1、Spring

1.1、简介

- Spring是一个轻量级控制反转(IoC)和面向切面(AOP)的容器框架。
- 2002年，首次推出了Spring框架的雏形：interface21框架。https://www.interface21.io/
- *Spring*框架即以interface21框架为基础,经过重新设计,并不断丰富其内涵,于*2004年3月24日*,发布了1.0正式版。
- Spring理念：使现有的技术更加容易使用，本身是一个大杂烩，整合了现有的技术框架
- 官网：https://spring.io/projects/spring-framework#overview
- 官方下载地址：https://repo.spring.io/release/org/springframework/spring/
- GitHub：https://github.com/spring-projects/spring-framework
- Maven

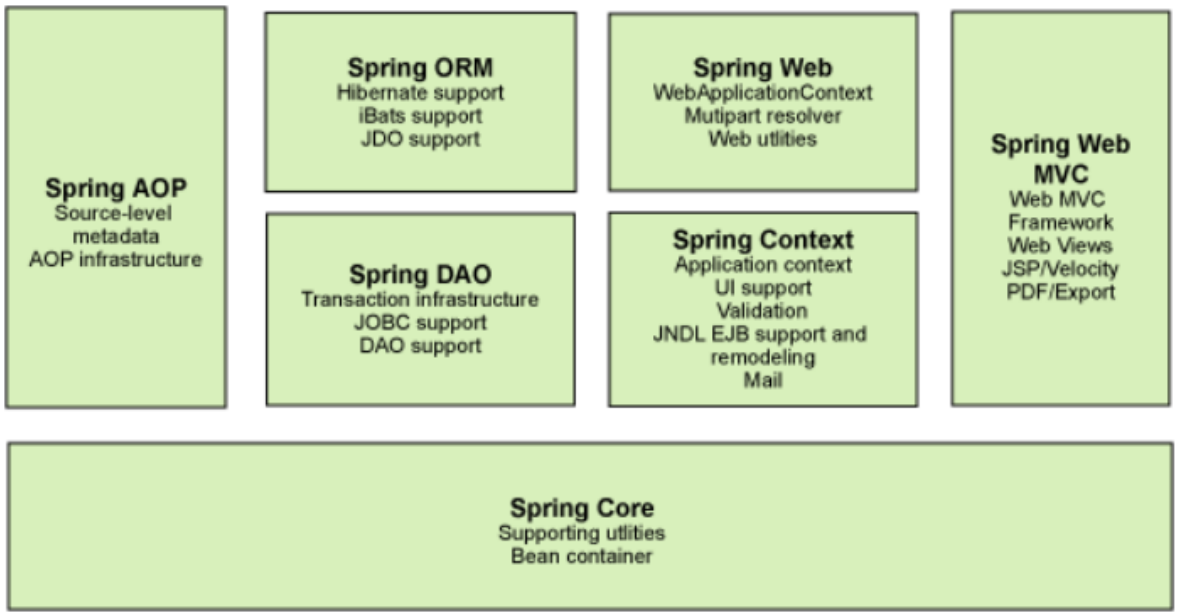
```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
```

1.2、优点

- Spring是一个开源的免费的框架(容器)
- Spring是一个轻量级的，非入侵式的框架
- 控制反转(IOC)，面向切面编程(AOP)
- 支持事务的处理，对框架整合的支持

Spring就是一个轻量级的控制反转（IOC）和面向切面（AOP）编程的框架

1.3、组成



2、IOC理论推导

1. UserDao接口

```java
public interface UserDao {
    void getUser();
}
```

2. UserDaoImpl实现类

```java
public class UserDaoImpl implements UserDao {
    public void getUser() {
        System.out.println("默认获取用户的数据");
    }
}
```

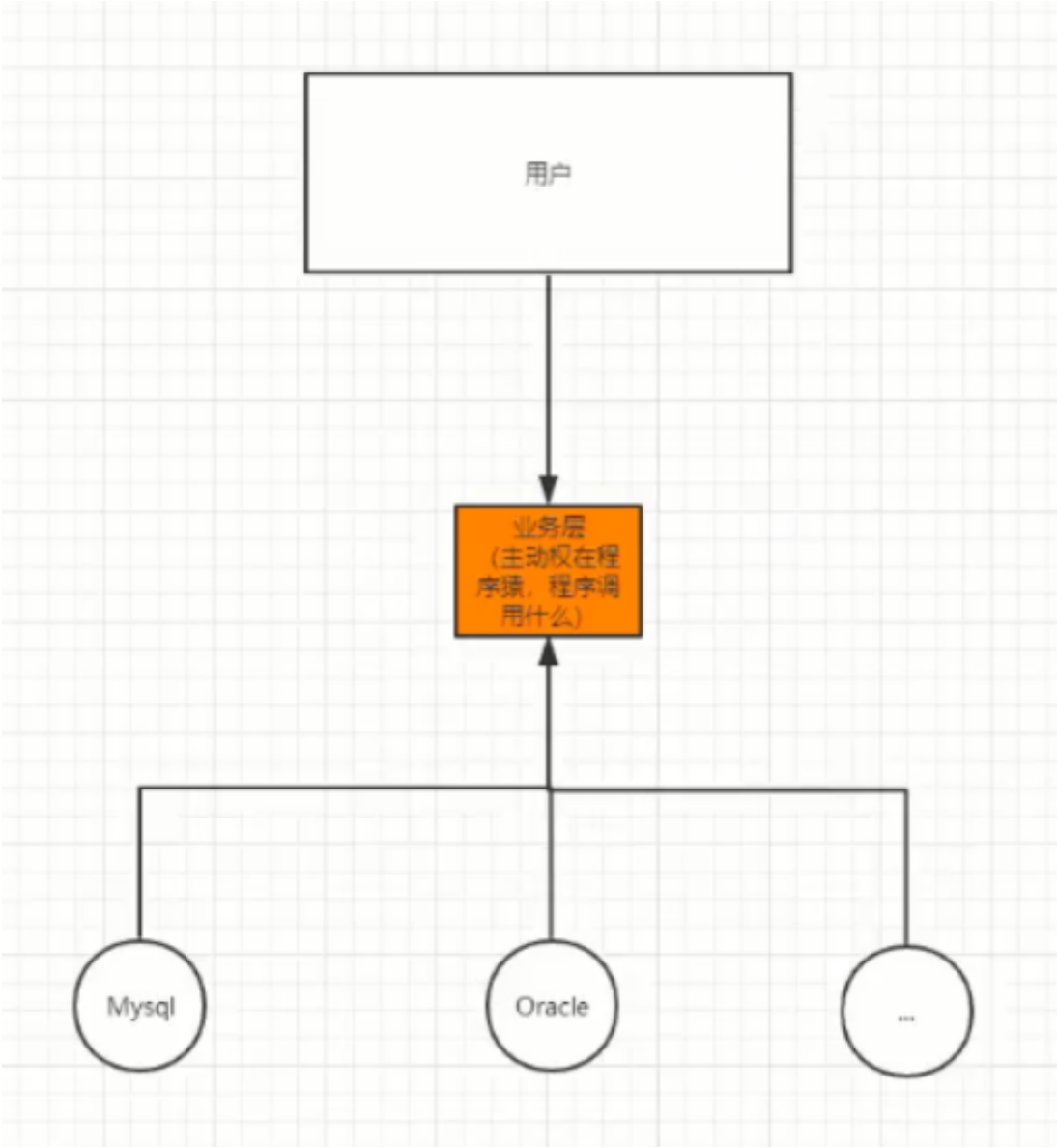
3. UserService业务接口

```java
public interface UserService {
    void getUser();
}
```

4. UserServiceImpl业务实现类

```java
public class UserServiceImpl implements UserService {
    private UserDao dao = new UserDaoImpl();
    public void getUser() {
        dao.getUser();
    }
}
```

在之前的业务中，用户的需求，可能会影响原来的代码，要根据需求去修改原来的代码。如果代码量大，修改一次特别麻烦

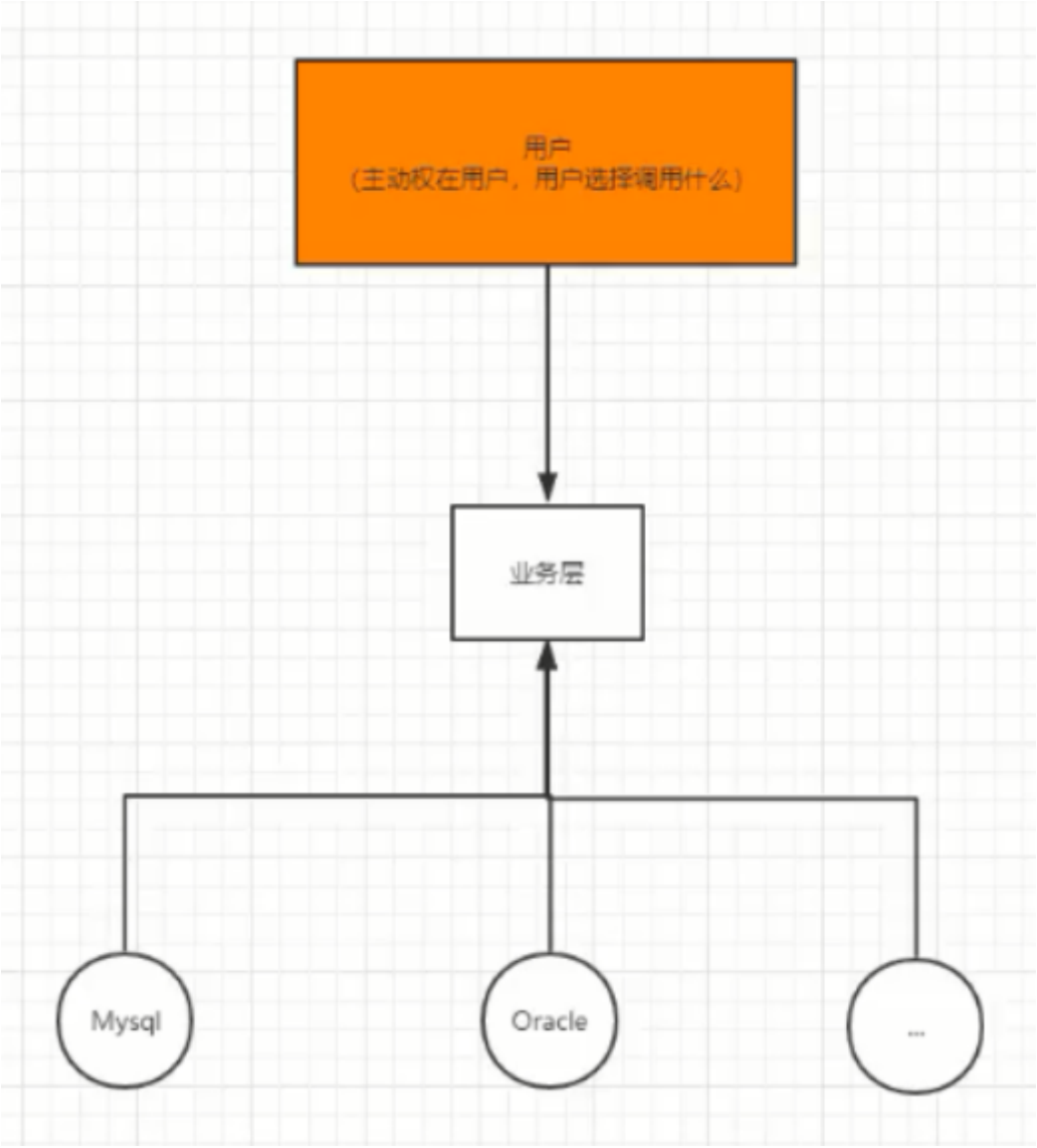


使用Set接口实现，已经发生了控制反转

```
private UserDao dao;

//利用set实现进行值得注入
public void setDao(UserDao dao) {
    this.dao = dao;
}
```

- 之前时程序主动创建对象，控制权在程序手上
- 使用set注入后，程序不再具有主动性，而是变成被动接受对象

这种思想，从本质上解决了问题，程序员不用在去管理对象的创建，降低了程序的耦合性，可以专注在业务的实现，这就是IOC的原型



IOC本质

控制反转(IOC)，是一种设计思想，DI(依赖注入)是实现IOC的一种方法，也有人认为DI只是IOC的另一种说法，没有IOC的程序中，我们使用面向对象编程，对象的创建与对象间的依赖关系完全硬编码在程序中，对象的创建由程序自己控制，控制反转后将对象的创建移交给第三方。控制反转就是：获得依赖对象的方式反转了。

采用XML方式配置Bean的时候，Bean的定义信息是和实现分离的，而采用注解的方式可以把两者合为一体，Bean的定义信息直接以注解的形式定义在实现类中，从而达到了零配置的目的

控制反转是一种通过描述（XML或注解）并通过第三方去生产或获取特定对象的方式。在spring中实现控制反转的时IOC容器，其实实现方法是依赖注入DI

3、HelloSpring

- 导入依赖包

```xml
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```

- 编写实体类

```java
public class Hello {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Hello{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

- 编写核心配置文件applicationContext.xml

```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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--  使用spring来创建对象，这些都称为bean
        bean = 对象
        id = 变量名
        class = 类的路径
        property 给对象中的属性赋值
        ref : 引用spring容器中创建好的对象
        value : 具体的值
    -->
    <bean id="hello" class="com.zh.pojo.Hello">
        <property name="name" value="spring"></property>
     </bean>

</beans>
```

- 测试

```java
import com.zh.pojo.Hello;
```

```java
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {

    public static void main(String[] args) {

        //获取spring的上下文对象
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

        //对象由spring来创建，管理，分配
        Hello hello = (Hello) context.getBean("hello");

        System.out.println(hello.toString());
    }
}
```

4、IOC创建对象的方式

1. 使用无参构造创建对象，默认

```xml
<!--  使用无参构造创建对象，默认
      对象必须有无参构造，否则无法初始化
-->
<bean id="user" class="com.zh.pojo.User">
    <property name="name" value="张恒"></property>
</bean>
```

2. 使用有参构造创建对象
    1. 下标赋值

```xml
<!--  1.使用下标赋值  -->
<bean id="user" class="com.zh.pojo.User">
    <constructor-arg index="0" value="张三"></constructor-arg>
</bean>
```

    2. 类型

```xml
<!--  2.通过数据类型创建    不建议使用
        基本数据类型可以直接写
        引用数据类型要写全路径
-->
<bean id="user" class="com.zh.pojo.User">
    <constructor-arg type="java.lang.String" value="李四">
</constructor-arg>
</bean>
```

    3. 参数名

```xml
<!--  3.直接通过参数名创建  -->
<bean id="user" class="com.zh.pojo.User">
    <constructor-arg name="name" value="王麻子"></constructor-arg>
</bean>
```

总结：在配置文件加载的时候，容器中的所有对象就已经初始化了

5、Spring配置

5.1、别名

```xml
<!--
    别名：如果添加了别名，我们也可以使用别名获取这个对象
    name:bean的id
    alias：别名NewName
-->
<alias name="user" alias="User123"/>
```

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

    //获取spring上下文
    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

    User user = (User) context.getBean("User123");

    user.show();
}
```

5.2、Bean配置

```xml
<!--
    id：Bean的唯一标识符，相当于对象名
    class：bean对象所对应的全限定名
    name：也是别名,name可以同时取多个别名
-->
<bean id="userT" class="com.zh.pojo.UserT" name="t,123"></bean>
```

5.3、Import

一般用于团队开发使用，可以将多个配置文件，导入合并成一个

假设项目多个人开发，三个人负责不同的类的开发，不同的类注册在不同的bean中，可以使用import将不同的bean.xml合并成功一个

- 张三：beans01.xml
- 李四：beans02.xml
- 王五：beans03.xml
- applicationContext.xml

```xml
<import resource="beans01.xml"/>
<import resource="beans02.xml"/>
<import resource="beans03.xml"/>
```

使用的时候直接使用applicationContext.xml就可以

6、DI依赖注入

6.1、构造器注入

见4、IOC创建对象的方式

6.2、Set方式注入【重点】

- 依赖注入：set注入
  - 依赖：bean对象的创建依赖容器
  - 注入：bean对象中的所有属性，由容器来注入

6.2.1、环境搭建

1. 复杂类型

```java
/**
 * 地址类
 */
public class Address {
    private String address;
}
```

2. 真实测试对象

```java
/**
 * 学生类
 */
public class Student {
    private String name;    //姓名
    private Address address;    //地址
    private String[] books;     //书籍
    private List<String> hobbys;    //爱好
    private Map<String,String> card; //学生卡
    private Set<String> games; //游戏
    private String wife; //妻子
    private Properties info; //配置类
}
```

3. applicationContext.xml

```xml
<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">
```

```
    <!--  DI 依赖注入-->
    <bean id="address" class="com.zh.pojo.Address">
        <property name="address" value="陕西省咸阳市"></property>
    </bean>
    <bean id="student" class="com.zh.pojo.Student">
        <!--  1.普通值注入      value  -->
        <property name="name" value="张恒"></property>
        <!--  2.bean注入       ref  -->
        <property name="address" ref="address"></property>
    </bean>
</beans>
```

4. 测试类

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


    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

    Student student = (Student) context.getBean("student");

    System.out.println(student.toString());
}
```

完善注入信息

```
<!--  1.普通值注入      value  -->
<property name="name" value="张恒"></property>
<!--  2.bean注入       ref  -->
<property name="address" ref="address"></property>
<!--  3.数组注入        array  -->
<property name="books">
    <array>
        <value>西游记</value>
        <value>红楼梦</value>
        <value>水浒传</value>
        <value>三国演义</value>
    </array>
</property>
<!--  4.list  -->
<property name="hobbys">
    <list>
        <value>听歌</value>
        <value>跳舞</value>
        <value>打篮球</value>
    </list>
</property>
<!--  5.Map  -->
<property name="card">
    <map>
        <entry key="电话" value="123456789"></entry>
        <entry key="学号" value="987654321"></entry>
    </map>
</property>
<!--  6.set  -->
```

```
<property name="games">
    <set>
        <value>LOL</value>
        <value>王者荣耀</value>
    </set>
</property>
<!--  7.null 空值注入  -->
<property name="wife">
    <null></null>
</property>
<!--  8.Properties 配置类-->
<property name="info">
    <props>
        <prop key="身份证">121546156181156</prop>
        <prop key="性别">男</prop>
    </props>
</property>
```

6.3、扩展方式注入

可以使用P命名空间和C命名空间进行注入

官方解释：

使用：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:c="http://www.springframework.org/schema/c"
       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">
    <!--
        p命名和c命名空间注入
        不能直接使用，需要导入xml约束
        p命名需要加上
            xmlns:p="http://www.springframework.org/schema/p"
        c命名需要加上
            xmlns:c="http://www.springframework.org/schema/c"
    -->
    <!--
        p命名空间
        可以直接注入属性值 property
        相当于set注入
```

```
    -->
    <bean id="user" class="com.zh.pojo.User" p:name="张三" p:age="18">
</bean>
    <!--
        c命名空间
        可以通过构造器注入 constructor-arg
    -->
    <bean id="user2" class="com.zh.pojo.User" c:name="李四" c:age="20">
</bean>
</beans>
```

测试

```
@Test
public void userTest(){

    ApplicationContext context = new
ClassPathXmlApplicationContext("userBeans.xml");

    User user = context.getBean("user2", User.class);

    System.out.println(user.toString());

}
```

注意事项：p命名和c命名空间不能直接使用，需要导入XML约束

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

6.4、bean的作用域

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of an HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.
application	Scopes a single bean definition to the lifecycle of a `ServletContext`. Only valid in the context of a web-aware Spring `ApplicationContext`.
websocket	Scopes a single bean definition to the lifecycle of a `WebSocket`. Only valid in the context of a web-aware Spring `ApplicationContext`.

```
<!--
    bean的作用域
        1.单例模式(spring默认机制)
            scope="singleton"
            只会创建一个对象
        2.原型模式
            scope="prototype"
            每次从容器中get，都会创建一个新的对象
        3.request,session,application
            只能在web开发中使用

-->
```

1. 单例模式（Spring默认机制）

```
<bean id="user2" class="com.zh.pojo.User" c:name="李四" c:age="20"
scope="singleton"></bean>
```

2. 原型模式：每次从容器中get的时候，都会产生一个新的对象

```
<bean id="accountService" class="com.something.DefaultAccountService"
scope="prototype"/>
```

3. request,session,application只能在web开发中使用

7、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

在Spring中有三种装配方式

1. 在xml中显示的装配
2. 在java中显示装配
3. 隐式的自动咋黄配bean

7.1、测试

环境搭建：一个人有两个宠物

```
public class People {
    private String name;
    private Dog dog;
    private Cat cat;
}
```

```
public class Dog {
    public void shout(){
        System.out.println("汪汪汪汪汪汪~");
    }
}
```

```java
public class Cat {
    public void shout(){
        System.out.println("喵喵喵~");
    }
}
```

7.2、ByName自动装配

```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
需要保证所有bean的id唯一，并且这个bean需要和自动注入的属性的set方法的值一致
-->
<bean id="dog" class="com.zh.pojo.Dog"></bean>
<bean id="cat" class="com.zh.pojo.Cat"></bean>
<bean id="people" class="com.zh.pojo.People" autowire="byName">
    <property name="name" value="张三"></property>
</bean>
```

7.3、ByType自动装配

```xml
<!--

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
    需要保证所有bean的class唯一，并且这个bean需要和自动注入的属性的类型一致
    被装配的bean可以不要id
    <bean class="com.zh.pojo.Dog"></bean>
    必须保证类型全局唯一
-->

<bean id="dog" class="com.zh.pojo.Dog"></bean>
<bean id="cat" class="com.zh.pojo.Cat"></bean>
<bean id="people" class="com.zh.pojo.People" autowire="byType">
    <property name="name" value="张三"></property>
</bean>
```

小结：

- ByName的时候，需要保证所有bean的id唯一，并且这个bean需要和自动注入的属性的set方法的值一致
- ByType的时候，需要保证所有bean的class唯一，并且这个bean需要和自动注入的属性的类型一致

7.4、使用注解实现自动装配

使用注解须知：

1. 导入约束

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

2. 配置注解的支持

```
<!--开启注解支持-->
<context:annotation-config/>
```

3. 完整配置

```
<?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.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-
context.xsd">
    <!--开启注解支持-->
    <context:annotation-config/>
</beans>
```

@Autowired

直接在属性上使用，也可以在set方法上使用
使用@Autowired可以省略set方法，前提是这个自动装配的属性在IOC容器中存在，且符合名字byname

@Nullable：字段标记了这个注解，说明这个字段可以为null

```
public People(@Nullable String name) {
    this.name = name;
}
```

测试代码

```
public class People {

    private String name;
    //如果Autowired的required属性为false，说明这个对象可以为null否则不允许为空
    @Autowired(required = false)
    private Dog dog;
    @Autowired
    private Cat cat;
}
```

@Qualifier

如果@Autowired自动装配的环境比较复杂。自动装配无法完成。可以使用@Qualifier(value = "xxxx")配合@Autowired使用。指定唯一的bean对象注入

```
<bean id="dog11" class="com.zh.pojo.Dog"></bean>
<bean id="dog22" class="com.zh.pojo.Dog"></bean>
<bean id="cat" class="com.zh.pojo.Cat"></bean>
<bean id="people" class="com.zh.pojo.People">
    <property name="name" value="张三"></property>
</bean>
```

```java
public class People {

    private String name;
    @Autowired
    @Qualifier(value = "dog22")
    private Dog dog;
    @Autowired
    private Cat cat;
}
```

@Resource：是java自带的注解，不是spring的

@Resource默认通过byname方法实现，如果找不到名字，则通过bytype实现。

自动装配的环境比较复杂，使用@Resource(name = "dog22")。指定指定唯一的bean对象注入

```java
public class People {

    private String name;
    @Resource(name = "dog22")
    private Dog dog;
    @Resource
    private Cat cat;
}
```

小结：

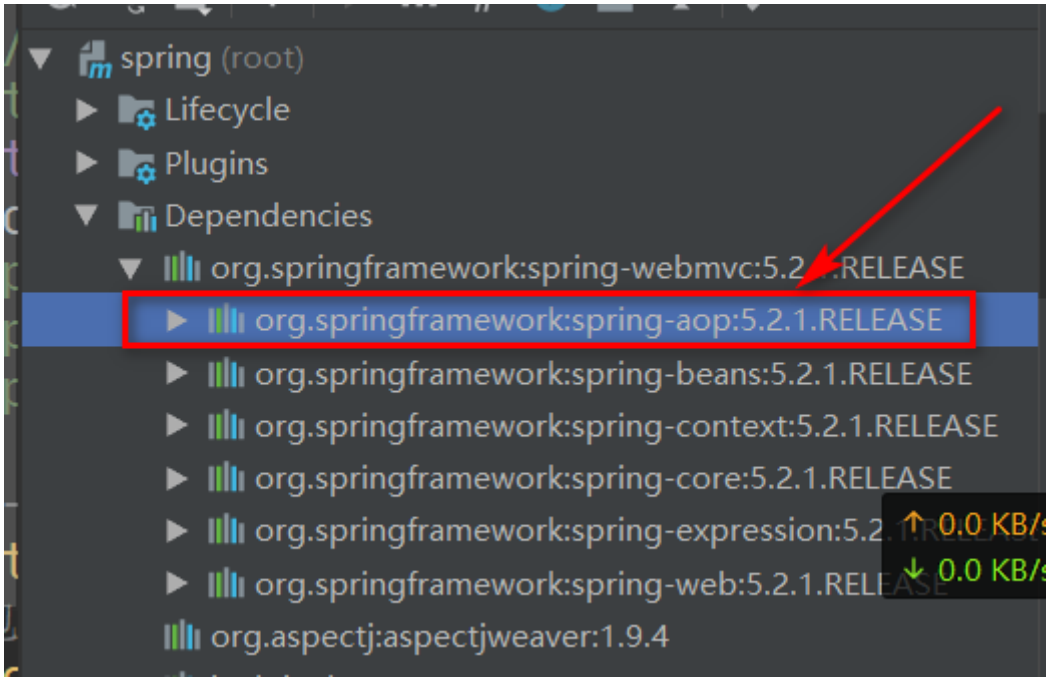
@Autowired和@Resource的区别

- 都是用来自动装配的，都可以放在属性字段上
- @Autowired通过ByType的方式实现的，而且必须要求这个对象存在。【常用】
- @Resource默认通过ByName的方式实现的，如果找不到名字，则通过ByType实现。如果两个都找不到的情况下，就报错。【常用】
- 执行顺序不同：@Autowired通过ByType的方式实现，@Resource默认通过ByName的方式实现

8、使用注解开发

在Spring4之后，要使用注解开发，必须导入aop的包

spring-webmvc已经导入



使用注解需要导入context约束，增加注解支持

```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:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
    <!--开启注解支持-->
    <context:annotation-config/>
    <!--指定要扫描的包，这个包下的注解就会生效-->
    <context:component-scan base-package="com.zh"/>
</beans>
```

1. bean
2. 属性如何注入

```java
//等价于<bean id="user" class="com.zh.model.User"/>
@Component
public class User {
    //相当于 <property name="name" value="李四"/>
    @Value("李四")
    public String name;
}
```

3. 衍生的注解

   @Component衍生注解。在web开发中，按照三层架构分层
   dao:　　@Repository
   service:　@Service
   controller: @Controller
   四个注解的功能都是一样的，都是将某个类装配到ioc容器中，装配bean

4. 自动装配
5. 作用域

```java
//等价于<bean id="user" class="com.zh.model.User"/>
@Component
//作用域
@Scope("singleton")
public class User {

    //相当于 <property name="name" value="李四"/>
    @Value("李四")
    public String name;

}
```

6. 小结

   xml与注解

   - xml更加万能，使用与任何场合！维护简单方便
   - 注解不是自己类使用不了，维护相对复杂

   xml与注解最佳实践

   - xml用来管理bean
   - 注解只完成属性注入
   - 使用注解过程中，必须开启注解支持，扫描对应的包

```xml
<!--开启注解支持-->
<context:annotation-config/>
<!--指定要扫描的包，这个包下的注解就会生效-->
<context:component-scan base-package="com.zh"/>
```

9、使用Java方式配置Spring

不使用Spring的xml配置，全权交给java

JavaConfig是Spring的一个子项目，在Spring4之后，成为一个核心功能

实体类

```java
//说明这个类被spring容器接管了，注册到了ioc容器中
@Component
public class User {

    //属性注入值
    @Value("张三")
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
```

```
        return "User{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

配置文件

```
//这个也会被spring容器托管，注册到容器中，因为它本来就是一个@Component
//@Configuration代表这是一个配置类，和applicationContext.xml是一样的
@Configuration
//扫描包
@ComponentScan("com.zh.pojo")
//整合配置类
@Import(MyConfig.class)
public class AppConfig {

    /*
     * 相当于一个<bean>标签
     * 方法名相当于bean标签的id
     * 返回值相当于bean的class
     *
     * 扫描包之后可以不写这个方法
     */
    @Bean
    public User user(){
        return new User();//就是返回要注入到bean的对象
    }

}
```

```
@Configuration
public class MyConfig {
}
```

测试类

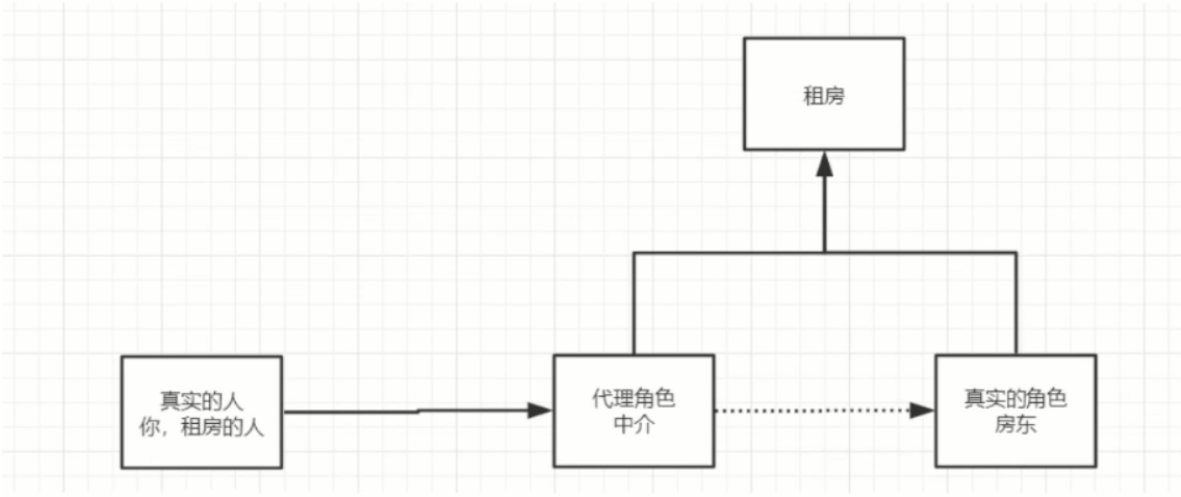
```
public static void main(String[] args) {

    //如果完全使用了配置类，只能通过AnnotationConfig上下文获取容器，通过配置类的class对象加载
    ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

    User user = context.getBean("user", User.class);

    System.out.println(user.getName());

}
```

10、代理模式

SpringAOP的底层就是代理模式

代理模式的分类：

- 静态代理
- 动态代理



10.1、静态代理

角色分析：

- 抽象角色：一般使用接口或者抽象类来解决
- 真实角色：被代理的角色
- 代理角色：代理真实角色，代理真实角色后，一般会做一些附属操作
- 客户：访问代理对象的人

代码步骤：

1. 接口

```
//租房
public interface Rent {
    public void rent();
}
```

2. 真实角色

```
//房东
public class Host implements Rent{
    public void rent() {
        System.out.println("房东要出租房子");
    }
}
```

3. 代理角色

```
//中介 代理角色
public class Proxy implements Rent{

    private Host host;

    public Proxy() {
    }
```

```
8
9       public Proxy(Host host) {
10          this.host = host;
11      }
12
13      public void rent() {
14          seeHouse();
15          host.rent();
16          hetong();
17          fare();
18      }
19
20      //看房
21      public void seeHouse(){
22          System.out.println("中介带你看房");
23      }
24
25      //签租赁合同
26      public void hetong(){
27          System.out.println("签租赁合同");
28      }
29
30      //收中介费
31      public void fare(){
32          System.out.println("收中介费");
33      }
34  }
```

4. 客户端访问代理角色

```
1   public class Client {
2
3       public static void main(String[] args) {
4           //房东要租房子
5           Host host = new Host();
6           //代理    代理角色会做一些附属操作
7           Proxy proxy = new Proxy(host);
8           //直接找中介即可
9           proxy.rent();
10
11      }
12
13  }
```

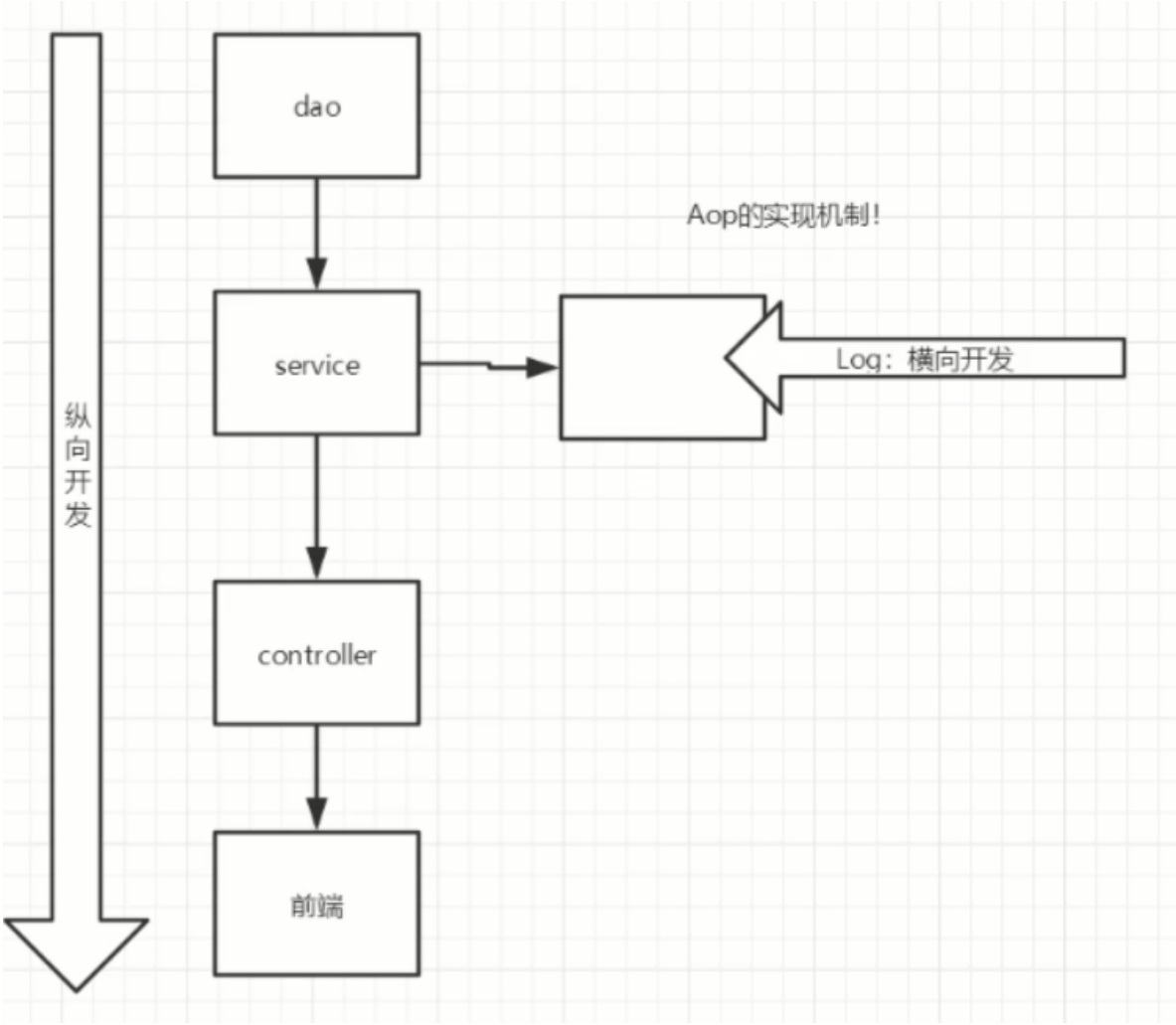
代理模式的好处：

- 可以使真实角色的操作更加纯粹，不用去关注一些公共业务
- 公共业务就交给代理角色，实现了业务分工
- 公共业务发生扩展的时候，方便集中管理

缺点：一个真实角色就会产生一个代理角色，代码量会翻倍，开发效率变低

10.2、静态代理2

代码spring10_proxy：demo02



10.3、动态代理

- 动态代理和静态代理角色一样
- 动态代理的代理类是动态生成的，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【使用】
  - 基于类：cglib
  - java字节码实现：javasist

需要了解两个类：Proxy，InvocationHandler：调用处理程序

工具类

```
package com.zh.demo04;

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

//会用这个类自动生成代理类
public class ProxyInvocationHandler implements InvocationHandler {

    //被代理的接口
    private Object target;

```

```
    public void setTarget(Object target) {
        this.target = target;
    }


    //生成得到代理类
    public Object getProx(){
        return
Proxy.newProxyInstance(this.getClass().getClassLoader(),target.getClass().g
etInterfaces(),this);
    }


    //处理代理实例，并返回结果
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {

        log(method.getName());
        //动态代理的本质，就是使用反射机制实现
        Object result = method.invoke(target, args);
        return result;
    }

    public void log(String msg){
        System.out.println("使用"+msg+"方法");
    }

}
```

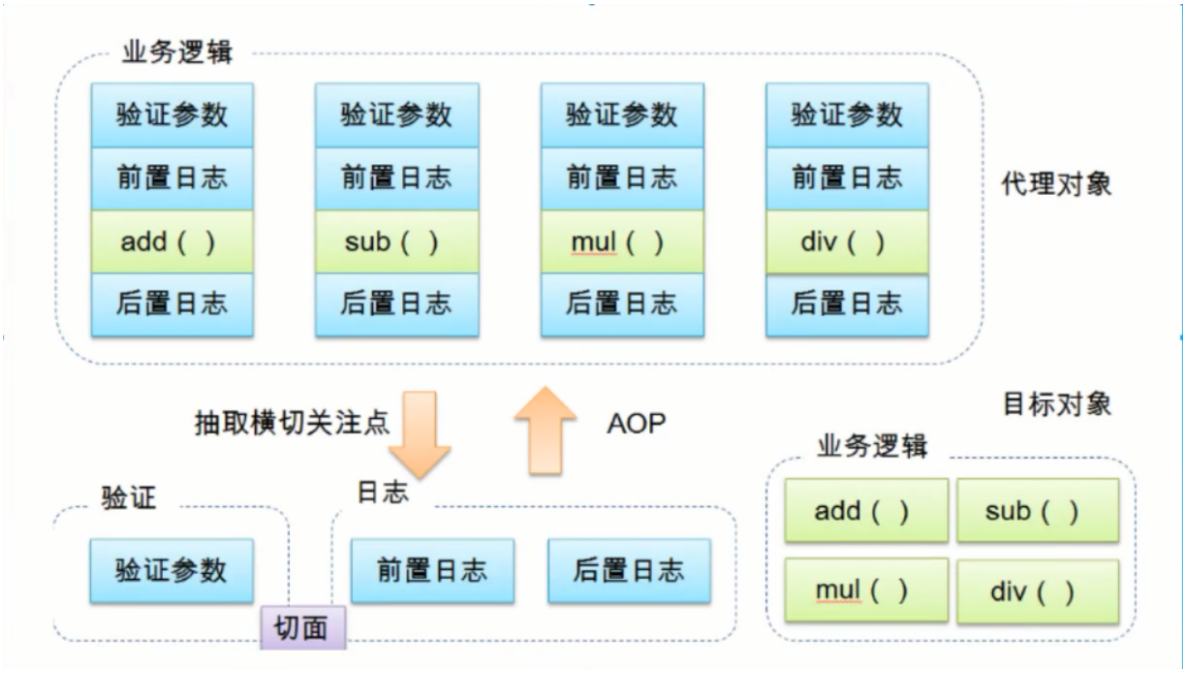
动态代理的好处：

- 可以使真实角色的操作更加纯粹，不用去关注一些公共业务
- 公共业务就交给代理角色，实现了业务分工
- 公共业务发生扩展的时候，方便集中管理
- 一个动态代理类代理的是一个接口，一般就是对应一类业务
- 一个动态代理类可以代理多个类，只有是实现同一个接口即可

11、AOP

11.1、什么是AOP

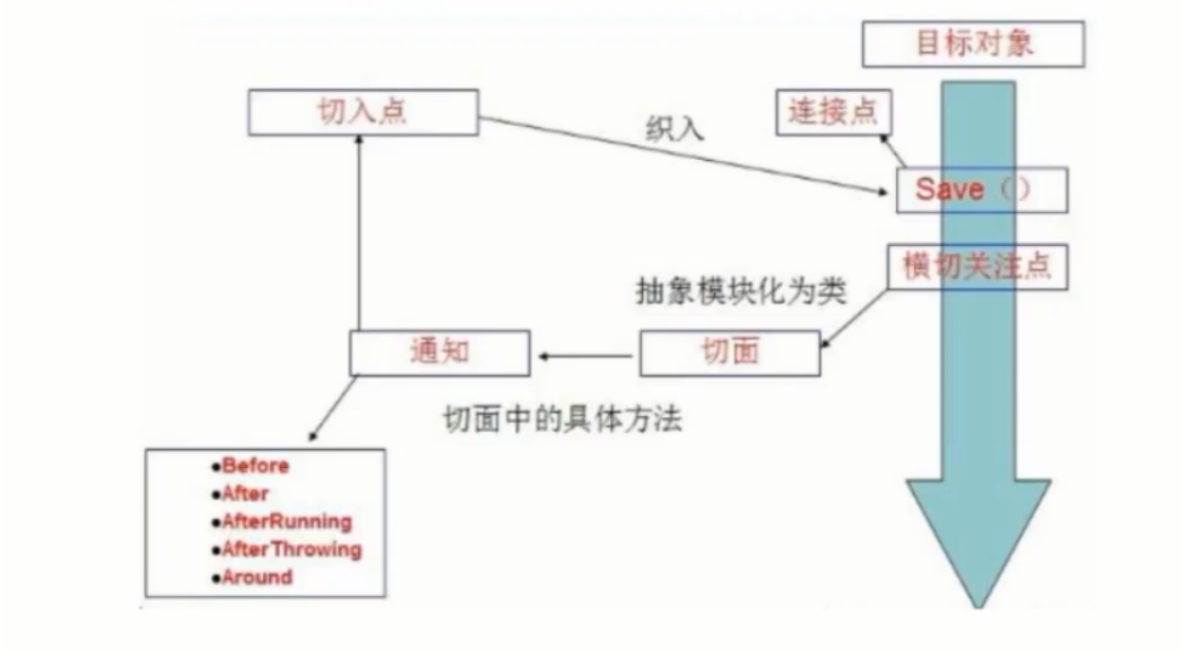
AOP(Aspect Oriented Programming)意为：面向切面编程，通过预编译方式和运行期间动态代理实现程序功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。



11.2、AOP在Spring中的作用

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。即是，与业务逻辑无关的，但是我们需要关注的部分，就是横切关注点。如日志，安全，缓存，事务等等....
- 切面（ASPECT）：横切关注点被模块化的特殊对象。即，它是一个类。
- 通知（Advice）：切面必须要完成的工作。即，它是类中的一个方法
- 目标（Target）：被通知对象
- 代理（Proxy）：向目标对象应用通知之后创建的对象
- 切入点（PointCut）：切面通知执行的"地点"的定义
- 连接点（JointPoint）：与切入点匹配的执行点



SpringAOP中，通过Advice定义横切逻辑，Spring中支持5中类型的Advice

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

即AOP在不改变原有代码的情况下，去增加新功能

11.3、环境搭建

【重点】使用AOP织入，需要导入一个依赖包

```xml
<!-- spring aop织入依赖包 -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.4</version>
</dependency>
```

```java
public interface UserService {
    void add();
    void delete();
    void update();
    void select();
}
```

```java
package com.zh.service;

public class UserServiceImpl implements UserService{
    public void add() {
        System.out.println("添加一个用户");
    }

    public void delete() {
        System.out.println("删除一个用户");
    }

    public void update() {
        System.out.println("修改一个用户");
    }
```

```
15
16      public void select() {
17          System.out.println("查询一个用户");
18      }
19  }
```

使用aop必须配置aop的约束

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <beans xmlns="http://www.springframework.org/schema/beans"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xmlns:aop="http://www.springframework.org/schema/aop"
5          xsi:schemaLocation="http://www.springframework.org/schema/beans
6           http://www.springframework.org/schema/beans/spring-beans.xsd
7            http://www.springframework.org/schema/aop
8             https://www.springframework.org/schema/aop/spring-aop.xsd">
9
10      <!--注册bean-->
11      <bean id="userService" class="com.zh.service.UserServiceImpl"></bean>
12
13  </beans>
```

11.4、AOP的三种实现方式

11.4.1、使用Spring的API接口

```
1   SpringAop中，通过Advice定义横切逻辑，spring中支持5种类型的Advice
2   通知类型              连接点                 实现接口
3   前置通知              方法前                 MethodBeforeAdvice
4   后置通知              方法后                 AfterReturningAdvice
5   环绕通知              方法前后                MethodInterceptor
6   异常抛出通知           方法抛出异常           ThrowsAdvice
7   引介通知              类中增加新的方法属性    IntroductionInterceptor
```

```
1   //前置通知MethodBeforeAdvice
2   public class Log implements MethodBeforeAdvice {
3       /*
4        *  method      要执行的目标对象的方法
5        *  objects     参数
6        *  o           目标对象
7        */
8       public void before(Method method, Object[] objects, Object o) throws
    Throwable {
9           System.out.println(o.getClass().getName()+"对象
    的"+method.getName()+"的方法被执行了");
10      }
11  }
```

```java
//后置通知
public class AfterLog implements AfterReturningAdvice {
    /*
     *  o           返回值
     *  method      要执行的目标对象的方法
     *  objects     参数
     *  o1          目标对象
     */
    public void afterReturning(Object o, Method method, Object[] objects,
Object o1) throws Throwable {
        System.out.println("执行了"+method.getName()+"方法。返回结果为："+o);
    }
}
```

```xml
<!--注册bean-->
<bean id="userService" class="com.zh.service.UserServiceImpl"></bean>
<bean id="log" class="com.zh.log.Log"></bean>
<bean id="afterLog" class="com.zh.log.AfterLog"></bean>

<!--方式一 ： 使用原生Spring Aop接口-->
<!--配置aop：需要导入aop的约束-->
<aop:config>
    <!--切入点 expression:表达式  execution(要执行的位置*****)
        *(修饰词) *(返回值) *(类名) *(方法名) *(参数)
    -->
    <aop:pointcut id="pointcut" expression="execution(*
com.zh.service.UserServiceImpl.*(..))"/>
    <aop:advisor advice-ref="log" pointcut-ref="pointcut"></aop:advisor>
    <aop:advisor advice-ref="afterLog" pointcut-ref="pointcut">
</aop:advisor>
</aop:config>
```

11.4.2、自定义实现AOP

```java
//自定义切入点类
public class DiyPointCut {

    public void befare(){
        System.out.println("=============方法执行前============");
    }

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

```xml
<!--方式二：自定义类-->
<bean id="diy" class="com.zh.diy.DiyPointCut"></bean>
<aop:config>
    <!--自定义切面 ref要引用的类-->
    <aop:aspect ref="diy">
        <!--切入点-->
        <aop:pointcut id="point" expression="execution(*
com.zh.service.UserServiceImpl.*(..))"/>
        <!--类中的方法-->
        <aop:before method="befare" pointcut-ref="point"/>
        <aop:after method="after" pointcut-ref="point"/>

    </aop:aspect>
</aop:config>
```

11.4.3、使用注解实现AOP

使用注解实现需要开启AOP注解支持

```xml
<!--方式三：使用注解实现aop-->
<bean id="annotationPointCut" class="com.zh.diy.AnnotationPointCut"></bean>
<!--开启aop注解支持-->
<aop:aspectj-autoproxy/>
```

```java
/**
 * 方式三：使用注解实现AOP
 */
//标注这个类是一个切面
@Aspect
public class AnnotationPointCut {

    @Before("execution(* com.zh.service.UserServiceImpl.*(..))")
    public void befare(){
        System.out.println("============方法执行前============");
    }

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


    /**
     * 环绕增强
     */
    @Around("execution(* com.zh.service.UserServiceImpl.*(..))")
    public void around(ProceedingJoinPoint jp) throws Throwable {

        System.out.println("环绕前");

        //获得方法签名
        Signature signature = jp.getSignature();
        System.out.println("signature:"+signature);
        //执行方法
        Object proceed = jp.proceed();

```

```
        System.out.println("环绕后");

        System.out.println(proceed);
    }
}
```

12、整合MyBatis

步骤：

1. 导入相关jar包
    - junit
    - mybatis
    - mysql
    - spring相关
    - aop织入
    - mybatis-spring：整合需要
    - spring-jdbc：spring操作数据库需要

```
<dependencies>
    <!-- spring核心包 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.2.1.RELEASE</version>
    </dependency>
    <!-- spring aop织入依赖包 -->
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.9.4</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.38</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.4.6</version>
    </dependency>
    <!--spring操作数据库需要-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.1.RELEASE</version>
    </dependency>
    <!-- mybatis和spring整合需要 -->
    <dependency>
```

```
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>2.0.3</version>
    </dependency>
</dependencies>
```

2. 编写配置文件

3. 测试

12.1、Mybatis

1. 编写实体类

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    private int id;
    private String name;
    private String pwd;
}
```

2. 编写核心配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">

<!--核心配置环境-->
<configuration>
    <typeAliases>
        <package name="com.zh.pojo"/>
    </typeAliases>

    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/mybatis"/>
                <property name="username" value="root"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>

    <mappers>
        <mapper class="com.zh.mapper.UserMapper"/>
    </mappers>

</configuration>
```

3. 编写接口

```java
public interface UserMapper {

    List<User> findAll();

}
```

4. 编写Mapper.xml

```xml
<?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.zh.mapper.UserMapper">
    <select id="findAll" resultType="user">
        select * from mybatis.user
    </select>
</mapper>
```

5. 测试

```java
@Test
public void findAll() throws IOException {
    String resources = "mybatis-config.xml";
    InputStream in = Resources.getResourceAsStream(resources);
    SqlSessionFactory sessionFactory = new
SqlSessionFactoryBuilder().build(in);
    SqlSession sqlSession = sessionFactory.openSession();

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

    List<User> list = mapper.findAll();
    for (User user : list) {
        System.out.println(user);
    }
    sqlSession.close();
}
```

12.2、mybatis-spring

官网：http://mybatis.org/spring/zh/index.html

MyBatis-Spring	MyBatis	Spring 框架	Spring Batch	Java
2.0	3.5+	5.0+	4.0+	Java 8+
1.3	3.4+	3.2.2+	2.1+	Java 6+

12.3、两种整合方法

12.3.1、SqlSessionTemplate

1. 编写数据源

```
<!--DataSource：使用Spring的数据源替换Mybatis的配置-->
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>
    <property name="username" value="root"/>
    <property name="password" value="123456"/>
</bean>
```

2. SqlSessionFactory

```
<!--SqlSessionFactory-->
<bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <!--mybatis所有的配置，可以在此处实现-->
    <!--别名-->
    <property name="typeAliasesPackage" value="com.zh.pojo"/>
    <!--当mybatis配置文件有设置的时候，绑定mybatis配置文件-->
    <property name="configLocation" value="classpath:mybatis-
config.xml"/>
    <!--映射Mapper.xml文件-->
    <property name="mapperLocations"
value="classpath:com/zh/mapper/*.xml"/>
 </bean>
```

3. SqlSessionTemplate

```
<!--
    SqlSessionTemplate：基于Spring 的事务配置来自动提交、回滚、关闭 session。
    SqlSessionTemplate：是sqlSession的模板
    只能使用构造器注入sqlSessionFactory，因为SqlSessionTemplate
    没有set方法
-->
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg index="0" ref="sqlSessionFactory"/>
</bean>
```

4. 需要给接口加实现类

```
public class UserMapperImpl implements UserMapper {

    //原来的操作都使用sqlSession来操作，现在使用SqlSessionTemplate
    private SqlSessionTemplate sqlSession;

    public void setSqlSession(SqlSessionTemplate sqlSession) {
        this.sqlSession = sqlSession;
    }

    public List<User> findAll() {

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

        return mapper.findAll();
    }
```

```
}
```

5. 将实现类注入到ioc中

```
<bean id="UserMapper" class="com.zh.mapper.UserMapperImpl">
    <property name="sqlSession" ref="sqlSession"/>
</bean>
```

6. 测试

```
@Test
public void findAll(){
    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

    UserMapper userMapper = context.getBean("UserMapper",
UserMapper.class);

    List<User> list = userMapper.findAll();

    for (User user : list) {
        System.out.println(user);
    }
}
```

12.3.2、SqlSessionDaoSupport

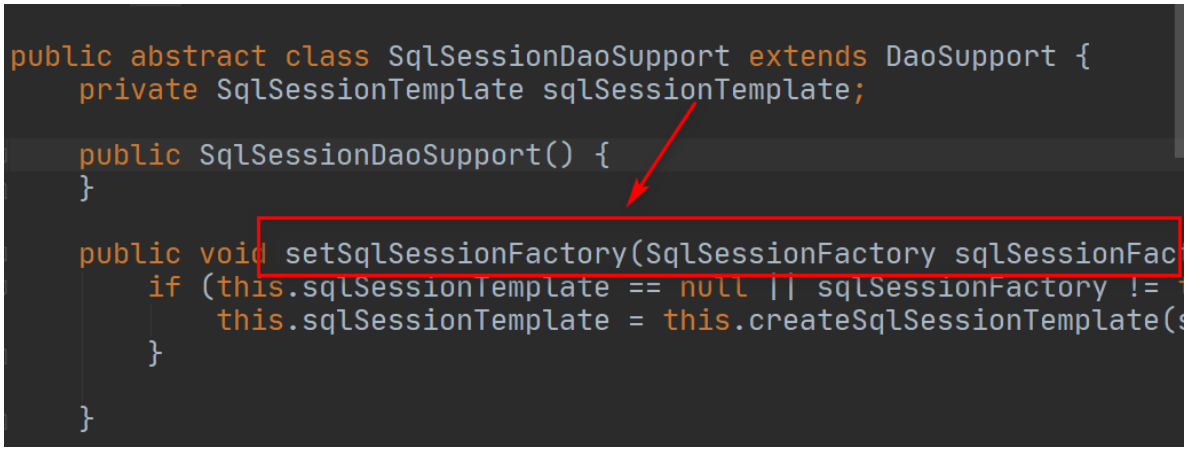
SqlSessionDaoSupport 是一个抽象的支持类，用来提供 SqlSession。调用 getSqlSession() 方法你会得到一个 SqlSessionTemplate

```
public class UserMapperImpl2 extends SqlSessionDaoSupport implements
UserMapper {

    public List<User> findAll() {
        SqlSession sqlSession = getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        return mapper.findAll();
    }
}
```

```
<bean id="UserMapper2" class="com.zh.mapper.UserMapperImpl2">
    <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>
```

使用SqlSessionDaoSupport 不需要在xml中配置SqlSessionTemplate

但是因为继承SqlSessionDaoSupport类，有sqlSessionFactory，所以需要注入sqlSessionFactory



13、声明式事务

13.1、回顾事务

- 把一组业务当成一个业务来做：要么都成功，要么都失败
- 事务在开发中，非常重要，涉及到数据的一致性
- 确保完整性和一致性

事务ACID原则

- 原子性
- 一致性
- 隔离性
  - 多个业务可能操作同一个资源，防止数据损坏
- 持久性
  - 事务一旦提交，结果都不会被影响，被持久化的写到存储器

例如：

delete删除语句错误

```
<mapper namespace="com.zh.mapper.UserMapper">
    <select id="findAll" resultType="user">
        select * from mybatis.user
    </select>

    <insert id="add" parameterType="user">
        insert into mybatis.user (id, name, pwd) values (#{id},#{name},#
{pwd});
    </insert>

    <delete id="delete" parameterType="int">
        deletes from mybatis.user where id = #{id}
    </delete>
</mapper>
```

```java
public List<User> findAll() {

    User user = new User(5, "小明", "123");

    SqlSession sqlSession = getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    mapper.add(user);
    mapper.delete(5);

    return mapper.findAll();
}
```

测试

```java
@Test
public void findAll(){
    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

    UserMapper userMapper = context.getBean("UserMapper2",
UserMapper.class);

    List<User> list = userMapper.findAll();

    for (User user : list) {
        System.out.println(user);
    }
}
```

```
r version for the right syntax to use near 'deletes from mybatis.user where id = 5' at line

ntax to use near 'deletes from mybatis.user where id = 5' at line 1
server version for the right syntax to use near 'deletes from mybatis.user where id = 5' at
```

<Filter criteria>

	id	name	pwd
1	1	张三	123456
2	2	李四	123456
3	3	王麻子	123456
4	4	王五	123456
5	5	小明	123

程序报错，但是不符合事务一致性，这一组业务中，在错误之前的sql操作已经执行了

13.2、Spring中的事务

- **声明式事务：AOP 不修改原有代码**
- 编程式事务：在代码中，声明事务的管理　修改了原有代码

13.3、声明式事务的使用

13.3.1、使用声明式事务要配置tx和aop的约束

```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:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">
</beans>
```

13.3.2、开启事务的声明

```xml
<!--开启声明式事务 固定写法-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!--<constructor-arg ref="dataSource" />-->
    <property name="dataSource" ref="dataSource"/>
</bean>
```

13.3.3、配置事务通知

```xml
<!--结合AOP实现事务的织入-->
<!--配置事务通知-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!--给那些方法配置事务 name-->
    <!--配置事务的传播特性 propagation
         Spring中七种Propagation类的事务属性详解：
         REQUIRED：支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。
         SUPPORTS：支持当前事务，如果当前没有事务，就以非事务方式执行。
         MANDATORY：支持当前事务，如果当前没有事务，就抛出异常。
         REQUIRES_NEW：新建事务，如果当前存在事务，把当前事务挂起。
         NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
         NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。
         NESTED：支持当前事务，如果当前事务存在，则执行一个嵌套事务，如果当前没有事务，就新建一个事务。
         一般查询业务设置为只读：read-only="true"  只读
    -->
    <tx:attributes>
        <!--<tx:method name="add" propagation="REQUIRED"/>
        <tx:method name="delete" propagation="REQUIRED"/>
        <tx:method name="update" propagation="REQUIRED"/>
        <tx:method name="findAll" read-only="true"/>-->
        <!--一般给所有的方法设置事务-->
        <tx:method name="*" propagation="REQUIRED"/>
```

```
    </tx:attributes>
</tx:advice>
```

13.3.4、配置事务切入

```
<!--配置事务切入-->
<aop:config>
                                                <!--给com.zh.mapper包下所有类的所有方法配置-->
    <aop:pointcut id="txPoinCut" expression="execution(* com.zh.mapper.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPoinCut"/>
</aop:config>
```