

# Spring 学习笔记

## 一、Spring 的基本概念

1、Spring 是一种轻量级的开源框架。

2、Spring 核心的两部分

aop: 面向切面编程，简单理解就是扩展功能不是修改源代码实现；

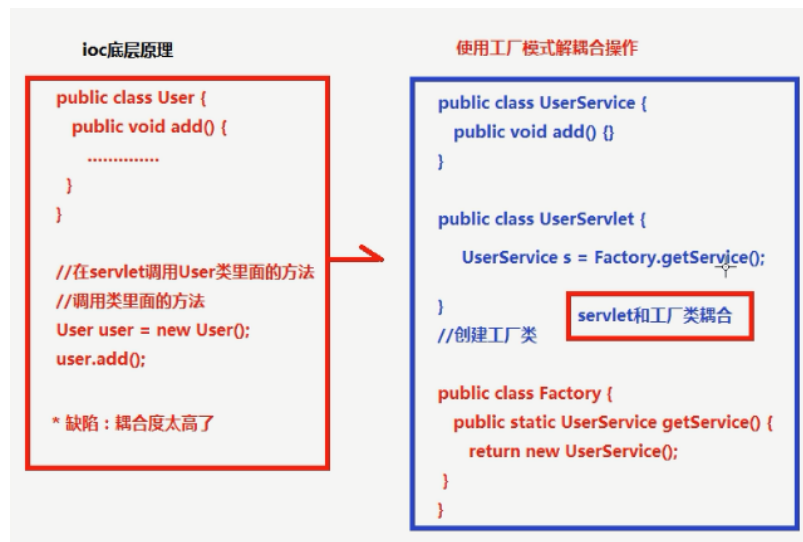
ioc: 控制反转，将对象的管理交给容器，主体程序需要的资源，只要调用就可以。

3、Spring 是一个一站式框架

Spring 在 javaee 三层结构中,为每一层都提供了不同解决方案: web 层: SpringMVC;  
service 层(业务逻辑层):Spring 的 ioc;dao 层(数据操作层):spring 的 jdbcTemplate。

## 二、Spring 的 ioc 原理

传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，并且需要负责依赖对象的整个生命周期，难于测试。如下图所示是传统应用程序与依赖对象之间的关系图，一种过渡的解决方案就是使用工厂模式，但是它依然存在缺点：工厂一般以单例模式出现，并且应用程序与工厂之间又会产生新的耦合关系。



有了 IoC 容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

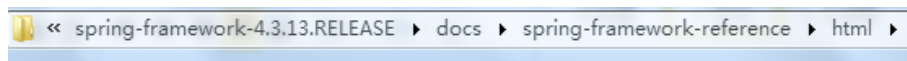


ioc 的底层原理，包括：xml 配置文件，dom4j 解析 xml 文件，工厂设计模式，反射机制。

### 三、编写 Spring 的配置文件

1、位置和文件名没有固定的要求。官方建议放在 `src` 目录下：名称为：`applicationContext.xml`。

2、引入 `schema` 约束，这个约束可以在 `spring` 包的 `doc` 目录下找：



该目录下最后一个文件，打开，翻到最后：

```
<?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="User" class="com.zxt.domain.User"></bean>

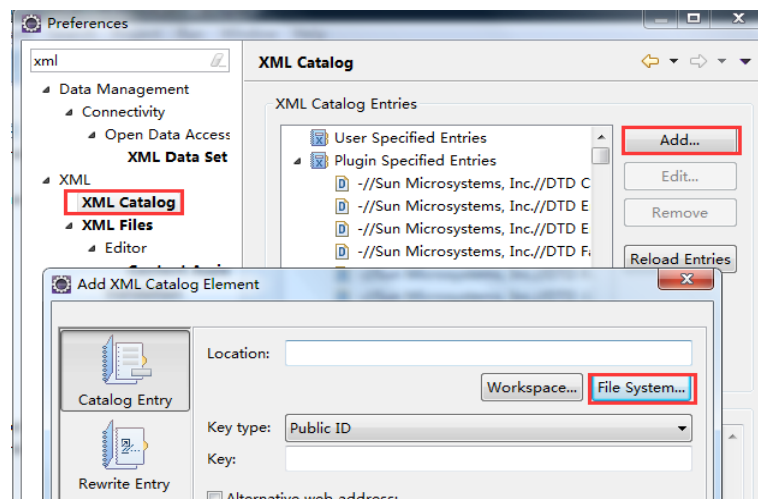
</beans>
```

加载配置文件的方式：

```
ApplicationContext context = new ClassPathXmlApplicationContext(
    "applicationContext.xml"); // 推荐使用（通过类路径）
```

```
ApplicationContext context = new FileSystemXmlApplicationContext(
    "src/applicationContext.xml"); // 通过文件路径
```

3、在 `eclipse` 下编写配置文件一般都会有相应的提示信息，如果没有则可以引入 `schema` 约束文件到 `eclipse` 中：（约束文件在 `spring` 包的 `schema` 文件夹下去找。）



## 四、Spring 的 bean 管理（xml 方式）

### 1、Bean 实例化的方式

1、在 spring 中通过配置文件创建对象

2、bean 实例化的三种方式

第一种：使用类的无参数构造方法创建（主要使用该方法），若对象没有无参构造函数，则会报错。

```
// getBean("User"): 默认使用类的无参构造方法构造类的对象
```

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

第二种：使用静态工厂创建：创建一个静态的工厂类方法，由该工厂类来实例化类的对象（bean 里面需要配置 **factory** 类的路径，以及创建对象的静态方法名：**factory-method=""**）。

第三种：使用实例工厂创建：创建一个非静态工厂类方法，由该工厂的实例化对象来创建类的对象（首先需要配置一个工厂类的实例对象，然后配置类的 **factory-bean=""** 为该工厂类的实例对象）。

### 2、Bean 的生命周期

1、实例化(当我们的程序加载 **applicationContext.xml** 文件)，把我们的 bean(前提是 **scope=singleton**)实例化到内存。

2、调用 **set** 方法设置属性。

3、如果你实现了 bean 名字关注接口(**BeanNameAware**)，则可以通过 **setBeanName** 获取 id 号。

4、如果你实现了 bean 工厂关注接口，(**BeanFactoryAware**)，则可以获取 **BeanFactory**。

5、如果你实现了 **ApplicationContextAware** 接口，则调用方法：

```
// 该方法传递 ApplicationContext
```

```
public void setApplicationContext(ApplicationContext arg0)
```

```
    throws BeansException {
```

```
    // TODO Auto-generated method stub
```

```
    System.out.println("setApplicationContext"+arg0);
```

```
}
```

6、如果 bean 和一个后置处理器关联，则会自动去调用 `Object postProcessBeforeInitialization` 方法。

7、如果你实现 `InitializingBean` 接口，则会调用 `afterPropertiesSet`。

8、如果自己在 `<bean init-method="init"/>` 则可以在 bean 定义自己的初始化方法。

9 如果 bean 和一个后置处理器关联，则会自动去调用 `Object postProcessAfterInitialization` 方法。

10、使用我们的 bean。

11、容器关闭。

12、可以通过实现 `DisposableBean` 接口来调用方法 `destory`。

13、可以在 `<bean destory-method="fun1"/>` 调用定制的销毁方法。

### 3、Bean 标签的常用属性

1、id 属性：类实例的名称，命名规则与一般的变量一样，在代码中可以根据 id 值来获取配置的对象。

2、class 属性：创建对象所在类的全路径。

3、name 属性：功能和 id 一样。一个区别：id 属性的值不可以包含特殊符号，而 name 值可以包含特殊符号（例如下划线等）。name 属性为旧版本中的使用方式，现在基本不用，而使用 id。

4、scope 属性：表示对象的范围。

scope 属性常用的取值：**singleton（默认值）：单例**。单例模式下 spring 框架初始化时就会实例化配置的对象并加载到内存。

```
<bean id="User" class="com.zxt.domain.User"></bean> 或者
```

```
<bean id="User" class="com.zxt.domain.User" scope="singleton"></bean>
```

```
// getBean("User"): 默认使用类的无参构造方法构造类的对象
User user = (User) context.getBean("User");
user.setName("zhangjike");
```

```
// scope属性默认值为 singleton单例模式，所以得到的两个对象其实是一样的
User user2 = (User) context.getBean("User");
```

```
System.out.println(user);
System.out.println(user2);
```

```
com.zxt.domain.User@4de8b406  
com.zxt.domain.User@4de8b406
```

prototype: 多例，多实例的应用场景：例如 action。

```
<bean id="User" class="com.zxt.domain.User" scope="prototype"></bean>
```

```
// getBean("User"): 默认使用类的无参构造方法构造类的对象  
User user = (User) context.getBean("User");  
user.setName("zhangjike");
```

```
// scope属性默认值为 singleton单例模式，所以得到的两个对象其实是一样的  
User user2 = (User) context.getBean("User");
```

```
System.out.println(user);  
System.out.println(user2);
```

```
com.zxt.domain.User@5afa04c  
com.zxt.domain.User@6ea12c19
```

request: 创建对象，并将对象放到 requests 域中

session: 创建对象，并将对象放到 session 域中

globalSession: 创建对象，并将对象放到 session 域中

## 4、Bean 属性注入

Java 程序中属性注入的方式：1、使用 set 方式；2、有参数构造函数；3、接口注入（不常用）。

第一种 使用set方法注入	第二种 有参数构造注入	第三种 使用接口注入
<pre>public class User {     private String name;     public void setName(String name) {         this.name = name;     } } User user = new User(); user.setName("abcd");</pre>	<pre>public class User {     private String name;     public User(String name) {         this.name = name;     } } User user = new User("lucy");</pre>	<pre>public interface Dao {     public void delete(String name); } public class DaoImpl implements Dao {     private String name;     public void delete(String name) {         this.name = name;     } }</pre>

在 Spring 框架中只支持前两种方式，即使用 set 方法（使用最多），和有参构造。

### 1、有参构造

```

public class Demo {

    private String name;

    public Demo(String name) {
        this.name = name;
    }

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

<!-- 使用有参构造函数进行属性注入 -->
<bean id="demo" class="com.zxt.domain.Demo">
    <constructor-arg name="name" value="pipixia"></constructor-arg>
</bean>

Demo demo = (Demo) context.getBean("demo");
demo.sayHello();

```

Hello, pipixia

## 2、set 方法注入

```

private String name;
private double price;

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

public void setPrice(double price) {
    this.price = price;
}

public void message() {
    System.out.println("bookname is " + name + " and price is " + price);
}

<!-- 使用set方法注入属性 -->
<bean id="book" class="com.zxt.domain.Book">
    <!-- name: 为对象中属性的名称, value: 需要为对象属性赋的值 -->
    <property name="name" value="易筋经"></property>
    <property name="price" value="30"></property>
</bean>

Book book = (Book) context.getBean("book");
book.message();

```

bookname is 易筋经 and price is 30.0

## 3、注入对象类型属性

(1)、首先在类中定义一个需要被注入的对象的属性，并创建其 `set` 方法，使用 `set` 方法注入该对象。

```
public class Book {  
    private String name;  
    private double price;  
    private User user;  
  
    public void setUser(User user) {  
        this.user = user;  
    }  
  
    public void message() {  
        System.out.println("bookname is " + name + " and price is " + price + " author is " + user.getName())  
    }  
}
```

(2)、注入对象类型的属性的配置。`Book` 和 `User` 都是一个类，所以首先需要配置 `Book` 和 `User` 的对象。

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

```



```

public class Person {

    private String[] args;
    private List<String> list;
    private Map<String, String> map;
    private Properties properties;

    public void getPersonMessage() {
        System.out.println("args: " + args);
        System.out.println("list: " + list);
        System.out.println("map: " + map);
        System.out.println("properties: " + properties);
    }
}

```

<!-- 配置复杂类型属性 -->

```
<bean id="person" class="com.zxt.domain.Person">
```

<!-- 数组 -->

```
<property name="args">
```

```
<list>
```

```
<value>小王</value>
```

```
<value>小马</value>
```

```
<value>小刘</value>
```

```
</list>
```

```
</property>
```

<!-- List -->

```
<property name="lsit">
```

```
<list>
```

```
<value>大众</value>
```

```
<value>宝马</value>
```

```
<value>奔驰</value>
```

```
</list>
```

```
</property>
```

<!-- Map -->

```
<property name="map">
```

```
<map>
```

```
<entry key="aa" value="aav"></entry>
```

```
<entry key="bb" value="bbv"></entry>
```

```
<entry key="cc" value="ccv"></entry>
```

```
</map>
```

```
</property>
```

<!-- Properties -->

```
<property name="properties">
```

```
<props>
```

```
<prop key="username" >root</prop>
```

```

        <prop key="password" >root</prop>
    </props>
</property>
</bean>

    Person person = (Person) context.getBean("person");
    person.getPersonMessage();

args: [Ljava.lang.String;@192b07fd
list: [大众, 宝马, 奔驰]
map: {aa=aav, bb=bbv, cc=ccv}
properties: {password=root, username=root}|

```

## 5、继承配置

```
public class Student
```

```
public class Grdate extends Student
```

在 beans.xml 文件中体现配置

```
<!-- 配置一个学生对象 -->
```

```
<bean id="student" class="com.hsp.inherit.Student">
```

```
    <property name="name" value="顺平" />
```

```
    <property name="age" value="30"/>
```

```
</bean>
```

```
<!-- 配置 Grdate 对象 -->
```

```
<bean id="grdate" parent="student" class="com.hsp.inherit.Grdate">
```

```
    <!-- 如果自己配置属性 name, age, 则会替换从父对象继承的数据 -->
```

```
    <property name="name" value="小明"/>
```

```
    <property name="degree" value="学士"/>
```

```
</bean>
```

## 五、Spring 的 bean 管理（注解方式）

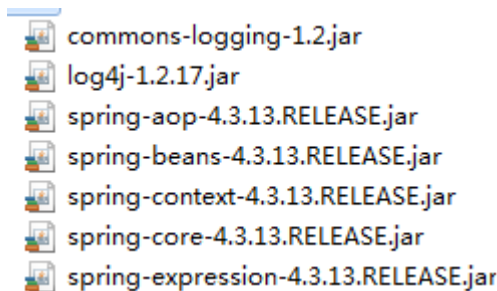
### 1、注解介绍

- 1、代码里面的特殊标记，使用注解可以完成某些功能。
- 2、注解的语法：`@注解名称(属性名称=属性值)`。
- 3、注解可以使用在类、方法、属性上面。

### 2、注解开发的准备

需要注意的是：注解开发可以减少配置文件的编写，提高效率，但是注解并不能完全代替配置文件，配置文件里面仍然需要一些配置。

- 1、注解开发除了需要导入基本的 spring 的 jar 包，还需要导入 aop 包：



- 2、编写 spring 配置文件，引入约束：

基本的 spring 开发，需要引入 beans 约束，要实现注解开发，还需要引入 context 约束。

#### 41.2.8 the context schema

The `<context>` tags deal with `ApplicationContext` configuration that relates to plumbing - that is, not usually beans that are important to an application but beans that do a lot of grunt work in Spring, such as `BeanFactoryPostProcessors`. The following snippet references the correct schema so that the namespace are available to you.

```
<?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">
```

- 3、在配置文件中，不需要再编写其他内容，只需要配置开启注解扫描即可：

```
<context:component-scan base-package="com.zxt"></ context:component-scan >
```

需要注意：当有多个包需要扫描时，可以使用逗号分开，也可以使用包的相同的前缀即可，例如 `com`。上面的配置会扫描类上，方法上，属性上的所有注解。

还有如下一个不常用的配置：（它只会扫描属性上的注解。）

```
<context:annotation-config></ context:annotation-config>
```

### 3、常用的注解

Spring 管理对象有: `@Component`、`@Controller`、`@Service`、`@Repository` 四个注解。目前它们的功能是一样的, `@Controller` 用在 web 层; `@Service` 用在业务层; `@Repository` 用在持久层。其中 `@Component` 是所有受 Spring 管理组件的通用形式, `@Component` 注解可以放在类的头上, `@Component` 不推荐使用。

#### 1、@Component

`@Component`: 是所有受 Spring 管理组件的通用形式, `@Component` 注解可以放在类的头上, `@Component` 不推荐使用。

#### 2、@Controller

`@Controller` 对应表现层的 Bean, 也就是 Action, 例如:

```
@Controller
@Scope("prototype")
public class UserController {

    public void controllerMethod() {
        System.out.println("我是一个controller方法");
    }
}
```

使用 `@Controller` 注解标识 `UserController` 之后, 就表示要把 `UserController` 交给 Spring 容器管理, 在 Spring 容器中会存在一个名字为 "userController" 的 Controller, 这个名字是根据 `UserController` 类名来取的。注意: 如果 `@Controller` 不指定其 `value(@Controller)`, 则默认的 bean 名字为这个类的类名首字母小写, 如果指定 `value(@Controller(value="userController"))` 或者 `(@Controller("userController"))`, 则使用 `value` 作为 bean 的名字。

这里的 `UserController` 还使用了 `@Scope` 注解, `@Scope("prototype")` 表示将类的范围声明为原型, 可以利用容器的 `scope="prototype"` 来保证每一个请求有一个单独的 Action 来处理, 避免 struts 中 Action 的线程安全问题。spring 默认 scope 是单例模式 (`scope="singleton"`)。Springmvc 的 controller 基于 servlet, 可以精确控制到方法上, 因此 controller 可以单例开发, Struts2 的 action 基于 filter, 基于类进行拦截请求的, 则是多例 prototype 开发。

```

public static void main(String[] args) {
    // 加载spring的配置文件
    context = new ClassPathXmlApplicationContext("beans.xml");

    UserController userController = (UserController) context.getBean("userController");
    userController.controllerMethod();
}

```

我是一个controller方法

### 3、@Service

@Service 对应的是业务层 Bean，例如：

```

@Service(value="userService")
public class UserService {

    public void serviceMethod() {
        System.out.println("我是一个service方法");
    }
}

```

@Service("userService")注解是告诉 Spring，帮我创建一个 UserService 的实例，bean 的名字必须叫做"userService"。这样当其他类需要使用 UserService 的实例时，就可以由 Spring 创建好的"userService"，然后注入给它：在类只需要声明一个 UserService 类型的属性变量来接收由 Spring 注入的"userService"即可，具体代码如下：

```

@Controller
@Scope("prototype")
public class UserController {

    @Resource(name="userService")
    public UserService userService;

    public void controllerMethod() {
        System.out.println("我是一个controller方法");
        userService.serviceMethod();
    }
}

public static void main(String[] args) {
    // 加载spring的配置文件
    context = new ClassPathXmlApplicationContext("beans.xml");

    UserController userController = (UserController) context.getBean("userController");
    userController.controllerMethod();
}

```

我是一个controller方法

我是一个service方法

#### 4、@Repository

@Repository 对应数据访问层 Bean，例如：

```
@Repository(value="userDao")
public class UserDao {

    public void daoMethod() {
        System.out.println("我是一个dao方法");
    }
}
```

@Repository(value="userDao")注解是告诉 Spring，让 Spring 创建一个名字叫“userDao”的 UserDao 实例。

当 Service 需要使用 Spring 创建的名字叫“userDao”的 UserDao 实例时，就可以使用@Resource(name = "userDao")注解告诉 Spring，Spring 把创建好的 userDao 注入给 Service 即可。

#### 4、属性的注入

属性的注入方式除了上面提到的@Resource 方法，还有@Autowired 和@Qualifier 方法。

##### 1、@Autowired

@Autowired 顾名思义，就是自动装配，其作用是为了消除代码 Java 代码里面的 getter/setter 与 bean 属性中的 property。当然，getter 看个人需求，如果私有属性需要对外提供的话，应当予以保留。

@Autowired 默认按类型匹配的方式，在容器查找匹配的 Bean，当有且仅有一个匹配的 Bean 时，Spring 将其注入@Autowired 标注的变量中。

```
@Service(value="userService")
public class UserService {

    public void serviceMethod() {
        System.out.println("我是一个service方法");
    }
}
```

```

@Controller
public class UserController {

    @Autowired
    public UserService userService;

    public void controllerMethod() {
        System.out.println("我是一个controller方法");
        userService.serviceMethod();
    }
}

public static void main(String[] args) {
    // 加载spring的配置文件
    context = new ClassPathXmlApplicationContext("beans.xml");

    UserController userController = (UserController) context.getBean("userController");
    userController.controllerMethod();
}

```

我是一个controller方法

我是一个service方法

若自动装配找不到一个匹配的 bean，则会报错，例如，将 UserService 的注解去掉，则 spring 容器中就没有一个 UserService 的实例了，自然找不到。

```

public class UserService {

    public void serviceMethod() {
        System.out.println("我是一个service方法");
    }
}

```

```

ronment).

```

```

Error creating bean with name 'userController': Unsatisfied dependency expressed through field 'userService';

```

此时可以设置，当找不到匹配类型的 bean 时，显示为空（null）

```

@Controller
public class UserController {

    @Autowired(required=false)
    public UserService userService;

    public void controllerMethod() {
        System.out.println("我是一个controller方法");
        System.out.println(userService);
        // |userService.serviceMethod();
    }
}

```

我是一个controller方法

null

## 2、Qualifier（指定注入 Bean 的名称）

当使用@Autowired 注入属性，匹配到该类型的多个不同 bean 实例时，会报错，如下实例，定义了两个 Service 的实例，都继承自 MyService 接口，这样当需要获取 MyService 的对象实例时，便不知道需要获取的具体是哪个对象的实例。

```
public interface MyService {

    public void serviceMethod();

}

@Service(value="bookService")
public class BookService implements MyService {

    public void serviceMethod() {
        System.out.println("我是一个BookService方法");
    }

}

@Service(value="userService")
public class UserService implements MyService {

    public void serviceMethod() {
        System.out.println("我是一个UserService方法");
    }

}

@Controller
public class UserController {

    @Autowired
    public MyService myService;

    public void controllerMethod() {
        System.out.println("我是一个controller方法");
        myService.serviceMethod();
    }

}

public static void main(String[] args) {
    // 加载spring的配置文件
    context = new ClassPathXmlApplicationContext("anno.xml");

    UserController userController = (UserController) context.getBean("userController");
    userController.controllerMethod();

    log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvironment).
    log4j:WARN Please initialize the log4j system properly.
    log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
    Exception in thread "main" org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating
        at org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor$AutowiredFieldc
        at org.springframework.beans.factory.annotation.InjectionMetadata.inject(InjectionMetadata.java:88)
```

出现这种情况通常有两种解决办法：

(1)、删除其中一个实现类，Spring 会自动去 base-package 下寻找 MyService 接口的实现类，发现 MyService 接口只有一个实现类，便会直接引用这个实现类。



(2)、实现类就是有多个该怎么办？此时可以使用@Qualifier 注解来指定 Bean 的名称：

```
@Controller
public class UserController {

    @Autowired
    @Qualifier("bookService")
    public MyService myService;

    public void controllerMethod() {
        System.out.println("我是一个controller方法");
        myService.serviceMethod();
    }
}
```

我是一个controller方法  
我是一个BookService方法

### 3、Resource

说一下@Resource 的装配顺序：

(1)、@Resource 后面没有任何内容，默认通过 name 属性去匹配 bean，找不到再按 type 去匹配；

(2)、指定了 name 或者 type 则根据指定的类型去匹配 bean；

(3)、指定了 name 和 type 则根据指定的 name 和 type 去匹配 bean，任何一个不匹配都将报错。

然后，区分一下@Autowired 和@Resource 两个注解的区别：

(1)、@Autowired 默认按照 byType 方式进行 bean 匹配，@Resource 默认按照 byName 方式进行 bean 匹配

(2)、@Autowired 是 Spring 的注解，@Resource 是 J2EE 的注解，这个看一下导入注解的时候这两个注解的包名就一清二楚了。

Spring 属于第三方的，J2EE 是 Java 自己的东西，因此，建议使用@Resource 注解，以减少代码和 Spring 之间的耦合。

## 六、配置文件和注解混合使用

### 1、创建对象操作使用配置文件实现

```
public class BookDao {

    public void bookDaoMethod() {
        System.out.println("这是BookDao的方法");
    }
}

<!-- 开启注解模式，会自动扫描com.zxt包下的注解 -->
<context:component-scan base-package="com.zxt"></context:component-scan>
|
<!-- 配置对象 -->
<bean id="bookDao" class="com.zxt.xmlanno.BookDao"></bean>
<bean id="ordersDao" class="com.zxt.xmlanno.OrdersDao"></bean>
<bean id="bookService" class="com.zxt.xmlanno.BookService"></bean>
```

### 2、注入属性的操作使用注解的方式

```
public class BookService {

    @Resource(name="bookDao")
    private BookDao bookDao;

    @Resource(name="ordersDao")
    private OrdersDao ordersDao;

    public void bookServiceMethod() {
        System.out.println("这是BookService的方法");
        bookDao.bookDaoMethod();
        ordersDao.ordersDaoMethod();
    }
}

public static void main(String[] args) {
    // 加载spring的配置文件
    context = new ClassPathXmlApplicationContext("xmlanno.xml");

    BookService bookService = (BookService) context.getBean("bookService");
    bookService.bookServiceMethod();
}
```

这是BookService的方法

这是BookDao的方法

这是OrdersDao的方法

## 七、AOP 基础

### 1、AOP 概念

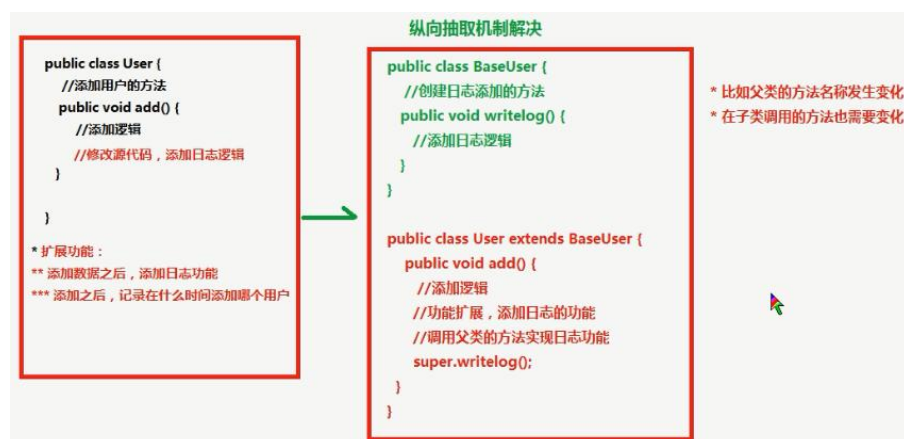
AOP 是什么(Aspect Oriented Programming), AOP 是一种编程范式, 提供从另一个角度来考虑程序结构以完善面向对象编程 (OOP)。

AOP 为开发者提供了一种描述横切关注点的机制, 并能够自动将横切关注点织入到面向对象的软件系统中, 从而实现了横切关注点的模块化。

AOP 能够将那些与业务无关, 却为业务模块所共同调用的逻辑或责任, 例如事务处理、日志管理、权限控制等, 封装起来, 便于减少系统的重复代码, 降低模块间的耦合度, 并有利于未来的可操作性和可维护性。

### 2、AOP 原理

AOP 将传统的纵向抽取机制变成横向抽取机制



### 第二种情况 没有接口情况

```
public class User {  
    public void add() {  
  
    }  
}
```

使用cglib动态代理，没有接口情况

//动态代理实现

\* 创建User类的子类的代理对象

\* 在子类里面调用父类的方法完成增强

## 3、AOP 操作术语

1、关注点：就是所关注的公共功能，比如像事务管理，就是一个关注点。表示“要做什么”。

2、连接点 (Joinpoint)：在程序执行过程中某个特定的点，通常在这些点需要添加关注点的功能，比如某方法调用的时候或者处理异常的时候。在 Spring AOP 中，一个连接点总是代表一个方法的执行。表示“在什么地方做”。简单理解：类里面可以被增强的方法，这些方法称为连接点。

3、切入点 (Pointcut)：匹配连接点的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。切入点表达式如何和连接点匹配是 AOP 的核心：Spring 缺省使用 AspectJ 切入点语法。简单理解：类里面可以被增强的方法很多，在实际操作中，实际增强的方法被称为切入点。

4、通知或增强 (Advice)：在切面的某个特定的连接点 (Joinpoint) 上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。通知表示“具体怎么做”。简单理解就是：增强的逻辑即增强，表示方法需要添加的功能。

前置通知：在方法之前执行；

后置通知：在方法之后执行；

异常通知：在方法出现异常时执行；

最终通知：在后置之后执行；

环绕通知：在方法之前和之后执行。

- 前置通知 (Before advice) :  
在某连接点之前执行的通知, 但这个通知不能阻止连接点前的执行 (除非它抛出一个异常)。
- 返回后通知 (After returning advice) :  
在某连接点正常完成后执行的通知: 例如, 一个方法没有抛出任何异常, 正常返回。
- 抛出异常后通知 (After throwing advice) :  
在方法抛出异常退出时执行的通知。
- 后通知 (After (finally) advice) :  
当某连接点退出的时候执行的通知 (不论是正常返回还是异常退出)。
- 环绕通知 (Around Advice) :  
包围一个连接点的通知, 如方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。

**5、切面/方面 (Aspect):** 一个关注点的模块化, 这个关注点可能会横切多个对象。综合表示“在什么地方, 要做什么, 以及具体如何做”。**简单理解: 把增强应用到切入点的过程称为切面。**

**6、引入 (Introduction):** 也被称为内部类型声明 (**inter-type declaration**)。为已有的类声明额外的方法或者某个类型的字段。**Spring** 允许引入新的接口 (以及一个对应的实现) 到任何被代理的对象。例如, 你可以使用一个引入来使 **bean** 实现 **IsModified** 接口, 以便简化缓存机制。

**7、目标对象 (Target Object):** 被一个或者多个切面所通知 (**advise**) 的对象。也有人把它叫做被通知 (**advised**) 对象。既然 **Spring AOP** 是通过运行时代理实现的, 这个对象永远是一个被代理 (**proxied**) 对象。

**8、织入 (Weaving):** 把切面连接到其它的应用程序类型或者对象上, 并创建一个被通知的对象的过程。也就是说织入是一个过程, 是将切面应用到目标对象从而创建出 **AOP** 代理对象的过程。这些可以在编译时 (例如使用 **AspectJ** 编译器), 类加载时和运行时完成。**Spring** 和其他纯 **Java AOP** 框架一样, 在运行时完成织入。

**9、AOP 代理 (AOP Proxy):** **AOP** 框架使用代理模式创建的对象, 从而实现在连接点处插入通知 (即应用切面), 就是通过代理来对目标对象应用切面。在 **Spring** 中, **AOP** 代理可以用 **JDK** 动态代理或 **CGLIB** 代理实现, 而通过拦截器模型应用切面。注意: **Spring** 引入的基于模式 (**schema-based**) 风格和 **@AspectJ** 注解风格的切面声明, 对于使用这些风格的用户来说, 代理的创建是透明的。

```

public class User {

    public void add() { }

    public void update() { }

    public void delete() { }

    public void findAll() { }

}

```

\* 连接点：类里面哪些方法可以被增强，这些方法称为连接点

\* 切入点：在类里面可以有很多的方法被增强，比如实际操作中，只是增强了类里面add方法和update方法，实际增强的方法称为切入点

\* 通知/增强：增强的逻辑，称为增强，比如扩展日志功能，这个日志功能称为增强

前置通知：在方法之前执行

后置通知：在方法之后执行

异常通知：方法出现异常

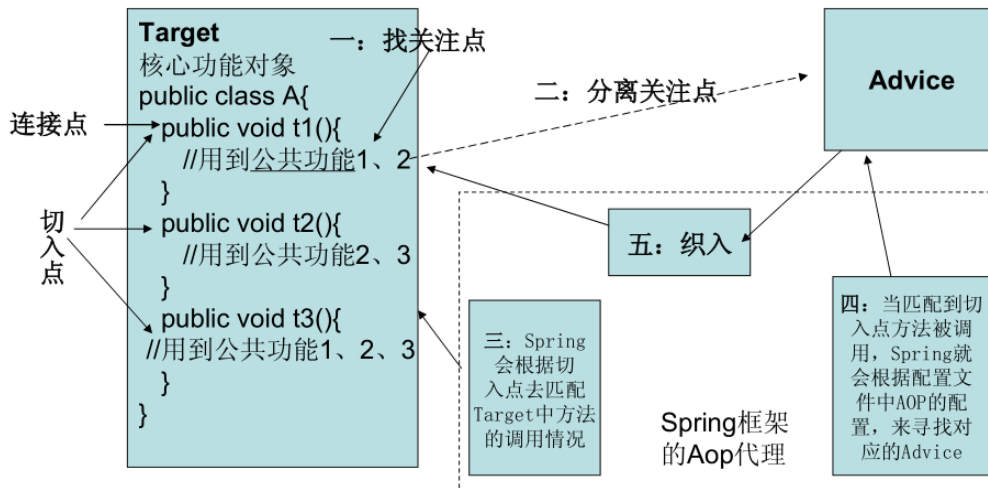
最终通知：在后置之后执行

环绕通知：在方法之前和之后执行

\* 切面：把增强应用到具体方法上面，过程称为切面

把增强用到切入点过程

## AOP基本运行流程



## 八、Spring 中的 AOP 操作

1、在 spring 里面进行 aop 操作，使用 AspectJ 实现。

(1)、AspectJ 不是 spring 的一部分，AspectJ 是一个面向切面的框架，它扩展了 Java 语言。AspectJ 定义了 AOP 语法，所以它有一个专门的编译器用来生成遵守 Java 字节编码规范的 Class 文件。和 spring 一起使用进行 aop 操作。

(2)、spring2.0 以后增加了对 AspectJ 的支持。





2、使用 AspectJ 有两种方式

(1)、基于 xml 配置文件的方法

(2)、基于 AspectJ 的注解方式

### 1、AOP 操作准备工作

1、AOP 操作除了需要导入基本的 jar 包之外，还需要导入 aop 相关的包。

 aopalliance.jar	2
 aspectjweaver-1.8.13.jar	2
 spring-aop-4.3.13.RELEASE.jar	2
 spring-aspects-4.3.13.RELEASE.jar	2

2、同样地创建 spring 的配置文件，需要导入 aop 约束。

```
<?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" <!--
       </beans>

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

    <!-- 开启注解模式，自动会扫描com.zxt包下的注解 -->
    <context:component-scan base-package="com.zxt"></context:component-scan>

</beans>
```

3、使用表达式配置切入点，切入点：实际增强的方法。

常用的表达式：

execution(<访问修饰符><返回类型><方法名>(<参数>)<异常>)

(1)、execution(\* com.zxt.Employe.addEmploye(..) )



(2)、`execution(* com.zxt.Employee.*(..) )`

(3)、`execution(* *.*(..) )`

(4)、匹配所有以 `save` 开头的方法，`execution(* save*(..) )`

如下实例：`Employee` 类为目标对象，其中的 `addEmployee` 方法为需要增强的方法，即切入点，`MyEmployee` 类中的方法用来增强 `addEmployee` 方法的逻辑，所以其中的方法为通知。

```
@Controller(value="employee")
public class Employee {

    public void addEmployee() {
        System.out.println("增加雇员的方法.....");
    }
}

@Controller(value="myEmployee")
public class MyEmployee {

    public void before() {
        System.out.println("前置通知.....");
    }
}
```

如下为配置 `aop` 的简单操作，首先就是配置切入点，使用上面说到的表达式的方法。接着配置切面（即将通知增强到方法的过程），这里可以配置不同类型的通知。

```
<!-- 配置aop操作 -->
<aop:config>
    <!-- 1、配置切入点 -->
    <aop:pointcut expression="execution(* com.zxt.aop.Employee.addEmployee(..))" id="pointcut1" />

    <!-- 2、配置切面：把增强用到方法上面的过程 -->
    <aop:aspect ref="myEmployee">
        <!-- 配置增强类型，method：表示增强类里面使用的哪个方法 -->
        <aop:before method="before" pointcut-ref="pointcut1"/>
    </aop:aspect>
</aop:config>

public static void main(String[] args) {
    // 加载spring的配置文件
    context = new ClassPathXmlApplicationContext("aop.xml");

    Employee employee = (Employee) context.getBean("employee");
    employee.addEmployee();
}
```

前置通知.....

增加雇员的方法.....



如下实例为配置一个环绕通知，环绕通知是最强大的一种通知，它会在方法执行的前后执行，所以环绕通知的方法需要额外的逻辑。

```
public void after() {
    System.out.println("后置通知.....");
}

// 环绕通知（在方法之前和方法之后都调用）
public void around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
    // 方法之前
    System.out.println("方法之前.....");

    // 需要增强的方法
    proceedingJoinPoint.proceed();

    // 方法之后
    System.out.println("方法之后.....");
}

<!-- 配置aop操作 -->
<aop:config>
    <!-- 1、配置切入点 -->
    <aop:pointcut expression="execution(* com.zxt.aop.Employee.addEmployee(..))" id="pointcut1" />

    <!-- 2、配置切面：把增强用到方法上面的过程 -->
    <aop:aspect ref="myEmployee">
        <!-- 配置增强类型，method：表示增强类里面使用的哪个方法 -->
        <aop:before method="before" pointcut-ref="pointcut1"/>

        <aop:after-returning method="after" pointcut-ref="pointcut1" />

        <aop:around method="around" pointcut-ref="pointcut1"/>
    </aop:aspect>
</aop:config>

public static void main(String[] args) {
    // 加载spring的配置文件
    context = new ClassPathXmlApplicationContext("aop.xml");

    Employee employee = (Employee) context.getBean("employee");
    employee.addEmployee();
}

- -
前置通知.....
方法之前.....
增加雇员的方法.....
方法之后.....
后置通知.....
```

## 2、Aspectj 的注解方式

首先注解开发需要导入的 jar 包，以及 spring 配置文件的约束都是同上面一样的。类似于 Spring 注解开发需要开启注解扫描，这里也需要开启 Aspectj 的注解支持。

```
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

```
<!-- 开启aop的注解支持 -->
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

然后编写目标类，以及用来增强目标类方法的类以及各种通知方法。需要注意不同种类的方法需要使用不同的配置，同时增强类也需要使用注解@Aspect(切面)。

```
@Controller(value="myTarget")
public class MyTarget {

    public void someMethod() {
        System.out.println("目标对象需要增强的方法.....");
    }
}
```

```
@Controller(value="enhance")
@Aspect
public class Enhance {

    @Before(value="execution(* com.zxt.aopanno.MyTarget.someMethod(..))")
    public void before() {
        System.out.println("前置通知.....");
    }

}
```

```
public static void main(String[] args) {
    // 加载spring的配置文件
    context = new ClassPathXmlApplicationContext("aop_anno.xml");

    MyTarget myTarget = (MyTarget) context.getBean("myTarget");
    myTarget.someMethod();
}
```

```
前置通知.....
目标对象需要增强的方法.....
```

## 九、Spring 的 jdbcTemplate

### 1、简介

1、spring 是一个一站式的框架,针对 javaee 的三层都有相应的解决技术。在 dao 层,使用 jdbcTemplate 技术。



2、spring 对不同的持久层技术都有自己的封装,如下图所示: jdbcTemplate 即 spring 对 jdbc 进行了封装。

ORM持久化技术	模板类
JDBC	org.springframework.jdbc.core.JdbcTemplate
Hibernate5.0	org.springframework.orm.hibernate5.HibernateTemplate
IBatis(MyBatis)	org.springframework.orm.ibatis.SqlMapClientTemplate
JPA	org.springframework.orm.jpa.JpaTemplate


3、jdbcTemplate 的使用和 dbutils 的使用类似。

### 2、jdbcTemplate 的 crud 操作

1、导入 jdbcTemplate 卡法所需 jar 包:

 spring-jdbc-4.3.13.RELEASE.jar	2
 spring-tx-4.3.13.RELEASE.jar	2

当然不能忘了数据库的驱动包:

 mysql-connector-java-5.1.6-bin.jar
--

2、创建对象,设置数据库信息:

```
// 设置数据信息
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql://localhost:3306/zxtblog");
dataSource.setUsername("root");
dataSource.setPassword("root");
```

3、创建 jdbcTemplate 对象,设置数据源:

```
// 创建jdbcTemplate对象,设置数据源
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
```

4、调用 `JdbcTemplate` 对象里面的方法，完成相关操作。

### 1)、添加

```
// 调用JdbcTemplate对象的方法操作数据库
// 创建SQL语句
String sql = "insert into user(name, password, purview, sex, email, tel) values(?, ?, ?, ?, ?, ?)";
int rows = jdbcTemplate.update(sql, "zxt", "123456", "admin", "男", "111@qq.com", "1111");
System.out.println(rows);
```

### 2)、修改

```
String sql = "update user set name=? where name=? ";
int rows = jdbcTemplate.update(sql, "123", "zxt");
System.out.println(rows);
```

### 3)、删除

```
String sql = "delete from user where name=? ";
int rows = jdbcTemplate.update(sql, "123");
System.out.println(rows);
```

### 4)、查询

`JdbcTemplate` 实现查询，有接口 `RowMapper`。`JdbcTemplate` 针对这个接口没有提供实现类，得到不同的类型数据需要自己封装。

如下图是 `Jdbc` 原始的操作，`RowMapper` 接口已经完成了相关的步骤，我们只需要遍历结果集，对数据进行封装。

```
try {
    Class.forName("com.mysql.jdbc.Driver");
    //创建连接
    conn = DriverManager.getConnection("jdbc:mysql:///spring_day03", "root", "root");
    //编写sql语句
    String sql = "select * from user where username=?";
    //预编译sql
    pstmt = conn.prepareStatement(sql);
    //设置参数值
    pstmt.setString(1, "lucy");
    //执行sql
    rs = pstmt.executeQuery();
    //遍历结果集
    while(rs.next()) {
        //得到返回结果值
        String username = rs.getString("username");
        String password = rs.getString("password");
        //放到user对象里面
        User user = new User();
        user.setUsername(username);
        user.setPassword(password);

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

查询的具体实现：

### 第一个：返回某一个值

```
queryForObject(String sql, Class<T> requiredType) : T - J ^
```

- (1) 第一个参数是 sql 语句
- (2) 第二个参数是返回类型的 class

```
// 查询数据表的记录数
String sql = "select count(*) from user";
int count = jdbcTemplate.queryForObject(sql, Integer.class);
System.out.println(count);
```

### 第二个：查询返回对象

- 1、首先对数据库中的表结构进行对象的封装

```
public class User {

    private int id;
    private String name;
    private String password;
    private String purview;
    private String sex;
    private String email;
    private String tel;
```

- 2、实现 RowMapper 接口

```
public class UserRowMapper implements RowMapper<User>{

    @Override
    // 第一个参数即数据库查询得到的结果集， rowNum为数据的行号
    public User mapRow(ResultSet rs, int rowNum) throws SQLException {

        // 从数据集里把数据得到
        int id = rs.getInt("id");
        String name = rs.getString("name");
        String password = rs.getString("password");
        String purview = rs.getString("purview");
        String sex = rs.getString("sex");
        String email = rs.getString("email");
        String tel = rs.getString("tel");

        // 把数据封装到对象里面
        User user = new User();
        user.setId(id);
        user.setName(name);
        user.setPassword(password);
        user.setPurview(purview);
        user.setSex(sex);
        user.setEmail(email);
        user.setTel(tel);

        return user;
    }
}
```

### 3、调用 jdbcTemplate 对象的方法实现查询

```
● queryForObject(String sql, Object[] args, RowMapper<T> rowMapper) : T
```

- (1) 第一个参数是 sql 语句
- (2) 第二个参数是 RowMapper 接口的实现类
- (3) 第三个参数是可变参数

```
// 查询返回对象
String sql = "select * from user where name=?";
User user = jdbcTemplate.queryForObject(sql, new UserRowMapper(), "admin");
System.out.println(user);
```

```
com.zxt.domain.User@3abfe836
```

### 第三个 查询返回 list 集合

```
● query(String sql, RowMapper<T> rowMapper) List<T>
```

- (1) 第一个参数是 sql 语句
- (2) 第二个参数是 RowMapper 接口的实现类，自己封装数据
- (3) 第三个参数是可变参数

```
// 查询返回对象列表
String sql = "select * from user";
List<User> list = jdbcTemplate.query(sql, new UserRowMapper());
System.out.println(list);
```

```
[com.zxt.domain.User@311d617d, com.zxt.domain.User@7c53a9eb, com.zxt.domain.User@ed17bee, com.
```

## 3、Spring 配置连接池

C3P0 是一个开源的 JDBC 连接池，它实现了数据源和 JNDI 绑定，支持 JDBC3 规范和 JDBC2 的标准扩展。目前使用它的开源项目有 Hibernate, Spring 等。

### 1、导入开发所需 jar 包：

```
c3p0-0.9.2.1.jar
mchange-commons-java-0.2.3.4.jar
```

### 2、创建 spring 配置文件，配置连接池。

以往 c3p0 连接池的代码实现如下：

```
ComboPooledDataSource dataSource = new ComboPooledDataSource();
dataSource.setDriverClass("com.mysql.jdbc.Driver");
dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/zxtblog");
dataSource.setUser("root");
dataSource.setPassword("root");
```

现在要用 spring 配置来实现：

### 1、首先配置 dataSource 数据信息

```
<!-- 配置c3p0的连接池 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <!-- 注入属性值 -->
    <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
    <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/zxtblog"></property>
    <property name="user" value="root"></property>
    <property name="password" value="root"></property>
</bean>
```

### 2、然后配置 jdbcTemplate 对象

```
<!-- 配置jdbcTemplate对象 -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

3、创建 service 和 dao，在 service 中注入 dao，在 dao 中注入 jdbcTemplate 对象，然后完成相应的数据库的操作。

```
@Service(value="userService")
public class UserService {

    @Resource(name="userDao")
    private UserDao userDao;

    // 添加用户
    public void addUser() {
        userDao.add();
    }
}

@Repository(value="userDao")
public class UserDao {

    @Resource(name="jdbcTemplate")
    private JdbcTemplate jdbcTemplate;

    // 实现一个添加用户的方法
    public void add() {
        String sql = "insert into user(name, password, purview, sex, email, tel) values(?, ?, ?, ?, ?, ?)";
        jdbcTemplate.update(sql, "zahngsan", "abcd", "common", "男", "234@qq.com", "654634");
    }
}
```

```
private static ApplicationContext context;

public static void main(String[] args) {
    context = new ClassPathXmlApplicationContext("applicationContext.xml");

    UserService userService = (UserService) context.getBean("userService");
    userService.addUser();
}
```

## 十、Spring 事务管理

### 1、什么是事务

事务(Transaction)是访问并可能更新数据库中各种数据项的一个程序执行单元(unit)。事务通常由高级数据库操纵语言或编程语言（如 SQL, C++或 Java）书写的用户程序的执行所引起，并用形如 `begin transaction` 和 `end transaction` 语句（或函数调用）来界定。事务由事务开始(begin transaction)和事务结束(end transaction)之间执行的全体操作组成。

例如：在关系数据库中，一个事务可以是一条 SQL 语句，一组 SQL 语句或整个程序。事务是恢复和并发控制的基本单位。

事务应该具有 4 个属性：原子性、一致性、隔离性、持续性。这四个属性通常称为 **ACID 特性**。

### 2、事务的特性

**原子性 (Atomicity):** 事务是一个原子操作，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):** 一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。在现实中的数据不应该被破坏。

**隔离性 (Isolation):** 可能有许多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):** 一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

## 1、Spring 中事务管理的两种方式

第一种：编程事务管理

第二种：声明式事务管理

（1）基于 xml 配置文件的方式

（2）基于注解的方式实现



## 2、Spring 事务管理的 api

Spring 事务管理的首要工作就是要配置事务管理器，PlatformTransactionManager：事务管理器接口。

Spring 框架为不同的持久层框架提供了不同的 PlatformTransactionManager 接口的实现类：

事务	说明
<a href="#">org.springframework.jdbc.datasource.DataSourceTransactionManager</a>	使用Spring JDBC或iBatis 进行持久化数据时使用
<a href="#">org.springframework.orm.hibernate5.HibernateTransactionManager</a>	使用Hibernate5.0版本进行持久化数据时使用
<a href="#">org.springframework.orm.jpa.JpaTransactionManager</a>	使用JPA进行持久化时使用
<a href="#">org.springframework.jdo.JdoTransactionManager</a>	当持久化机制是Jdo时使用
<a href="#">org.springframework.transaction.jta.JtaTransactionManager</a>	使用一个JTA实现来管理事务，在一个事务跨越多个资源时必须使用

## 3、Spring 事务管理的转账实例

1、首先在数据库中创建两个转账用的账号：

id	username	salary
1	小王	10000
2	小李	10000

```
public class AccountRowMapper implements RowMapper<Account>{

    @Override
    // 第一个参数即数据库查询得到的结果集， rowNum为数据的行号
    public Account mapRow(ResultSet rs, int rowNum) throws SQLException {

        // 从数据集里把数据得到
        int id = rs.getInt("id");
        String username = rs.getString("username");
        int salary = rs.getInt("salary");

        // 把数据封装到对象里面
        Account account = new Account();
        account.setId(id);
        account.setUsername(username);
        account.setSalary(salary);

        return account;
    }
}
```

2、创建 service 和 dao

**Service:** 业务逻辑层: 主要完成用户的逻辑;

**Dao:** 数据操作层: 单纯的操作数据, 不涉及用户业务逻辑;

转账业务: 小王转账 1000 元给小李。分为两个步骤: 小王的钱少 1000, 小李的钱多 1000, 这两个步骤是统一的, 即一个事务。

```
@Repository(value="accountDao")
public class AccountDao {

    @Resource(name="jdbcTemplate")
    private JdbcTemplate jdbcTemplate;

    // 根据账户名, 返回该账户信息
    public Account getAccount(String username) {
        String sql = "select * from account where username=?";
        Account account = jdbcTemplate.queryForObject(sql, new AccountRowMapper(), username);

        return account;
    }
}

@Repository(value="ordersDao")
public class OrdersDao {

    @Resource(name="jdbcTemplate")
    private JdbcTemplate jdbcTemplate;

    // 账户钱数减少
    public void lessMoney(Account ac, int money) {
        String sql = "update account set salary=salary-? where username=?";
        jdbcTemplate.update(sql, money, ac.getUsername());
    }

    // 账户钱数增加
    public void moreMoney(Account ac, int money) {
        String sql = "update account set salary=salary+? where username=?";
        jdbcTemplate.update(sql, money, ac.getUsername());
    }
}

@Service(value="ordersService")
public class OrdersService {

    @Resource(name="ordersDao")
    private OrdersDao ordersDao;

    @Resource(name="accountDao")
    private AccountDao accountDao;

    // 转账业务(小王向小李转账1000)
    public void accountMoney() {
        // AccountDao获取账户信息
        Account xiaowang = accountDao.getAccount("小王");
        Account xiaoli = accountDao.getAccount("小李");

        // 转账业务
        ordersDao.lessMoney(xiaowang, 1000);
        // 此过程可能发生异常, 导致一个钱少了, 但是另一个的钱却没有多, 要保证两个过程都执行, 需要使用事务
        ordersDao.moreMoney(xiaoli, 1000);
    }
}
```

解决方案：使用事务，当出现异常时，会将操作回滚。

## 4、声明式事务管理（xml 配置）

事务管理的配置分为三步：

- （1）配置事务管理器（这一步无论是在 xml 配置形式中，还是注解形式中都是必须的）。
- （2）配置事务增强（即要使用事务的方法）
- （3）配置切面（将事务使用到该方法中）

```
<!-- 第一步配置事务管理器 -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <!-- 注入dataSource -->
  <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 第二步 配置事务增强 -->
<tx:advice id="txadvice" transaction-manager="transactionManager">
  <!-- 作事务操作 -->
  <tx:attributes>
    <!-- 设置进行事务操作方法的匹配 -->
    <tx:method name="account*" />
  </tx:attributes>
</tx:advice>

<!-- 第三步 配置切面 -->
<aop:config>
  <!-- 切入点 -->
  <aop:pointcut expression="execution(* com.zxt.accountservice.OrdersService.*(..))" id="pointcut1" />

  <!-- 切面 -->
  <aop:advisor advice-ref="txadvice" pointcut-ref="pointcut1" />
</aop:config>
```

这里需要注意：mysql 的数据库引擎并不都是支持事务的，所以很有可能上面的配置完成之后，当出现异常，数据库的操作并没有回滚。所以需要对 mysql 数据库进行相应的设置。

在 sql 命令输入框输入修改数据表存储引擎的命令：ALTER TABLE 数据表名称 TYPE = INNODB； 使之支持事务操作。

## 5、声明式事务管理（注解）

首先仍然需要配置事务管理器，然后开启事务的注解

```
<!-- 第一步配置事务管理器 -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <!-- 注入dataSource -->
  <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 第二步 开启事务注解 -->
<tx:annotation-driven transaction-manager="transactionManager" />
```

再在需要开启事务管理的方法上添加注解：@Transactional

```
@Service(value="ordersService")
public class OrdersService {

    @Resource(name="ordersDao")
    private OrdersDao ordersDao;

    @Resource(name="accountDao")
    private AccountDao accountDao;

    // 转账业务(小王向小李转账1000)
    @Transactional
    public void accountMoney() {
        // AccountDao获取账户信息
        Account xiaowang = accountDao.getAccount("小王");
        Account xiaoli = accountDao.getAccount("小李");

        // 转账业务
    }
}
```

## 十一、Spring 整合 web 项目

### 1、原理

#### 1、加载 Spring 的配置文件

```
// 加载spring的配置文件
ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");
```

若每次要使用 ioc 容器里面的对象时都要加载配置文件，功能上可以实现，但是性能低下。可不可以让整个 web 项目只加载一次而所有地方都可以使用？

2、实现思路：把加载配置文件和创建对象的过程，在服务器启动的时候完成。

3、实现原理：关键技术：(1) ServletContext 对象 (2) 监听器

具体实现：在服务器启动的时候，每个项目创建一个 ServletContext 对象；

在 ServletContext 对象创建的时候，使用监听器可以具体到 ServletContext 对象是在什么时候创建的；

使用监听器监听到 ServletContext 对象创建的时候，加载 spring 的配置文件，创建配置文件中配置的对象；


把创建出来的对象放到 ServletContext 域里面（setAttribute 方法）

获取对象的时候到 ServletContext 域里面获取（getAttribute 方法）

### 2、配置

上述步骤在 spring 里面不需要自己写，spring 里面封装了一个监听器，使用的时候只需要配置即可。在 web.xml 文件中。

配置监听器之前，需要导入 spring 整合 web 项目的 jar 包。

 spring-web-4.3.13.RELEASE.jar

```
<!-- 配置监听器 -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

完成监听器的配置之后，还需要指定配置文件的位置，否则 spring 不知道去哪里加载配置文件。

```

public class ContextLoader {
    /**
     * Config param for the root WebApplicationContext id,
     * to be used as serialization id for the underlying BeanFactory: {@value}
     */
    public static final String CONTEXT_ID_PARAM = "contextId";

    /**
     * Name of servlet context parameter (i.e., {@value}) that can specify the
     * config location for the root context, falling back to the implementation's
     * default otherwise.
     * @see org.springframework.web.context.support.XmlWebApplicationContext#DEFA
     */
    public static final String CONFIG_LOCATION_PARAM = "contextConfigLocation"

<!-- 指定spring配置文件的位置 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>class:applicationContext.xml</param-value>
</context-param>

```