Spring

Not Given

2022 年 7 月 12 日

目录

第一章：Spring 的基本概念

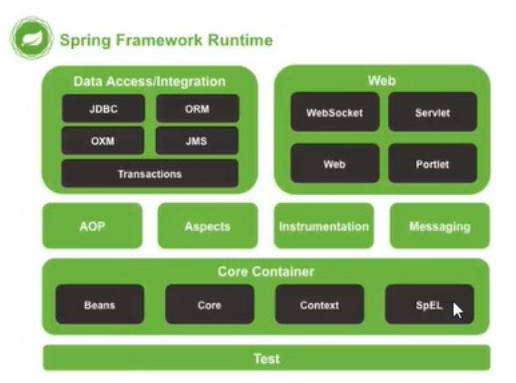
一、Spring 的简介

一种分层的轻量级开发框架，以控制反转和面向切面编程为内核

二、Spring 的优势

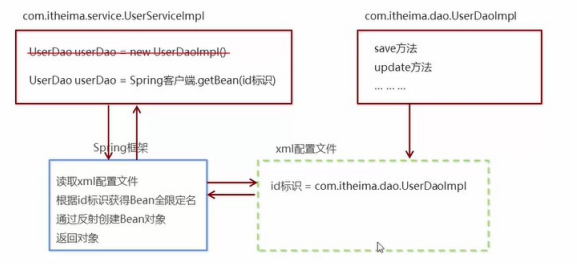
1. 方便解耦，简化开发。之前业务层需要调用 Dao 层创建对象，而使用 Spring 后可以调用 Ioc 容器创建对象
2. AOP 编程的支持
3. 声明式事务的支持
4. 方便程序的测试
5. 方便集成各种优秀的框架
6. 降低 javaEE 的使用难度

三、Spring 的体系结构



四、Spring 程序开发步骤

1. 运行步骤



2. 开发步骤

① 导入 Spring 开发的基本包坐标
② 编写 Dao 接口和实现类
③ 创建 Spring 核心配置文件
④ 在 Spring 配置文件中配置 UserDaoImpl
⑤ 使用 Spring 的 API 获得 Bean 实例

第二章：Spring 的配置文件

一、Bean 标签的配置

(一) 基本标签

1. 格式：`<Bean id=" " class=" "> </Bean>`
2. id：唯一性标识，在配置文件内部 id 不允许重复
3. class：全类名，通过 User 类中的无参构造创建对象
4. scope：范围配置

取值范围	说明
singleton	默认值，单例的
prototype	多例的
request	WEB 项目中，Spring 创建一个 Bean 的对象，将对象存入到 request 域中
session	WEB 项目中，Spring 创建一个 Bean 的对象，将对象存入到 session 域中
global session	WEB 项目中，应用在 Portlet 环境，如果没有 Portlet 环境那么globalSession 相当于 session

单例和多例的区别

5. 生命周期配置：除了需要在 User 类中定义方法，还需要在 Bean 中进行相关配置

   (1) init-method：初始化方法。`init-method="init"`

   (2) destory-method：销毁方法。`destory-method="destory"`

(二) 依赖注入

1. 概念：当调用者需要另一个角色协助时，不再需要直接调用，而是通过 Spring 中的控制反转功能进行管理。通常是把持久层对象传入业务层
2. 注入分类：普通数据类型，引用数据类型（Bean 对象），集合数据类型
3. 普通类型注入方式：`<property name=" 被调用数据对应属性名" value=" 属性值"></property>`
4. 引用类型注入方式

   (1) set 方法：为被调用对象创建 set 方法，之后在配置文件的 Bean 中添加标签

   `<property name=" 被调用对象对应属性名" ref=" 被调用对象在 Bean 中对应的 id"></property>`

   (2) 构造器：创建以被调用对象为参数的构造函数，之后在配置文件中添加标签

   `<constructor-arg name=" 被调用对象对应属性名" ref=" 被调用对象在 Bean 中对应的 id"></constructor-arg>`

5. 集合类注入方式：Map，List 等

```
<property name="strlist">
    <list>
        <value> </value>
    </list>
</property>
```

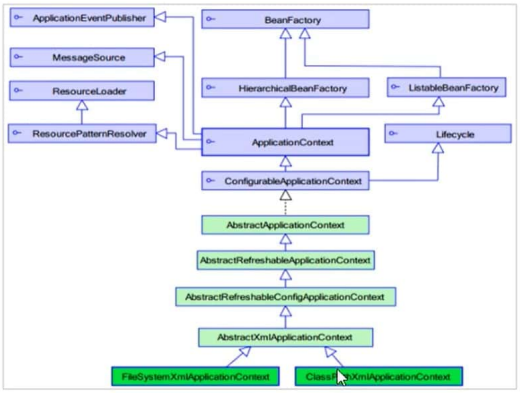
二、引入其他配置文件

1. 目的：分模块开发
2. 方式：`<import resource=" 文件路径"`

第三章：Spring 的 Api

一、ApplicationContext 继承体系

1. AC 的概念与作用：接口类型，可以通过其实例获得 Spring 中的 Bean 对象
2. 继承体系



3. 实现类

1) ClassPathXmlApplicationContext

它是从类的根路径下加载配置文件推荐使用这种

2) FileSystemXmlApplicationContext

它是从磁盘路径上加载配置文件，配置文件可以在磁盘的任意位置。

3) AnnotationConfigApplicationContext

当使用注解配置容器对象时，需要使用此类来创建 spring 容器。它用来读取注解。

二、getBean() 方法

1. 使用 id：可以在有多个同类型 Bean 时使用，需要进行强制类型转化
2. 使用类的字节码对象类型：不可以在有多个同类型 Bean 时使用，不需要进行强制类型转化

```
public static void main(String[] args) {
    //ApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
    ApplicationContext app = new FileSystemXmlApplicationContext( configLocation: "C:\\Users\\apple\\Idea
    //UserService userService = (UserService) app.getBean("userService");
    UserService userService = app.getBean(UserService.class);
    userService.save();
}
```

第四章：配置数据源

一、数据源的简介

1. 同义词：连接池
2. 作用
   - 数据源(连接池)是提高程序性能如出现的
   - 事先实例化数据源，初始化部分连接资源
   - 使用连接资源时从数据源中获取
   - 使用完毕后将连接资源归还给数据源
3. 开发步骤
   ① 导入数据源的坐标和数据库驱动坐标
   ② 创建数据源对象
   ③ 设置数据源的基本连接数据
   ④ 使用数据源获取连接资源和归还连接资源

二、手动创建数据源

(一) 直接创建

1. 导入坐标
2. 测试 C3P0 数据源

```java
public class DataSourceTest {

    @Test
    //测试手动创建 c3p0 数据源
    public void test1() throws Exception {
        ComboPooledDataSource dataSource = new ComboPooledDataSource();
        dataSource.setDriverClass("com.mysql.jdbc.Driver");
        dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/test");
        dataSource.setUser("root");
        dataSource.setPassword("root");
        Connection connection = dataSource.getConnection();
        System.out.println(connection);
        connection.close();
    }

}
```

3. 测试 Druid 数据源

```java
public class DataSourceTest {

    @Test
    //测试手动创建 druid 数据源
    public void test2() throws Exception {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/test");
        dataSource.setUsername("root");
        dataSource.setPassword("root");
        DruidPooledConnection connection = dataSource.getConnection();
        System.out.println(connection);
        connection.close();
    }
```

(二) 使用配置文件创建数据源

1. 目的：解耦合
2. 步骤

   (1) 创建 jdbc.properties 配置文件

   (2) 读取配置文件，不需要扩展名

   (3) 创建数据源对象，并设置连接参数

```java
public class DataSourceTest {

    @Test
    //测试手动创建 c3p0 数据源(加载properties配置文件)
    public void test3() throws Exception {
        //读取配置文件
        ResourceBundle rb = ResourceBundle.getBundle("jdbc");
        String driver = rb.getString( key: "jdbc.driver");
        String url = rb.getString( key: "jdbc.url");
        String username = rb.getString( key: "jdbc.username");
        String password = rb.getString( key: "jdbc.password");
        //创建数据源对象  设置连接参数
        ComboPooledDataSource dataSource = new ComboPooledDataSource();
        dataSource.setDriverClass(driver);
        dataSource.setJdbcUrl(url);
        dataSource.setUser(username);
        dataSource.setPassword(password);

        Connection connection = dataSource.getConnection();
        System.out.println(connection);
        connection.close();
    }
```

三、使用 Spring 创建数据源

(一) 不使用配置文件

1. 方式：用 Bean 配置数据源
2. 步骤

   (1) 在 pom 文件中导入 Spring 的坐标

   (2) 在 applicationContext.xml 文件中配置 Bean

```xml
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
    <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/test"></property>
    <property name="user" value="root"></property>
    <property name="password" value="root"></property>
</bean>
```

   (3) 获取 Bean

```java
@Test
//测试Spring容器产生数据源对象
public void test4() throws Exception {
    ApplicationContext app = new ClassPathXmlApplicationContext( configLocation: "applicationContext.xml");
    DataSource dataSource = app.getBean(DataSource.class);
    Connection connection = dataSource.getConnection();
    System.out.println(connection);
    connection.close();
}
```

(二) 使用配置文件

1.4 抽取jdbc配置文件

applicationContext.xml加载jdbc.properties配置文件获得连接信息。

首先，需要引入context命名空间和约束路径:

- 命名空间：xmlns:context="http://www.springframework.org/schema/context"
- 约束路径：http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd

```xml
<context:property-placeholder location="classpath:jdbc.properties"/>
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="${jdbc.driver}"/>
    <property name="jdbcUrl" value="${jdbc.url}"/>
    <property name="user" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

第五章：注解开发

一、简介

1. Spring 特点：轻代码，重配置
2. 注解相对于配置文件优势：简化配置，提高开发速度

二、原始注解

(一) 概述

1. 功能：代替 Bean 的配置
2. 注解概览

Spring原始注解主要是替代<Bean>的配置

注解	说明
@Component	使用在类上用于实例化Bean
@Controller	使用在web层类上用于实例化Bean
@Service	使用在service层类上用于实例化Bean
@Repository	使用在dao层类上用于实例化Bean
@Autowired	使用在字段上用于根据类型依赖注入
@Qualifier	结合@Autowired一起使用用于根据名称进行依赖注入
@Resource	相当于@Autowired+@Qualifier，按照名称进行注入
@Value	注入普通属性
@Scope	标注Bean的作用范围
@PostConstruct	使用在方法上标注该方法是Bean的初始化方法
@PreDestroy	使用在方法上标注该方法是Bean的销毁方法

(二) 具体配置方式

1. 组件扫描：告诉 Spring 哪个包下有注解需要扫描

注意：

使用注解进行开发时，需要在applicationContext.xml中配置组件扫描，作用是指定哪个包及其子包下的Bean需要进行扫描以便识别使用注解配置的类、字段和方法。

```
<!--注解的组件扫描-->
<context:component-scan base-package="com.itheima"></context:component-scan>
```

2. 实例化 Bean

```
//<bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"></bean>
@Component("userDao")
public class UserDaoImpl implements UserDao {
    public void save() {
        System.out.println("save running...");
    }
}
```

3. 依赖注入

```
@Component("userService")
public class UserServiceImpl implements UserService {

    //<property name="userDao" ref="userDao"></property>
    @Autowired
    @Qualifier("userDao")
    private UserDao userDao;
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
```

(三) 其他形式的配置

1. 普通类型数据注入

```
@Value("itcast")
private String driver;
```

直接注入

```
@Value("${jdbc.driver}")
private String driver;
```

使用配置文件

2. 注解范围配置

```
@Service("userService")
//@Scope("prototype")
@Scope("singleton")
public class UserServiceImpl implements UserService {

    @Value("${jdbc.driver}")
    private String driver;
```

三、新注解

(一) 概述

1. 原始注解的不足

使用上面的注解还不能全部替代xml配置文件，还需要使用注解替代的配置如下：

- 非自定义的Bean的配置：<bean>
- 加载properties文件的配置：<context:property-placeholder>
- 组件扫描的配置：<context:component-scan>
- 引入其他文件：<import>

2. 新注解概览

注解	说明
@Configuration	用于指定当前类是一个 Spring 配置类，当创建容器时会从该类上加载注解
@ComponentScan	用于指定 Spring 在初始化容器时要扫描的包。 作用和在 Spring 的 xml 配置文件中的 <context:component-scan base-package="com.itheima"/>一样
@Bean	使用在方法上，标注将该方法的返回值存储到 Spring 容器中
@PropertySource	用于加载.properties 文件中的配置
@Import	用于导入其他配置类

3. 新注解的核心思想：用一个包含注解的配置类来代替配置文件

(二) 具体配置方式

1. 创建核心配置类

```
@Configuration
//<context:component-scan base-package="com.itheima"/>
@ComponentScan("com.itheima")
//<context:property-placeholder location="classpath:jdbc.properties"/>
@PropertySource("classpath:jdbc.properties")
public class SpringCofiguration {

    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;
```

2. 加载配置类

```
public class UserController {

    public static void main(String[] args) {
        //ClassPathXmlApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
        ApplicationContext app = new AnnotationConfigApplicationContext(SpringCofiguration.class);
        UserService userService = app.getBean(UserService.class);
        userService.save();
    }
```

第六章：Spring 集成 Junit

一、简介

1. 之前方法的缺陷：测试时需要先创建容器、获取对象，操作繁琐

   在测试类中，每个测试方法都有以下两行代码：

```
ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");
IAccountService as = ac.getBean("accountService",IAccountService.class);
```

   这两行代码的作用是获取容器，如果不写的话，直接会提示空指针异常。所以又不能轻易删掉。

2. 解决方法：让 SpringJunit 创建容器，并直接在测试类中注入测试 Bean

3. 操作步骤

   ① 导入spring集成Junit的坐标
   ② 使用@Runwith注解替换原来的运行期
   ③ 使用@ContextConfiguration指定配置文件或配置类
   ④ 使用@Autowired注入需要测试的对象
   ⑤ 创建测试方法进行测试

二、具体操作

1. 导入坐标

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>5.0.5.RELEASE</version>
</dependency>
```

2. 编写测试类

```
@RunWith(SpringJUnit4ClassRunner.class)
//@ContextConfiguration("classpath:applicationContext.xml")
@ContextConfiguration(classes = {SpringCofiguration.class})
public class SpringJunitTest {

    @Autowired
    private UserService userService;

    @Autowired
    private DataSource dataSource;

    @Test
    public void test1() throws SQLException {
        userService.save();
```

第七章：SpringMVC

一、Spring 集成 Web 环境

(一) 基本步骤

1. 添加 Servlet 相关坐标

```
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <version>2.2.1</version>
    <scope>provided</scope>
```

2. 覆写 Servlet 方法

```
public class UserServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOExcep
        ApplicationContext app = new ClassPathXmlApplicationContext( configLocation: "applicationContext.xm
        UserService userService = app.getBean(UserService.class);
        userService.save();
    }
}
```

3. 在 Web.xml 中配置 Servelt

```
<servlet>
    <servlet-name>UserServlet</servlet-name>
    <servlet-class>com.itheima.web.UserServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>UserServlet</servlet-name>
    <url-pattern>/userServlet</url-pattern>
</servlet-mapping>
```

(二) 存在的问题及解决

1. 存在的问题：多次加载 xml 文件并创建容器

```
ApplicationContext app = new ClassPathXmlApplicationContext( configLocation: "applicationContext.xm
UserService userService = app.getBean(UserService.class);
```

2. 解决方法：监听 ServletContext，并在监听类中加载文件并创建容器，创建 ApplicationContext 应用上下文对象，再将其放到最大域中

```
public class ContextLoaderListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent servletContextEvent) {
        ApplicationContext app = new ClassPathXmlApplicationContext( configLocation: "applicationContext.xml
        //将Spring的应用上下文对象存储到ServletContext域中
        ServletContext servletContext = servletContextEvent.getServletContext();
        servletContext.setAttribute( s: "app",app);
        System.out.println("spring容器创建完毕....");
    }

    public void contextDestroyed(ServletContextEvent servletContextEvent) {

    }
}
```

(三) 使用 Spring 中的监听器

1. 简介

上面的分析不用手动实现，Spring提供了一个监听器**ContextLoaderListener**就是对上述功能的封装，该监听器内部加载Spring配置文件，创建应用上下文对象，并存储到**ServletContext**域中，提供了一个客户端工具**WebApplicationContextUtils**供使用者获得应用上下文对象。

2. 步骤

所以我们需要做的只有两件事:

① 在**web.xml**中配置**ContextLoaderListener**监听器（导入spring-web坐标）

② 使用**WebApplicationContextUtils**获得应用上下文对象**ApplicationContext**

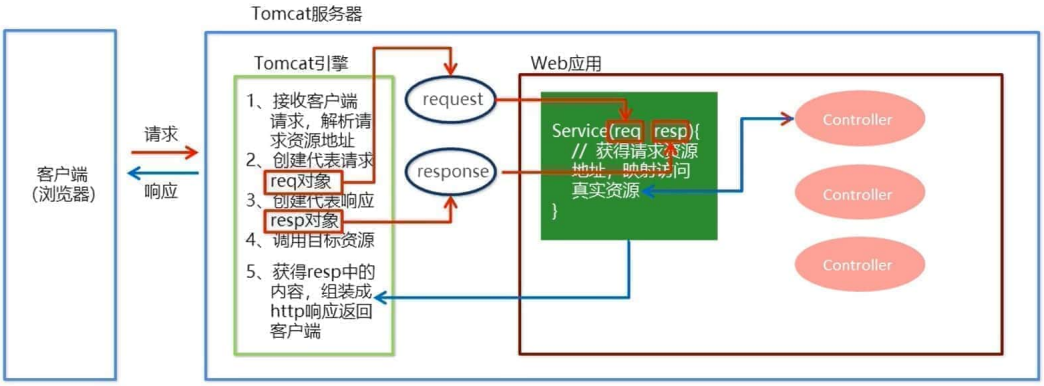
二、SpringMVC 简介

1. 概述

SpringMVC是一种基于Java的实现**MVC设计模型**的请求驱动类型的轻量级**Web框架**，属于**SpringFrameWork**的后续产品，已经融合在Spring Web Flow中。

SpringMVC已经成为目前最主流的MVC框架之一，并且随着Spring3.0的发布，全面超越Struts2，成为最优秀的MVC框架。它通过一套注解，让一个简单的Java类成为处理请求的控制器，而无须实现任何接口。同时它还支持**RESTful**编程风格的请求。

2. Web 层处理基本结构



3. 开发步骤

(1) 导入 SpringMVC 包

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.0.5.RELEASE</version>
</dependency>
```

(2) 配置 Servlet，即 SpringMVC 前端控制器

```
<!--配置SpringMVC的前端控制器-->
<servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

(3) 编写 Controller，将 Controller 使用注解配置到 Spring 容器中

```
@Controller
public class UserController {

    @RequestMapping("/quick")
    public String save(){
        System.out.println("Controller save running....");
        return "success.jsp";
    }

}
```

(4) 配置组件扫描，在 spring-mvc.xml 中进行配置，只扫描 Web 层。并在 Web.xml 的 Servlet 标签内配置加载文件

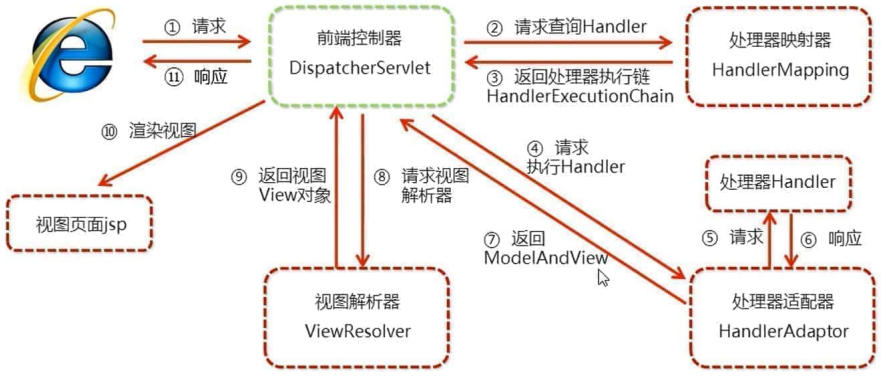
```
ion context not configured for this file                    Configure application context
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.or
                           http://www.springframework.org/schema/context http://www.springframework.or

    <!--Controller的组件扫描-->
    <context:component-scan base-package="com.itheima.controller"/>

</beans>
```

```
<servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

三、SpringMVC 组件解析

1. SpringMVC 底层执行流程

① 用户发送请求至前端控制器DispatcherServlet。
② DispatcherServlet收到请求调用HandlerMapping处理器映射器。
③ 处理器映射器找到具体的处理器(可以根据xml配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
④ DispatcherServlet调用HandlerAdapter处理器适配器。
⑤ HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。
⑥ Controller执行完成返回ModelAndView。
⑦ HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet。
⑧ DispatcherServlet将ModelAndView传给ViewReslover视图解析器。
⑨ ViewReslover解析后返回具体View。
⑩ DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。DispatcherServlet响应用户。



2. 注解 @RequestMapping 解析

(1) 作用：建立 url 与处理请求的方法之间的对应关系，可以用在类上或方法上

```
public class UserController {

    // 请求地址  http://localhost:8080/user/quick
    @RequestMapping(value="/quick")
    public String save(){
        System.out.println("Controller save running....");
        return "/success.jsp";
    }

}
```

(2) 其他属性

- **value**：用于指定请求的URL。它和path属性的作用是一样的
- **method**：用于指定请求的方式
- **params**：用于指定限制请求参数的条件。它支持简单的表达式。要求请求参数的key和value必须和配置的一模一样

例如：

- **params = {"accountName"}**，表示请求参数必须有accountName
- **params = {"moeny!100"}**，表示请求参数中money不能是100

3. 组件扫描：SpringMVC 主要扫描 Web 层，其他的归 Spring 扫描
4. xml 文件配置
   (1) 功能：进行组件扫描，或者添加组件
   (2) 示例：可以进行视图解析器的配置，为 return 的链接自动添加前缀和后缀；也可以配置 Json 转换器，将封装好的对象转化为 Json 字符串格式

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/jsp/"></property>
    <property name="suffix" value=".jsp"></property>
</bean>
```

配置视图解析器

```
<!--配置处理器映射器-->
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
    <property name="messageConverters">
        <list>
            <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverte
        </list>
    </property>
```

配置 Json 转化器

四、SpringMVC 获取请求参数

(一) 请求数据类型概述

客户端请求参数的格式是：**name=value&name=value……**

服务器端要获得请求的参数，有时还需要进行数据的封装，SpringMVC可以接收如下类型的参数：

- 基本类型参数
- POJO类型参数
- 数组类型参数
- 集合类型参数

(二) 获取基本类型数据

1. 在 Controller 类中添加与请求中一致的参数，获得请求时会直接传入数据

```
http://localhost:8080/itheima_springmvc1/quick9?username=zhangsan&age=12
```

```
@RequestMapping("/quick9")
@ResponseBody
public void quickMethod9(String username,int age) throws IOException {
    System.out.println(username);
    System.out.println(age);
}
```

2. 使用数据

(三) 获取 POJO 数据

1. 当类的属性名与请求参数一致时，会自动封装

```
http://localhost:8080/itheima_springmvc1/quick9?username=zhangsan&age=12
```

```
public class User {
    private String username;
    private int age;
    getter/setter…
}
@RequestMapping("/quick10")
@ResponseBody
public void quickMethod10(User user) throws IOException {
    System.out.println(user);
}
```

2. 使用数据

(四) 获取数组类型数据

1. 当类参数中的数组名称与请求参数名称一致时，会自动封装

```
http://localhost:8080/itheima_springmvc1/quick11?strs=111&strs=222&strs=333
```

```
@RequestMapping("/quick11")
@ResponseBody
public void quickMethod11(String[] strs) throws IOException {
    System.out.println(Arrays.asList(strs));
}
```

2. 使用数据

(五) 获取集合类型数据

1. 将集合参数包含到一个 POJO 中

```
public class VO {
    private List<User> userList;

    public List<User> getUserList() {
        return userList;
    }
```

2. 设置这个对象为 Controller 类的参数，接收数据
3. 特殊情况-AJAX and JSON
   (1) 方式：当使用 AJAX 提交时，可以指定接收格式为 JSON，并在服务器端用 @RequestBody 注解标注，从而对集合数据进行直接封装
   (2) 示例

```
$.ajax({
    type:"POST",
    url:"${pageContext.request.contextPath}/user/quick15",
    data:JSON.stringify(userList),
    contentType:"application/json;charset=utf-8"
});
```

ajax.jsp

```
public void save15(@RequestBody List<User> userList) throws IOException {
    System.out.println(userList);
}
```

UserController.java

(六) 静态资源的访问

1. 存在问题：SpringMVC 的前端控制器接管了所有的 Servlet，无法在该虚拟目录下找到静态资源
2. 解决方式

   (1) 单独配置静态资源的访问路径，mapping 配置了找资源的地址，location 配置了哪个目录下的资源是对外开放的

```
<!--开发资源的访问-->
<mvc:resources mapping="/js/**" location="/js/"/>
<mvc:resources mapping="/img/**" location="/img/"/>
```

   (2) 设置当 MVC 找不到资源时，使用原始容器去找静态资源

```
<mvc:default-servlet-handler/>
```

(七) 请求数据乱码的问题

1. 存在问题：当请求体数据为中文时，可能出现乱码
2. 解决方式：在 web.xml 中配置全局过滤器

当post请求时，数据会出现乱码，我们可以设置一个过滤器来进行编码的过滤。

```
<filter>
   <filter-name>CharacterEncodingFilter</filter-name>
   <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
   <init-param>
     <param-name>encoding</param-name>
     <param-value>UTF-8</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>CharacterEncodingFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

(八) 参数绑定

1. 存在问题：请求参数中的键名与类中的参数不一致
2. 解决方式：使用注解设置参数绑定

注解@RequestParam还有如下参数可以使用：

- **value**：与请求参数名称
- **required**：此在指定的请求参数是否必须包括，默认是true，提交时如果没有此参数则报错
- **defaultValue**：当没有指定请求参数时，则使用指定的默认值赋值

```
@RequestMapping("/quick14")
@ResponseBody
public void quickMethod14(@RequestParam(value="name",required =
false,defaultValue = "itcast") String username) throws IOException {
    System.out.println(username);
}
```

(九) 自定义类型转换器

1. 目的：将来自客户端的请求数据进行类型转换
2. 方式

   (1) 定义转换器类，实现 Converter 接口

```java
public class DateConverter implements Converter<String, Date> {
    public Date convert(String dateStr) {
        //将日期字符串转换成日期对象 返回
        SimpleDateFormat format = new SimpleDateFormat( pattern: "yyyy-MM-dd");
        Date date = null;
        try {
            date = format.parse(dateStr);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return date;
    }
}
```

(2) 在配置文件中声明转换器

```xml
<!--声明转换器-->
<bean id="conversionService" class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <list>
            <bean class="com.itheima.converter.DateConverter"></bean>
        </list>
    </property>
</bean>
```

(3) 在 <annotation-driven> 中引用

```xml
<!--mvc的注解驱动-->
<mvc:annotation-driven conversion-service="conversionService"/>
```

(十) 获取请求头

1. 获取一般请求头

1. @RequestHeader

使用@RequestHeader可以获得请求头信息，相当于web阶段学习的request.getHeader(name)

@RequestHeader注解的属性如下：

- **value**：请求头的名称
- **required**：是否必须携带此请求头

```java
@RequestMapping("/quick17")
@ResponseBody
public void quickMethod17(
        @RequestHeader(value = "User-Agent",required = false) String
headerValue){
    System.out.println(headerValue);
}
```

2. 获取 Cookie 的值

2. @CookieValue

使用@CookieValue可以获得指定Cookie的值

@CookieValue注解的属性如下：

- **value**：指定cookie的名称
- **required**：是否必须携带此cookie

(十一) 文件上传

1. 客户端三要素

1. 文件上传客户端三要素

- 表单项type= “file”
- 表单的提交方式是post
- 表单的enctype属性是多部分表单形式，及enctype= “multipart/form-data”

```
<form action="${pageContext.request.contextPath}/quick20" method="post"
    enctype="multipart/form-data">
    名称: <input type="text" name="name"><br>
    文件: <input type="file" name="file"><br>
    <input type="submit" value="提交"><br>
</form>
```

2. 多部分表单的请求体

2. 文件上传原理

- 当form表单修改为多部分表单时，request.getParameter()将失效。
- enctype= “application/x-www-form-urlencoded” 时，form表单的正文内容格式是:
  key=value&key=value&key=value
- 当form表单的enctype取值为Mutilpart/form-data时，请求正文内容就变成多部分形式:

3. 单文件上传

(1) 在 pom.xml 导入 fileupload 和 io 的坐标

```
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.2.2</version>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.4</version>
</dependency>
```

(2) 配置文件上传解析器

② 配置文件上传解析器

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!--上传文件总大小-->
    <property name="maxUploadSize" value="5242800"/>
    <!--上传单个文件的大小-->
    <property name="maxUploadSizePerFile" value="5242800"/>
    <!--上传文件的编码类型-->
    <property name="defaultEncoding" value="UTF-8"/>
</bean>
```

(3) 编写文件上传代码

③ 编写文件上传代码

```
@RequestMapping("/quick20")
@ResponseBody
public void quickMethod20(String name,MultipartFile uploadFile) throws
IOException {
    //获得文件名称
    String originalFilename = uploadFile.getOriginalFilename();
    //保存文件
    uploadFile.transferTo(new File("C:\\upload\\"+originalFilename));
}
```

4. 多文件上传：设置多个表单参数和类参数。当多个文件名称一致时，服务端也可以使用数组接收

多文件上传，只需要将页面修改为多个文件上传项，将方法参数MultipartFile类型修改为MultipartFile[]即F

```
<h1>多文件上传测试</h1>
<form action="${pageContext.request.contextPath}/quick21" method="post"
enctype="multipart/form-data">
    名称：<input type="text" name="name"><br>
    文件1：<input type="file" name="uploadFiles"><br>
    文件2：<input type="file" name="uploadFiles"><br>
    文件3：<input type="file" name="uploadFiles"><br>
    <input type="submit" value="提交"><br>
</form>
```

upload.jsp

```
@RequestMapping("/quick21")
@ResponseBody
public void quickMethod21(String name,MultipartFile[] uploadFiles) throws
IOException {
    for (MultipartFile uploadFile : uploadFiles){
        String originalFilename = uploadFile.getOriginalFilename();
        uploadFile.transferTo(new File("C:\\upload\\"+originalFilename));
    }
}
```

UserController.java

五、SpringMVC 的数据响应

(一) 响应方式概述

1）页面跳转
- 直接返回字符串
- 通过ModelAndView对象返回

2）回写数据
- 直接返回字符串
- 返回对象或集合

(二) 页面跳转-返回字符串形式

1. 过程：将返回的字符串与视图解析器的前后缀拼接后跳转
2. 前缀：可以决定是请求转发（forward）还是重定向（redirect）
3. 数据的存储与调用：可以使用 Model 对象

(三) 页面跳转-返回 ModelAndView

1. 创建 ModelAndView 对象，或者将 ModelAndView 设置为类的参数。使用第二种时，当方法被调用，会自动被传递实参

```
@RequestMapping(value="/quick2")
public ModelAndView save2(){
    /*
        Model:模型 作用封装数据
        View: 视图 作用展示数据
     */
    ModelAndView modelAndView = new ModelAndView();
    //设置视图名称
    modelAndView.setViewName("success");
    return modelAndView;
}
```

```
@RequestMapping(value="/quick3")
public ModelAndView save3(ModelAndView modelAndView){
    modelAndView.addObject( attributeName: "username", attributeValue: "itheima");
    modelAndView.setViewName("success");
    return modelAndView;
}
```

2. 设置视图与模型数据。视图是一个网页（html 或 jsp），模型可以看做一个数据存储域
3. 返回视图，并从域中拿取数据
4. 其他注意事项：也可以使用原始的 Request 对象进行数据的存取，但是不常用。使用框架后应当尽量使用框架封装好的数据类型

(四) 回写数据-返回字符串

1. Web 原始方式：将 Response 对象设定为类的参数，调用 Response 中的方法

```
@RequestMapping("/quick4")
public void quickMethod4(HttpServletResponse response) throws
IOException {
    response.getWriter().print("hello world");
}
```

2. MVC 注解方式：使用注解告知 MVC 是回写数据而非页面跳转，再 return

```
@RequestMapping("/quick5")
@ResponseBody
public String quickMethod5() throws IOException {
    return "hello springMVC!!!";
}
```

(五) 回写数据-返回对象或集合

1. 常用方式：返回对象，再通过配置好的转化器将对象转化为 Json 字符串

```
//期望SpringMVC自动将User转换成json格式的字符串
public User save10() throws IOException {
    User user = new User();
    user.setUsername("lisi");
    user.setAge(30);

    return user;
}
```

2. 配置自动方式：在 spring-mvc.xml 中配置 mvc 注解驱动

在方法上添加**@ResponseBody**就可以返回json格式的字符串，但是这样配置比较麻烦，配置的代码比较多，因此，我们可以使用mvc的注解驱动代替上述配置。

```
<!--mvc的注解驱动-->
<mvc:annotation-driven/>
```

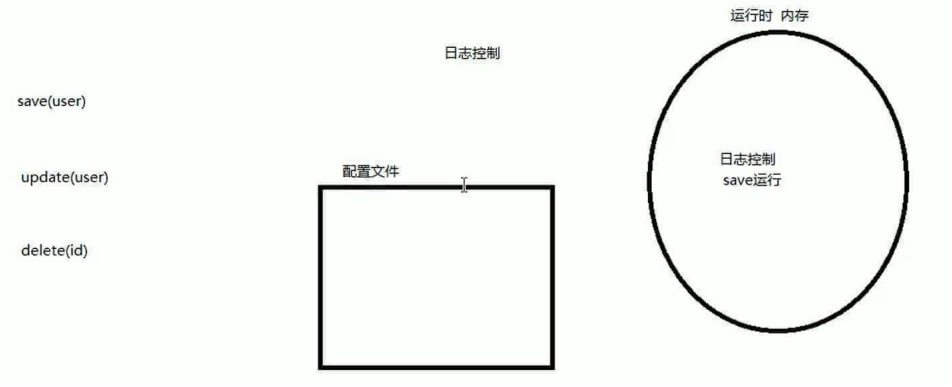
在 SpringMVC 的各个组件中，**处理器映射器**、**处理器适配器**、**视图解析器**称为SpringMVC 的三大组件。
使用<mvc:annotation-driven>自动加载 RequestMappingHandlerMapping（处理映射器）和 RequestMappingHandlerAdapter（处 理 适 配 器 ），可用在Spring-xml.xml配置文件中使用 <mvc:annotation-driven>替代注解处理器和适配器的配置。
同时使用<mvc:annotation-driven>默认底层就会集成jackson进行对象或集合的json格式字符串的转换。

第八章：AOP

一、AOP 的简介

(一) 简介

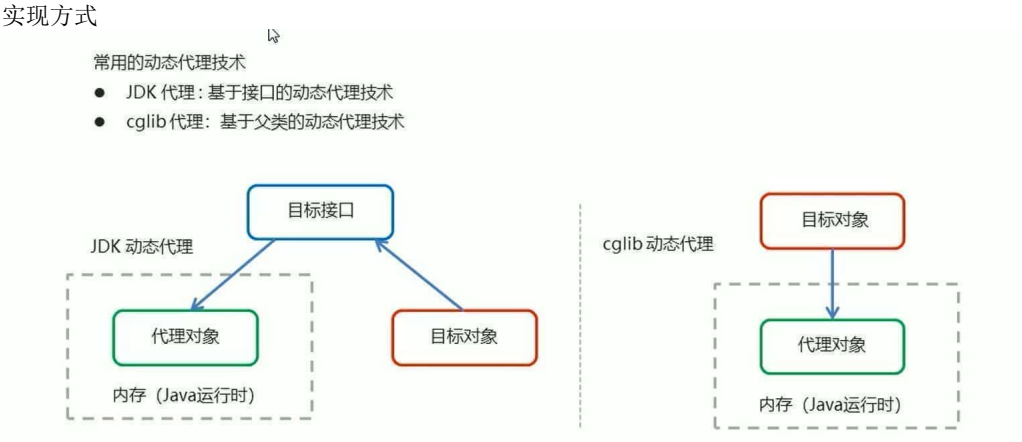
1. 含义：面向切面编程，是通过预编译方式和运行期间动态代理实现程序功能统一维护的一种技术
2. 基本思想：通过配置文件将不同方法结合在一起，实现代码层面的解耦合。其中，目标方法叫做切面，另外的叫做功能增强方法



3. 作用：在程序运行期间，在不修改源码的前提下进行功能增强
4. 优势：减少重复代码，提高开发效率，并且便于维护
5. 相关概念

- Target（目标对象）：代理的目标对象
- Proxy（代理）：一个类被AOP织入增强后，就产生一个结果代理类
- Joinpoint（连接点）：所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法，因为spring只支持方法类型的连接点
- Pointcut（切入点）：所谓切入点是指我们要对哪些Joinpoint进行拦截的定义
- Advice（通知/增强）：所谓通知是指拦截到Joinpoint之后所要做的事情就是通知
- Aspect（切面）：是切入点和通知（引介）的结合
- Weaving（织入）：是指把增强应用到目标对象来创建新的代理对象的过程。spring采用动态代理织入，而AspectJ采用编译期织入和类装载期织入

6. 实现方式



7. 开发时的注意事项

1. 需要编写的内容

- 编写核心业务代码（目标类的目标方法）
- 编写切面类，切面类中有通知(增强功能方法)
- 在配置文件中，配置织入关系，即将哪些通知与哪些连接点进行结合

2. AOP 技术实现的内容

Spring 框架监控切入点方法的执行。一旦监控到切入点方法被运行，使用代理机制，动态创建目标对象的代理对象，根据通知类别，在代理对象的对应位置，将通知对应的功能织入，完成完整的代码逻辑运行。

3. AOP 底层使用哪种代理方式

在 spring 中，框架会根据目标类是否实现了接口来决定采用哪种动态代理的方式。

(二) 基于 jdk 的动态代理

1. 编写目标类和功能增强类
2. 编写动态代理。真正实操时由框架提供代理，不需要编写具体代码

```
public static void main(String[] args) {

    //目标对象
    final Target target = new Target();

    //增强对象
    final Advice advice = new Advice();

    //返回值 就是动态生成的代理对象
    TargetInterface proxy = (TargetInterface) Proxy.newProxyInstance(
            target.getClass().getClassLoader(), //目标对象类加载器
            target.getClass().getInterfaces(), //目标对象相同的接口字节码对象数组
            new InvocationHandler() {
                //调用代理对象的任何方法 实质执行的都是invoke方法
                public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
                    advice.before(); //前置增强
                    Object invoke = method.invoke(target, args);//执行目标方法
                    advice.afterReturning(); //后置增强
                    return invoke;
                }
            }
```

(三) 基于 cglib 的动态代理

1. 编写目标类和功能增强类
2. 编写动态代理

```
//1、创建增强器
Enhancer enhancer = new Enhancer();
//2、设置父类（目标）
enhancer.setSuperclass(Target.class);
//3、设置回调
enhancer.setCallback(new MethodInterceptor() {
    public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws T
        advice.before(); //执行前置
        Object invoke = method.invoke(target, args);//执行目标
        advice.afterReturning(); //执行后置
        return invoke;
    }
});
//4、创建代理对象
Target proxy = (Target) enhancer.create();


proxy.save();
```

二、xml 方式实现 AOP

(一) 基本步骤

1. 导入 AOP 坐标

```xml
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.4</version>
</dependency>
```

2. 创建目标接口和目标类（内部有切点）

```java
public class Target implements TargetInterface {
    public void save() { System.out.println("save running....."); }
}
```

3. 创建切面类（内部有增强方法）

```java
public class MyAspect {

    public void before(){
        System.out.println("前置增强.....");
    }

}
```

4. 使用 Bean 的方式将目标对象和切面对象交给 Spring 管理

```xml
<!--目标对象-->
<bean id="target" class="com.itheima.aop.Target"></bean>

<!--切面对象-->
<bean id="myAspect" class="com.itheima.aop.MyAspect"></bean>
```

5. 配置织入

```xml
<!--配置织入：告诉spring框架 哪些方法(切点)需要进行哪些增强(前置、后置...)-->
<aop:config>
    <!--声明切面-->
    <aop:aspect ref="myAspect">
        <!--切面：切点+通知-->
        <aop:before method="before" pointcut="execution(public void com.itheima.aop.Target.save())"></a
    </aop:aspect>
</aop:config>
```

(二) 切点表达式的写法

1. 作用：用于配置切点

2. 通用语法

表达式语法：

```
execution([修饰符] 返回值类型 包名.类名.方法名(参数))
```

- 访问修饰符可以省略
- 返回值类型、包名、类名、方法名可以使用星号* 代表任意
- 包名与类名之间一个点 . 代表当前包下的类，两个点 .. 表示当前包及其子包下的类
- 参数列表可以使用两个点 .. 表示任意个数，任意类型的参数列表

例如：

```
execution(public void com.itheima.aop.Target.method())
execution(void com.itheima.aop.Target.*(..))
execution(* com.itheima.aop.*.*(..))
execution(* com.itheima.aop..*.*(..))
execution(* *..*.*(..))
```

(三) 通知的写法

通知的配置语法：

```
<aop:通知类型 method=“切面类中方法名” pointcut=“切点表达式”></aop:通知类型>
```

名称	标签	说明
前置通知	<aop:before>	用于配置前置通知。指定增强的方法在切入点方法之前执行
后置通知	<aop:after-returning>	用于配置后置通知。指定增强的方法在切入点方法之后执行
环绕通知	<aop:around>	用于配置环绕通知。指定增强的方法在切入点方法之前和之后都执行
异常抛出通知	<aop:throwing>	用于配置异常抛出通知。指定增强的方法在出现异常时执行
最终通知	<aop:after>	用于配置最终通知。无论增强方式执行是否有异常都会执行

(四) 切点表达式的抽取

1. 含义：当切点表达式一样时，可以进行抽取
2. 方式：用对于表达式 id 的引用代替具体的表达式

当多个增强的切点表达式相同时，可以将切点表达式进行抽取，在增强中使用 pointcut-ref 属性代替 pointcut 属性来引用抽取后的切点表达式。

```xml
<aop:config>
    <!--引用myAspect的Bean为切面对象-->
    <aop:aspect ref="myAspect">
        <aop:pointcut id="myPointcut" expression="execution(* com.itheima.aop.*.*(..))"/>
        <aop:before method="before" pointcut-ref="myPointcut"></aop:before>
    </aop:aspect>
</aop:config>
```

三、注解方式实现 AOP

(一) 基本步骤

1. 导入 AOP 坐标
2. 创建目标接口和目标类（内部有切点）
3. 创建切面类（内部有增强方法）
4. 将两种类交给 Spring 容器

```java
@Component("target")
public class Target implements TargetInterface {
    public void save() {
        System.out.println("save running.....");
        //int i = 1/0;
    }
}
```

5. 使用注解指定切面，配置通知与切点表达式

```java
@Component("myAspect")
@Aspect //标注当前MyAspect是一个切面类
public class MyAspect {

    //配置前置通知
    @Before("execution(* com.itheima.anno.*.*(..))")
    public void before() { System.out.println("前置增强......"); }

    public void afterReturning() { System.out.println("后置增强....."); }

    //Proceeding JoinPoint: 正在执行的连接点===切点
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("环绕前增强....");
        Object proceed = pjp.proceed();//切点方法
        System.out.println("环绕后增强....");
        return proceed;
    }
```

6. 配置组件扫描和 aop 自动代理

```xml
<!--组件扫描-->
<context:component-scan base-package="com.itheima.anno"/>

<!--aop自动代理-->
<aop:aspectj-autoproxy/>
```

(二) 通知的写法

通知的配置语法：@通知注解("切点表达式")

名称	注解	说明
前置通知	@Before	用于配置前置通知。指定增强的方法在切入点方法之前执行
后置通知	@AfterReturning	用于配置后置通知。指定增强的方法在切入点方法之后执行
环绕通知	@Around	用于配置环绕通知。指定增强的方法在切入点方法之前和之后都执行
异常抛出通知	@AfterThrowing	用于配置异常抛出通知。指定增强的方法在出现异常时执行
最终通知	@After	用于配置最终通知。无论增强方式执行是否有异常都会执行

(三) 切点表达式的抽取

同 xml 配置 aop 一样，我们可以将切点表达式抽取。抽取方式是在切面内定义方法，在该方法上使用@Pointcut 注解定义切点表达式，然后在在增强注解中进行引用。具体如下：

```java
@@Component("myAspect")
@Aspect
public class MyAspect {
    @Before("MyAspect.myPoint()")
    public void before(){
        System.out.println("前置代码增强.....");
    }
    @Pointcut("execution(* com.itheima.aop.*.*(..))")
    public void myPoint(){}
}
```