对象在实例化后可能处于不完全初始化的状态，对依赖的访问需要确保线程安全性。

**AOP核心概念**：

切面（Aspect）：切面是横切关注点的模块化单元，它定义了在哪里、何时以及如何应用横切逻辑。

连接点（Join Point）：连接点是在应用执行过程中能够插入切面的点，比如方法的调用、异常的抛出等。

通知（Advice）：通知定义了在连接点上执行的操作，包括前置通知（before）、后置通知（after）、环绕通知（around）等。

切点（Pointcut）：切点定义了一组连接点，通知将会在这些连接点上被执行。

**解决了以下问题**：

分离关注点：AOP允许将横切关注点（如日志、安全、事务等）从主要业务逻辑中分离出来，使得关注点之间的耦合性降低，增加了代码的可维护性和可重用性。

避免重复性代码：通过AOP，可以将多个模块共同需要的横切逻辑封装到一个切面中，避免了在各个模块中重复编写相同的代码。

集中化管理：AOP可以将横切逻辑集中管理，使得系统的整体架构更加清晰，易于理解和维护。

**场景**

日志记录：记录方法的调用和返回值，以及异常信息。

事务管理：在方法执行前开启事务，在方法执行后提交或回滚事务。

权限控制：检查用户是否具有执行特定操作的权限。

性能监控：统计方法的执行时间、频率等性能指标。

异常处理：捕获方法执行过程中抛出的异常并进行处理。

**映射文件（Mapper）的作用和结构**

作用：

映射文件（Mapper）用于定义 SQL 语句、存储过程调用和 SQL 查询结果与 Java 对象之间的关系。

提供了面向对象的查询方式，使得开发者不需要直接编写繁琐的 JDBC 代码。

结构：

一个典型的 MyBatis 映射文件包括以下几个部分：

mapper：根元素，用于表示一个 mapper 文件。

namespace：指定该 mapper 文件对应的命名空间，一般使用对应的接口名。

select/insert/update/delete：定义具体的 SQL 操作。

resultMap：定义结果映射，将查询结果映射到 Java 对象。

parameterMap：定义参数映射（较少使用）。

示例：

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.example.mapper.UserMapper">

<select id="findUserById" parameterType="int" resultType="com.example.model.User">

SELECT \* FROM users WHERE id = #{id}

</select>

<insert id="insertUser" parameterType="com.example.model.User">

INSERT INTO users (name, email) VALUES (#{name}, #{email})

</insert>

</mapper>

**MyBatis 的动态 SQL 是如何实现的？**

实现方式：

MyBatis 提供了一套基于 XML 标签的动态 SQL 语法，允许在映射文件中编写条件判断和循环等控制结构，以生成动态 SQL 语句。主要通过 <if>, <choose>, <when>, <otherwise>, <trim>, <where>, <set> 等标签实现。

应用场景：

动态 SQL 常用于根据不同条件生成不同的查询语句、更新语句等。比如，根据用户输入的查询条件动态构建 SQL 查询。

示例：

<select id="findUsers" parameterType="map" resultType="com.example.model.User">

SELECT \* FROM users

<where>

<if test="name != null">

AND name = #{name}

</if>

<if test="email != null">

AND email = #{email}

</if>

</where>

</select>

**SqlSessionFactory 和 SqlSession 的区别和用途**

**SqlSessionFactory：**

作用：它是 MyBatis 的核心对象之一，用于创建 SqlSession 对象。

用途：SqlSessionFactory 是线程安全的，通常在应用程序启动时被创建并在整个应用程序生命周期内保持存在。它负责读取 MyBatis 配置文件和映射文件，建立连接池等资源，并为应用程序提供 SqlSession 对象。

示例：

String resource = "mybatis-config.xml";

InputStream inputStream = Resources.getResourceAsStream(resource);

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

**SqlSession：**

作用：它是 MyBatis 执行 SQL 的主要接口，代表了与数据库的一次会话。

用途：SqlSession 不推荐在多个线程间共享。每次数据库操作都需要获取一个新的 SqlSession 实例，操作完成后应及时关闭以释放资源。通过 SqlSession 可以执行映射文件中定义的 SQL 语句，如 select, insert, update, delete。

示例：

try (SqlSession session = sqlSessionFactory.openSession()) {

UserMapper mapper = session.getMapper(UserMapper.class);

User user = mapper.findUserById(1);

}

**Spring中的Bean的作用域：**

Singleton（单例）

Singleton是Spring默认的作用域。在单例模式下，Spring容器只会创建该Bean的一个实例，并且所有对该Bean的请求（注入或者通过getBean方法获取）都会返回同一个实例。

应用场景：

当Bean不需要基于状态来区分时，可以使用Singleton。例如，数据访问层的DAO对象、Service对象等通常都适合使用Singleton，因为它们的状态不会随每次请求的变化而变化。

示例：

@Service

public class UserService {

// Singleton scope by default

// Implementation details

}

2. Prototype（原型）

Prototype作用域告诉Spring容器每次请求时都要创建一个新的Bean实例。每次调用getBean()方法时，Spring都会返回一个新的实例。

应用场景：

当Bean的状态需要在每次请求或每次使用时重新初始化时，应该使用Prototype。例如，对于用户请求的每个新实例或会话，可能需要一个新的对象实例。

示例：

@Component

@Scope("prototype")

public class User {

// Prototype scope

// Implementation details

}

3. Request（请求）

Request作用域是在Web应用中特定于HTTP请求的生命周期。每个HTTP请求都会创建一个新的Bean实例，该Bean仅在当前HTTP请求内有效。

应用场景：

当需要在同一个HTTP请求内共享一个Bean实例时，可以使用Request作用域。例如，在处理用户请求时，可能需要在多个组件之间共享数据。

示例：

@Component

@Scope(value = WebApplicationContext.SCOPE\_REQUEST, proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class ShoppingCart {

// Request scope

// Implementation details

}

4. Session（会话）

Session作用域是在Web应用中特定于用户会话的生命周期。Spring容器会为每个用户会话创建一个唯一的Bean实例，该实例在整个用户会话期间有效。

应用场景：

当需要在整个用户会话期间保持状态时，可以使用Session作用域。例如，保存用户的登录信息、购物车内容等。

示例：

@Component

@Scope(value = WebApplicationContext.SCOPE\_SESSION, proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class UserSession {

// Session scope

// Implementation details

}

5. Global Session（全局会话）

Global Session作用域是在Portlet应用中特定于Portlet全局会话的生命周期。类似于Session作用域，但是仅在Portlet环境中有效。

应用场景：

在使用Portlet作为Web应用框架时，可以使用Global Session作用域来管理在全局Portlet会话中共享的Bean实例。

示例：

@Component

@Scope(value = WebApplicationContext.SCOPE\_GLOBAL\_SESSION, proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class PortletSessionBean {

// Global Session scope

// Implementation details

}

**6.使用注解实现Spring中的IOC和AOP**

**IOC**

关键注解：

@Component：标记一个类为Spring的组件类，Spring会自动检测并注册该类为Bean。

@Autowired：标记在字段、构造函数或方法上，表示需要自动注入依赖。

@Configuration：标记一个类为配置类，类似于XML配置文件，用于定义Bean。

@Bean：标记在方法上，用于显式地定义一个Bean。

示例：

定义一个组件类：

import org.springframework.stereotype.Component;

@Component

public class UserService {

public void performTask() {

System.out.println("Performing a task");

}

}

自动注入依赖：

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Component

public class UserController {

@Autowired

private UserService userService;

public void execute() {

userService.performTask();

}

}

配置类：

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

@Bean

public UserService userService() {

return new UserService();

}

}

**AOP**

关键注解：

@Aspect：标记一个类为切面类。

@Before、@After、@Around、@AfterReturning、@AfterThrowing：定义切点方法，在目标方法执行前后或替代目标方法执行。

@Pointcut：定义切点表达式，用于描述哪些方法需要被切面增强。

示例：

定义一个切面类：

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.springframework.stereotype.Component;

@Aspect

@Component

public class LoggingAspect {

@Before("execution(\* com.example.UserService.\*(..))")

public void logBefore() {

System.out.println("Logging before method execution");

}

}

配置类启用AOP：

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration

@EnableAspectJAutoProxy

public class AppConfig {

// Other bean definitions

}

**注解与XML配置方式的对比**

注解配置：

优点：更加简洁：代码即配置，减少了XML文件的冗余。

更具可读性：开发者可以更直观地看到依赖关系和配置。

强类型检查：编译时就能发现配置错误。

缺点：侵入式：需要修改源代码来添加注解。

不适合某些复杂配置：在处理复杂条件逻辑时，注解可能不如XML灵活。

XML配置：

优点：非侵入式：无需修改源代码，可以在外部进行配置。

灵活性高：适用于复杂的配置场景，例如条件配置、环境切换等。

缺点：冗长：XML文件可能非常庞大且难以维护。

可读性差：配置分散在多个文件中，不如注解直观。

**1.映射文件的基本结构**

映射文件通常以.xml为后缀，位于classpath下的某个目录中，例如src/main/resources/mappers/。

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.example.MyMapper">

<!-- SQL语句定义 -->

<select id="selectById" parameterType="int" resultType="com.example.User">

SELECT \* FROM users WHERE id = #{id}

</select>

<!-- 参数映射 -->

<insert id="insertUser" parameterType="com.example.User">

INSERT INTO users (username, password) VALUES (#{username}, #{password})

</insert>

<!-- 结果映射 -->

<resultMap id="userResultMap" type="com.example.User">

<id property="id" column="id"/>

<result property="username" column="username"/>

<result property="password" column="password"/>

</resultMap>

</mapper>

2. 定义映射关系

namespace属性：指定Mapper接口或者Mapper XML文件的命名空间，用于唯一标识该映射文件或Mapper接口。

<mapper namespace="com.example.MyMapper">

<!-- SQL语句、结果映射等 -->

</mapper>

id属性：定义每个SQL语句或映射的唯一标识符。

<select id="selectById" ...>

<!-- SQL语句 -->

</select>

3. 参数映射

parameterType属性：指定SQL语句中的参数类型。可以是Java对象、基本类型或Map等。

<select id="selectById" parameterType="int" ...>

SELECT \* FROM users WHERE id = #{id}

</select>

在SQL语句中使用#{}语法来引用参数，其中#{id}对应于parameterType中指定的参数。

4. 结果映射

resultType属性：指定SQL查询的结果类型，通常是一个Java Bean或者基本类型。

<select id="selectById" resultType="com.example.User">

SELECT \* FROM users WHERE id = #{id}

</select>

或者使用resultMap元素进行更详细的结果映射定义：

<resultMap id="userResultMap" type="com.example.User">

<id property="id" column="id"/>

<result property="username" column="username"/>

<result property="password" column="password"/>

</resultMap>

<select id="selectById" resultMap="userResultMap">

SELECT \* FROM users WHERE id = #{id}

</select>

在结果映射中，<id>和<result>元素用来映射数据库列（column）到Java对象的属性（property）。

5. 动态SQL

MyBatis还支持动态SQL，通过<if>、<choose>、<when>、<otherwise>等元素来实现条件判断和动态SQL拼接，使得SQL语句的构建更加灵活和复用性更高。

6. 其他元素

除了上述基本元素外，还有一些其他常用的元素如<update>、<delete>、<include>等，用于定义更新操作、删除操作以及包含其他映射片段。

**Spring IOC 的实现**

定义 Bean 类

首先，创建两个简单的 Java Bean 类：UserService 和 UserRepository。

UserRepository.java

package com.example.repository;

public class UserRepository {

public void save() {

System.out.println("Saving user...");

}

}

UserService.java

package com.example.service;

import com.example.repository.UserRepository;

public class UserService {

private UserRepository userRepository;

// 设值注入

public void setUserRepository(UserRepository userRepository) {

this.userRepository = userRepository;

}

public void registerUser() {

userRepository.save();

System.out.println("User registered!");

}

}

2. 配置 XML 文件

接下来，在 resources 目录下创建 applicationContext.xml 文件，定义 Bean 及其依赖注入。

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- 定义 UserRepository Bean -->

<bean id="userRepository" class="com.example.repository.UserRepository"/>

<!-- 定义 UserService Bean 并注入 UserRepository -->

<bean id="userService" class="com.example.service.UserService">

<!-- 设值注入 -->

<property name="userRepository" ref="userRepository"/>

</bean>

</beans>

3. 使用 Spring IOC 容器

最后，编写一个简单的 Java 程序，加载 Spring IOC 容器并获取 Bean 实例。

package com.example;

import com.example.service.UserService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

public static void main(String[] args) {

// 加载 Spring IOC 容器

Application Contextcontext = new ClassPathXmlApplicationContext("applicationContext.xml");

// 获取 UserService Bean 实例

UserService userService = (UserService) context.getBean("userService");

// 调用 UserService 方法

userService.registerUser();

}

}