Spring web开发的三大容器

所谓容器就是管理对象的地方，博客链接

https://www.cnblogs.com/liujia1990/p/9024884.html

spring容器

管理service和dao的容器，因此在spring的配置文件中不扫描带有`Controller`注解的类，对应`applicationContext.xml`文件

springMVC容器

管理controller的容器，因此在springMVC的配置文件中只扫描带有`Controller`注解的类，并且springMVC的拦截器也是由其管理的，如：

```
<mvc:interceptors>
  <mvc:interceptor>
   <mvc:mapping path="/employee/**" ></mvc:mapping>
   <bean class="com.smart.core.shiro.LoginInterceptor" ></bean>
  </mvc:interceptor>
</mvc:interceptors>
```

对应`springMVC.xml`文件

> spring容器是springMVC的父容器，后者可以访问前者的bean，前者不可以访问后者，因此在controller中访问service时需要将其注入springMVC容器

web容器

管理servlet、Listener、Filter的容器，他们都是在web容器的掌控范围内，但是不在以上两个容器的掌控范围，因此无法在这些类中直接使用spring注解的方式注入需要的对象，容器无法识别。

但我们有时候又确实会有这样的需求，比如在容器启动的时候，做一些验证或者初始化操作，这时可能会在监听器里用到bean对象；又或者需要定义一个过滤器做一些拦截操作，也可能会用到bean对象。但是前提是：servelt容器在实例化监听器或过滤器对象时，要确保spring容器已经初始化完成。

而spring容器的初始化也是由Listener（ContextLoaderListener）完成，因此只需在web.xml中先配置初始化spring容器的Listener（见ssm整合的第五项），然后在配置自己的Listener。

1 Spring

> Spring简化了开发，它整合了现有的技术，Spring是为解决软件开发的复杂性而创建的，使用简单的JavaBean替换了EJB，提供了企业应用功能，具有简单性、松耦合性、可测试性等许多优点，可以用它开发任何应用。
>
> 为了降低 Java开发的复杂性，Spring采取以下四种关键策略：
>
> - 基于POJO的轻量级和最小侵入式编程
> - 通过依赖注入和面向接口实现松耦合
> - 基于切面和惯例进行声明式编程
> - 通过切面和模板减少样板式代码

> 一言蔽之：Spring是一个轻量级的控制反转(IOC)和面向切面(AOP)编程的容器框架

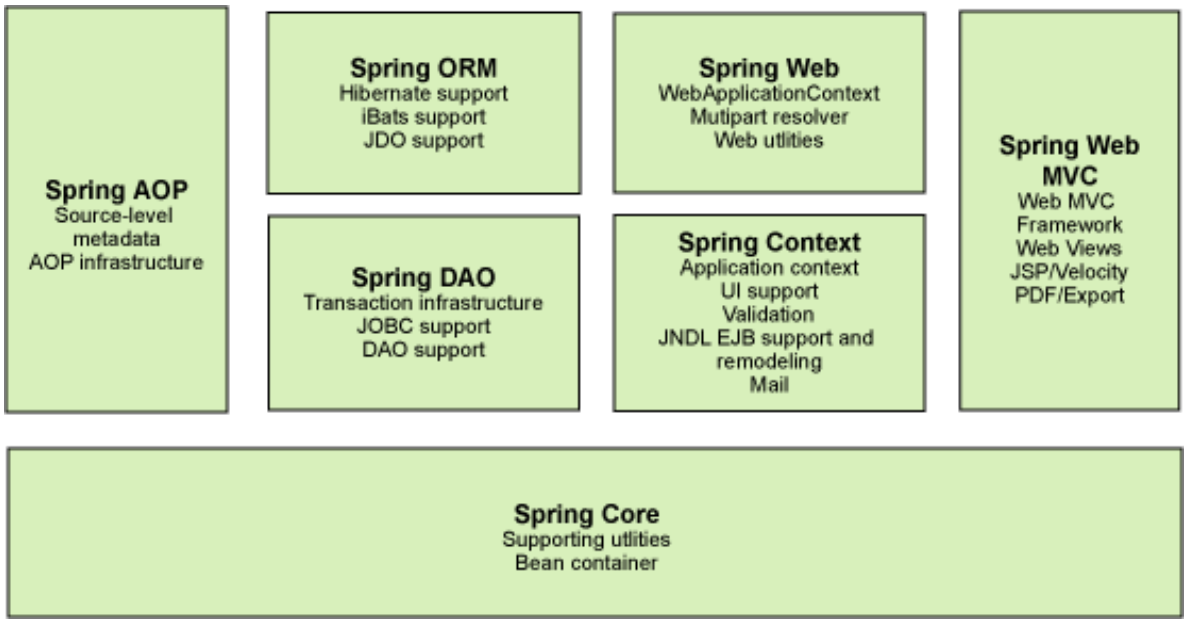
1.1 Spring的历史

2002年，interface21框架推出，Spring框架以该框架为原型，经过重新设计，并不断丰富其内涵，于2004年3月24日发布1.0版本，Rod Johnson 是Spring框架的创始人

1.2 Spring的优点

- 免费的开源框架
- 轻量级、非侵入式的框架
- 支持事务处理，支持框架整合
- 控制反转和面向切面编程

1.3 Spring的组成



2 Spring的配置

```xml
<?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:p="http://www.springframework.org/schema/p"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd ">

</beans>
```

2.1Spring配置文件标签

2.1.1 <beans>

这个标签是配置文件的根标签，其他标签都在这个标签中书写

其子标签有：

1. bean
2. alias
3. beans
4. import
5. description

2.1.2 <bean>

向IOC容器注入一个POJO

```
<bean id="user" class="com.sundehui.domain.User"/>
```

其主要子标签为：

1. property: 为POJO的域设置初始值，但是只有该域有Setter方法才可以，否则报错
2. constructor-arg: 为POJO的构造方法传值，其方法见使用有参构造的方法

其属性有：

1. id: 容器中对象的标识符
2. class：容器对象类型的全限定类名
3. name: 取别名，可以用空格分隔取多个别名
4. scope: 配置bean作用域，它的值有：
    - singleton/true: 单例模式，默认
    - prototype/false: 原型模式
    - request: 针对每一个http请求，创建一个新bean，同时该bean仅在当前HTTP request内有效
    - session：针对每一次HTTP请求都会产生一个新的bean，同时该bean仅在当前HTTP session内有效
    - application:
    - websocket

    > request,session,application,websocket这三个作用域需要再web.xml做额外的配置

    ```
    <!--Servlet 2.4及以上的web容器-->

    <web-app>
    ...
    <listener>
       <listener-class>

    org.springframework.web.context.request.RequestContextListener
       </listener-class>
    </listener>
    ```

```
...
</web-app>

<!--Servlet2.4以前的web容器,那么你要使用一个javax.servlet.Filter的实现-->
<web-app>
..
<filter>
 <filter-name>requestContextFilter</filter-name>
 <filter-class>
     org.springframework.web.filter.RequestContextFilter
  </filter-class>
</filter>

<filter-mapping>
 <filter-name>requestContextFilter</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
...
</web-app>
```

- 自定义scope, spring的作用域由接口org.springframework.beans.factory.config.Scope来定义，自定义自己的作用域只要实现该接口即可

2.1.3 <alias>

为bean起别名，如

```
<bean id="user" class="com.sundehui.domain.User" name="别名1,别名2,..."/>
<alias name="user" alias="sundehui"/> <!--可以使用sundehui作为id获取容器中的对象-->
```

bean 的属性`name`也可以用于起别名，可以取多个别名，所以alias没卵用

2.1.4 <import>

一般用于多个团队开发, 它可以将其他配置文件导入:

```
<import resource="beans1.xml"/>
<import resource="beans2.xml"/>
<import resource="beans3.xml"/>
```

2.2 用Java代码进行配置

可以完全不用xml文件做配置，只用java!

JavaConfig是Spring的一个子项目，在Spring4之后，它称为了Spring的核心功能，并且为官方推荐来配置Spring。

```
import com.sundehui.domain.User;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
```

```java
import org.springframework.context.annotation.Import;
@Configuration//表明这个类是一个配置类
@ComponentScan("com.sundehui")//开启包扫描
@Import(Config2.class)//导入另外一个配置类
public class AppConfig {

    //相当于一个bean标签，beanId为方法名，返回值包含class属性
    @Bean
    public User getUser(){
        return new User();
    }
}

```

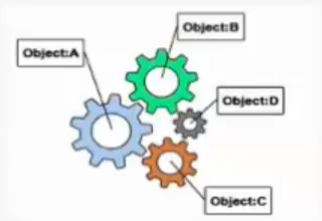
3 IOC

IOC是Spring的核心，Spring使用了多种方式实现了IOC，可以用XML配置文件，可以用注解，也可以零配置。Spring容器在初始化时先读取配置文件，根据配置文件或元数据创建与组织对象存入IOC容器，使用时再从容器中取出需要的对象。即对象的创建、管理、装配交给了Spring。

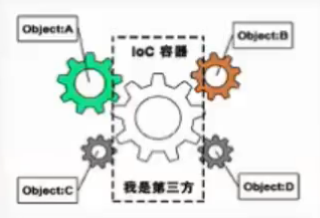
3.1 控制反转

对程序的控制权从程序员转为了调用者，即**获得依赖对象的方式反转**了。

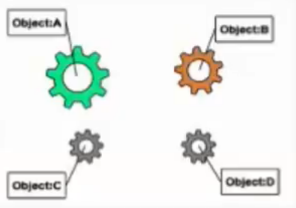
> 在没有IOC的程序中，对象的创建硬编码在程序中，对象的创建由该程序自己进行，控制反转后将对象的创建转移给第三方。如下图，ioc的存在实现了图2的解耦过程。



• 图1：耦合的对象



• 图2：解耦的过程



• 图3：理想的系统

3.2 DI（依赖注入）

依赖注入是IOC的实现方式

3.2.1 什么是依赖注入?

3.2.1.1 依赖

bean对象的创建依赖于容器

3.2.1.2 注入

bean对象的域由容器注入

注入方式有三种

1. 构造器注入
2. **setter注入**

```java
public class Student {
 private String name;
 private User user;
 private String[] books;
 private List<String> hobbys;
 private Map<String,String> card;
 private Set<String> games;
 private Properties info;
 private String couple;

 setters...;
 getters...;
}
```

```xml
<bean id="user" class="com.sundehui.domain.User"/>
<bean id="student" class="com.sundehui.domain.Student">
<!-- 普通值注入， value-->
 <property name="name" value="孙德辉"/>
<!-- bean注入，ref-->
 <property name="user" ref="user"/>
<!-- 数组注入-->
 <property name="books">
     <array>
         <value>《鲁宾孙漂流记》</value>
         <value>《钢铁是怎样炼成的》</value>
         <value>《呐喊》</value>
     </array>
 </property>
<!-- list注入-->
 <property name="hobbys">
     <list>
         <value>看电视</value>
         <value>看书</value>
         <value>打游戏</value>
     </list>
 </property>
<!-- Map注入-->
 <property name="card">
     <map>
         <entry key="idCard" value="13432123323432123432"/>
         <entry key="moneyCard" value="13432123323432123432"/>
     </map>
 </property>
<!-- Set注入-->
 <property name="games">
     <set>
         <value>王者荣耀</value>
         <value>cod</value>
     </set>
 </property>
<!-- 空指针注入-->
 <property name="couple">
```

```
        <null/>
    </property>
<!-- properties注入-->
    <property name="info">
        <props>
            <prop key="学号">1611650716</prop>
            <prop key="性别">male</prop>
        </props>
    </property>
</bean>
```

3. 拓展注入

```
xmlns:p="http://www.springframework.org/schema/p"<!--p明明空间,property-->
xmlns:c="http://www.springframework.org/schema/c"<!--c命名空间,constructor-arg-->
```

3.2.2 使用哪种方式注入?

it is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies.Note that use of the @Required annotation on a setter method can be used to make the property be a required dependency; however, **constructor injection with programmatic validation** of arguments is preferable.

3.2.3 循环依赖

https://blog.csdn.net/qq_36381855/article/details/79752689

3.3 IOC创建对象的方式

1. 使用无参构造方法（默认）。
2. 使用有参构造方法:

```
<bean id="user" claass="com.sundehui.domain.User">
    <!--第一种方法，使用参数列表的下标-->
    <constructor-arg index="0" value="sundehui"/>
    <!--第二种方法，使用参数列表的类型，不建议使用-->
    <constructor-arg type="java.lang.String" value="sundehui"/>
    <!--第二种方法，使用参数列表的形参名-->
    <constructor-arg type="name" value="sundehui"/>
</bean>
```

3.4 IOC创建对象的时机

配置文件被加载后就创建了对象，从容器中取出的对象都是同一个，即应用了单例模式。

4 bean的自动装配

- 自动装配是Spring满足bean依赖的一种方式
- Spring会在上下文中自动寻找，并自动给bean装配属性

4.1 装配的三种方式

4.1.1 xml中显式配置

4.1.2 Java代码中显式配置

4.1.3 隐式自动装配

基于xml的自动装配

- byType

```xml
<bean class="com.kuang.pojo.Cat"/>
<bean class="com.kuang.pojo.Dog"/>

<!--
    byName: 会自动在容器上下文中查找，和自己对象set方法后面的值对应的 beanid!
    byType: 会自动在容器上下文中查找，和自己对象属性类型相同的bean!
    -->
<bean id="people" class="com.kuang.pojo.People" autowire="byType">
    <property name="name" value="小狂神呀"/>
</bean>
```

- byName

```xml
<!--
    byName: 会自动在容器上下文中查找，和自己对象set方法后面的值对应的 beanid!
    -->
<bean id="people" class="com.kuang.pojo.People" autowire="byName">
    <property name="name" value="小狂神呀"/>
</bean>
```

基于注解的自动装配

1. 导入约束

`xmlns:context="http://www.springframework.org/schema/context"`

```
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd"
```

2. 配置注解的支持

`<context:annotation-config/>`

3. 注解
    - **@Autowired**：一般放在域或方法上

      它的作用是，自动把**在IOC容器中的对象**给当前类的域装配，要求当前类的域名和IOC的对象id一样，是byName的，它要求域不为空，如果可以为空，可将其required属性设置为false
    - **@Qualifier(value="beanId")**：和@Autowired组合使用，显示指出为注解的域装配那个对象

- **@Resource**:Java的原生注解，它默认用byName，找不到名字用byType方式装配bean，也可以用name属性指定bean

5 Spring的注解开发

- 装配注解见上文
- @Component：放在类上，说明说明这个类被Spring管理了
- @Value：为被@Compent注解的类装配属性值，放在域或setter上
- @Component 的衍生注解
  - @Repository --- dao层
  - @Service --- service层
  - @Controller --- controller层
- @Scope：配置bean的作用域

6 代理模式

即真实角色委托代理角色去做真实角色想做的事

好处:

1. 使真实角色的操作更加纯粹，不用去关注一些公共业务
2. 公共业务交给代理角色，实现了业务的分工
3. 公共业务发生拓展时，方便集中管理

6.1 角色分析

- 抽象角色：一般用接口
- 真实角色：被代理的角色
- 代理角色：代理真实角色，它一般做一些增强操作
- 客户角色：访问代理对象的角色

6.2 静态代理

以租房这一事件举例：

1. 抽象角色，为真实角色和代理角色提供公共方法

```
public interface AbstraceRole {
     void rent ();
}
```

2. 真实角色，实现了抽象角色

```
public class RealRole implements AbstraceRole {
    public void rent() {
        System.out.println("我是房东，我要出租房子");
    }
}
```

3. 代理角色，实现了抽象角色

```
public class ProxyRole implements AbstraceRole {
    //代理角色以组合的方式代理真实角色
    private RealRole realRole;

    public ProxyRole(RealRole realRole) {
        this.realRole = realRole;
    }
    //实现真实角色功能
    public void rent (){
        realRole.rent();
    }
    public void fare(){
        sout("收租房费");
    }
}
```

4. //客户角色，从访问真实角色转为访问代理角色

```
public class ClientRole {
    public static void main(String[] args) {
        RealRole realRole = new RealRole();
        ProxyRole proxyRole = new ProxyRole(realRole);
        proxyRole.rent();
    }
}
```

静态代理的缺点：

1. 一个真实角色就需要为其创建一个代理角色，代码量较大

6.3 动态代理

- 动态代理的角色划分和静态代理一样
- 动态代理的代理类时动态生成的，不是直接写好的
- 动态代理分为两大类：基于接口的动态代理，基于类的动态代理
    - 基于接口：JDK动态代理 (重点)
    - 基于类：cglib
    - java字节码实现：Javasist

`Proxy`和`InvocationHandler`时JDK动态代理的核心类和接口

> **Proxy** 类提供了创建动态代理类和实例的静态方法，也是其所创建的代理类的超类
>
> **InvocationHandler** 是由代理实例的**调用处理程序**实现的接口
>
> > 每个代理实例都有一个与之关联的**调用处理程序**，当在代理类上调用方法时，该**方法调用**将被编码并分派到其调用处理程序的invoke()方法

1. 抽象角色: 同静态代理
2. 真实角色：同静态代理
3. 调用处理程序

```java
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class ProxyInvocationHandler<T> implements InvocationHandler {
    private T target;

    public void setTarget(T target) {
        this.target = target;
    }
    //获取代理类，代理类实现了真实角色实现的接口
    public T getProxyObj(){
        return (T)Proxy.newProxyInstance(this.getClass().
getClassLoader(),

target.getClass().getInterfaces(),
                                         this);
    }
    //当在代理类上调用方法时，该方法调用将被编码并分派到其调用处理程序的invoke()方法
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        Object invoke = method.invoke(target, args);
        return invoke;
    }
}
```

4. 客户角色

```java
public class Client {
    public static void main(String[] args) {
        //创建真实角色
        RealRole realRole = new RealRole();
        //创建调用处理程序
        ProxyInvocationHandler<RealRole> pih = new
ProxyInvocationHandler();
        //注入真实角色对象到调用处理程序
        pih.setTarget(realRole);
        //获取代理对象
        AbstraceRole proxyObj =  pih.getProxyObj();
        //调用代理对象方法
        proxyObj.rent();
    }
}
```

动态代理的好处：

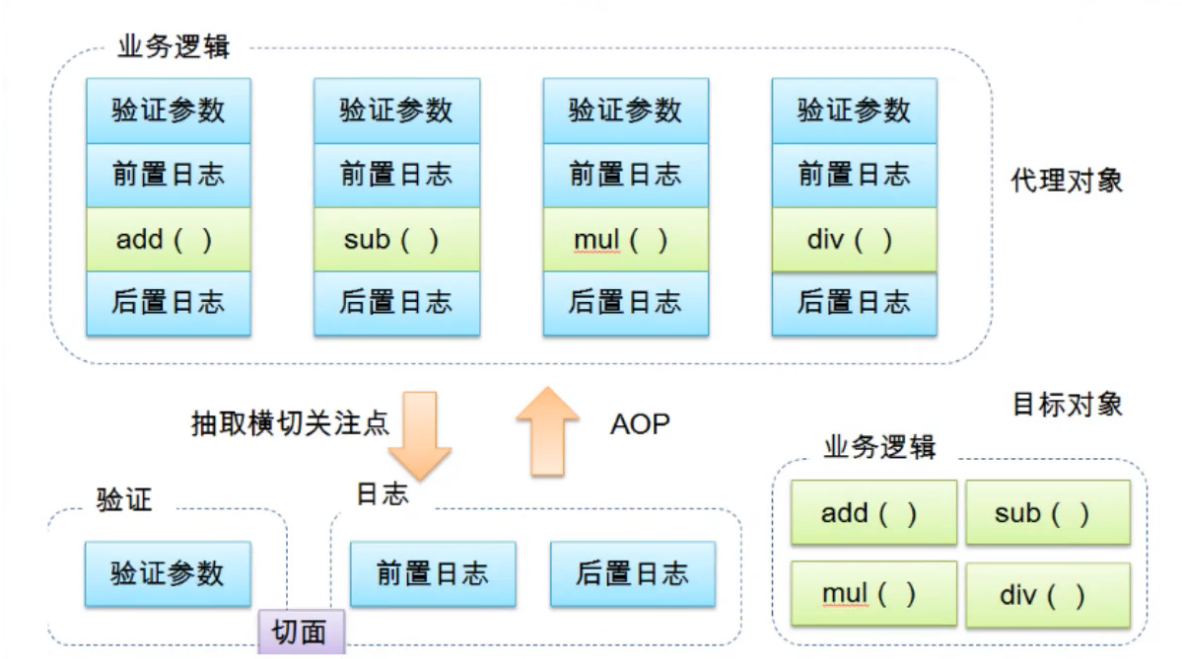
1. 一个动态代理代理的是一个接口，一般对应一类业务
2. 一个动态代理类代理了多个类，只要这些类实现了同一个接口就可以
3. 代码量相对于静态代理大大减少

7 AOP

在不改变源代码的情况下进行增强.

7.1 什么是AOP?

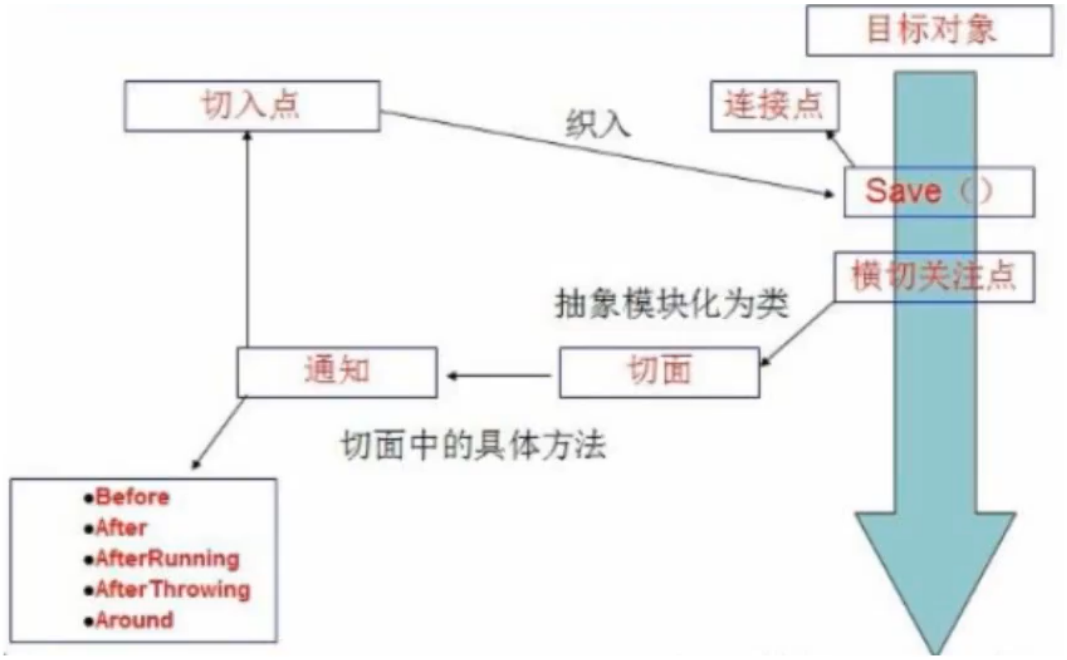
AOP（Aspect Oriented Programming）意为面向切面编程，通过**预编译**和**运行期动态代理**实现程序功能的统一维护的一种技术。它是OOP的延续，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使业务逻辑各部分之间的耦合度降低，提高程序可用性，提高开发效率。



7.2 AOP在Spring中的应用

提供声明式事务，允许用户自定义切面

- 横切关注点：跨越应用程序多个模块的方法或功能。即，与我们的业务无关的，但我们需要关注的部分，就是横切关注点。如日志、安全、缓存、事务……
- 切面 (Aspect)：横切关注点被模块化的特殊对象，即一个类
- 通知 (Advice)：切面必须要完成的工作，即类的一个方法
- 目标 (Target)：被通知的对象
- 代理 (Proxy)：向目标对象应用通知之后创建的对象
- 切入点 (CutPoint)：切面通知执行的”地点“的定义
- 连接点 (JoinPoint)：与切入点匹配的执行点



Spring支持五种类型的Advice，这些都是接口，是Spring的原生api

通知类型	连接点	实现接口
前置通知	方法方法前	org.springframework.aop.MethodBeforeAdvice
后置通知	方法后	org.springframework.aop.AfterReturningAdvice
环绕通知	方法前后	org.aopalliance.intercept.MethodInterceptor
异常抛出通知	方法抛出异常	org.springframework.aop.ThrowsAdvice
引介通知	类中增加新的方法属性	org.springframework.aop.IntroductionInterceptor

7.3 Spring中实现AOP

首先导入包：

```
<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.5</version>
</dependency>
```

方式1：使用Spring的接口

前置通知：`MethodBeforeAdvice`

```java
/*method: 被代理的对象的方法对象
* args: 被代理的接口的方法的参数列表
* target: 被代理的对象
*/
void before(Method method, Object[] args, @Nullable Object target) throws Throwable;
```

后置通知：`AfterReturningAdvice`

```java
/* returnValue：被代理对象的方法的返回值
*  method：同上
*  agrs：同上
*  target：被代理的对象
*/
void afterReturning(@Nullable Object returnValue,
                    Method method,
                    Object[] args,
                    @Nullable Object target) throws Throwable;
```

配置文件需要增加的内容

```xml
<!--注册bean-->
<bean id="userService" class="com.sundehui.service.UserServiceImpl"/>
<bean id="logAfter" class="com.sundehui.aop.LogAfter"/>
<bean id="logBefore" class="com.sundehui.aop.LogBefore"/>
<!--配置aop-->
<aop:config>
  <!--使用expression表达式指定切入点-->
  <!--execution(修饰符  返回值  包名.类名/接口名.方法名(参数列表)), * 表示任意，..表示任意参数列    表-->
  <aop:pointcut id="pointCUt"
        expression="execution(* * com.sundehui.service.UserServiceImpl.*(..))"/>
   <!--执行环绕增加-->
  <aop:advisor advice-ref="logBefore" pointcut-ref="pointCUt"/>
  <aop:advisor advice-ref="logAfter" pointcut-ref="pointCUt"/>
</aop:config>
```

方式2：自定义类实现AOP

1. 自定义一个切面类

```java
public class MyAop {
    public void before (){
        System.out.println("==========方法执行前=============");
    }

    public void after (){
        System.out.println("==========方法执行后=============");
    }
}
```

2. 在配置文件中配置aop

```xml
<!--引入自定义切面类-->
<bean id="myAop" class="com.sundehui.aop.MyAop"/>
<aop:config>
  <!--将aop的切面引用指向自定义切面类-->
  <aop:aspect ref="myAop">
     <!--设置切入点-->
    <aop:pointcut id="pointCut"
         expression="execution(* com.sundehui.service.UserServiceImpl.*(..))"/>
    <!--方法执行前通知-->
    <aop:before method="before" pointcut-ref="pointCut"/>
    <!--方法执行后通知-->
    <aop:after method="after" pointcut-ref="pointCut"/>
  </aop:aspect>
</aop:config>
```

方式3: 注解实现aop

1. 自定义一个切面类

```java
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect//标注该类是一个切面
public class MyAop {
    //execution表达式定义切入点
    @Before("execution(* com.sundehui.service.UserServiceImpl.*(..))")
    public void before (){
        System.out.println("=========方法执行前=============");
    }

    @After("execution(* com.sundehui.service.UserServiceImpl.*(..))")
    public void after (){
        System.out.println("=========方法执行后=============");
    }
}
```

2. 在配置文件中添加

```xml
<!--将自定义切面类给Spring托管-->
<bean id="myAop" class="com.sundehui.aop.MyAop"/>
<!--开启aop的注解支持,false表示使用JDK的动态代理,true表示cglib-->
<aop:aspectj-autoproxy proxy-target-class="false"/>
```

8 整合mybatis

8.1 引入jar包

```
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.2.1.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.2.1.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.4.6</version>
        </dependency>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>2.0.3</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.9.5</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.47</version>
        </dependency>
```

共同配置(配置文件均在spring的配置文件中书写)

```
 <!--引入了外部配置文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>
    <!--    Spring整合mybatis框架-->
    <!--1，配置连接池-->
    <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>
    <!--2，配置SqlSessionFactory工厂-->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <!--在idea中,这里必须为树状目录,点分目录识别不到-->
```

```
        <property name="mapperLocations"
value="classpath:com/sundehui/dao/*Dao.xml"/>
        <property name="configLocation"
value="classpath:mybatisConfig.xml"/>
    </bean>
```

方式一: 自动扫描,创建代理对象

```
  <bean id="mapperScanner"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
       <property name="basePackage" value="com.sundehui.dao"/>
      <!--如果只有一个数据源,此处不用配置-->
<!--   <property name="sqlSessionFactoryBeanName"
value="sqlSessionFactory"/>-->
    </bean>
```

方式二: 实现dao接口,向实现类注入SqlSessionTemplate对象

1. 配置文件

```
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
        <constructor-arg index="0" ref="sqlSessionFactory"/>
</bean>
```

2. 实现dao接口

```
import com.sundehui.domain.Account;
import org.mybatis.spring.SqlSessionTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import java.util.List;

public class IAccountDaoImpl implements IAccountDao {

    @Autowired
    private SqlSessionTemplate sqlSession;

    @Override
    public List<Account> findAll() {
        return sqlSession.getMapper(IAccountDao.class).findAll();
    }
}

```

3. 将实现类注入容器

```
<bean id="accountDaoImpl" class="com.sundehui.dao.IAccountDaoImpl"/>
```

方式三: 继承SqlSessionDaoSupport类并实现dao接口

`SqlSessionDaoSupport` 类已经包 `SqlSession` 对象封装好了,其子类只需要通过 `getSqlSession()` 方法就可以获得 `SqlSession` 对象.

1. 继承SqlSessionDaoSupport类并实现dao接口

```
public class IAccountDaoImpl extends SqlSessionDaoSupport implements
IAccountDao {
    @Override
    public List<Account> findAll() {
        return getSqlSession().getMapper(IAccountDao.class).findAll();
    }
}
```

2. 配置文件

```
 <bean id="accountDaoImpl" class="com.sundehui.dao.IAccountDaoImpl">
        <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>
```

9 声明式事务

声明式事务:不改变源代码,使用AOP进行事务管理

编程式事务:在源代码中进行事务管理

依赖:

```
<dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.2.1.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>5.2.1.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.2.1.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.47</version>
        </dependency>
        <dependency>
```

```
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.2</version>
        </dependency>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>com.mchange</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.5.2</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.12</version>
            <scope>provided</scope>
        </dependency>
```

9.1 事务

事务涉及到数据一致性问题, 要么都成功要么都失败

9.2 配置声明式事务

方法一: 标准配置

```
    <!--开启事务-->
    <bean id="txManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--结合aop实现事务织入-->
    <!--配置事务通知-->
    <tx:advice id="txAdvice" transaction-manager="txManager">
    <!--给那些方法配置事务， propagation 事务的传播特性-->
        <tx:attributes>
            <tx:method name="*" propagation="REQUIRED"/>
        </tx:attributes>
    </tx:advice>

    <!--配置事务切入-->
    <aop:config>
        <aop:pointcut id="txPointCut"
```

```
                        expression="execution(* com.sundehui.dao.*.*(..))"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut"/>
    </aop:config>
```