Spring

主讲：崔译

一、程序员的春天

> Spring 是一个 业务逻辑组件 框架， 有两大核心: IOC / AOP
>
> Spring 框架只关心一件事情
>
> Bean （对象）的生命周期
>
> 对象创建
>
> 属性赋值
>
> 方法调用
>
> 销毁
>
> 动态的代理对象
>
> 从对象的创建 到 销毁的 整个过程的任意节点，Spring 都可以参与

二、作用

用于实现程序设计的两大原则

- 高低原则

  高内聚，低耦合

  一个类中的方法与方法之间的相关度 高，一个类只关心一件事情（UserDao）

  类和类之间的耦合度 低，当一个类发生改变，不影响其他类

  ```
  class UserService{
   private UserDao userDao;
   public void add(User user){
  userDao = ObjectFactory.getObject("userDao");
   userDao.insert(user);
   }
  }
  ```

  使用接口（UserDao）分离 调用者（UserService）和 实现者（UserDaoImpl）

- 开闭原则

  对扩展开， 对修改闭

  一个类一旦完成（被打成jar包），应该容易扩展已有方法，不能修改已有代码

```
class UserServiceProxy extends UserService{
 public void add(User user){
 begin();
 super.add(user);
 commit();
 }
}
```

三、HelloWorld

1、导入依赖

```
<!--Spring 的核心包-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>${spring.version}</version>
</dependency>

<!--Spring IOC 的核心包-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-beans</artifactId>
  <version>${spring.version}</version>
</dependency>

<!--Spring 上下文，提供IOC 的注解-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>${spring.version}</version>
</dependency>
```

2、创建java类

```
public class SomeClass {
    public void a()
    {
        System.out.println("aaa");
    }
    public void b()
    {
        System.out.println("bbb");
    }
}
```

3、编写Spring配置文件

- 配置文件名，建议使用`applicationContext.xml`
- 配置文件放在`resources`文件夹中，该文件夹应该是`Resources Root`
- 创建方式：
    - 右键 -- > new
    - XML Configuration File
    - Spring Config

    （当且仅当导入Spring jar 包后，该选项才会出现）

```
<?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="sc" class="SomeClass"></bean>
</beans>
```

4、测试类

```
ApplicationContext ac = new
            ClassPathXmlApplicationContext("applicationContext.xml");
SomeClass sc = (SomeClass) ac.getBean("sc");
sc.a();
sc.b();
```

四、IOC

1、概念

`Inversion of Control`控制反转 ， 将对象的创建交给Spring ， 由Spring 管理对象的整个生命周期

2、IOC 的几种方式

2-1 使用无参构造创建对象

```
<bean id="sc" class="SomeClass"></bean>
```

2-2 使用有参构造创建对象

```
<bean id="sc" class="SomeClass">
        <!--指定构造器参数-->
        <constructor-arg index="0" type="java.lang.String" value="aaaaaa">
        </constructor-arg>
        <constructor-arg index="1" type="java.lang.Integer" value="88">
        </constructor-arg>
</bean>
```

2-3 调用类的静态方法

```
<!--调用com.itany.SomeClass的getInstance方法，传入aaa参数，创建对象
        对象类型 未知！！！！
-->
<bean id="sc" class="SomeClass" factory-method="getInstance">
        <constructor-arg index="0" type="java.lang.String" value="aaa">
        </constructor-arg>
</bean>

<bean id="d" class="SomeClass" factory-method="getDate"></bean>
```

2-3 调用类的非静态方法

```
<bean id="sc" class="SomeClass" factory-method="getInstance">
       <constructor-arg index="0" type="java.lang.String" value="aaa">
        </constructor-arg>
</bean>
<!--使用sc对应的类中的getList方法创建java.util.ArrayList对象-->
<bean id="list" class="java.util.ArrayList"
               factory-bean="sc" factory-method="getList">
        <constructor-arg index="0" type="int" value="5"></constructor-arg>
</bean>
```

2-4 综合案例

给定如下代码，完成Spring配置

```
ApplicationContext ac = new
  ClassPathXmlApplicationContext("applicationContext.xml");
//Calendar c1 = Calendar.getInstance();
Calendar c = (Calendar) ac.getBean("cal");
System.out.println(c);

// 获取到执行 update t_user set name = 'aaa' 后 受影响的行数
// con = DriverManager.getConnection(url,user,pwd);
// ps = con.prepareStatement(sql);
// int count = ps.executeUpdate();
Integer count = (Integer) ac.getBean("count");
System.out.println(count);
```

```
<bean id="cal" class="java.util.Calendar" factory-method="getInstance">
</bean>

<bean id="con" class="java.sql.DriverManager"
      factory-method="getConnection">
   <constructor-arg index="0" type="java.lang.String"


                    value="jdbc:mysql://localhost:3306/ums?
```

```
useUnicode=true&amp;characterEncoding=utf8">
  </constructor-arg>
  <constructor-arg index="1" type="java.lang.String" value="root">
  </constructor-arg>
  <constructor-arg index="2" type="java.lang.String" value="root">
  </constructor-arg>
</bean>

<bean id="ps" class="java.sql.PreparedStatement" factory-bean="con"
      factory-method="prepareStatement">
    <constructor-arg index="0" type="java.lang.String"
                     value="update t_user set username = 'aaa'">
    </constructor-arg>
</bean>

<bean id="count" class="java.lang.Integer" factory-bean="ps"
      factory-method="executeUpdate">
</bean>
```

2-5 使用FactoryBean

简介

- 是一个Bean
- 是一个对象工厂类型的Bean
- 用于创建对象的Bean

实现方式

1. 编写FactoryBean

```
public class DateFactoryBean implements FactoryBean<Date> {
    private int year;
    private int month;
    private int date;
    public DateFactoryBean(int year, int month, int date) {
        this.year = year;
        this.month = month;
        this.date = date;
    }
    @Override
    public Date getObject() throws Exception {
        // 创建对象，将创建好的对象进行返回
        Calendar c = Calendar.getInstance();
        c.set(year,month,date);
        return c.getTime();
    }
    @Override
    public Class<?> getObjectType() {
        // 返回创建的对象的class
        return Date.class;
    }

    @Override
```

```
    public boolean isSingleton() {
        // 是否是单例对象
        return false;
    }
}
```

2. 配置FactoryBean

```
<bean id="someDate" class="DateFactoryBean">
  <constructor-arg index="0" value="2018"></constructor-arg>
  <constructor-arg index="1" value="10"></constructor-arg>
  <constructor-arg index="2" value="22"></constructor-arg>
</bean>
```

3、IOC-DI

3-1 概念

`Dependency Injection` 依赖注入：为Spring管理的对象的属性 注入值

配置xml 方式实现DI，属性必须有get/set

3-2 简单数据类型

> 包括 基本数据类型，包装类，字符串，Class

```
<bean id="someBean" class="SomeBean">
  <property name="username" value="张三"></property>
  <property name="married" value="true"></property>
  <property name="age" value="22"></property>
  <property name="cls" value="java.util.Calendar"></property>
</bean>
```

3-3 集合类型

```
<bean id="otherBean" class="OtherBean">
  <property name="arr">
    <array>
      <value>arr1</value>
      <value>arr2</value>
    </array>
  </property>
  <property name="someList">
    <list>
      <value>list1</value>
      <value>list2</value>
    </list>
  </property>


  <property name="someSet">
```

```
    <set>
      <value>set1</value>
      <value>set2</value>
    </set>
  </property>

  <property name="someMap">
    <map>
      <entry key="k1" value="v1"></entry>
      <entry key="k2" value="v2"></entry>
    </map>
  </property>

  <property name="somePros">
    <props>
      <prop key="pk1">pv1</prop>
      <prop key="pk2">pv2</prop>
    </props>
  </property>
</bean>
```

3-4 Resource

`org.springframework.core.io.Resource`

```
<property name="res" value="classpath:applicationContext.xml"></property>
```

3-5 对象类型

```
<bean id="someBean" class="SomeBean">
        <!--ref 引用，值是另一个Bean 的id值-->
        <property name="otherBean" ref="otherBean"></property>
</bean>
<bean id="otherBean" class="OtherBean">
</bean>
```

3-6 属性编辑器

编辑器代码

```
public class AddressProperyEditor extends PropertyEditorSupport {
    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        // 参数text 是注入的值 此处text 代表 江苏-南京

        Address addr  = new Address();
        addr.setCity(text.split("-")[1]);
        addr.setProvince(text.split("-")[0]);
        // 最后一行代码是一样的

        // 将编辑好的对象（addr） 放入到 对应的bean（Person）
```

```
        //  的属性中（address）
        setValue(addr);
    }
}
```

相关配置

```
<!--配置 用户自定义编辑器,全局有效-->
<bean class=
      "org.springframework.beans.factory.config.CustomEditorConfigurer">
  <property name="customEditors">
    <map>
      <!--key代表属性类型，value 代表对应的编辑器
                    当遇到key 类型的属性时，使用value类型的属性编辑器
      -->
      <entry key="com.itany.Address" value="com.itany.AddressProperyEditor">
      </entry>
    </map>
  </property>
</bean>
```

4、配置继承关系

```
class Parent{
  private String name;
  // set/get
}
class Child extends Parent{
  private int age;
  // set / get
}
```

```
<bean id="p" class="com.Parent">
    <property name="name" value="b"></property>
</bean>
<bean id="c" class="com.Child" parent="p">
    <property name="name" value="a"></property>
</bean>
<!--
Child c = ac.getBean("c")
c.getName();==>?
-->
```

5、访问外部配置文件

```
<!--配置外部配置文件-->
<bean class=
"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location" value="classpath:some.properties">
        </property>
</bean>

<bean class="com.itany.Person" id="person">
  <property name="name" value="${abc}"></property>
  <property name="address" value="江苏-南京"></property>
</bean>
```

6、对象的生命周期

6-1 生命周期钩子

生命周期	备注
实例化	调用构造方法创建对象（默认在Spring 容器启动时被调用）
DI	依赖注入（对象被创建时调用）
postProcessBeforeInitialization	在初始化方法执行之前
初始化方法	不是构造方法，存在于类中的一个普通方法，使用init-method指定
postProcessAfterInitialization	在初始化方法执行之后
就绪、使用	此时对象已经创建完成,可以getBean 使用
销毁方法	不是finallize,在对象从Spring容器中销毁时，被调用的方法（不是从jvm销毁），使用destroy-method指定
对象销毁	-------

6-2 初始化方法和销毁方法

```
public class SomeClass {
    public void someInitMethod() {
        System.out.println("someInitMethod");
    }
    public void someDesMethod(){
        System.out.println("Destroy....");
    }
}
```

```
<!--
   init-method 值是方法名，用于指定初始化方法
   destroy-method 值是方法名，用于指定销毁方法
-->
<bean id="someClass" class="day02.SomeClass"
      init-method="someInitMethod"
      destroy-method="someDesMethod">
</bean>
```

6-3 修改对象实例化时机

```
<!--
        lazy-init:
            default: 默认值，行为和false一致
            false：不进行延迟初始化，即当ac容器被创建时，立即初始化对象
            true： 进行延迟初始化，即当ac容器被创建时，不立即初始化对象，
                    当第一次从容器中获取对象是初始化
-->
<bean id="someClass" class="day02.SomeClass" lazy-init="true"  >
</bean>
```

6-4 配置对象的Scope

Spring容器中的对象默认是单例的，可以使用`scope`属性配置对象的创建方式

```
<!--
     scope
            singleton 默认值，单例对象
            prototype 每次从容器中获取时，都会创建新对象
            request   每次请求创建新对象
            session   每个会话一个对象
 -->
<bean id="someClass" class="day02.SomeClass"  scope="prototype"></bean>
```

6-5 后处理bean

所谓的 后处理bean 指的是 实现了 特定 接口的 bean ，用于对容器中其他的所有的Bean 进行后处理操作

后处理操作：在 DI 之后 和 init-method 之后执行的代码片段

特定接口：

- `BeanPostProcessor`

```
public class SomeBeanPostProcessor implements BeanPostProcessor {

    private Resource location;

    public Resource getLocation() {

        return location;
```

```
    }

    public void setLocation(Resource location) {
        this.location = location;
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
        // 在init-method之前，在DI之后
//        System.out.println(beanName+"的属性注入完成");

        try {
            Properties p = new Properties();
            p.load(location.getInputStream());
//            System.out.println(p);

//            if(bean instanceof SomeClass)
//            {
//                SomeClass temp = (SomeClass) bean;
//                String value = temp.getName();
//                value = value.substring(2,value.length()-1);
//                value = p.getProperty(value);
//                temp.setName(value);
//            }

            Class cls = bean.getClass();
            Field[] fields = cls.getDeclaredFields();
            for(Field  f : fields)
            {
                f.setAccessible(true);
                String key = f.get(bean)+"";
                key = key.substring(2,key.length()-1);
                String value = p.getProperty(key);
                f.set(bean,value);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        return bean;
    }
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        // 在init-method 之后
//        System.out.println(beanName+"初始化方法完成");
        return bean;
    }
}
```

- `BeanFactoryPostProcessor`

```
public class SomeBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory
configurableListableBeanFactory) throws BeansException {
        System.out.println("ac容器被创建完成后调用的方法。。。。。");
    }
}
```

7、BeanFactory 和 FactoryBean

BeanFactoy	FactoryBean
是对象工厂，就是容器	是一个Bean
用于管理所有的对象	用于对象的创建
本身就是一个接口	实现FactoryBean<T>接口，使用getObject创建对象
ApplicationContext 继承了该接口	

8、属性的自动装配

使用`autowire`属性 实现 对象属性的 自动装配。

值：

- no : 不使用自动装配，此时必须使用`property`为属性注入值
- default： 默认值，行为和no 一致
- byName

  根据属性名自动装配，Spring 会在容器中寻找一个id值 和 属性名 完全一致的bean

  将找到的bean 注入到 对应属性中

  如果找不到：什么事也不会发生

  如果找到了，但是类型不匹配：报错
- byType

  根据属性类型自动装配

  Spring 会在容器中寻找一个bean，它的 的类型 和 属性类型 完全一致（或者是其子类、实现类）的bean，将其注入到对应属性中

  如果找不到：什么事也不会发生

  如果找到了，并且不止一个，报错
- constructor

  根据构造方法参数，使用byType方式注入

```
<bean autowire="byType" id="userService"
       class="ums.service.UserServiceImpl">
</bean>
```

五、IOC-注解

注解	作用
@Component	定义Bean，相当于`<bean></bean>`，可以使用参数指定bean的id，默认是类名（首字母小写）
@Controller	等同于@Component，用于controller层（action层）
@Service	等同于@Component，用于service层
@Repository	等同于@Component，用于dao层
@Value	属性注入，对应3-2、 3-3、 3-4
@AutoWired	对应的是 3 - 5
@PostConstruct	init-method
@PreDestroy	destroy-method
@Lazy	修改对象实例化时机
@Scope	修改对象创建时机（单例，多例）
@ComponentScan	扫包，用于扫描类上的注解
@Configuration	在Spring配置文件中，有很多的配置项不是配置在特定类上的，而是一些基础配置（例如 扫包配置，读取properties文件...），这些配置会写在java类中，而这个java类，需要添加@Configuration，简单来说：@Configuration注解相当于applicationContext.xml
@Bean	在@Configuration中注册Bean，一般用于第三方的Bean

```
@Component
@Scope("prototype")
@Lazy
public class SomeClass {

    @Value("aaaaaaa")
    private String name;

    @Value("${abc}")
    private int age;


    @Value("classpath:log4j.properties")
```

```
    private Resource resource;

    @Autowired
    private OtherClass otherClass;

    @PostConstruct
    public void init()
    {
        System.out.println("init");
    }

    @PreDestroy
    public void destroy()
    {
        System.out.println("des...");
    }

    @Override
    public String toString() {
        return "SomeClass{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", resource=" + resource +
                ", otherClass=" + otherClass +
                '}';
    }
}
```

```
@Configuration
@ComponentScan({"annotation","day02"})
public class ApplicationConfig {


    // 在Configuration中 使用@Bean注解 + 方法 来 注册 Bean
    // 方法返回值是bean的对象，方法名是bean的id
    @Bean
    public PropertyPlaceholderConfigurer propertyPlaceholderConfigurer()
    {
        PropertyPlaceholderConfigurer ppc = new PropertyPlaceholderConfigurer();
        Resource location = new ClassPathResource("some.properties");
        ppc.setLocation(location);
        return ppc;
    }

}
```

```
public class Test {
    public static void main(String[] args) {
        ApplicationContext ac = new
          AnnotationConfigApplicationContext(ApplicationConfig.class);
        SomeClass sc = (SomeClass) ac.getBean("someClass");
        System.out.println(sc);
    }
}
```

六、AOP

1、概念

`Aspect Oriented Programming` 面向（Oriented）切面（Aspect）编程

将程序中的交叉业务逻辑（事务、日志、访问控制、缓存）提取出来，封装成 `切面`

由Spring 容器，在适当的时候（编译期 / **运行期**），将 `切面` 动态的 `织入` 到具体的业务逻辑中

以事务为例

- 事务代码不再出现在service实现类的方法中
- 事务代码独立存在于方法中
- 事务代码 和 业务逻辑代码 相互独立，不再存在 **静态代理类（代理类的java文件）**

2、实现原理

使用 **`动态代理`** 技术实现AOP

动态代理： 使用Proxy 类，动态的在内存中创建代理类对象，不需要静态的编写代理类（不存在java文件）

程序员只需要关心 切面逻辑，不需要编写代理类

Spring 实现动态代理的两种方式

1. 对于实现了 某些 接口的 类，使用 jdk 自带的 `java.lang.reflect.Proxy` 类创建动态代理类和代理对象, 使用实现相同接口的方式
2. 对于没有实现任何接口的类，使用 `CGLIB` 的 `Proxy` 类实现动态代理，使用 继承目标类的方式实现动态代理

对于没有实现任何接口，并且被final修饰的类，无法创建动态代理对象

3、HelloWorld

3-1 导入依赖

```
<!--SpringAOP 核心包-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>${spring.version}</version>
</dependency>

<!--SpringAOP 切点表达式jar-->
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>${aspect.version}</version>
</dependency>
```

3-2 编写切面

```
public class UserDaoAdvisor {
    public void before()
    {
        System.out.println("dao方法开始执行");
    }

    public void after()
    {
        System.out.println("执行结束");
    }
}
```

3-3 配置Bean

```
<bean id="userDao" class="aop01.UserDaoImpl"></bean>
<bean id="advis" class="aop02.UserDaoAdvisor"></bean>

<!--告诉Spring
            在 执行aop01.UserDaoImpl中的所有的方法前，执行advis的before方法
            在 执行aop01.UserDaoImpl中的所有的方法后，执行advis的after方法
    -->
<aop:config>
  <!--aop01.UserDaoImpl中的所有的方法-->
  <aop:pointcut id="pc" expression="within(aop01.UserDaoImpl)"></aop:pointcut>

  <aop:aspect ref="advis">
    <aop:before method="before" pointcut-ref="pc"></aop:before>
    <aop:after method="after" pointcut-ref="pc"></aop:after>
  </aop:aspect>
</aop:config>
```

4、几个概念

4-1 切面

提取出来的交叉业务逻辑代码片段

4-2 通知

由 切面 代码 组成的类，普通java类

配置通知，配置一个普通的bean `<bean id="" class=""></bean>`

4-3 织入

将通知 和 目标类 目标方法 动态的 耦合在 一起的 过程

4-4 切点

通知的 织入 位置

5、通知类型

- 前置通知 before : 在方法执行前执行的通知
- 后置通知 after ： 在方法执行后执行的通知
  - after-returning : 在方法正常返回后执行的通知
  - after-throwing : 在方法抛出异常后执行的通知
- 环绕通知 around: 在方法执行前、后 执行的通知

```
public class UserDaoAdvisor {
    // 除了环绕通知，其他通知都可以提供一个参数，类型是org.aspectj.lang.JoinPoint
    // 通过该参数，可以获取到要执行的方法的相关信息
    public void before(JoinPoint jp)
    {
        // 获取的是方法签名
        Signature signature = jp.getSignature();

        // 方法名
        String methodName = signature.getName();
        // 调用方法的对象类型 这个方法声明在哪个类中
        Class cls= signature.getDeclaringType();
        // 方法访问修饰符
        int modifiers = signature.getModifiers();
        String mode = Modifier.toString(modifiers);


        // 获取参数列表
        Object[] args = jp.getArgs();
        System.out.println(Arrays.toString(args));

        // 调用方法的对象
        Object target = jp.getTarget();
        System.out.println(target);

        System.out.println("before:" + cls.getName() + "  " + mode + " " + methodName );
    }
```

```
    public void after()
    {
        System.out.println("执行结束");
    }

    public void afterReturing(Object retValue)
    {
        System.out.println("方法正常返回:" + retValue);
    }

    public void afterThrow(Exception ex)
    {
        System.out.println("方法抛出异常" + ex.getMessage());
    }

    //对于环绕通知，会阻塞 目标方法 执行  Filter
    //需要通过代码手动放行目标方法  chain.doFilter(req,res);
    // 要求：
    //    1. 方法参数是ProceedingJoinPoint
    //    2. 方法throws Throwable
    //    3. 方法返回值类型写Object
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("环绕---前");
        // 继续执行目标方法，该方法存在异常，不要try - catch
        // 目标方法的返回值被proceed方法返回
        Object obj = pjp.proceed();
        System.out.println("环绕---后:" + obj);
        return obj;
    }

}
```

```
<!--告诉Spring
            在 执行aop01.UserDaoImpl中的所有的方法前，执行advis的before方法
            在 执行aop01.UserDaoImpl中的所有的方法后，执行advis的after方法
    -->

<aop:config>
  <!--aop01.UserDaoImpl中的所有的方法-->
  <aop:pointcut id="pc" expression="within(aop01.UserDaoImpl)"></aop:pointcut>

  <aop:aspect ref="advis">
    <aop:before method="before" pointcut-ref="pc"></aop:before>
    <aop:after method="after" pointcut-ref="pc"></aop:after>
    <aop:after-returning method="afterReturing"
                         pointcut-ref="pc" returning="retValue">
    </aop:after-returning>
    <aop:after-throwing method="afterThrow" pointcut-ref="pc" throwing="ex">
    </aop:after-throwing>
    <aop:around method="around" pointcut-ref="pc"></aop:around>
  </aop:aspect>

</aop:config>
```

6、切点表达式

Spring AOP 使用 AspectJ 切点表达式来定义切入位置

1、within

语法：`within(包名.类名)`

作用：切入类的所有方法

2、execution

语法: `execution(表达式)`

表达式语法

```
modifiers? ret-type declearing-class?.methodName(param) throws?
访问修饰符? 返回值类型 包名.类名.方法名(参数类型列表) 抛出的异常?
? 代表可有可无
```

```
public void s(java.lang.String,java.lang.Integer)
public void com.itany.service.UserService.add(com.itany.entity.User)
* com.itany.service.impl.*.*(..)
* com..*Service.*(..)
* s(*)
* s(..)
```

七、AOP-注解

```
@Component
@Aspect
public class SomeAdvisor {


    @Pointcut("within(aop03.SomeClass)")
    public void pc() {}


    @Before("pc()")
    public void before(JoinPoint jp)
    {

        System.out.println("before:" );
    }

    @After("pc()")
    public void after()
    {
        System.out.println("执行结束");

    }
```

```
    @AfterReturning(value = "pc()",returning = "retValue")
    public void afterReturing(Object retValue)
    {
        System.out.println("方法正常返回:" + retValue);
    }

    @AfterThrowing(value = "pc()",throwing = "ex")
    public void afterThrow(Exception ex)
    {
        System.out.println("方法抛出异常" + ex.getMessage());
    }

    @Around("pc()")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("环绕---前");
        Object obj = pjp.proceed();
        System.out.println("环绕---后:" + obj);
        return obj;
    }

}
```

```
@Configuration
@ComponentScan("aop03")
// 启用AspectJ注解的自动代理
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class AppConfig {
}
```

```
public class Test {
    public static void main(String[] args) {
        ApplicationContext ac =
                new AnnotationConfigApplicationContext(AppConfig.class);
        SomeClass sc = ac.getBean(SomeClass.class);
        sc.a();
    }
}
```

八、Spring 相关类

类名	作用
FactoryBean	工厂对象，是对象创建的一种方式
CustomEditorConfigurer	属性编辑器，DI的一种方式，需要注入customEditors属性
PropertyPlaceholderConfigurer	访问外部配置文件
BeanPostProcessor	对对象(bean)进行后处理操作
BeanFactoryPostProcessor	对 对象工厂（beanFactory）进行后处理操作

九、整合Mybatis

1、添加依赖

```
<!-- Spring-Mybatis 开始-->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>${mybaits.version}</version>
</dependency>

<!--Mybatis 对Spring的支持-->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>${mybatis-spring.version}</version>
</dependency>

<!--Mybatis分页插件-->
<dependency>
  <groupId>com.github.pagehelper</groupId>
  <artifactId>pagehelper</artifactId>
  <version>${pagehelper.version}</version>
</dependency>

<!--分页插件的依赖包-->
<dependency>
  <groupId>com.github.jsqlparser</groupId>
  <artifactId>jsqlparser</artifactId>
  <version>${jsqlparser.version}</version>
</dependency>
```

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>${spring.version}</version>
</dependency>
<!-- Spring-Mybatis 结束-->

<!--mysql驱动包-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>${mysql.version}</version>
</dependency>
<!--数据库连接池-->
<dependency>
  <groupId>commons-dbcp</groupId>
  <artifactId>commons-dbcp</artifactId>
  <version>${dbcp.version}</version>
</dependency>
<!--数据库连接池的依赖包-->
<dependency>
  <groupId>commons-pool</groupId>
  <artifactId>commons-pool</artifactId>
  <version>${pool.version}</version>
</dependency>
```

2、配置xml

```
<!--配置扫包-->
<context:component-scan base-package="com.itany.service">
</context:component-scan>

<!--配置数据源-->
<!--Spring内置数据源-->
<!--<bean id="source"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">-->
<!---->
<!--</bean>-->


<!--commons-dbcp-->
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location" value="classpath:datasource.properties"></property>
</bean>
<bean id="ds" class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${jdbc.driver}"></property>
  <property name="url" value="${jdbc.url}"></property>
  <property name="password" value="${jdbc.password}"></property>
  <property name="username" value="${jdbc.username}"></property>

  <property name="maxActive" value="5"></property>
```

```
  <property name="maxWait" value="5000"></property>
  <property name="initialSize" value="5"></property>
</bean>


<!--配置SqlSessionFactoryBean-->
<bean class="org.mybatis.spring.SqlSessionFactoryBean">

  <property name="dataSource" ref="ds"></property>
  <property name="mapperLocations" value="classpath:mappers/*Mapper.xml"></property>
  <property name="typeAliasesPackage" value="com.itany.entity"></property>
  <property name="plugins">
    <array>
      <bean class="com.github.pagehelper.PageInterceptor">
        <property name="properties">
          <!--<value>-->
          <!--pageSizeZero=true-->
          <!--reasonable=true-->
          <!--</value>-->
          <props>
            <prop key="pageSizeZero">true</prop>
            <prop key="reasonable">true</prop>
          </props>
        </property>
      </bean>
    </array>
  </property>
</bean>


<!--配置Dao接口，
        让Mybatis 扫描所有的dao接口，并创建对象
    -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
  <!--配置Dao接口所在包的  路径！！！  可以使用 分号 或者 逗号 分隔多个路径-->
  <property name="basePackage" value="com/itany/dao"></property>
</bean>
```

十、整合事务

1、添加依赖

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>${spring.version}</version>
</dependency>
```

2、配置事务管理器

```
 <!--配置事务管理器-->
<bean id="tr"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="ds"></property>
</bean>
```

3、启用注解事务

```
<!--启用注解事务-->
<tx:annotation-driven transaction-manager="tr" />
```

4、添加事务注解

```
//作用于所有方法
@Transactional(propagation = Propagation.REQUIRED,rollbackFor = Throwable.class)
public class UserServiceImpl {

    @Autowired
    private UserDao userDao;

    public void delete(int id)
    {
        userDao.deleteById(id);
    }

    // 会和类上的Transactional合并覆盖，相当于继承
    @Transactional(readOnly = true)
    public void findAll()
    {

    }
}
```

5、事务的特性

- A 原子性
- C 一致性
- I 隔离性
- D 永久性

6、并发情况下，数据库产生的问题

> 1、脏读：
>
> 事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据
>
> 一个事务读取到另一个事务没有提交的数据
>
> 2、不可重复读：

> 事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。
>
> 3、幻读：
>
> 系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统 管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读
>
> 一个事务多次读取的数据量不一致

7、事务的隔离级别

- TRANSACTION_NONE

  不支持事务

- TRANSACTION_READ_UNCOMMITED

  脏读、不可重复读 和 幻读（虚读）都可以发生

- TRANSACTION_READ_COMMITED

  脏读、不可重复读不会发生 、 幻读（虚读）可以发生（**主流数据库不支持**）

- TRANSACTION_SERIALIZABLE

  将并发操作 改成串行操作 牺牲了效率

  脏读、不可重复读 和 幻读（虚读）都不可以发生（**牺牲了效率**）

8、事务的传播策略

```
public enum Propagation {
    // 需要事务，当前方法如果有事务，使用该事务，否则开启新事务
    REQUIRED(0),
    // 需要事务，如果没有：什么事也不做
    SUPPORTS(1),
    // 需要事务，如果没有：报错
    MANDATORY(2),
    // 需要事务， 不论有没有， 都开启新事务
    REQUIRES_NEW(3),
    // 不需要事务，如果有：什么事也不做
    NOT_SUPPORTED(4),
    // 不需要事务，如果有：报错
    NEVER(5),
    // 不需要事务，如果有：就使用
    NESTED(6);
}
```

9、事务的只读优化

Spring 对于查询操作，提供了只读优化 `readOnly`，实现只读事务