Spring

狂神文档：https://www.cnblogs.com/renxuw/p/12994080.html

官方文档：https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#spring-core

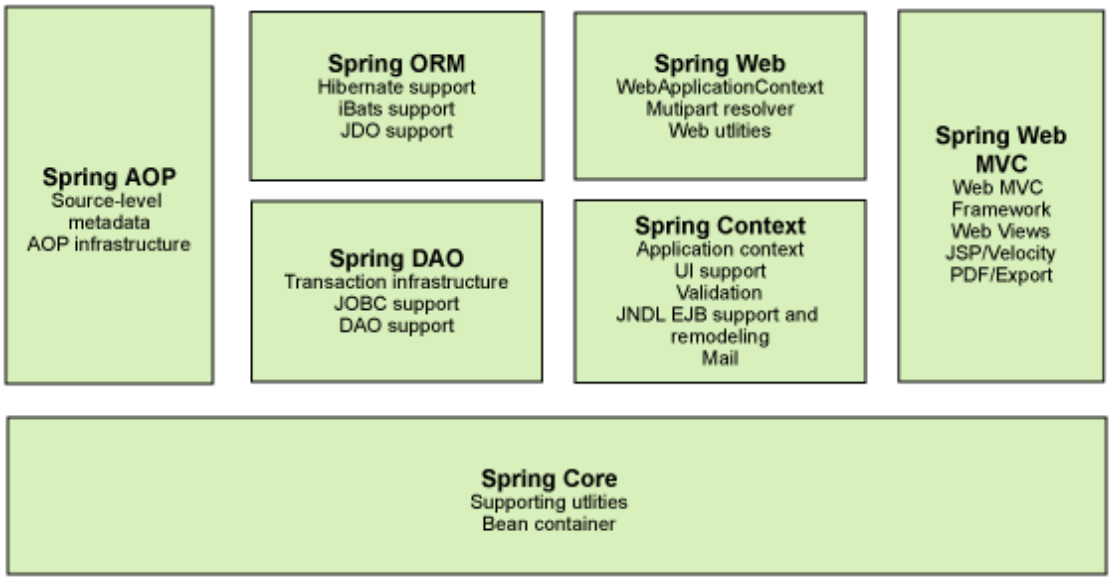
1. Spring的优点

- Spring是一个开源的免费框架
- 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

  总结：Spring是一个轻量级的控制反转（IOC）、面向切面编程（AOP）的框架！

---

2. Spring的组成模块



- **核心容器**：核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 `BeanFactory`，它是工厂模式的实现。 `BeanFactory` 使用 *控制反转* （IOC） 模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。
- **Spring 上下文**：Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。
- **Spring AOP**：通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架中。所以，可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。
- **Spring DAO**：JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。

- **Spring ORM**： Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。
- **Spring Web 模块**： Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。
- **Spring MVC 框架**： MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。

---

3. 扩展

> Spring Boot：
>
> 一个快速开发的脚手架；
>
> 基于SpringBoot开源快速开发当个微服务；
>
> 约定大于配置；
>
> 学习SpringBoot的基础是需要完全掌握Spring以及SpringMVC的基础，如今大多数的公司都是基于SpringBoot进行快速开发的
>
> Spring Cloud：
>
> SpringCloud是基于SpringBoot实现的；

---

4. IOC控制反转

> **工程1：spring-01-ioc1**

4.1 传统的业务实现方法

1、UserDao 接口

2、UserDaoImpl 实现类

3、UserService 业务接口

4、UserServiceImpl 业务实现类

在我们之前的业务中，用户的需求可能会影响我们原来的代码，我们需要根据用户的需求去修改原代码！如果程序代码量十分大，修改一次的成本代价十分高。

我们使用一个Set接口实现，已经发生了革命性的变化

```
private UserDao userDao;

// 利用set进行动态实现值的注入
public void setUserDao(UserDao userDao) {
    this.userDao = userDao;

```

- 之前，程序是主动创建对象，控制权在程序员手上！
- 使用了set注入之后，程序不再具有主动性，而是变成了被动的接收对象！

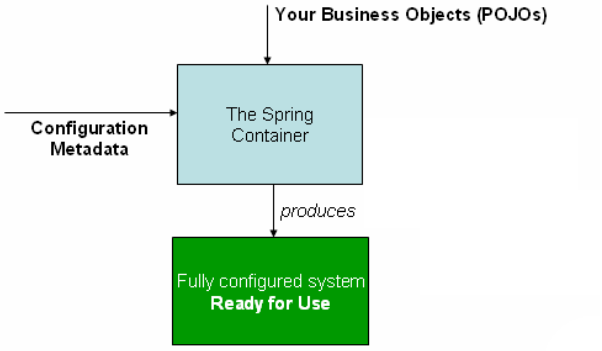
这种思想，从本质上解决了问题，我们程序员不用再去管理对象的创建了。系统的耦合性大大降低了，可以更加专注的在业务的实现上了，这是IOC的原型。

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 。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转移给第三方，所谓的控制反转就是：获得依赖对象的方式反转了。

IOC是spring的核心内容：使用了多种方式完美的实现了IOC，可以使用XML配置，也可以使用注解，新版本的Spring也可以零配置实现IOC。

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容器中，程序使用时再从IOC容器中取出需要的对象。

下图显示了Spring的工作原理的高级视图。您的应用程序类与配置元数据结合在一起，因此，在 `ApplicationContext` 创建和初始化后，您将拥有一个完全配置且可执行的系统或应用程序。



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

4.2 XML的配置文件：

> **工程2：spring-02-hellospring**

```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">

</beans>
```

4.3 第一个Spring程序

- 编写一个Hello实体类

```java
public class Hello {
   private String name;

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

```

```
    public void show(){
        System.out.println("Hello,"+ name );
    }
}
```

- 编写我们的spring文件，这里我们命名为beans.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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--使用Spring来创建对象，在Spring中这些都统称为Bean
        类型 变量名 = new 类型();
        Hello hello =new Hello();

        id = 变量名;
        class = new 类型();
        bean = 对象     new Hello();
        property 相当于给对象中的属性设置一个值
    -->
    <bean id="hello" class="com.abraham.pojo.Hello">
        <property name="str" value="Spring"/>
    </bean>

</beans>
```

- 我们可以去进行测试了 .

```
public class MyTest02 {
    public static void main(String[] args) {
        // 获取Spring的上下文对象
        ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
        // 我们的对象现在都在Spring中管理了，我们要使用，直接在里面取出来就可以了
        Hello hello = (Hello) context.getBean("hello");
        System.out.println(hello.toString());
    }
}
```

思考：

- Hello 对象是谁创建的？【hello 对象是由Spring创建的
- Hello 对象的属性是怎么设置的？hello 对象的属性是由Spring容器设置的

这个过程就叫控制反转：

- 控制：谁来控制对象的创建，传统应用程序的对象是由程序本身控制创建的，使用Spring后，对象是由Spring来创建的
- 反转：程序本身不创建对象，而变成被动的接收对象 .

依赖注入：就是利用set方法来进行注入的.

IOC是一种编程思想，由主动的编程变成被动的接收，要实现不同的操作，只需要在xml配置文件中进行修改，所谓的IoC,一句话搞定：对象由Spring 来创建 、管理、装配！

可以通过newClassPathXmlApplicationContext去浏览一下底层源码 .

---

5. IOC创建对象的方式

> **工程3：spring-03-ioc2**：beans.xml

5.1 使用无参构造创建对象

【默认方法】

5.2 使用有参构造创建对象

- 方法一：下标赋值；

```
<bean id="user" class="com.abraham.pojo.User">
    <constructor-arg index="0" value="long"/>
</bean>
```

- 方法二：类型；(不推荐使用)

```
<bean id="user" class="com.abraham.pojo.User">
    <constructor-arg type="java.lang.String" value="long"/>
</bean>
```

- 方法三：直接通过参数名；

```
<bean id="user" class="com.abraham.pojo.User">
    <constructor-arg name="name" value="long"/>
</bean>
```

总结：在配置文件加载的时候，容器中管理的对象就已经初始化了。

---

6. Spring配置

> **工程3：spring-03-ioc2**：beans.xml

6.1 别名（alias）

如果添加了别名，我们也可以使用别名来获取到这个对象

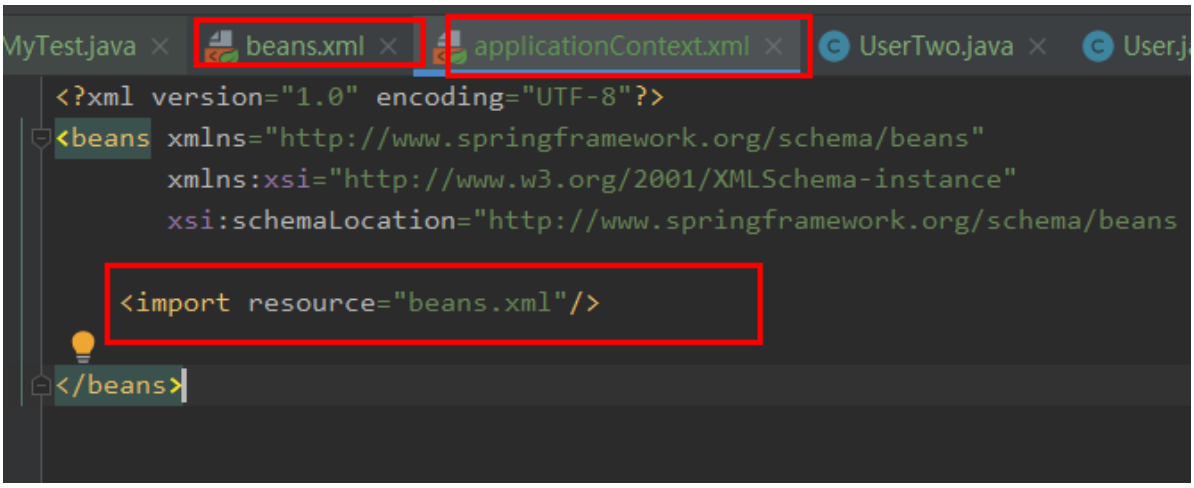
```
<alias name="user" alias="user2"/>
```

6.2 Bean的配置

```
<!--
    id：bean的唯一标识符，也就是相当于我们学的对象名
    class：bean对象所对应的权限定名：包名+类名
    name：别名,而且name可以同时取多个别名,分别可以用空格、逗号、分号进行分割
-->
<bean id="userTwo" class="com.abraham.pojo.UserTwo"
name="woshibieming,userTwo2 userTwo3;userTwo4">

</bean>
```

6.3 import

一般用于团队开发使用，它可以将多个配置文件导入合并为一个。



假设，现在项目有多个人进行不同的类开发，不同的类需要注册在不同的配置文件beans中，我们可以利用import将所有人的beans文件导入到总的xml文件：applicationContext.xml中，使用的时候直接使用一个总的配置文件就好，如上图所示。如果内容相同则会被合并为一个。

---

7. 依赖注入（DI）

> **工程4：spring-04-di**

7.1 构造器注入

【前文章节5已经描述】

7.2 Set方式注入【重点】

- 依赖注入：

  依赖：bean对象的创建依赖于容器；

  注入：bean对象中的所有属性，由容器来注入。

【环境搭建】

1、复杂类型

```java
public class Address {
    private String address;

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

2、真实测试对象

```java
public class Student {
    private String name;
    private Address address;
    private String[] books;
    private List<String> hobbys;
    private Map<String ,String> card;
    private Set<String > games;
    private Properties info;
    private String wife;
}
```

3、【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">

    <bean id="address" class="com.abraham.pojo.Address">
        <property name="address" value="福建省龙岩市"/>
    </bean>

    <bean id="student" class="com.abraham.pojo.Student">
        <!--第一种，普通值注入，value-->
        <property name="name" value="long"/>

        <!--第二种，Bean注入，ref-->
        <property name="address" ref="address"/>

        <!--第三种，数组注入，array-->
        <property name="books">
            <array>
                <value>红楼梦</value>
                <value>西游记</value>
                <value>水浒传</value>
            </array>
        </property>

        <!--第四种，List注入-->
        <property name="hobbys">
            <list>
```

```
                <value>听歌</value>
                <value>敲代码</value>
                <value>看电影</value>
            </list>
        </property>

        <!--第五种，Map注入-->
        <property name="card">
            <map>
                <entry key="身份证" value="666666"/>
                <entry key="银行卡" value="888888"/>
            </map>
        </property>

        <!--第六种，Set注入-->
        <property name="games">
            <set>
                <value>LOL</value>
                <value>COC</value>
                <value>BOB</value>
            </set>
        </property>

        <!--第七种，null注入-->
        <property name="wife">
            <null/>
        </property>

        <!--第八种，properties注入-->
        <property name="info">
            <props>
                <prop key="学号">2016551206</prop>
                <prop key="性别">girl</prop>
                <prop key="emial">1486460308@qq.com</prop>
            </props>
        </property>
    </bean>
</beans>
```

4、测试类

```
public class MyTest04 {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        Student student = (Student) context.getBean("student");
        System.out.println(student.toString());

        /**
         * name='long'
         * address=Address{address='福建省龙岩市'}
         * books=[红楼梦, 西游记, 水浒传]
         * hobbys=[听歌, 敲代码, 看电影]
         * card={身份证=666666, 银行卡=888888}
         * games=[LOL, COC, BOB]
         * info={学号=2016551206, 性别=girl, emial=1486460308@qq.com}
         * wife='null'
```

```
        */
    }
}
```

7.3 其它方式注入

- **我们可以使用c命名空间和p命名空间进行注入：**

使用前要导入约束依赖：

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

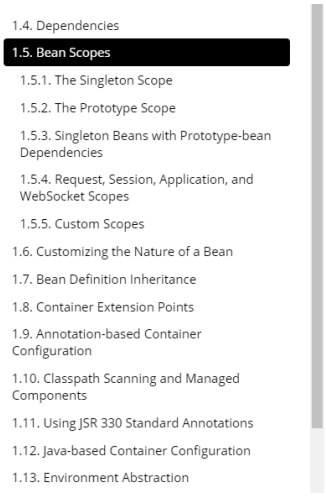
```
 xmlns:c="http://www.springframework.org/schema/c"
```

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

    <!--p命名空间注入，可以直接注入属性的值：property-->
    <bean id="user" class="com.abraham.pojo.User" p:name="long" p:age="24"/>

    <!--c命名空间注入，可以通过构造器注入属性的值：construct-args-->
    <bean id="user2" class="com.abraham.pojo.User" c:name="long"
c:age="24"/>

</beans>
```

P命名空间相当于通过无参构造注入，C命名空间相当于通过有参构造器注入参数；

7.4 Bean的作用域

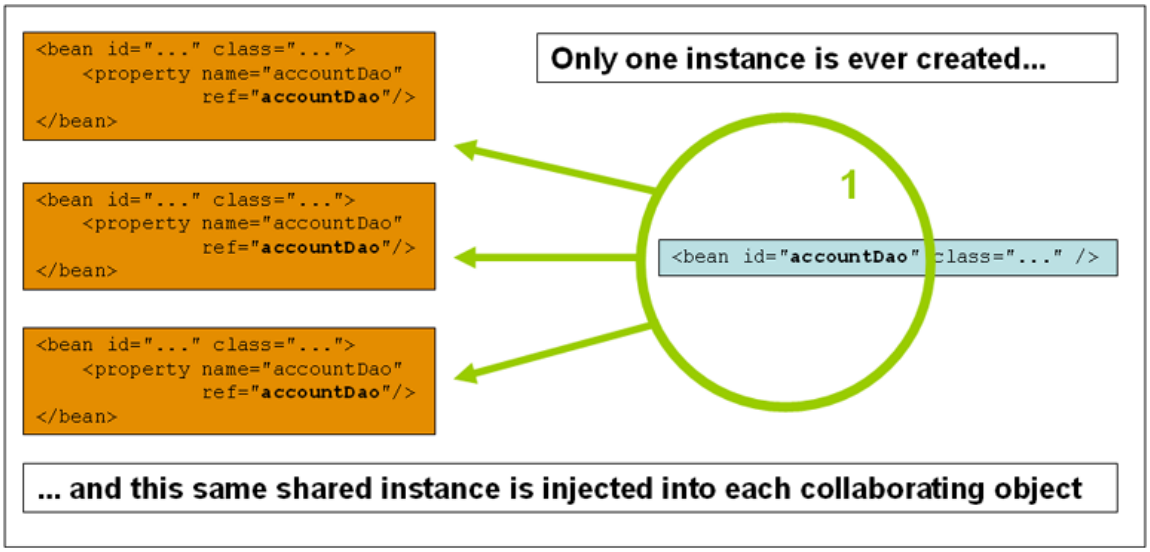


The following table describes the supported scopes:

Table 3. Bean scopes

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`.

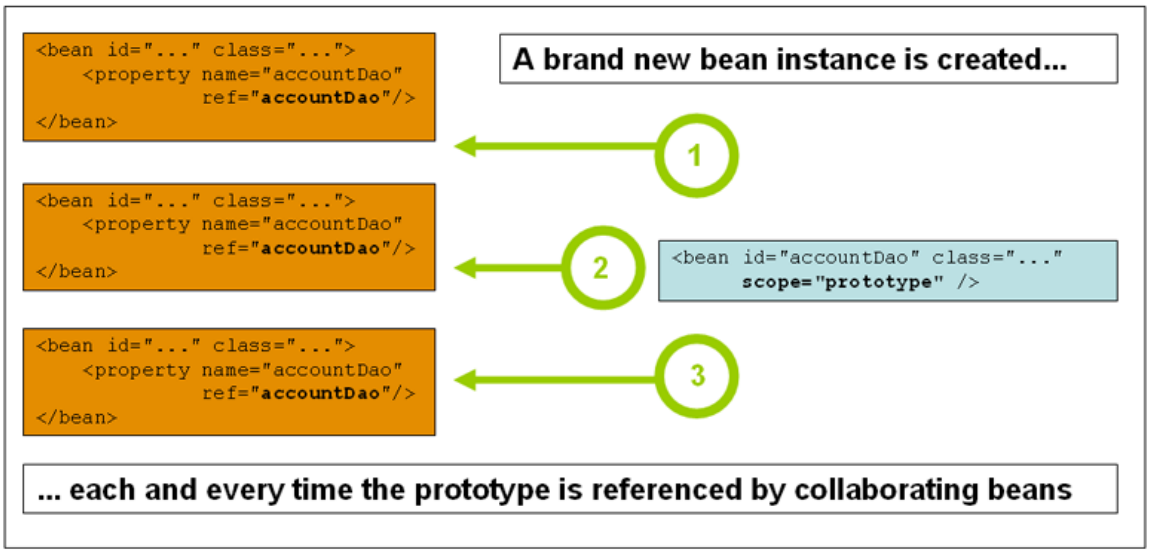
单例模式（Spring默认的机制）：



- 当一个bean的作用域为Singleton，那么Spring IoC容器中只会存在一个共享的bean实例，并且所有对bean的请求，只要id与该bean定义相匹配，则只会返回bean的同一实例。Singleton是单例类型，就是在创建起容器时就同时自动创建了一个bean的对象，不管你是否使用，他都存在了，每次获取到的对象都是同一个对象。注意，Singleton作用域是Spring中的缺省作用域。要在XML中将bean定义成singleton，可以这样配置：

```
<!-- the following is equivalent, though redundant (singleton scope is the
default) -->
<bean id="accountService" class="com.something.DefaultAccountService"
scope="singleton"/>
```

原型模式：每次从容器get的时候都会产生新的对象。



- 当一个bean的作用域为Prototype，表示一个bean定义对应多个对象实例。Prototype作用域的bean会导致在每次对该bean请求（将其注入到另一个bean中，或者以程序的方式调用容器的getBean()方法）时都会创建一个新的bean实例。Prototype是原型类型，它在我们创建容器的时候并没有实例化，而是当我们获取bean的时候才会去创建一个对象，而且我们每次获取到的对象都不是同一个对象。根据经验，对有状态的bean应该使用prototype作用域，而对无状态的bean则应该使用singleton作用域。在XML中将bean定义成prototype，可以这样配置：

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

其余的request、session、application这些只能在web中使用到

---

8. Bean的自动装配

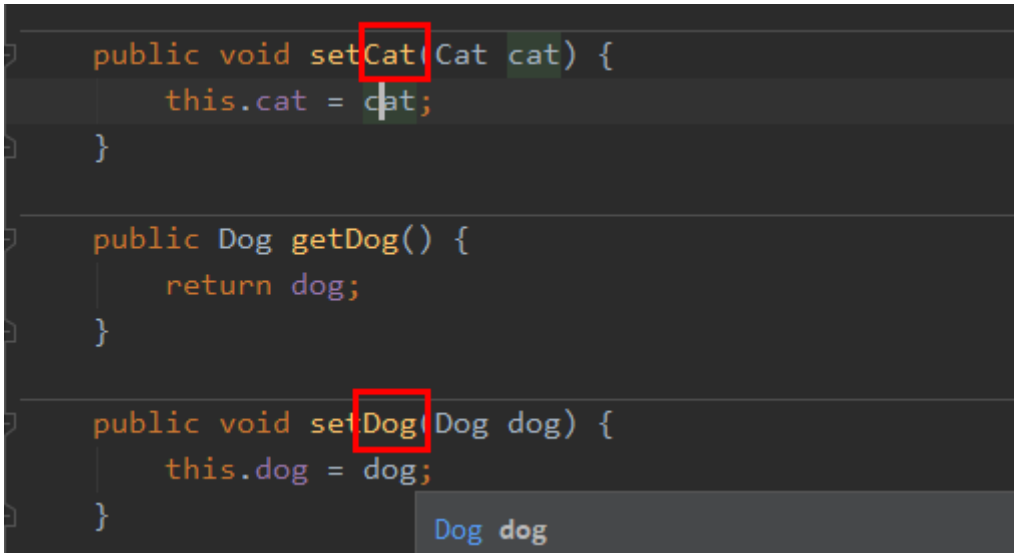
> **工程5：spring-05-autowired1**
>
> - 自动装配是Spring满足依赖的一种方式；
> - Spring会在上下文中自动寻找，并给Bean装配属性；
>
> 在Spring中有三种装配方式
>
> 1、在xml中显示的配置
>
> 2、在Java中显示配置
>
> 3、隐式的自动装配Bean【重要】

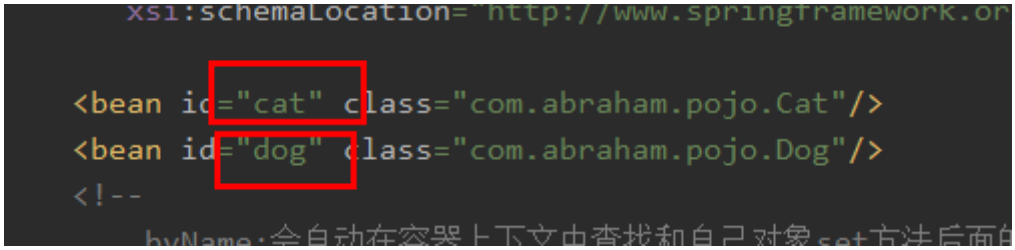
8.1 装配方式：ByName和ByType自动装配

```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">

    <bean id="cat" class="com.abraham.pojo.Cat"/>
    <bean id="dog666" class="com.abraham.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bean-id;【按名称自动装配】
        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 (必须保证装配全局唯一,可省略id参数);【按类型自动装配】
    -->
    <bean id="people" class="com.abraham.pojo.People" autowire="byType"><!--autowire="byName"-->
        <property name="name" value="long"/>
<!--        <property name="dog" ref="dog"/>-->
<!--        <property name="cat" ref="cat"/>-->
    </bean>
</beans>
```

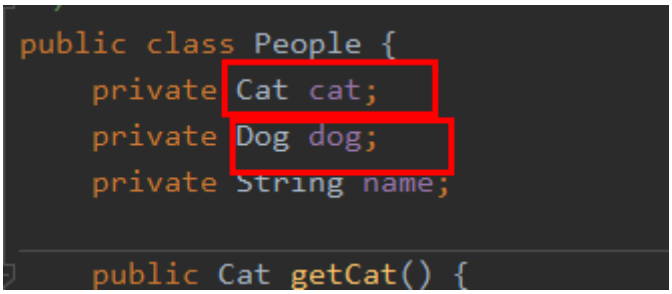
小结：

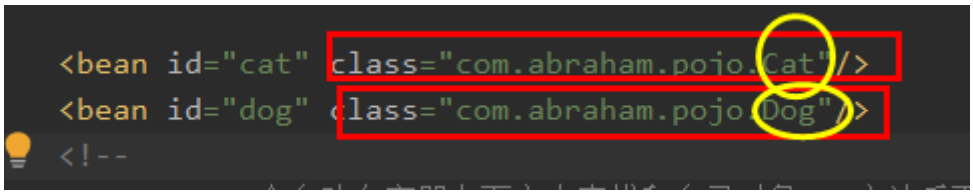
- 当一个bean节点带有 autowire byName的属性时；**【按名称自动装配】**
  - 将查找其类中所有的set方法名，例如setCat，获得将set去掉并且首字母小写的字符串，即cat；
  - 去spring容器中寻找是否有此字符串名称id的对象；
  - 如果有，就取出注入；如果没有，就报空指针异常；
- byName需要保证所有的bean的id唯一，并且这个bean需要和自动注入的属性的set方法的值一致；





- 使用autowire byType首先需要保证：同一类型的对象，在spring容器中唯一；如果不唯一，会报不唯一的异常；**【按类型自动装配】**
- byType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类型一致；





8.2 使用注解实现自动装配

jdk1.5开始支持注解，spring2.5开始支持注解；

使用注解比在xml中配置自动装配更方便；

使用注解须知：

1、导入约束；context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；context:annotation-config/

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

    <context:annotation-config/>

</beans>
```

与往常一样，您可以将它们注册为单独的bean定义，但也可以通过在基于XML的Spring配置中包含以下标记来隐式注册它们（注意，包括 `context` 名称空间）：

@Autowired 注解 【按类型装配 byType】

直接在属性上使用或者是在set方式上使用；也能在构造方法上使用；

```java
public class People {
    @Autowired  // 在属性上使用@Autowired注解
    private Cat cat;
//    @Autowired
    private Dog dog;
    private String name;

    public Cat getCat() {
        return cat;
    }

    public void setCat(Cat cat) {
        this.cat = cat;
    }

    public Dog getDog() {
        return dog;
    }
```

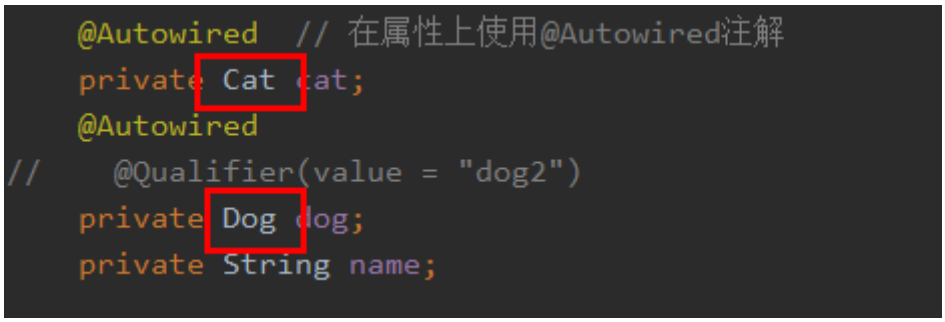
```
19
20      @Autowired  // 在set方法上使用@Autowired注解
21      public void setDog(Dog dog) {
22          this.dog = dog;
23      }
24
25      public String getName() {
26          return name;
27      }
28
29      public void setName(String name) {
30          this.name = name;
31      }
32  }
```

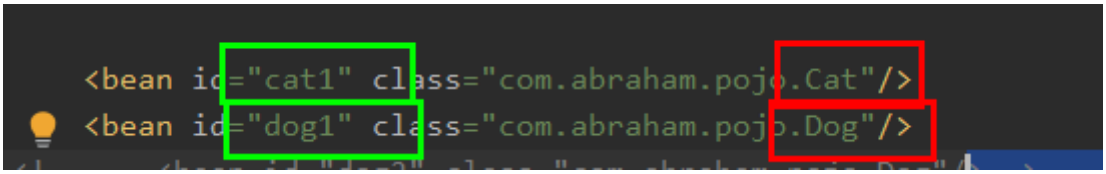
- 使用了@Autowired注解之后我们可以不用编写set方法了，前提是你这个自动在装配的属性在IOC容器中存在且符合ByName的名字要求；
- @Autowired(required = false) 如果显示定义了@Autowired的required属性值为false。说明这个对象可以为null，否则不允许为空，该方法的使用与注解@Nullable的使用相同；

```
1  @Nullable    // 该注解标记的字段可以为null值
```

```
public class SimpleMovieLister {

    @Autowired
    public void setMovieFinder(@Nullable MovieFinder movieFinder) {
        ...
    }
}
```

- **@Autowired标注的属性类型可以与xml文件中的bean字段的id不相同，也可以识别并自动装配成功；（xml文件中类型唯一的情况，因为@Autowired注解先匹配class中的属性类型是否一致）**





- **但是如果@Autowired自动装配的环境比较复杂（class中的属性类型不唯一，且id与属性类型的名称均不一致），自动装配无法通过一个注解【@Autowired】来完成的时候，我们可以使用**

@Qualifier(value="xxx")去配置@Autowired的使用，指定一个唯一的bean对象注入；

```
public class People {
    @Autowired  // 在属性上使用@Autowired注解
    private Cat cat;
    @Autowired
    @Qualifier(value = "dog2")
    private Dog dog;
    private String name;
```

```
<bean id="cat1" class="com.abraham.pojo.Cat"/>
<bean id="dog1" class="com.abraham.pojo.Dog"/>
<bean id="dog2" class="com.abraham.pojo.Dog"/>
<bean id="dog3" class="com.abraham.pojo.Dog"/>
<bean id="people" class="com.abraham.pojo.People" />
```

- @Autowired是根据**类型**自动装配的，加上@Qualifier则可以根据byName的方式自动装配;
- @Qualifier不能单独使用。

`@Resource` 注解【按照名称装配 ByName】

该注解用法类似@Autowired注解

```
1 public class People {
2     @Resource   // 该注解为java的原生注解，与@Autowired注解的功能相似
3 //    @Autowired  // 在属性上使用@Autowired注解
4     private Cat cat;
5     @Autowired
6     @Qualifier(value = "dog2")
7     private Dog dog;
8     private String name;
9 }
```

- **如果@Resource标注的属性名称与bean中的id值不相同的情况下，也可以根据class的属性类型进行成功装配（byType）**

```
public class People {
    @Resource()   // 该注解为java的原生注解，与@Autowired注解的功能相同
//    @Autowired  // 在属性上使用@Autowired注解
    private Cat cat;
    @Autowired
```

```
<bean id="cat1" class="com.abraham.pojo.Cat"/>
```

- **但是如果@Resource自动装配的环境比较复杂（bean中的id与标注的属性名称均不一致，且class中的属性类型不唯一），自动装配无法自动装配，需要配合name属性指定一个唯一的bean对象注入；**

```
public class People {
    @Resource(name = "cat1")    // 该注解为java的原生注解，与@Autowired注解的功能相同
//    @Autowired  // 在属性上使用@Autowired注解
    private Cat cat;
    @Autowired
```

```
    <bean id="cat1" class="com.abraham.pojo.Cat"/>
    <bean id="cat2" class="com.abraham.pojo.Cat"/>
```

小结：@Autowired注解与@Resource注解的区别：

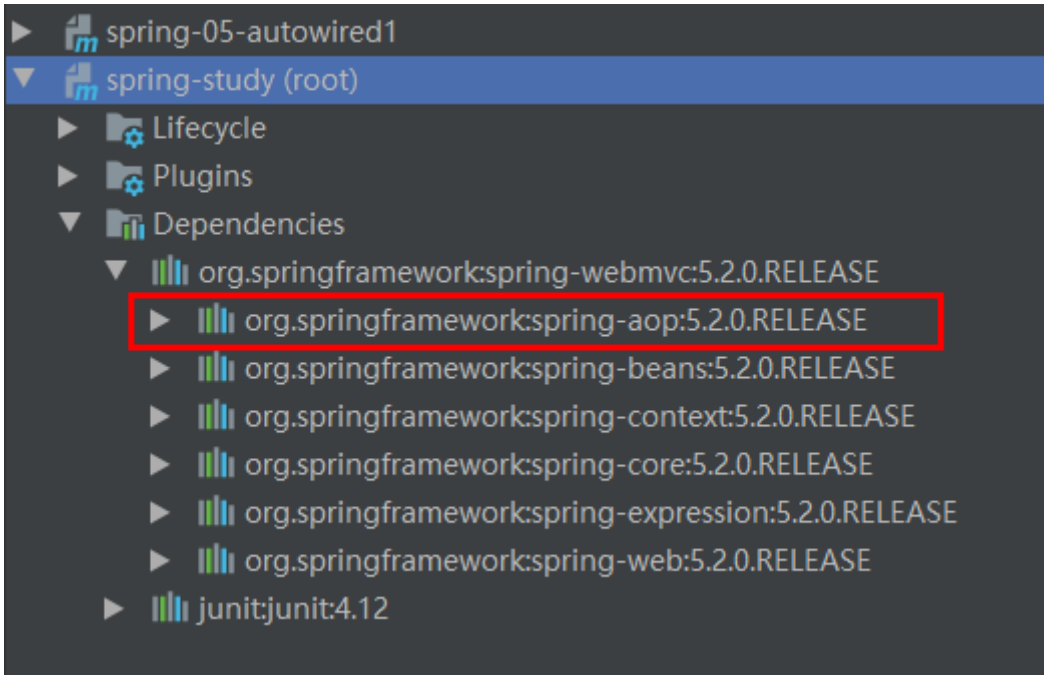
- 都是用来自动装配，都可以放在属性字段上；
- `@Autowired` 默认**按照类型装配**（ByType），默认情况下必须要求依赖对象必须存在，如果要允许null 值，可以设置它的required属性为false，如：@Autowired(required=false)，如果我们想使用名称装配可以结合@Qualifier注解进行使用
- `@Resource` 默认**按照名称装配**（ByName），名称可以通过name属性进行指定。如果没有指定name属性，当注解写在字段上时，默认取字段名进行按照名称查找，如果注解写在setter方法上默认取属性名进行装配。当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。
- 它们的作用相同都是用注解方式注入对象，但执行顺序不同。@Autowired先byType，@Resource先byName。

---

9. 使用注解开发

工程6：spring-06-anno

在Spring4之后，如果要使用注解开发必须保证aop的包导入成功；



使用注解之前需要导入context约束，增加注解的支持；

```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/beans/spring-context.xsd">

    <context:annotation-config/>

</beans>
```

9.1 Bean

【参照前文章节8】

9.2 属性如何注入

```java
/**
 * @Component 注解等价于在applicationContext.xml文件中注册，即：<bean id="user" class="com.abraham.pojo.User"/>
 * @Component 组件类
 */
@Component
public class User {

    // 相当于：<bean id="user" class="com.abraham.pojo.User">
    //           <property name="name" value="long"/>
    //        </bean>
    @Value("long")
    public String name;

    @Value("abraham")
    public void setName(String name) {
        this.name = name;
    }
}
```

@Value注解可以给属性赋值，在属性上和set方法上使用均可；

9.3 衍生的注解

@Component（pojo）的衍生注解，我们在web开发中会按照mvc三层架构分层：

- dao 【@Repository】
- service 【@Service】 等价于Impl实现类
- controller 【@Controller】

以上这四个注解的功能是一样的，都是实现将某个类注册到Spring中，装配Bean；

9.4 自动装配

```
- @Autowired：自动装配，通过类型、名字识别；
- @Qualifier：如果@Autowired不能唯一自动装配上属性，则通过该注解的value字段赋值为false来解决即@Qulifier(value = false)；
- @Nullable：如果某字段标记了该注解，说明这个字段可以为null；
- @Resource：自动装配，通过类型、名字识别，类似于@Autowired注解，是java原生注解；
```

9.5 作用域

具体类同7.4节的内容

```
@Scope   设置作用域：singleton：为单例模式    prototype：为原型模式
```

9.6 小结

XML与注解的对比：

- xml更加万能，适用于任何场合，维护更加方便；
- 注解 如果不是自己的类就使用不了，维护相对复杂；

XML与注解的最佳实践：

- xml用来管理Bean；
- 注解只负责完成属性的注入；
- 我们在使用的过程中只需要注意：必须让注解生效，要开启注解的支持；

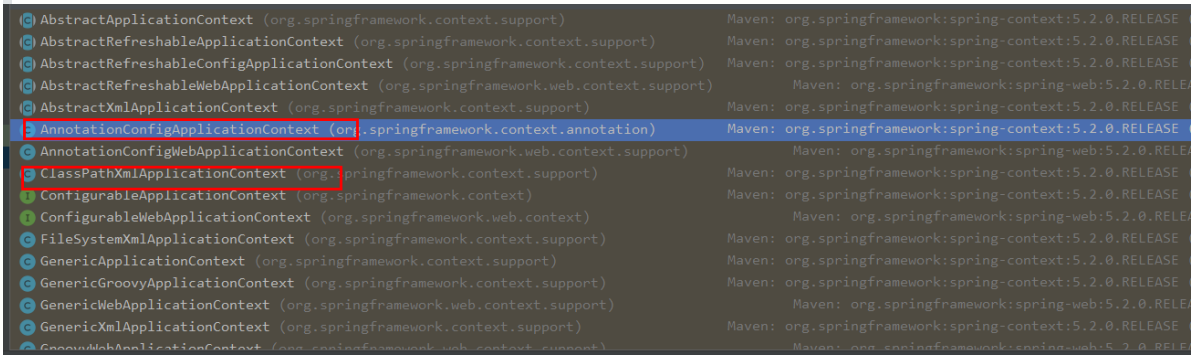
---

10. 使用java的方式配置Spring

> **工程7：spring-07-appconfig**

该章节要实现完全不使用spring的xml配置，全权交给 Java来做！！

Javaconfig是Spring的一个子项目，在Spring4 之后变成了一个核心功能。



实体类：

```java
@Component
public class User {
    private String name;

    public String getName() {
```

```
        return name;
    }

    @Value("long")  // 注入值
    public void setName(String name) {
        this.name = name;
    }

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

```

配置类：

```
@Configuration  // 代表这是一个配置类，和之前学习的bean.xml是一样的
                // 这个注解也会被Spring容器托管，注册到容器中，因为它本来就是一个Component
@ComponentScan("com.abraham.pojo")
@Import(LongConfig2.class)  // 把另外一个配置类导进来可以一起用
public class LongConfig {

    // 注册一个Bean，相当于一个Bean标签，这个方法的名字就是相当于bean标签中的id属性，这个方法的返回值就相当于bean标签中的class属性
    @Bean
    public User user(){ // 方法名等于Bean的id
        return new User();  //返回要注入的bean对象
    }
}
```

测试类：

```
public class MyTest07 {
    public static void main(String[] args) {
        // 如果完全使用了配置类方式去做，我们就只能通过AnnotationConfigApplicationContext上下文来获取容器，通过配置类的class对象加载
        ApplicationContext context = new AnnotationConfigApplicationContext(LongConfig.class);
        User user = (User) context.getBean("user");
        System.out.println(user.getName());
    }
}
```

---

11. 代理模式

> **工程8：spring-08-proxy**

为什么要学习代理模式？　因为这就是Spring AOP的底层。　【Spring AOP】与【Spring MVC】是重点

代理模式的分类：

- 静态代理
- 动态代理

11.1 静态代理

> demo1

角色分析：

- 抽象角色：一般会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，代理真实角色后我们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：抽象角色
public interface Rent {
    public void rent();
}
```

2. 真实角色

```
// 房东：真实角色
public class Host implements Rent {

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

3. 代理角色

```
// 代理角色
public class Proxy {
    private Host host;

    public Proxy() {

    }

    public Proxy(Host host) {
        this.host = host;
    }

    // 代理房东租房
```

```
    public void rent(){
        seeHouse();
        host.rent();
        fare();
    }

    // 中介看房
    public void seeHouse(){
        System.out.println("中介带你看房");
    }

    // 收中介费
    public void fare(){
        System.out.println("中介要收中介费");
    }
}
```

4. 客户端访问代理角色

```
// 租客
public class Client {
    public static void main(String[] args) {
        // 房东要出租房子
        Host host = new Host();
//        host.rent();
        // 代理，中介帮房东出租房子，但是中介一般会有一些其他的附属属性
        Proxy proxy = new Proxy(host);
        // 客户不要面对房东，直接找中介租房
        proxy.rent();
    }
}
```

代理模式的优点：

- 可以使真实角色的操作更加纯粹，不用去关注一些公共的业务；
- 公共业务就交给代理角色，实现了业务的分工；
- 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，代码量就会翻倍，导致开发效率降低；

我们想要静态代理的好处，又不想要静态代理的缺点，所以，就有了动态代理!

11.2 静态代理再理解

> **demo2包**

1. 接口（抽象角色）

```java
// 抽象角色
public interface UserService {
    public void add();
    public void delete();
    public void update();
    public void select();
}
```

2. 真实角色

```java
// 真实角色
public class UserServiceImpl implements UserService {
    public void add() {
        System.out.println("增加了一个用户");
    }

    public void delete() {
        System.out.println("删除了一个用户");
    }

    public void update() {
        System.out.println("修改了一个用户");
    }

    public void select() {
        System.out.println("查询了一个用户");
    }
}
```

3. 代理角色

需求来了，现在我们需要增加一个日志功能，怎么实现！

- 思路1：在实现类上增加代码【麻烦】
- 思路2：使用代理来做，能够不改变原来的业务情况下，实现此功能就是最好的了！

```java
// 代理角色
public class UserServiceProxy implements UserService{

    private UserService userService;

    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    public void add() {
        log("add");
        userService.add();
    }

    public void delete() {
        log("delete");
        userService.delete();
    }

    public void update() {
        log("update");
```

```
        userService.update();
    }

    public void select() {
        log("select");
        userService.select();
    }

    // 日志方法
    public void log(String msg){
        System.out.println("[Debug] 使用了" + msg + "方法! ");
    }
}
```

4. 用户

```
// 用户
public class Customer {
    public static void main(String[] args) {
        UserService userService = new UserServiceImpl();
//        userService.delete();
        UserServiceProxy proxy = new UserServiceProxy();
        proxy.setUserService(userService);

        proxy.delete();

        proxy.add();
    }
}
```

我们在不改变原来的代码的情况下，实现了对原有功能的增强，这是AOP中最核心的思想

聊聊AOP：纵向开发，横向开发

11.3 动态代理

> **demo03、demo04包**

- 动态代理和静态代理的角色一样；
- 动态代理的代理类是动态生成的，不是我们直接写好的；
- 动态代理分为两大类：基于接口的动态代理和基于类的动态代理；
  - 基于接口—— JDK 动态代理 【这里学习的】
  - 基于类—— cglib
  - Java字节码实现—— Javassist：简单、快速、直接使用Java编码的形式

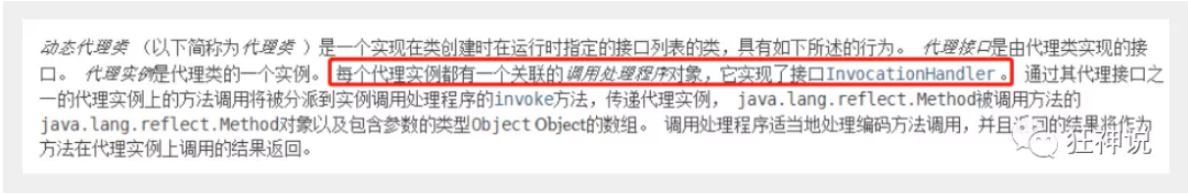
JDK的动态代理需要了解两个类

核心 : InvocationHandler 和 Proxy ， 打开JDK帮助文档看看

【InvocationHandler：调用处理程序】

```
Object invoke(Object proxy, 方法 method, Object[] args);
//参数
//proxy - 调用该方法的代理实例
//method -所述方法对应于调用代理实例上的接口方法的实例。方法对象的声明类将是该方法声明的接口，它可以是代理类继承该方法的代理接口的超级接口。
//args -包含的方法调用传递代理实例的参数值的对象的阵列，或null如果接口方法没有参数。原始类型的参数包含在适当的原始包装器类的实例中，例如java.lang.Integer或java.lang.Boolean 。
```

【Proxy：代理】




```
//生成代理类
public Object getProxy(){
    return Proxy.newProxyInstance(this.getClass().getClassLoader(),
                                  rent.getClass().getInterfaces(),this);
}
```

【重点】 理解反射机制（补学）

动态代理的好处：

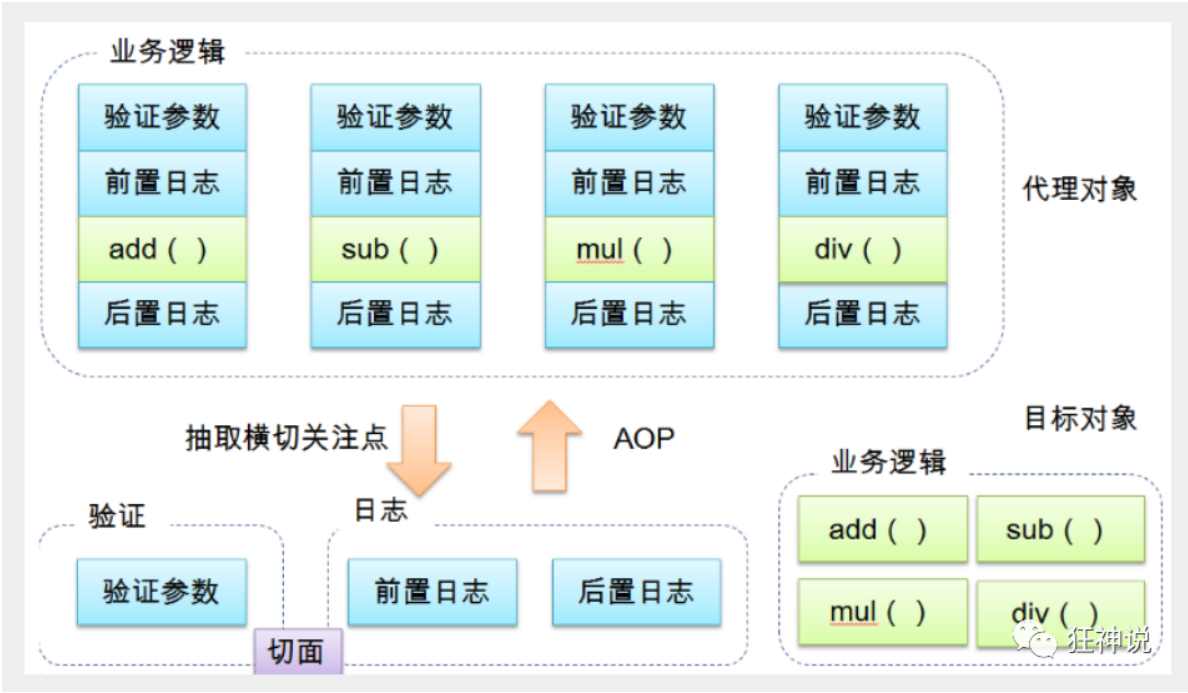
- 可以使真实角色的操作更加纯粹，不用去关注一些公共的业务；
- 公共业务就交给代理角色，实现了业务的分工；
- 公共业务发生扩展的时候，方便集中管理；
- 一个动态代理类代理的是一个接口，一般就是一类业务；
- 一个动态代理类可以代理多个类，只要是实现了同一个接口即可；

---

12. AOP面向切面编程

11.1 什么是AOP

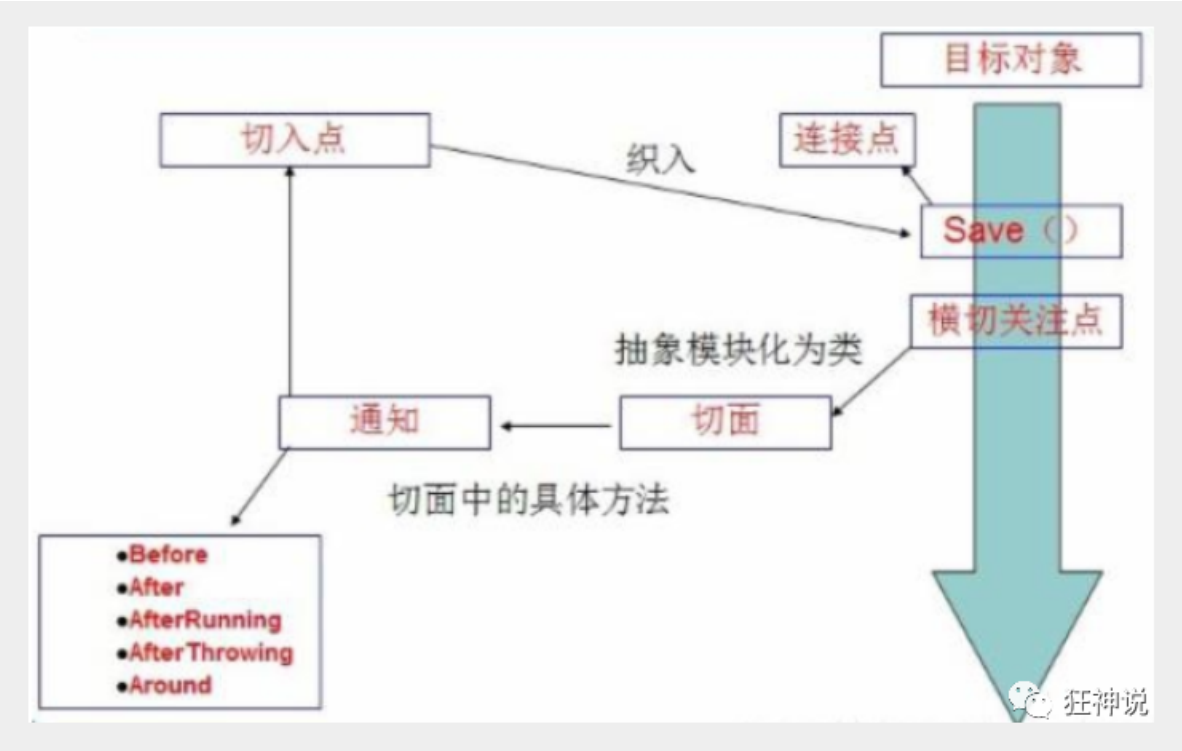
AOP意为：**面向切面编程**，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。



11.2AOP在Spring中的作用

提供声明式事物，允许用户自定义切面

- 横向关注点：：跨越应用程序多个模块的方法或功能。通常是业务逻辑的实现功能，如日志，安全，缓存、事务等等；
- 切面：横切关注点被模块化的特殊对象。通常是一个类；
- 通知：切面必须要完成的工作。通常是这个类中的方法；
- 目标：通常是方法的对象；
- 代理：向目标对象应用通知之后创建的对象。
- 切入点：切面通知 执行的"地点"的定义。
- 连接点：与切入点匹配的执行点。



11.3 使用Spring实现AOP

【重点】导入Spring AOP的依赖包

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.6</version>
</dependency>
```

方式一：使用Spring的API 接口 【主要是Spring API接口实现】【最强大的】

配置文件：

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

    <!--方式一：使用原生的Spring API 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 （ * * * * * *）
        -->
        <aop:pointcut id="pointcut" expression="execution(*
com.abraham.service.UserServiceImpl.*(..))"/>
        <!--执行环绕增加-->
        <aop:advisor advice-ref="log" pointcut-ref="pointcut"/>
        <aop:advisor advice-ref="afterLog" pointcut-ref="pointcut"/>
    </aop:config>
```

代理文件：

```
public class BeforeLog implements MethodBeforeAdvice {

    // method：要执行的目标对象的方法
    // objects/args：参数
    // o/target：目标对象
    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println(target.getClass().getName() + "的" + method.getName() + "被执行了！");

    }
}
```

```
public class AfterLog implements AfterReturningAdvice {

    // object/returnValue:返回值
    public void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws Throwable {
        System.out.println("执行了 " + method.getName() + "方法，返回的结果为：" + returnValue);
    }
}

```

测试文件：

```
public class MyTest {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        // 动态代理代理的是一个接口（注意）
        UserService userService = (UserService) context.getBean("userService");

        userService.select();
    }
}
```

- 在使用spring框架配置AOP的时候，不管是通过XML配置文件还是注解的方式都需要定义pointcut”切入点”

  例如定义切入点表达式 **execution (* com.sample.service.impl..*. *(..))**

- execution()是最常用的切点函数，其语法如下所示：

  整个表达式可以分为五个部分：

  1. **execution()**：表达式主体；
  2. **第一个 * 号**：表示返回类型， *号表示所有的类型；
  3. **包名**：表示需要拦截的包名，后面的两个句点表示当前包和当前包的所有子包，com.sample.service.impl包、子孙包下所有类的方法；
  4. **第二个 * 号**：表示类名，*号表示所有的类；
  5. ***(..)**：最后这个星号表示方法名，*号表示所有的方法，后面括弧里面表示方法的参数，两个句点表示任何参数；

方式二：使用自定义类来实现AOP 【主要是切面的定义】

配置文件：

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

代理文件：

```
public class DiyPointCut {

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

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

测试文件：**同方式一；**

方式三：使用注解实现

注解支持：

```
<?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: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/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">
```

注解文件：

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

自定义的代理文件：

```java
// 使用注解方式实现AOP
@Aspect // 该注解标注这个类是一个切面
public class AnnotationPointCut {

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

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

    // 在环绕增强中，我们可以给定一个参数，代表我们要获取处理切入的点
    @Around("execution(* com.abraham.service.UserServiceImpl.*(..))")
    public void around(ProceedingJoinPoint pjp){
        System.out.println("========环绕前========");

        // 获得签名,返回被执行的方法名称
        Signature signature = pjp.getSignature();
        System.out.println("  " + signature);

        // 执行方法
        try {
            Object proceed = pjp.proceed();
//            System.out.println("###" + proceed);
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }

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

---

13. 整合Mybatis

>

步骤:

- 导入相关的jar包
  - junit
  - MyBatis
  - MySQL数据库
  - spring相关
  - AOP织入
  - MyBatis-spring【new】

```xml
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
```

```
            <version>4.12</version>
            <scope>test</scope>
        </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.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.1.9.RELEASE</version>
        </dependency>
        <!--Spring操作数据库需要一个spring-jdbc-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.1.9.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.9.6</version>
        </dependency>

        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>2.0.2</version>
        </dependency>
    </dependencies>
```

- 编写配置文件
- 测试

12.1 回忆MyBatis

1. 编写实体类；
2. 编写核心配置文件；
3. 编写接口；
4. 编写Mapper.xml；
5. 测试；

12.2 MyBatis-Spring

- **实现方法一**

1. 编写数据源；

```
<!--DataSource(数据源):使用Spring的数据源替换MyBatis的配置 c3p0 bdcp druid-->
<!--这里我们使用Spring提供的jdbc:
org.springframework.jdbc.datasource.DriverManagerDataSource-->
<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?
useSSL=true&amp;useUnicode=true&amp;characterEncoding=utf-8"/>
    <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" />
    <!--绑定MyBatis的配置文件：mybatis-config-->
    <property name="configLocation" value="classpath:mybatis-
config.xml"/>
    <property name="mapperLocations"
value="classpath:com/abraham/mapper/*.xml"/>
</bean>
```

3. SqlSessionTemplate；

```
<!--SqlSessionTemplate：就是我们使用的sqlSession-->
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
    <!--只能使用构造器注入sqlSessionFactory，因为它没有set方法-->
    <constructor-arg index="0" ref="sqlSessionFactory"/>
</bean>
```

4. 需要给接口增加实现类；

```
public class UserMapperImpl implements UserMapper {

    // 在原来我们的所有操作都使用sqlSession来执行，现在都使用
SqlSessionTemplate；
    private SqlSessionTemplate sqlSession;

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

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

```
}
```

5. 将自己写的实现类注入到Spring中；

```
<?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">

    <import resource="spring-dao.xml"/>

    <!--方式一 -->
    <bean id="userMapper" class="com.abraham.mapper.UserMapperImpl">
        <property name="sqlSession" ref="sqlSession"/>
    </bean>

</beans>
```

6. 测试使用即可；

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

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

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

- **实现方法二**

1. 编写数据源；【同上】
2. SqlSessionFactory；【同上】
3. SqlSessionTemplate；【同上】
4. 修改实现类；

```
public class UserMapperImplTwo extends SqlSessionDaoSupport implements
UserMapper {
    public List<User> selectUser() {
//        SqlSession sqlSession = getSqlSession();
//        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
//        return mapper.selectUser();
        return getSqlSession().getMapper(UserMapper.class).selectUser();
    }
}
```

5. 修改bean配置的注入；

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

```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">

    <import resource="spring-dao.xml"/>

    <!--方式一 -->
    <bean id="userMapper" class="com.abraham.mapper.UserMapperImpl">
        <property name="sqlSession" ref="sqlSession"/>
    </bean>

    <!--方式二-->
    <bean id="userMapperTwo"
class="com.abraham.mapper.UserMapperImplTwo">
        <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
    </bean>
</beans>
```

6. 修改Test类;

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

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

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

总结：整合到spring以后可以完全不要mybatis的配置文件，除了这些方式可以实现整合之外，我们还可以使用注解来实现，这个等我们后面学习SpringBoot的时候还会测试整合！

---

14. 声明式事务

14.1 回顾事务

- 把一组业务当成一个业务来做；要么成功，要么都失败；
- 事务在项目开发中，十分的重要，涉及到数据一致性的问题，不能马虎；
- 确保完整性和一致性；

事务的ACID原则：

- 原子性（atomicity）
  - 事务是原子性操作，由一系列动作组成，事务的原子性确保动作要么全部完成，要么完全不起作用;
- 一致性（consistency）
  - 一旦所有事务动作完成，事务就要被提交。数据和资源处于一种满足业务规则的一致性状态中;
- 隔离性（isolation）

    - 可能多个事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏;
- 持久性（durability）
    - 事务一旦完成，无论系统发生什么错误，结果都不会受到影响。通常情况下，事务的结果被写到持久化存储器中;

14.2 spring中的事务管理

Spring在不同的事务管理API之上定义了一个抽象层，使得开发人员不必了解底层的事务管理API就可以使用Spring的事务管理机制。Spring支持编程式事务管理和声明式的事务管理。

编程式事务管理

- 将事务管理代码嵌到业务方法中来控制事务的提交和回滚
- 缺点：必须在每个事务操作业务逻辑中包含额外的事务管理代码

声明式事务管理

- 一般情况下比编程式事务好用。
- 将事务管理代码从业务方法中分离出来，以声明的方式来实现事务管理。
- 将事务管理作为横切关注点，通过aop方法模块化。Spring中通过Spring AOP框架支持声明式事务管理。

使用Spring管理事务，注意头文件的约束导入 : tx

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

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

事务管理器

- 无论使用Spring的哪种事务管理策略（编程式或者声明式）事务管理器都是必须的。
- 就是 Spring的核心事务管理抽象，管理封装了一组独立于技术的方法。

JDBC事务

```
<!--配置声明式事务-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <constructor-arg ref="dataSource" />
</bean>
```

配置好事务管理器后我们需要去配置事务的通知

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!--给哪些方法配置事务-->
    <!--配置事务的传播特性：new propagation-->
    <tx:attributes>
        <tx:method name="add" propagation="REQUIRED"/>
        <tx:method name="delete" propagation="REQUIRED"/>
        <tx:method name="update" propagation="REQUIRED"/>
        <tx:method name="select" read-only="true"/>
        <tx:method name="*" propagation="REQUIRED"/>   <!--所有方法-->
    </tx:attributes>
</tx:advice>
```

声明式事务：配置AOP

```
<!--配置事务的切入-->
<aop:config>
    <aop:pointcut id="txPointCut" expression="execution(*
com.abraham.mapper.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut"/>
</aop:config>
```

配置文件：

```

<!--结合AOP实现事务的织入-->
<!--配置事务通知：-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!--给哪些方法配置事务-->
    <!--配置事务的传播特性：new propagation-->
    <tx:attributes>
        <tx:method name="add" propagation="REQUIRED"/>
        <tx:method name="delete" propagation="REQUIRED"/>
        <tx:method name="update" propagation="REQUIRED"/>
        <tx:method name="select" read-only="true"/>
        <tx:method name="*" propagation="REQUIRED"/>    <!--所有方法-->
    </tx:attributes>
</tx:advice>

<!--配置事务的切入-->
<aop:config>
    <aop:pointcut id="txPointCut" expression="execution(*
com.abraham.mapper.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut"/>
</aop:config>
```

配置事务的传播特性：propagation

一、在声明式的事务处理中，要配置一个切面，其中就用到了propagation，表示打算对这些方法怎么使用事务，是用还是不用，其中propagation有七种配置，REQUIRED、SUPPORTS、MANDATORY、REQUIRES_NEW、NOT_SUPPORTED、NEVER、NESTED。默认是REQUIRED。

二、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。

为什么需要事务?

- **如果不配置事务，可能存在数据提交不一致的问题；**
- **如果我们不在spring中配置声明式事务，我们就需要在代码中手动配置；**
- **事务在项目的开发中十分的重要，涉及到数据的一致性和完整性；**