# spring用途

用途：

1. 能够整合其它框架--控制反转的思想
2. 声明式的事务管理--aop面向切面编程思想

# 准备工作

1. 准备好spring需要的jar包

方法1： 使用maven仓库

方法2： 使用我给大家准备好的jar

2) 编写一个spring的配置文件

# 控制反转

IOC Inversion of Controll

servlet

new MyServlet() --> tomcat服务器 (web容器)

doGet()

doPost()

spring 容器

dao, service, controller

所谓的控制反转，将某一类对象的创建权，以及某些方法的调用权，交由容器负责。

spring中的控制反转：

1）

<!-- 受到spring容器管理的类的定义

id="唯一标识"

class="受管的类型"

-->

<bean id="userDao" class="com.yihang.dao.UserDaoJdbc"></bean>

2）

// 1)根据配置文件创建spring容器

ClassPathXmlApplicationContext spring

= new ClassPathXmlApplicationContext("config/spring.xml");

// 2)获取容器中的对象

UserDaoJdbc dao1 = spring.getBean(UserDaoJdbc.class); // 根据类型获取

UserDaoJdbc dao2 = (UserDaoJdbc) spring.getBean("userDao"); // 根据id(名字)获取

UserDaoJdbc dao3 = spring.getBean("userDao", UserDaoJdbc.class);

// 3)使用容器中的对象

dao1.save(user);

# 依赖注入

依赖注入 dependency inject

所谓的依赖注入，通过spring容器管理类之间的依赖关系

控制反转和依赖注入的意义： 配合面向接口编程，可以实现层与层之间的弱耦合

dao层代码发生了变化，也导致service层代码受到影响，这时dao和service是强耦合

提倡弱耦合，首先要面向接口编程，其次在配置文件中建立service和dao的依赖关系，而不是写死在java代码里（靠spring 控制反转和依赖注入来实现）

# 控制反转能控制什么

## 如何控制bean的创建

1. 管理 java.sql.Date （没有无参构造的类）
2. 管理java.util.Calendar (仅有工厂方法的类)
3. 工厂类 FactoryBean -- spring最强大的控制对象创建的手段

## 控制bean的实例个数

<bean id="" class="" scope="">

scope=singleton|prototype， 如果不写scope默认为singleton

singleton 单例

prototype 多例

单例注入多例时，多例的特性被破坏，需要使用<aop:scoped-proxy/>来解决

<bean class="service" scope="singleton">

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

</bean>

<bean class="dao" scope="prototype">

<aop:scoped-proxy/>

</bean>

## bean初始化销毁

## 懒惰初始化

默认spring容器一创建，容器内所有单例对象都被创建，并调用init

懒惰初始化是指，对单例而言，使用时才创建，才调用init

<bean id="" class="" lazy-init="true|false">

## bean 后处理器

可以用来对容器中刚创建出来的对象做进一步的改造和增强

BeanPostProcessor

# 依赖注入有哪些方式

1. set方法注入

<property name="属性名" ref="" value=""/>

1. 构造方法注入

<constructor-arg index="构造参数下标" ref="" value=""/>

1. 注解注入

@Autowired 注解，加在要注入的私有属性上

也可以加在set方法上，或构造方法上

利用类型来匹配要注入的对象

# 注解方式

@Autowired 依赖注入

@Component @Repository @Service @Controller 控制反转

1. 给注解控制的bean添加id @Component("bean id")
2. FactoryBean类可以通过注解管理
3. 控制bean的初始化和销毁
   1. @PostConstruct 初始化
   2. @PreDestroy 销毁
4. 控制懒惰初始化 @Lazy 只对单例类有效
5. 控制bean的Scope @Scope(scopeName="singleton|prototype")
6. 注解方式控制后处理器

# 值注入

也是依赖注入，只是不是建立与其它bean的依赖关系，而是仅给属性赋值

占位符值注入：值写在spring的容器配置之外，\*.properties 或 系统的环境变量 当中

set注入 <property name="属性名" value="" ref=""/>

构造方法注入 <constructor-arg index="参数下标" value="" ref=""

/>

@Autowired (注入bean) @Value (注入值)

# 示例：整合mybatis

# 声明式事务管理

<!-- 事务管理器 -->

<bean id="transactionManager"

f class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

<!-- 启用 @Transactional注解 -->

<tx:annotation-driven transaction-manager="transactionManager"/>

在需要进行事务控制的方法上添加@Transactional注解

// 代理模式, 代理对象, 代理类

class OrderService$$EnhancerBySpringCGLIB$$4e47a71e

extends OrderService {

@Override

public void pay(int orderId, Map<Integer, Integer> map) {

try {

// 开始事务

orderService.pay(orderId, map); // 目标对象

// 提交事务

} catch(RuntimeException e) {

// 回滚事务

}

}

}

事务特性

1. 在业务对象的方法执行过程中，如果出现的是未检查异常(RuntimeException 或 Error)，默认会回滚事务. 检查异常仍然会导致事务提交

如果希望检查异常也回滚：

@Transactional(rollbackFor=Exception.class)

2) 如果业务方法内，自行捕捉了异常，spring认为你的业务方法执行正常，所以会让事务提交而不是回滚

3) readOnly 只读事务

对于只有查询的业务方法，将事务设置为只读事务，对于某些数据库能够提高效率

mysql 设置只读事务，可以提高效率

oracle 设置只读事务，无效

多次查询操作，要保障数据的一致性，就需要把多次查询操作控制在一个事务内，并设置为只读事务（隔离级别）

1. timeout 事务超时

当一个事务运行时间较长，不希望无限等待时，需要进行超时控制。

注意超时时间的计算

1. isolation 隔离级别

read\_uncommitted 未提交读

read\_committed 提交读 （oralce默认） 脏读

repeatable\_read 可重复读 (mysql默认）脏读，不可重复读

serializable 序列化读 脏读，不可重复读，幻读

隔离级别越高，一次事务内数据的一致性越高，但相应的性能越低。

oracle不支持repeatable\_read

# 面向切面编程

Aspect Oriented Programming

切面 面向 编程

声明式事务管理，是面向切面编程思想的体现

把一些能够重用的代码从正常的业务代码中提取出来，放入一个单独的通知类中;

另外需要将通知类和正常业务结合起来，通过切点来定义哪些业务方法需要添加通知;

最后通过代理将业务类(目标类)和通知类结合在一起。

(aspect)切面 = 通知+切点

(advice)通知类--包含可重用代码的类，例如事务通知、缓存通知、日志通知...

(pointcut)切点表达式--定义匹配规则，决定哪些业务方法需要加通知

(proxy)代理 -- 负责结合目标和通知

代理

通知1-> 通知2-> ... 通知n -> 目标(业务)

(target) 目标 -- 通常指业务对象

## spring 中面向切面编程步骤

对业务方法进行效率统计，看每个方法运行花费时间

1. 添加aop相关的jar包
2. 切面类 @Aspect

其中包含通知和切点

通知--对应切面类中的一个方法

切点--是通知方法上的一个注解

1. 配置

切面类也需要交给容器管理

启用aop的注解

## 切点表达式

within(包名.类名) -- 匹配到类中的所有方法

execution(修饰符 返回值 包.类.方法名(参数)) -- 匹配到类中的某些方法

通配符\* 表示任意类型

\* 用在参数时，代表一个任意类型的参数

.. 用在参数，表示参数的个数和类型都是任意的

@annotation(注解)

## 通知类型

@Around 环绕通知 -- 功能最强大, 可以对目标的调用做完全的控制

@Before 前置通知 -- 在目标方法之前被运行

@After 后置通知 -- 在目标方法之后被运行(无论正常或异常)

@AfterReturning -- 在目标方法正常结束后被运行

@AfterThrowing -- 在目标方法出现异常时被运行

## 代理对象的生成原理

jdk 自带动态代理技术: 只能针对接口生成代理

cglib 第三方的库: 不仅能针对接口生成代理，还能够生成子类代理

javassist 第三方的库

BeanPostProcessor

可以在spring在每个bean初始化时被调用，在初始化后，一旦发现此bean与切点表达式匹配，把它用代理对象替换掉

# spring mvc

## mvc 的概念

model 模型 -- 数据以及操作数据的业务逻辑，狭义就是指数据

view 视图 -- 数据的展现形式

controller 控制器 -- 联系模型和视图

spring-webmvc 实现了mvc模式，让程序的扩展性，可维护性提高，让各个类各司其职

## maven 开发springmvc的步骤

1. 加入了maven依赖

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.3.6.RELEASE</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.8.7</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.0.1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

1. 配置统一入口

在web.xml中配置：

<!-- 1.配置所有请求的入口

2.初始化spring容器

3.在应用程序启动时，初始化此servlet

-->

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<!-- 当请求匹配不到别的servlet时，就会将这个请求交给<url-pattern>为/的servlet来处理 -->

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

1. 编写控制器类

@Controller

public class HelloController {

// 请求 映射

@RequestMapping("/hello")

public String hello() {

System.out.println("进入了hello方法");

return "index"; // 返回值代表视图名字

}

}

在spring配置文件中：

<!-- 搜索Controller -->

<context:component-scan base-package="com.yihang.controller"/>

<!-- 配置视图解析器 -->

<mvc:view-resolvers>

<mvc:jsp prefix="/WEB-INF/" suffix=".jsp"/> <!-- 获取到controller方法返回的视图名 添加前缀和后缀，将整个结果作为jsp完整路径让请求转发到此jsp -->

<!-- 前缀 /WEB-INF/ 后缀 .jsp

/WEB-INF/index.jsp

-->

</mvc:view-resolvers>

## 接收请求参数

### 三种方式

方式1： 直接将请求参数与方法参数相对应, 方法参数的类型与参数的实际值对应.

常见的类型springmvc都能完成转换, 日期类型需要使用 @DateTimeFormat注解完成转换

<mvc:annotation-driven/>

方式2： 使用javabean来接收参数, 每个请求参数和javabean的属性相对应

方式3：

传统请求参数格式： ?参数名=参数值&

路径参数格式 路径/{参数}

/user/1

/user/2

### 请求中的中文乱码问题

post 需要配置spring提供的CharacterEncodingFiliter

get 如果采用tomcat 8.0 ，无需任何设置

tomcat 7.0以下的版本 , server.xml

<Connector URIEncoding="utf-8">

### 接收多个参数值

格式1： 参数名=值1&参数名=值2...

格式2： 参数名=值1,值2,...

springmvc中

方法上直接加一个 @RequestParam List<> 参数名

用javabean 定义属性 List<> 参数名

### 处理上传文件

<form method="post" action="" enctype="multipart/form-data">

多部分/表单数据

正常的表单数据格式：

name=%E4%B8%AD%E5%8D%88&age=32&birthday=2017-1-1&hobby=1&hobby=2

多部分/表单数据格式：

------WebKitFormBoundaryHi28hLL7wBpdAGIh

Content-Disposition: form-data; name="name"

中文

------WebKitFormBoundaryHi28hLL7wBpdAGIh

Content-Disposition: form-data; name="age"

38

------WebKitFormBoundaryHi28hLL7wBpdAGIh

Content-Disposition: form-data; name="birthday"

2017-1-1

------WebKitFormBoundaryHi28hLL7wBpdAGIh

Content-Disposition: form-data; name="hobby"

1

------WebKitFormBoundaryHi28hLL7wBpdAGIh

Content-Disposition: form-data; name="hobby"

2

------WebKitFormBoundaryHi28hLL7wBpdAGIh--

java中处理 mutilpart格式的表单

1. commons-fileupload

<dependency>

<groupId>commons-fileupload</groupId>

<artifactId>commons-fileupload</artifactId>

<version>1.3.1</version>

</dependency>

spring配置文件中

<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

</bean>

1. servlet 3.0

<servlet>

...

<multipart-config>

<!-- <location></location> 用来指定上传文件使用的临时目录 -->

<!-- <max-file-size></max-file-size> 每个上传文件的最大大小 （字节） -->

<!-- <max-request-size></max-request-size> 整个请求的最大长度 （字节） -->

<!-- <file-size-threshold></file-size-threshold> 文件大小阈值 -->

</multipart-config>

</servlet>

spring配置文件中（不加也可以）：

<bean id="multipartResolver" class="org.springframework.web.multipart.support.StandardServletMultipartResolver"></bean>

## 请求中的问题处理

### 静态资源（图片，html，css，js）访问时404

原因：springmvc dispatcherServlet配置的 / 路径覆盖了tomcat 用来处理静态资源的 defaultServlet的 /路径

解决方法：

在spring配置文件中加入

<mvc:default-servlet-handler/>

### 请求参数的类型转换问题 400

原因：springmvc在处理请求参数时，无法将字符串转换为java中的类型，此时会出现400错误

例子 字符串转整数，或字符串转日期

### 请求参数空导致 400

请求参数非空检查，出现在加了@RequestParam注解的参数上，如果请求参数为null，那么也会报400错误

解决方法：可以通过 @RequestParam(required=true|false)来进行控制

@RequestParam(defaultValue="默认值", name="请求参数名")

@PathVariable中的占位参数总是必须的

### 应用首页

http://ip:port/应用名称 --> 跳转至应用程序首页面

方法一：

写一个控制器

@Controller

public class IndexController {

@RequestMapping("/")

public String foo() {

return "index";

}

}

方法二：

使用view-controller，直接将请求路径映射到视图

<!-- 用来专门映射一些无需任何业务处理的路径，把这些路径直接映射到视图名 -->

<mvc:view-controller path="/" view-name="index"/>

### 自定义错误页面

400

404

500

在web.xml定义：

<error-page>

<location>/error</location>

</error-page>

定义controller：

@Controller

public class StatusCodeController {

@RequestMapping("/error")

public String error() {

return "error";

}

}

可以通过自定义异常的方式，将异常转为对应的响应状态码

@ResponseStatus(code = HttpStatus.BAD\_REQUEST)

public class MyException extends Exception {

public MyException(String message) {

super(message);

}

}

## 惯例优于配置

用一些规定来减少配置

1. 控制器方法如果没有返回结果，那么springmvc就会将@RequestMapping的路径作为视图名
2. 建议每个控制器类，对应一个独立的jsp目录

控制器内的每个方法和此目录下的jsp文件相对应

1. 建议将jsp放入web-inf下，不要直接访问，而是通过控制器请求转发间接访问

## 模型

Model 接口

ExtendedModelMap 实现类 同时实现了 Model接口和Map接口

支持链式调用

Model model = new ExtendedModelMap();

// 传统写法

model.addAttribute("key", value);

model.addAttribute("key", value);

// 链式写法

model.addAttribute("key", value).addAttribute("key", value);

model.addAttribute(对象);

model.addAttribute(List,Set,数组)

### 惯例优于配置

命名规则：

1. 根据对象的类型 将类型首字母小写，作为model中的名称
2. 如果是集合

List<User> userList

Set<User> userList

User[] userList

### 模型的几种变体

方式1: 在控制器方法上使用 Model类型的特殊参数

方式2: 在控制器方法上使用 ModelMap类型的特殊参数

方式3: 在控制器方法上使用 Map 类型的特殊参数

方式4: 给控制器方法参数，添加一个@ModelAttribute 注解， 加了此注解的参数就会被存入模型集合当中

## 请求重定向

### 重定向使用经验

相对独立的两个操作之间，跳转使用重定向

联系紧密的两个操作之间，跳转使用请求转发

### 重定向传值

方法1： 使用HttpSession

@RequestMapping("/redirect/test1")

public String test1(HttpSession session) {

session.setAttribute("username", "张三");

return "redirect:/redirect/test2";

}

@RequestMapping("/redirect/test2")

public void test2(HttpSession session) {

Object value = session.getAttribute("username");

System.out.println("值是： " + value);

}

方法2： 利用在重定向地址后追加请求参数的方式进行传值

@RequestMapping("/redirect/test1")

public String test1(Model model) {

model.addAttribute("username", "张三");

return "redirect:/redirect/test2";

}

第二次请求时 /redirect/test2?username=张三

@RequestMapping("/redirect/test2")

public void test2(String username) {

System.out.println("username:" + username);

}

或者：

@RequestMapping("/redirect/test1")

// RedirectAttributes 专门用来处理重定向时的数据

public String test1(RedirectAttributes ra) {

ra.addAttribute("username", "张三");

return "redirect:/redirect/test2";

}

第二次请求时 /redirect/test2?username=张三

@RequestMapping("/redirect/test2")

public void test2(String username) {

System.out.println("username:" + username);

}

注意：

1. 在重定向传参时， RedirectAttributes 的优先级高于 Model
2. Model.addAttribute(), RedirectAttributes.addAttribute()的局限性是只能传递简单类型的参数

方法3：

@RequestMapping("/redirect/test1")

// RedirectAttributes 专门用来处理重定向时的数据

public String test1(RedirectAttributes ra) {

User user = new User();

user.setName("张三");

user.setAge(18);

user.setHobby(Arrays.asList(1, 2, 3));

user.setBirthday(new Date());

// 在重定向地址后追加请求参数的方式

ra.addAttribute("username", "张三");

// session.setAttribute("user", user);

ra.addFlashAttribute("user", user);

return "redirect:/redirect/test2";

}

@RequestMapping("/redirect/test2")

// Object user = session.getAttribute("user");

// model.addAttribute("user", user);

// session.removeAttribute("user");

public void test2(String username, ModelMap model) {

User user = (User) model.get("user");

System.out.println("user:" + user);

System.out.println("username:" + username);

}

## 视图

View

public String test() {

return "视图名";

}

视图解析器

<mvc:jsp prefix="/WEB-INF/" suffix=".jsp"/>

new JSLTView("/WEB-INF/视图名.jsp");

生成excel视图

1）添加maven依赖

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi</artifactId>

<version>3.9</version>

</dependency>

1. 控制器方法返回视图对象

@RequestMapping("/view/test2")

public View test2(Model model) {

List<User> list = new ArrayList<>();

list.add(new User("张三", 18));

list.add(new User("李四", 28));

list.add(new User("王五", 38));

list.add(new User("赵六", 48));

// 将数据存入模型

model.addAttribute("list", list);

// 创建excel视图对象

return new AbstractXlsView() {

@SuppressWarnings("unchecked")

@Override

protected void buildExcelDocument(Map<String, Object> model, Workbook workbook, HttpServletRequest request,

HttpServletResponse response) throws Exception {

// workbook 是工作簿

Sheet sheet = workbook.createSheet("学生工作表"); // 创建工作表

List<User> list = (List<User>) model.get("list");

int i = 0;

for (User user : list) {

Row row = sheet.createRow(i++); // 创建一行

row.createCell(0).setCellValue(user.getName());

row.createCell(1).setCellValue(user.getAge());

}

}

};

}