" IOC "是一种思想。" IOC容器 "是一种容器。
容器： 能管理对象生命周期（创建该对象 到 销毁该对象）的东西。
IOC容器与非IOC容器的区别：IOC容器在创建出一个对象时会对该对象的成员变量进行赋值。而非IOC容器只会创建出对象，不会对创建出来的这个对象的成员变量进行赋值。
IOC容器给创建出的对象的成员变量进行赋值的这个现象被称为DI (依赖注入)。

spring中只需要一个配置文件，名字可以任意起(可以起为applicationContext.xml 或 spring-ioc.xml)，作用是在创建IOC容器对象的时候要将这个配置文件的名称传进去。
如：ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");

IOC容器对应的接口是BeanFactory接口。我们需要使用的代表IOC容器的类是ClassPathXMLApplicationContext类。

想要让IOC容器去管理一个对象，就在配置文件中为这个对象配置。IOC容器管理的对象就是由它创建出来的对象。IOC容器只会管理由它创建出来的对象的生命周期。
被IOC容器管理的javaBean对象的生命周期： 1. 被创建并被依赖注入。
2. 被初始化。 IOC容器创建出对象并依赖注入之后会去调用与这个对象对应的的init-method属性值指定的在这个对象所属的类中的方法。
3. 被使用。
4. 被销毁。 IOC容器要被关闭时，会去调用与这个对象对应的的destroy-method属性值指定的在这个对象所属的类中的方法。
注意：用于关闭IOC容器对象的close()方法是在ConfigurableApplicationContext类中2020/8/2定义的，ApplicationContext类中并没有定义close()方法。

一个IOC容器对象被创建出来后，这个IOC容器对象就会立刻查看xml配置文件，根据其中的scope属性值为"singleton"的创建出各个与之对应JavaBean对象。如果是scope属性值为"prototype"的，IOC容器是在调用getBean()时才去创建与之对应的JavaBean对象(每次调用都创建一个新的)。注意：scope属性值为"singleton"的对应的对象的生命周期是完全由IOC容器管理，
scope属性值为"singleton"的对应的对象是在容器关闭时被销毁。但scope属性值为"prototype"的对应的对象的生命周期中的销毁不是由IOC容器管理(IOC容器调用close()方法时不会调用它的destroy方法，也不会将它销毁)，它是在没有被引用时由JVM的垃圾收集器进行销毁。
的scope属性的默认值为"singleton"。
中的init-method用于指定将class属性值(JavaBean类)中的哪个方法作为初始化方法，这个初始化方法会被IOC容器创建出该JavaBean对象之后调用。
中的destroy-method用于指定将class属性值(JavaBean类)中的哪个方法作为销毁方法，这个销毁方法会被IOC容器要销毁该JavaBean对象之前调用。
如果我们会去获取某个对应的JavaBean对象，则需要给这个设置id属性。
获取JavaBean对象的时候推荐是使用ApplicationContext.getBean(String, Class)这个方法。(内部先根据String类型参数的值获取到对象(Object类型)，然后将Object类型转为第二个参数指定的类型，然后返回该对象。)
value属性值或的值只能是字面值(能用字符串表示出来的值)。
标签中的属性都能用标签表示，例如：value属性能用表示。
value属性值中如果含有特殊字符，则要将这个value属性写成，然后将value属性值用 包起来，即写成
注：只能写在中。

util:map/util:map、util:list/util:list、util:set/util:set都是一个特殊的。

如果spring配置文件中有一个的class属性值指定的类实现了BeanPostProcessor接口(注意在实现postProcessBeforeInitialization(Object bean, String beanName)与postProcessAfterInitialization(Object bean, String beanName)的时候一定要将参数bean的值返回)，则IOC容器在对其它的各个对应的对象进行初始化之前，都会调用
这个类中的postProcessBeforeInitialization方法，在初始化之后，都会调用这个类中的postProcessAfterInitialization方法。

依赖注入有两种方式：1. 利用该对象的set方法为其成员变量赋值。对应于　　注：写property的name属性值的时候用Alt + /提示。

当IOC容器查看xml配置文件发现时，IOC容器就用newInstance()方法去创建JavaBean对象。然后利用set方法对这个JavaBean对象进行依赖注入。

所以要求该JavaBean类中必须含有无参的构造方法。

可以在中对内部的一个的ref属性指定的对象的属性值进行修改，步骤是：

1. 在这个中定义一个
2. 然后将它的name属性值以级联的形式写为要进行修改的对象的属性的名字。

可以在的内部写，即，意思就是：将内部的这个对应的对象的地址赋给这个对应的成员变量。示例：

可以在的内部写，即，意思就是：将内部的这个对应的List对象的地址赋给这个对应的成员变量。示例：

```
//user1是一个的id属性值，意思就是将user1对应的对应的对象的地址add进这个List对象中。
// 意思就是将这个对应的对象的地址add进这个List对象中。
```

...

还可以在的内部写或，都是和同理。

可以在<property></property>的内部写<map></map>，即<property><map></map></property>，意思就是：将内部的这个<map></map>对应的Map对象的地址赋给这个<property></property>对应的成员变量。示例：

```
<property name="users">
    <map>
        <entry key="user1">      //这个<entry>表示这个Map对象中的第一个键值对，"键"为"user"，"值"为
            <ref bean="user"/>    id属性值为user的<bean>对应的对象的地址
        </entry>
        <entry key="user2" value-ref="user"></entry>
    </map>
</property>
```

在一个<bean>中可以用 ${"键"} 的方式来获取到一个.properties文件中的"值"。常用于：配置与数据库相关的信息时。

如果要想在Spring配置文件中使用一个xxx.properties文件：

方式1：使用

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="classpath:xxx.properties"></property> //"classpath:"的意思就是bin目录，source folder中
</bean>
```

的东西将来都是在bin目录下的。

方式2：使用<context:property-placeholder location="classpath:xxx.properties"/> ，这种方式就是第1种方式的简便写法。

注： 是这个org.springframework.beans.factory.config.PropertyPlaceholderConfigurer类实现的“${"键"}获取.properties文件中的"值" ”的功能，
只有在<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"></bean>中指定过的.properties文件中的
"值"才能用 ${"键"} 的方式获取。
如果想指定多个.properties文件则有且仅有两种方式：
方式1：<context:property-placeholder location="classpath:xxx1.properties,classpath:xxx2.properties"/>
方式2：<context:property-placeholder location="classpath:xxx1.properties" ignore-unresolvable="true"/>
<context:property-placeholder location="classpath:xxx2.properties" ignore-unresolvable="true"/>

注意：在.properties文件中写与数据库信息相关的键值对时，"键"起名请以"jdbc. "开头，这样能避免一些问题的发生。
示例：jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/test
jdbc.username=root
jdbc.password=123

2. 利用该对象的构造方法为其成员变量赋值。对应于 <constructor-arg></constructor-arg>
当IOC容器查看xml配置文件发现<constructor-arg></constructor-arg>时，IOC容器就会根据每个<constructor-arg></constructor-arg>的index属性值和value属性值从
当前的javaBean中查找与之匹配的构造方法(一旦找到匹配的，就立即停止查找。查找构造方法的顺序与各个构造方法在类中的定义顺序无关)，然后用这个匹配的构造方法
去创建javaBean对象并对创建出来的这个javaBean对象进行依赖注入。
如果<constructor-arg></constructor-arg>中是没有写index属性的，则利用反射找构造方法的时候，是用各个<constructor-arg>的value属性值按书写顺序去对应各个参数，
如果各个value属性值都能被解析为与之对应的参数的类型的数据，则选中该构造方法。注意，反射在找构造方法时的顺序和类中定义构造方法的顺序是无关的。为了让反射
能精确找到你所希望的构造方法，就可以在<constructor-arg>中使用type属性说明value属性值的类型，type属性值要写全类名。index属性(从0开始)是用来指定数据要传给
构造方法的哪个参数。

IOC容器对象.getBean("aaa", xxx.class);//如果aaa对应的对应的是一个实现了FactoryBean接口的javaBean的对象，则getBean()方法内部会去调用这个javaBean对象的getObject()方法，并将
getObject()方法的返回值返回。由于getObject()方法返回的对象是由我们自己创建出来的，并不是由IOC容器创建出来的，所以IOC容器不会对这个对象进
行管理，这个对象与IOC容器没关系。
IOC容器对象.getBean("aaa", xxx.class);//如果aaa对应的对应的是一个没有实现FactoryBean接口的javaBean的对象，则getBean()方法是将这个javaBean对象返回。

----------注解----------

补： 注解语法：@注解类名(方法名=值)，如果括号中只写一个方法名并且方法名是value，则"value="可以省略
不要去使用中的autowire属性。

组件：在类名上方写上了 @Component 、 @Repository 、@Service 、 @Controller 其中的任意一个注解的类。这4个注解不能写在接口上。
在IOC容器眼中，@Repository 、@Service、 @Controller 就是 @Component。
@Component 、 @Repository 、@Service、 @Controller 这4个注解能标在任意一个类的类名上。
@Repository是用来写在Dao层的类的类名上。
@Service是用来写在Service层的类的类名上。
@Controller是用来写在Controller层的类的类名上。Controller层就是控制层，Servlet属于控制层。

@Component的value属性等价于中的id属性。
如果没有指定 @Component的value属性的值，则系统隐式将value属性的值设置为修饰的类的类名首字母小写。

@Component
public class AAA {　　等价于
 …
}

```
    @Component
```

IOC容器扫描 public class AAA {　　等价于　　IOC容器读取
　　…
　　}

IOC容器只有读到配置文件中<context:component-scan base-package="包名" >/context:component-scan才会对该包中及其子包中的被 @Component、@Repository、@Service、@Controller 修饰的类进行扫描。
　context:component-scan中只能有context:include-filter, context:exclude-filter其中的一种。如果有了context:include-filter，就要将context:component-scan的
　use-default-filters属性的值设置为false。如果有了context:exclude-filter，就要将context:component-scan的use-default-filters属性的值设置为true。注：use-default-filters属性
　的默认值为true。context:component-scan中的context:include-filter可以有多个，并且type属性值可以不同。context:component-scan中的context:exclude-filter可以有多个，并且
　type属性值可以不同。
　<context:include-filter type="annotation"
expression="org.springframework.stereotype.Component"/> : 读取到这句话时，IOC容器就只会对<context:component-scan base-package="包名" >
　指定的包中及其子包中的被 @Component修饰的类进行扫描。
　<context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Component"/> : 读取到这句话时，IOC容器就不会对<context:component-scan base-package="包名" >
　指定的包中及其子包中的被 @Component修饰的类进行扫描。
　<context:include-filter type="assignable" expression="XXX"/> : 读取到这句话时，IOC容器就只会对<context:component-scan base-package="包名" >
　指定的包中及其子包中的被 @Component、@Repository、@Service、@Controller修饰的并且类名是XXX的类进行扫描。
　<context:exclude-filter type="assignable" expression="XXX"/> : 读取到这句话时，IOC容器就不会对<context:component-scan base-package="包名" >
　指定的包中及其子包中的被 @Component、@Repository、@Service、@Controller修饰的并且类名是XXX的类进行扫描。

@Autowired注解的位置： 1. 成员变量上　2.方法上　3.注解上
IOC容器扫描到在成员变量上的@Autowired注解时，会给这个成员变量去找JavaBean对象。然后将找到的对象的地址直接赋值给这个成员变量，而不是通过这个成员变量的set方法，所以不需要定义这个成员
变量的set方法。找的规则是找类型与这个成员变量的类型相同或是成员变量类型子类的对象。如果找不到则报错。如果找到了多个，则从这些对象中找id是与这个成员变量名相同的对象，如果找不到则报错。
如果想"找类型与这个成员变量的类型相同或是成员变量类型子类的对象"找不到时不报错，就将@Autowired的required属性值设为false。
如果想"如果找到了多个，则从这些对象中找id是与这个成员变量名相同的对象"找不到时不报错，就指定找不到时将这些对象中的哪一个对象的地址拿去设置：在@Autowired下方写@Qualifier("对象id")。

IOC容器扫描到在一个方法上的@Autowired注解时，会去调用这个方法(会给这个方法的形式参数去找JavaBean对象，然后将找到的对象的地址直接赋值给这个形式参数)。

注意！！补："IOC容器扫描到在成员变量上的@Autowired注解时，会给这个成员变量去找JavaBean对象。"：如果找到的这个JavaBean对象有Aspectj为它创建出来的代理对象，则是将找到的这个JavaBean对象
　　的代理对象的地址赋给成员变量，就不是将这个找到的JavaBean对象的地址赋给成员变量了。
----------杂----------

没有aaa这种写法，只能写成aaa。

配置文件中的属性之间需要有空格隔开。

一个配置文件中不能存在两个相同的id值。

类路径(classpath): 说的就是bin目录。

JavaBean：一个类中的每个成员变量都有与之对应的set或get方法，并且有一个无参构造方法，则这个类就是一个javaBean。属性的类型是get方法的返回值类型或set方法的参数类型，

属性的名称是set/get方法的方法名除去"set"/"get"后首字母小写。

xml中的每一个表示的是IOC容器创建出的一个JavaBean对象，而不是表示一个JavaBean类。//表示与对应？？？？？

IOC容器想要调用一个对象的xx方法，则这个对象只能是由它管理的对象。

如果给设置了autowire属性，则IOC容器读取到这个属性时会按autowired属性值指定的方式去给这个对象的每个属性找。
　如果属性值为byName，则找id值与属性名相同的，然后将找到的对应的对象的地址设置给成员变量。
　如果属性值为byType，则找class属性值是与属性类型相同或是属性类型子类的，找到多个就报错。

----------AOP----------

AOP(面向切面编程)：将非核心代码写到一个类中，而不和核心代码写在同一个类中。然后利用动态代理在invoke方法中让核心代码所属的类的对象去调用核心代码，让非核心代码所属的类的对象去调用非核心代码。

术语：

目标类：核心代码所属的类。一个核心代码对应于目标类中的一个目标方法。

切面类：非核心代码所属的类。

增强(也叫通知)：切面类中的方法，一个非核心代码对应于一个增强。

横切关注点：一个非核心代码就是一个横切关注点。

/连接点与切入点这两个名词很难理解，我放弃了**/**

连接点：InvocationHandler接口的实现类实现的invoke方法中的能调用切面类方法的各个地方。???

切入点：InvocationHandler接口的实现类实现的invoke方法中的调用了切面类方法的地方。???

连接点：代理类的方法中的invoke方法中的能调用切面类方法的各个地方。???

切入点：代理类的方法中的invoke方法中的调用了切面类方法的地方。???

动态代理类：在运行时产生的代理类。动态代理类继承Proxy类。没有继承Proxy类的类，就不是代理类。

调用Proxy.newProxyInstance()方法时，是将目标类实现的接口们传给interfaces参数。
Proxy.newProxyInstance方法根据传入的第二个参数interfaces动态生成一个代理类(继承了Proxy类)，这个代理类实现了interfaces中的接口。返回的代理对象就是这个生成的代理类的对象。

传入的第一个参数loader是用于加载生成的代理类的(传入用目标类对象获取的类加载器)。

动态代理类的构造方法中会调用Proxy类中的构造方法：protected Proxy(InvocationHandler h) 。也就是说创建的动态代理类的对象内部有一个成员变量InvocationHandler h：用于指向
在调用Proxy.newProxyInstance()时，接收的InvocationHandler对象。

实现InvocationHandler接口的invock()方法时，参数Object proxy不一定要进行使用。我看了很多源码里面都没有用到proxy的值。很多源码中都是将当前的代理类对象的地址传给了参数proxy，
但是虽然传了，却没有在方法体中用到proxy参数的值。要是我们自己去实现invock()方法，可以传null值给参数proxy。

生成的代理类中的每一个方法中的内容都只是：让InvocationHandler对象去调用invoke()方法。(生成的代理类中的每一个方法都是将定义它的接口中的与它对应的那个方法传给invoke()的参数method)。
生成的代理类中的每一个方法在调用invoke()方法时，都是将this传给参数proxy。

根据面向切面编程的思想创建出目标类和切面类之后，需要定义一个InvocationHandler的实现类，(要让这个实现类的对象能获取到目标类对象和切面类对象，并且要使一个这个实现类的对象只对应一个目标对象，也就是要使一个这个实现类的对象只能获取到一个目标类对象。在实现的invock方法中，会要这个InvocationHandler实现类对象去获取目标类对象和切面类对象)，然后将这个InvocationHandler实
现类的对象传给Proxy.newInstance()方法，Proxy.newInstance()方法会创建出代理类和代理类对象。然后去使用这个代理类对象。

想知道AspectJ是如何创建动态代理类的就去看JdkDynamicAopProxy这个类。JdkDynamicAopProxy类实现了InvocationHandler接口，JdkDynamicAopProxy对象能获取到目标类对象和切面类对象，
并且一个JdkDynamicAopProxy对象只能获取到一个目标类对象，在JdkDynamicAopProxy类实现的invock()方法中：要JdkDynamicAopProxy对象去获取了目标类对象和切面类对象。
JdkDynamicAopProxy类
中定义了一个用来返回动态代理类对象的方法，在这个方法中调用了Proxy.newInstance()方法(并将this传给了InvocationHandler类型的参数)。
(注： 是ProxyFactoryBean类调用ProxyCreateSupport类，ProxyCreateSupport类调用DefaultAopProxyFactory类，DefaultAopProxyFactory类调用JdkDynamicAopProxy类)

AspectJ的代码中定义了JdkDynamicAopProxy类。AspectJ会为每一个目标类对象创建出一个JdkDynamicAopProxy类对象。
AspectJ的作用就是为每一个目标类对象创建出代理类和代理类对象。

IOC容器读取到spring配置文件中的aop:aspectj-autoproxy/时，就能认识与AspectJ有关的那些注解，就会使用AspectJ。

IOC容器扫描到一个类中的@Aspect注解时，IOC容器才会将这个类当作一个切面类。
IOC容器扫描到切面中的@Before、@AfterReturning、@After、@Around、@AfterThrowing注解时，IOC容器才会将方法当作一个通知。
如果通知的参数列表中有(JoinPoint joinPoint)，则这个通知被调用时，IOC容器会将当前和这个通知合作的目标方法的信息封装成一个JoinPoint对象传进去。
@AfterReturning有一个returning属性，只要给一个@AfterReturning指定returning属性的值，然后在这个@AfterReturning修饰的方法的参数列表中定义一个Object类型的参数名为returning属性值的参数，
IOC容器就会将目标方法执行后的返回值传给名为returning属性值的这个参数。
@AfterThrowing有一个throwing属性，只要给一个@AfterThrowing指定throwing属性的值，然后在这个@AfterThrowing修饰的方法的参数列表中定义一个Exception类型的参数名为throwing属性值的参数，
IOC容器就会将目标方法抛出的异常对象传给名为throwing属性值的这个参数。
如果IOC容器使用了AspectJ，则IOC容器在创建一个javaBean对象时，会看这个javaBean有没有出现在某个切面类中的@Before/@AfterReturning/@After/@Around/@AfterThrowing的value属性值中，
如果有，就说明这个javaBean是一个目标类，就会让AspectJ为这个javaBean对象创建出代理类和代理类对象。
如果一个javaBean对象是有代理类对象的，那么ApplicationContext.getBean("这个javaBean对象对应的的id属性值")就不是返回这个javaBean对象，而是返回这个javaBean对象的代理对象。
(注： 有出现在@Before/@AfterReturning/@After/@Around/@AfterThrowing的value属性值中的类就是目标类。)
execution(* com.xukan.beans..(..))：
　第一个：意思是不管访问修饰符和返回值是什么
　第二个：意思是不管类名是什么
　第三个*：意思是不管方法名是什么
　.. ：意思是不管参数列表中是什么内容

可以定义一个类，然后在这个类中定义一个方法(方法体中不用写内容)，然后用@PointCut(value="xxx")去修饰这个方法，然后将一个切面类中的一个@Before/@AfterReturning/@After/@Around/@AfterThrowing的value属性值写为" 刚刚定义的这个类的全类名.被@PointCut(value="内容")修饰的这个方法的名字 + () "，
这样就等价于将这个@Before/@AfterReturning/@After/@Around/@AfterThrowing的value属性值写为"xxx"。

AspectJ创建的InvocationHandler接口的实现类实现的invoke()方法中：
　在让目标对象调用方法之前，判断有没有切面对象所属的类中含有value属性值是能匹配当前目标对象要调用的方法的@Before。如果有，就会去调用这个切面类对象的这个@Before修饰的方法。

```
invoke()方法中的内容(伪代码)：
    try{
        判断有没有切面对象所属的类中含有value属性值是能匹配当前目标对象要调用的方法的
@Before。如果有，就会去调用这个切面类对象的这个@Before修饰的方法。
        让目标对象调用方法。
        判断有没有切面对象所属的类中含有value属性值是能匹配当前目标对象调用了的方法的
@AfterReturning。如果有，就会去调用这个切面类对象的这个@AfterReturning修饰的方法。
    } catch(Exception e) {
        判断有没有切面对象所属的类中含有value属性值是能匹配当前目标对象调用了的方法的
@AfterThrowing。如果有，就会去调用这个切面类对象的这个@AfterThrowing修饰的方法。
        e.printStackTrace();
    } finally {
        判断有没有切面对象所属的类中含有value属性值是能匹配当前目标对象调用了的方法的@After。
如果有，就会去调用这个切面类对象的这个@After修饰的方法。
    }
```

什么时候要用AOP：当你想添加内容到一个类的方法中，但又不想不修改这个类中的代码时。

**********测试

```
public class Main {
  Calculator cc= new Calculator();
  public void m() {


    Class clazz = cc.getClass();
    Calculate c = (Calculate)Proxy.newProxyInstance(clazz.getClassLoader(),
clazz.getInterfaces(), new InvocationHandler() {

        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
            System.out.println(method.getDeclaringClass() + "。。。");
            return method.invoke(cc, args);//这里的cc之前隐含了"外部类名.this"，内部类
中可以使用"外部类名.this"，"外部类名.this"就是：使用这个内部类对象的
                                外部类对象的地址。
        }
    });
    System.out.println(c.add(1, 2));

}

public static void main(String[] args) {
    Main main = new Main();
    main.m();
}
```

}

**********代理工厂类(代理工厂对象可以创建出代理对象)

```
package cn.itcast.demo3;

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

/**
```

- @author cxf
  *
  */
  public class ProxyFactory {
  private Object targetObject;//目标对象
  private BeforeAdvice beforeAdvice;//前置增强
  private AfterAdvice afterAdvice;//后置增强

  /**
    - 用来生成代理对象
    - @return
      /
      public Object createProxy() {
      /
        - 1. 给出三大参数
          /
          ClassLoader loader = this.getClass().getClassLoader();
          Class[] interfaces = targetObject.getClass().getInterfaces();
          InvocationHandler h = new InvocationHandler() {
          public Object invoke(Object proxy, Method method, Object[] args)
          throws Throwable {
          /
            - 在调用代理对象的方法时会执行这里的内容
              */
              // 执行前置增强
              if(beforeAdvice != null) {
              beforeAdvice.before();
              }

          Object result = method.invoke(targetObject, args);//执行目标对象的目标方法
          // 执行后置增强
          if(afterAdvice != null) {
          afterAdvice.after();
          }

          // 返回目标对象的返回值
          return result;
          }
          };
          /*
        - 2. 得到代理对象
          */
          Object proxyObject = Proxy.newProxyInstance(loader, interfaces, h);
          return proxyObject;
          }

  public Object getTargetObject() {
  return targetObject;
  }
  public void setTargetObject(Object targetObject) {
  this.targetObject = targetObject;
  }
  public BeforeAdvice getBeforeAdvice() {

```
    return beforeAdvice;
}
public void setBeforeAdvice(BeforeAdvice beforeAdvice) {
    this.beforeAdvice = beforeAdvice;
}
public AfterAdvice getAfterAdvice() {
    return afterAdvice;
}
public void setAfterAdvice(AfterAdvice afterAdvice) {
    this.afterAdvice = afterAdvice;
}
}
```

----------声明式事务管理----------

IOC容器读取到spring配置文件中的tx:annotation-driven/时，才能认识与事务管理有关的那些注解。
transaction-manager属性的值要写为事务管理器对象对应的的id值。
含有@Transactional的类就是目标类。事务管理器类(是一个切面类，被@Aspect修饰)中定义的@Before/@After/...的value属性值能匹配所有被@Transactional修饰的方法。
事务管理器类中的@Before修饰的方法中的内容: " Connection.setAutoCommit(false); "
事务管理器类中的@AfterThrowing修饰的方法中的内容: 根据当前目标方法上的@Transactional的rollbackFor/rollbackForClassName/noRollbackFor/noRollbackForClassName属性的值
进行Connection.rollback();
事务管理器类中的@AfterReturning修饰的方法中的内容: Connection.commit();

@Transactional的各个属性:

1. Propergation ： 当我们没写Properation属性的时候，系统隐式将这个属性写上并将它的属性值设为" Propagation.REQUIRED "。

如果一个方法被Properation属性值为Propagation.REQUIRED的@Transactional修饰：当这个方法被另外一个也被@Transactional修饰的方法调用时，在这次调用中，这个方法上面的@Transactional就等于没写(即@Transactional失效)。

如果一个方法被Properation属性值为Propagation.REQUIRES_NEW的@Transactional修饰：当这个方法被另外一个也被@Transactional修饰的方法调用时，在这次调用中，这个方法的@Transactional不会失效。

(上面这段话的详细版本：

如果一个方法被Properation属性值为Propagation.REQUIRED的@Transactional修饰：当在另外一个也被@Transactional修饰的方法的方法体中要" 这个方法所属的类的对象去调用这个方法时 "，在这次调用中，这个方法上面的@Transactional就等于没写(即@Transactional失效)。

如果一个方法被Properation属性值为Propagation.REQUIRES_NEW的@Transactional修饰：当在另外一个也被@Transactional修饰的方法的方法体中要" 这个方法所属的类的对象去调用这个方法时 "，在这次调用中，这个方法的@Transactional不会失效。)

注：要想理解Propergation属性，有一点很关键："IOC容器扫描到在成员变量上的@Autowired注解时，会给这个成员变量去找JavaBean对象。"：如果找到的这个JavaBean对象有Aspectj为它创建出来的代理对象，则是将找到的这个JavaBean对象的代理对象的地址赋给成员变量，就不是将这个找到的JavaBean对象的地址赋给成员变量了。

示例程序：

```
@Service
public class CashierImpl implements Cashier {

    @Autowired
```

```
                private BookShopService bookShopService; //如果BookShopService类中含有@Transactional，则IOC容器赋给成员变量bookShopService的是代理对象的地址


@Transactional(propagation=Propagation.REQUIRED,noRollbackFor=RuntimeException.class)
                @Override
                public void checkOut(String username, List<String> isbns) {
                    for (String isbn : isbns) {
                        System.out.println(bookShopService);
                        bookShopService.buyBook(username, isbn); //如果BookShopService类的buyBook方法是被Properation属性值
                                                                                为Propagation.REQUIRES_NEW的@Transactional修饰的，则buyBook方法上的@Transactional
                                                                                不会失效。代理类中的
                                                                                事务管理器类中的@Before修饰的方法中的内容：
                                                                                if(@Transactional没失效){
                                                                                 Connection.setAutoCommit(false);
                                                                                }
                    }
                }
            }
```

```
    2. isolation ：isolation属性值一般都设为" Isolation.READ_COMMITTED "。隔离级别为"读已提交 "的意思是：只会从数据库中读取数据(数据库中的数据都是已经被提交的)。

    3. read： 如果设置read属性值为true，则在当前方法中不能发送增删改语句到数据库中，只能发送查询语句。
            如果设置read属性值为false，则在当前方法中可以发送增删改查语句到数据库中。

    4. timeout： timeout属性指定的时间的单位是秒
```

事务的并发说的是不是就是发送sql语句的一个线程 与发送sql语句的另一个线程之间的并发。。？？？
并发是不是可以看javaWeb中ThreadLocal的那集？？？
事务管理的意思就是要让发送到数据库的同属于一个业务的各条sql语句在同一个事务中 ？？？
只有事务管理器才能够进行事务管理。？？？
spring中，我们要让事务管理器帮我们进行事务管理。？？？

@Transactional的意思是：叫事务管理器实现："@Transactional修饰的方法中发送到数据库的sql语句都是在一个事务中"的效果。

事务嵌套依赖了数据库中的savePoint。

Spring的事务管理器接口是PlatformTransactionManager, 我们要用的基于JDBC的事务管理器类是DataSourceTransactionManager。

Spring中只有通过事务管理器对象才能使用事务。

----------草稿----------

Spring将javaBean进行了分类，一种是实现了FactoryBean接口的javaBean,这种javaBean类被称为"工厂Bean"。另外一种是没有实现FactoryBean接口的javaBean，这种javaBean被称为"普通bean"。

根据面向切面编程的思想创建出目标类和切面类之后，需要定义一个InvocationHandler的实现类，(要让这个实现类的对象能获取到目标类对象和切面类对象，并且要使一个这个实现类的对象只对应一个目标对象，也就是要使一个这个实现类的对象只能获取到一个目标类对象。在实现的invock方法中，会要这个InvocationHandler实现类对象去获取目标类对象和切面类对象)，然后将这个InvocationHandler实
现类的对象传给Proxy.newInstance()方法，Proxy.newInstance()方法会创建出代理类和代理类对象。然后去使用这个代理类对象。

想知道AspectJ是如何创建动态代理类的就去看JdkDynamicAopProxy这个类。JdkDynamicAopProxy类实现了InvocationHandler接口，JdkDynamicAopProxy对象能获取到目标类对象和切面类对象，并且一个JdkDynamicAopProxy对象只能获取到一个目标类对象，在JdkDynamicAopProxy类实现的invock()方法中：要JdkDynamicAopProxy对象去获取了目标类对象和切面类对象。
JdkDynamicAopProxy类
中定义了一个用来返回动态代理类对象的方法，在这个方法中调用了Proxy.newInstance()方法(并将this传给了InvocationHandler类型的参数)。
(注： 是ProxyFactoryBean类调用ProxyCreateSupport类，ProxyCreateSupport类调用DefaultAopProxyFactory类，DefaultAopProxyFactory类调用JdkDynamicAopProxy类)

AspectJ的代码中定义了JdkDynamicAopProxy类。AspectJ会为每一个目标类对象创建出一个JdkDynamicAopProxy类对象。
AspectJ的作用就是为每一个目标类对象创建出代理类和代理类对象。

根据面向切面编程的思想创建出目标类和切面类之后，需要定义一个InvocationHandler的实现类(这个实现类的对象要能获取到目标类对象和切面类对象，因为要让实现的invock方法中能？？？用到目标类对象和切面
类对象)，然后将这个InvocationHandler实现类的对象传给Proxy.newInstance()方法，Proxy.newInstance()方法会创建出代理类和代理类对象。然后去使用这个代理类对象。
想知道AspectJ是如何创建动态代理类的就去看JdkDynamicAopProxy这个类。JdkDynamicAopProxy类实现了InvocationHandler接口，在这个类中能获取到目标对象和切面对象，这个类实现的invock()方法中去获取了目标对象和切面对象。这个类中定义了一个用来返回动态代理类对象的方法，在这个方法中调用了Proxy.newInstance()方法(并将this传给了InvocationHandler类型的参数)。
(注： 是ProxyFactoryBean类调用ProxyCreateSupport类，ProxyCreateSupport类调用DefaultAopProxyFactory类，DefaultAopProxyFactory类调用JdkDynamicAopProxy类)

AspectJ的代码中定义了一个InvocationHandler的实现类(JdkDynamicAopProxy类)。
AspectJ会为每一个目标类对象创建出一个JdkDynamicAopProxy对象(内部存了这个目标类对象)，创建代理类和代理类对象。

事务的并发说的是不是就是发送sql语句的一个线程 与发送sql语句的另一个线程之间的并发。。？？？
并发是不是可以看javaWeb中ThreadLocal的那集？？？
事务管理的意思就是要让发送到数据库的同属于一个业务的各条sql语句在同一个事务中 ？？？
只有事务管理器才能够进行事务管理。？？？
spring中，我们要让事务管理器帮我们进行事务管理。？？？

@Transactional的意思是：叫事务管理器实现："@Transactional修饰的方法中发送到数据库的sql语句都是在一个事务中"的效果。
@T@Transactional写在类上表示：？？

autowired是将代理对象注入。
那我如果不用autowired，而是自己去new一个对象，赋给bookShopService成员变量，则会@Transactional(required_new)是不是会没效果？？是的。说明了@Transactional(required_new)有用就是因为
autowired是将代理对象注入的原因。

1. Propergation ： 当我们没写Properation属性的时候，系统隐式将这个属性写上并将它的属性值设为" Propagation.REQUIRED "。
   如果一个方法被Properation属性值为Propagation.REQUIRED的@Transactional修饰：当在另外一个也被@Transactional修饰的方法的方法体中要" 这个方法所属的类的
   对象去调用这个方法时 "，在这次调用中，这个方法上面的@Transactional就等于没写(即@Transactional失效)。
   如果一个方法被Properation属性值为Propagation.REQUIRES_NEW的@Transactional修饰：当在另外一个也被@Transactional修饰的方法的方法体中要" 这个方法所属的类的
   对象去调用这个方法时 "，在这次调用中，这个方法的@Transactional不会失效。