IT楠老师spring教程

> **B站：IT楠老师 公众号：IT楠说jiava QQ群： 1083478826 新知大数据**

制作不易、如果觉的好不妨打个赏：

一、spring简介

> **Spring 框架是 Java 应用最广的框架，它的成功来源于理念，而不是技术本身。**
>
> **它的理念包括 IoC (Inversion of Control，控制反转) 和 AOP(Aspect Oriented Programming，面向切面编程)。**

1、Spring 中常用术语：

> **框架：是能完成一定功能的半成品。**

框架能够帮助我们完成的是：项目的整体框架、一些基础功能、规定了类和对象如何创建，如何协作等，当我们开发一个项目时，框架帮助我们完成了一部分功能，我们自己再完成一部分，那这个项目就完成了。

> **非侵入式设计**

从框架的角度可以理解为：无需继承框架提供的任何类 这样我们在更换框架时，之前写过的代码几乎可以继续使用。

> **轻量级和重量级：**

轻量级是相对于重量级而言的，轻量级一般就是非入侵性的、所依赖的东西非常少、资源占用非常少、部署简单等等，其实就是比较容易使用，而重量级正好相反。

1、javaBean javaBean是一种Java语言写成的可重用组件。为写成JavaBean，类必须是具体和公共的，并且具有无参数的构造器。JavaBean通过提供符合一致性设计模式的公共方法将内部域暴露成员属性。更多的是一种规范，即包含一组set和get方法的java对象。javaBean可以使应用程序更加面向对象，可以把数据封装起来，把应用的业务逻辑和显示逻辑分离开，降低了开发的复杂程度和维护成本。

2、pojo （Plain Ordinary Java Object）简单的Java对象，实际就是普通JavaBeans，是为了避免和EJB混淆所创造的简称。 其中有一些属性及其getter、setter方法的类，没有业务逻辑，有时可以作为VO（value-object）或DTO（Data Transfer Object）来使用。不允许有业务方法，也不能携带connection之类的方法。 与javaBean相比，javaBean则复杂很多，JavaBean是可复用的组件，对JavaBean并没有严格的规范，理论上讲，任何一个Java类都可以是一个Bean。但通常情况下，由于JavaBean是被容器创建的，所以JavaBean应具有一个无参的构造器。另外，通常JavaBean还要实现Serializable接口用于实现Bean的持久性。一般在web应用程序中建立一个数据库的映射对象时，我们只能称他为POJO。用来强调它是一个普通的对象，而不是一个特殊的对象，其主要用来指代哪些没有遵从特定的java对象模型、约定或框架（如EJB）的java对象。理想的将，一个POJO是一个不受任何限制的java对象 3、entity 实体bean，一般是用于ORM对象关系映射，一个实体映射成一张表，一般无业务逻辑代码。负责将数据库中的表记录映射为内存中的Entity对象，事实上，创建一个EntityBean对象相当于创建一条记录，删除一个EntityBean对象会同时从数据库中删除对应记录，修改一个Entity Bean时，容器会自动将Entity Bean的状态和数据库同步。

传统javabean与spring中的bean的区别

javabean已经没人用了

springbean可以说是javabean的发展, 但已经完全不是一回事儿了

用处不同：传统javabean更多地作为值传递参数，而spring中的bean用处几乎无处不在，任何组件都可以被称为bean。

写法不同：传统javabean作为值对象，要求每个属性都提供getter和setter方法；但spring中的bean只需为接受设值注入的属性提供setter方法。

生命周期不同：传统javabean作为值对象传递，不接受任何容器管理其生命周期；spring中的bean有spring管理其生命周期行为。

所有可以被spring容器实例化并管理的java类都可以称为bean。

原来服务器处理页面返回的值都是直接使用request对象，后来增加了javabean来管理对象，所有页面值只要是和javabean对应，就可以用类.GET属性方法来获取值。javabean不只可以传参数，也可以处理数据，相当与把一个服务器执行的类放到了页面上，使对象管理相对不那么乱（对比asp的时候所有内容都在页面上完成）。

spring中的bean，是通过配置文件、javaconfig等的设置，有spring自动实例化，用完后自动销毁的对象。让我们只需要在用的时候使用对象就可以，不用考虑如果创建类对象（这就是spring的注入）。一般是用在服务器端代码的执行上。

容器

在日常生活中容器就是一种盛放东西的器具，从程序设计角度看就是装对象的的对象，因为存在放入、拿出等操作，所以容器还要管理对象的生命周期。

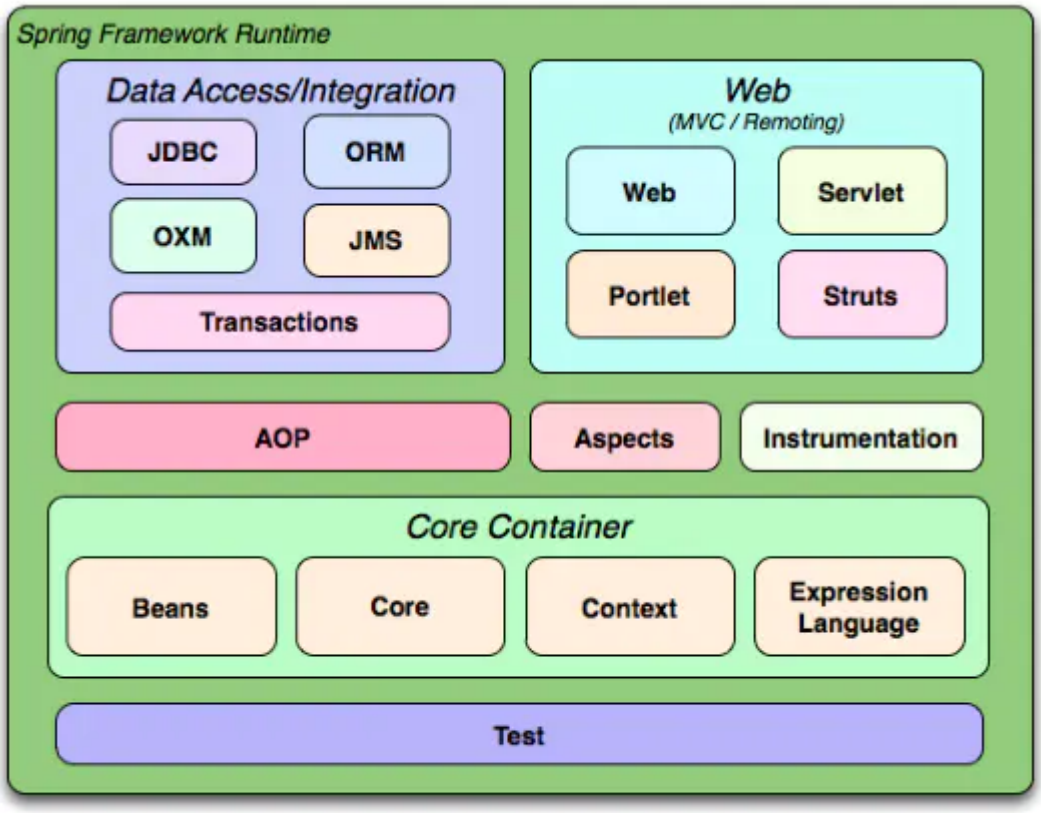
2、Spring 的优势

- 低侵入 / 低耦合 （降低组件之间的耦合度，实现软件各层之间的解耦）
- 声明式事务管理（基于切面和惯例）
- 方便集成其他框架（如MyBatis、Hibernate）
- 降低 Java 开发难度
- Spring 框架中包括了 J2EE 三层的每一层的解决方案（一站式）

3、Spring 能帮我们做什么

- Spring 能帮我们根据配置文件创建及组装对象之间的依赖关系。
- Spring 面向切面编程能帮助我们无耦合的实现日志记录，性能统计，安全控制。
- Spring 能非常简单的帮我们管理数据库事务。
- Spring 还提供了与第三方数据访问框架（如Hibernate、JPA）无缝集成，而且自己也提供了一套JDBC访问模板来方便数据库访问。
- Spring 还提供与第三方Web（如Struts1/2、JSF）框架无缝集成，而且自己也提供了一套Spring MVC框架，来方便web层搭建。
- Spring 能方便的与Java EE（如Java Mail、任务调度）整合，与更多技术整合（比如缓存框架）。

4、Spring 的框架结构



Overview of the Spring Framework

> **核心容器(Spring core)**

核心容器提供Spring框架的基本功能。Spring以bean的方式组织和管理Java应用中的各个组件及其关系。Spring使用BeanFactory来产生和管理Bean，它是工厂模式的实现。BeanFactory使用控制反转(IoC)模式将应用的配置和依赖性规范与实际的应用程序代码分开。BeanFactory使用依赖注入的方式提供给组件依赖。

Spring上下文(Spring context)

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，如JNDI、EJB、电子邮件、国际化、校验和调度功能。

Spring面向切面编程(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模块

DAO模式主要目的是将持久层相关问题与一般的的业务规则和工作流隔离开来。Spring 中的DAO提供一致的方式访问数据库，不管采用何种持久化技术，Spring都提供一直的编程模型。Spring还对不同的持久层技术提供一致的DAO方式的异常层次结构。

Spring ORM模块

Spring 与所有的主要的ORM映射框架都集成的很好，包括Hibernate、JDO实现、TopLink和IBatis SQL Map等。Spring为所有的这些框架提供了模板之类的辅助类，达成了一致的编程风格。

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。Web层使用Web层框架，可选的，可以是Spring自己的MVC框架，或者提供的Web框架，如Struts、Webwork、tapestry和jsf。

Spring MVC框架(Spring WebMVC)

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。Spring的MVC框架提供清晰的角色划分：控制器、验证器、命令对象、表单对象和模型对象、分发器、处理器映射和视图解析器。Spring支持多种视图技术。

5、个人体验

- spring中的几个名词，如IOC，DI，AOP，声明式事务，对初学者而言可能比较难以理解
- 但是当我们使用了spring经过了一定时间的开发之后，我们每个人其实就已经离不开spring了
- spring的理念真正颠覆了java的开发模式，java的发生离不开spring的支持
- 正如他的名字一样，spring给java开发带来了春天般的生机

6、准备工厂设计模式

工厂模式分为简单工厂模式，工厂方法模式和`抽象工厂模式`，它们都属于设计模式中的创建型模式。其主要功能都是帮助我们把对象的实例化部分抽取了出来，目的是降低系统中代码耦合度，并且增强了系统的扩展性。本文对这三种模式进行了介绍并且分析它们之间的区别。

（1）简单工厂设计模式

简单工厂模式最大的优点在于实现对象的创建和对象的使用分离，将对象的创建交给专门的工厂类负责，但是其最大的缺点在于工厂类不够灵活，增加新的具体产品需要修改工厂类的判断逻辑代码，而且产品较多时，工厂方法代码将会非常复杂。

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public interface Car {
    /**
     * 汽车运行的接口方法
     */
    void run();
}
```

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Benz implements Car {
    public void run() {
        System.out.println("奔驰 is running! ");
    }
}

public class Bike implements Car {
    public void run() {
        System.out.println("我只有自行车! ");
    }
}

public class Bmw implements Car {
    public void run() {
        System.out.println("宝马 is running! ");
    }
}
```

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class CarFactory {

    public static Car getCar(String type){
        if("benz".equalsIgnoreCase(type)){
            //其中可能有很复杂的操作
            return new Benz();
        }else if("bmw".equalsIgnoreCase(type)){
            //其中可能有很复杂的操作
```

```
            return new Bmw();
        }else {
            return new Bike();
        }
    }
}
```

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Client {
    public static void main(String[] args) {
        Car bmw = CarFactory.getCar("bmw");
        bmw.run();
        Car benz = CarFactory.getCar("benz");
        benz.run();
    }
}
```

（2）工厂方法模式

我们说过java开发中要遵循开闭原则，如果将来有一天我想增加一个新的车，那么必须修改CarFactory，就不太灵活。解决方案是使用工厂方法模式。

我们为每一个车都构建成一个工厂：

先抽象一个工厂接口

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public interface Factory {
    /**
     * 统一的创建方法
     * @return
     */
    Car create();
}
```

然后针对每一个产品构建一个工厂方法

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class BenzFactory implements Factory {
```

```
    public Car create() {
        //中间省略一万行代码
        return new Benz();
    }
}

public class BmwFactory implements Factory {
    public Car create() {
        //中间省略一万行代码
        return new Bmw();
    }
}

public class BikeFactory implements Factory {
    public Car create() {
        //中间省略一万行代码
        return new Bike();
    }
}
```

应用场景

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Client {
    public static void main(String[] args) {
        Factory benzFactory = new BenzFactory();
        Car benz = benzFactory.create();
        benz.run();
        Factory bmwFactory = new BmwFactory();
        Car bmw = bmwFactory.create();
        bmw.run();
    }
}
```

好处

此模式中，通过定义一个抽象的核心工厂类，并定义创建产品对象的接口，创建具体产品实例的工作延迟到其工厂子类去完成。这样做的好处是核心类只关注工厂类的接口定义，而具体的产品实例交给具体的工厂子类去创建。当系统需要新增一个产品是，无需修改现有系统代码，只需要添加一个具体产品类和其对应的工厂子类，使系统的扩展性变得很好，符合面向对象编程的开闭原则。

缺点

工厂方法模式虽然扩展性好，但是增加了编码难度，大量增加了类的数量，所以怎么选择还是看实际的需求。

二、 IOC 容器

首先聊聊控制反转

- 这不是什么技术，而是一种设计思想，就是将原本在程序中手动创建对象的控制权，交由Spring框架来管理。
- 以往的思路：若要使用某个对象，需要自己去负责对象的创建
- 反转的思路：若要使用某个对象，只需要从 Spring 容器中获取需要使用的对象，不关心对象的创建过程，也就是把创建对象的控制权反转给了Spring框架
- 好莱坞法则： Don't call me ,I'll call you

一个例子

家政服务为例：

过年了，今天楠哥（哪里都是张三）需要给家里打扫个卫生，于是想请几个钟点工来擦擦玻璃。

解决方案有几种：

1、自己主动的去，询问查找，看看谁认识钟点工，有了联系方式后，自己打电话邀约，谈价钱。

2、直接打电话给家政公司，提出要求即可。

第一种方式，就是我们之前接触的创建对象的方式，自己主动new，而第二种就是spring给我们提供的另外一种方式叫IOC，家政公司就像是一个大容器，能为我们提供很多的服务。

又过了几天，楠哥想清理一下油烟机，直接打电话给家政公司，师傅很快到达，帮助楠哥清理了油烟机。

聊聊例子中的问题

一定会有人问，那家政公司从哪里来啊！

1、自行构建

我们可以使用配置文件，或者注解的方式定义一下咱们自己的容器里存放的东西。

2、使用别人的

一定会有很多有钱人，成立自己的各类公司，他们的这些服务都可以集成在咱们的容器里，为我们提供很多强大的功能，比如spring自带的就很多template模板类。

1、操练一把

引入依赖（加入spring framework）

```
<dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.7</version>
            <scope>test</scope>
        </dependency>
        <!-- 引入web直接会将所有的依赖项全部依赖过来 -->
```

```
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.2.6.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.12</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.8</source> <!-- 源代码使用的JDK版本 -->
                    <target>1.8</target> <!-- 需要生成的目标class文件的编译版本 -->
                    <encoding>UTF-8</encoding><!-- 字符集编码 -->
                </configuration>
            </plugin>
        </plugins>
    </build>
```

写一个类（定义一个家政服务员类型）

```
/**
 * author IT楠老师
 * date 2020/5/13
 */
public class User {
    private String name;

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

    public User() {
    }

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

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

```
    }

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

配置文件（定义一下家政的创建方式，并把它注入容器）

```
<?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:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

</beans>
```

XML Schema命名空间作用：

1. 避免命名冲突，像Java中的package一样
2. 将不同作用的标签分门别类（像Spring中的tx命名空间针对事务类的标签，context命名空间针对组件的标签）

头文件解释：

1. xmlns="http://www.springframework.org/schema/beans" 声明xml文件默认的命名空间，表示未使用其他命名空间的所有标签的默认命名空间。
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 声明XML Schema 实例，声明后就可以使用schemaLocation 属性了，标准组织发布的。
3. xmlns:aop="http://www.springframework.org/schema/aop" 声明前缀为aop的命名空间，aop是简写，至是全命名空间
4. xsi:schemaLocation=" http://www.springframework.org/schema/beans

   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

这个地方，制定了某个命名空间的schema文件的具体位置

所以我们需要什么样的标签的时候，就引入什么样的命名空间和Schema 定义就可以了。

```
<?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就是java对象 ， 由Spring创建和管理-->
    <bean id="jiazheng" class="com.xinzhi.entity.User">
        <property name="name" value="IT楠老师"/>
    </bean>
</beans>
```

测试（用时从容器拿，而不是new）

```
@Test
public void testHello(){
    ApplicationContext applicationContext =
        new ClassPathXmlApplicationContext("applicationContext.xml");
    User user = applicationContext.getBean("jiazheng");
    System.out.println(user);
}
```

结果（效果一样）

```
User{name='IT楠老师'}
```

2、详细说说其中的几个对象

ApplicationContext是spring继BeanFactory之外的另一个核心接口或容器，允许容器通过应用程序上下文环境创建、获取、管理bean。为应用程序提供配置的中央接口。在应用程序运行时这是只读的。

```
public interface ApplicationContext extends EnvironmentCapable, ListableBeanFactory,
HierarchicalBeanFactory,
        MessageSource, ApplicationEventPublisher, ResourcePatternResolver {
}
```

一个ApplicationContext提供:

- 访问应用程序组件的Bean工厂方法。从org.springframework.beans.factory.ListableBeanFactory继承。
- 以通用方式加载文件资源的能力。继承自ResourcePatternResolver 接口。
- 向注册侦听器发布事件的能力。继承自ApplicationEventPublisher接口。
- 解析消息的能力，支持国际化。继承自MessageSource接口。

ConfigurableApplicationContext：

```
public interface ConfigurableApplicationContext extends ApplicationContext, Lifecycle,
Closeable {
}
```

该接口提供了根据配置创建、获取bean的一些方法，其中主要常用的实现包括：ClassPathXmlApplicationContext、FileSystemXmlApplicationContext等。提供了通过各种途径去加载实例化bean的手段。

ClassPathXmlApplicationContext是ConfigurableApplicationContext的一个实现

该类提供了很多记载资源的方式，该类可以加在类路劲下的xml资源文件，当然如果想加在其他地方的资源，可以使用FileSystemXmlApplicationContext这个类。

```
/**
     * Create a new ClassPathXmlApplicationContext, loading the definitions
     * from the given XML file and automatically refreshing the context.
     * @param configLocation resource location
     * @throws BeansException if context creation failed
     */
public ClassPathXmlApplicationContext(String configLocation) throws BeansException {
    this(new String[] {configLocation}, true, null);
}
```

3、体会依赖注入（DI）

对象是有属性的，属性是需要赋值的。通常的方式是：

```
Student student = new Student();
student.setName("张三");
student.setAge(23);
Teacher teacher = new Teacher();
student.setTeacher(teacher);
```

这需要我们手动使用set方法去赋值操作，当然我们还可以：

```
Student student = new Student("李四",12,teacher);
```

使用User的构造方法去赋值。

但是归根结底，都需要我们自己去赋值。

控制反转（IOC）也叫依赖注入（DI）的核心思想是，构建对象（包括初始化和赋值）都不需要人为操作，而是将这个权利交付给容器来进行。

（1）IOC有啥好处?

使用USB外部设备比使用内置硬盘有什么好处

- 第一、USB设备作为电脑主机的外部设备，在插入主机之前，与电脑主机没有任何的关系，只有被我们连接在一起之后，两者才发生联系，具有相关性。所以，无论两者中的任何一方出现什么的问题，都不会影响另一方的运行。这种特性体现在软件工程中，就是可维护性比较好，非常便于进行单元测试，便于调试程序和诊断故障。代码中的每一个Class都可以单独测试，彼此之间互不影响，只要保证自身的功能无误即可，这就是组件之间低耦合或者无耦合带来的好处。
- 第二、USB设备和电脑主机的之间无关性，还带来了另外一个好处，生产USB设备的厂商和生产电脑主机的厂商完全可以是互不相干的人，各干各事，**他们之间唯一需要遵守的就是USB接口标准。**这种特性体现在软件开发过程中，好处可是太大了。每个开发团队的成员都只需要关心实现自身的业务逻辑，完全不用去关心其它的人工作进展，因为你的任务跟别人没有任何关系，你的任务可以单独测试，你的任务也不用依赖于别人的组件，再也不用扯不清责任了。所以，在一个大中型项目中，团队成员分工明确、责任明晰，很容易将一个大的任务划分为细小的任务，开发效率和产品质量必将得到大幅度的提高。
- 第三、同一个USB外部设备可以插接到任何支持USB的设备，可以插接到电脑主机，也可以插接到DV机，USB外部设备可以被反复利用。在软件工程中，这种特性就是可复用性好，我们可以把具有普遍性的常用组件独立出来，反复利用到项目中的其它部分，或者是其它项目，当然这也是面向对象的基本特征。显然，IOC不仅更好地贯彻了这个原则，提高了模块的可复用性。符合接口标准的实现，都可以插接到支持此标准的模块中。
- 第四、同USB外部设备一样，模块具有热插拔特性。IOC生成对象的方式转为外置方式，也就是把对象生成放在配置文件里进行定义，这样，当我们更换一个实现子类将会变得很简单，只要修改配置文件就可以了，完全具有热插拔的特性。

暂且如此体会，以后的学习中会渗入理解。

（2）构造器注入

编写javaBean

```
/**
 * author IT楠老师
 * date 2020/5/13
 */
public class Dog {

    private String  name;
    private int age;

    public Dog(){}

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name;}
    public void setName(String name) {
```

```
        this.name = name;
    }

    public int getAge() {   return age; }
    public void setAge(int age) {
        this.age = age;
    }
}
```

无参构造

```
<bean id="goldilocks" class="com.xinzhi.entity.Dog" />
```

有参构造

注意：value属性注入的是基本类型，如果是引用类型要使用ref连接其他的bean。

```
<!-- 推荐使用，根据名字注入 -->
<bean id="goldilocks" class="com.xinzhi.entity.Dog">
    <constructor-arg name="name" value="goldilocks"/>
    <constructor-arg name="age" value="11"/>
</bean>
```

```
<!--根据参数顺序注入 -->
<bean id="goldilocks" class="com.xinzhi.entity.Dog">
    <constructor-arg index="0" value="goldilocks"/>
    <constructor-arg index="1" value="11"/>
</bean>
```

```
<!-- 直接注入也是按顺序注入，了解 -->
<bean id="goldilocks" class="com.xinzhi.entity.Dog">
    <constructor-arg value="goldilocks"/>
    <constructor-arg value="11"/>
</bean>
```

```
<!-- 按照参数类型注入，了解 -->
<bean id="goldilocks" class="com.xinzhi.entity.Dog">
    <constructor-arg type="java.lang.String" value="goldilocks"/>
    <constructor-arg type="java.lang.Integer" value="11"/>
</bean>
```

（3）set方法注入

要求被注入的属性，必须有set方法。

- set方法的方法名由set + 属性首字母大写
- 如果属性是boolean类型，没有set方法，是 is + 属性首字母大写 .

Address.java

```
/**
 * author IT楠老师
 * date 2020/5/18
 */
public class Address {
    private String addressInfo;
    public String getAddressInfo() {
        return addressInfo;
    }
    public void setAddressInfo(String addressInfo) {
        this.addressInfo = addressInfo;
    }
}
```

User.java

```
/**
 * author IT楠老师
 * date 2020/5/13
 */
public class User {
    private String name;
    private Address address;
    //爱好
    private String[] hobbies;
    //职务
    private List<String> duties;
    //家庭关系
    private Map<String,String> familyTies;
    //购物车商品
    private Set<String> carts;
    //工作经历
    private Properties workExperience;
    //女儿
    private String daughter;
    ...省略setter和getter，toString
}
```

配置文件，看各种类型的参数如何注入

```
<?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.xinzhi.entity.Address" >
        <property name="addressInfo" value="珠琳旺角大厦"/>
    </bean>
    <bean id="user" class="com.xinzhi.entity.User">
        <!-- 基本属性注入 -->
        <property name="name" value="楠哥" />
        <!-- 引用类型注入 -->
        <property name="address" ref="address"/>
        <!-- 数组注入 -->
        <property name="hobbies">
            <array>
                <value>写程序</value>
                <value>吃</value>
                <value>喝</value>
                <value>漂亮姑娘</value>
                <value>赢钱</value>
            </array>
        </property>
        <!-- list注入 -->
        <property name="duties">
            <list>
                <value>IT老师</value>
                <value>我儿子的爸爸</value>
            </list>
        </property>
        <!-- set注入 -->
        <property name="carts">
            <set>
                <value>纸尿裤</value>
                <value>玩具</value>
            </set>
        </property>
        <!-- map注入 -->
        <property name="familyTies">
            <map>
                <entry key="father" value="张某某" />
                <entry key="mather" value="钟某某" />
            </map>
        </property>
        <!-- property注入 -->
        <property name="workExperience">
            <props>
                <prop key="first">电厂职工</prop>
                <prop key="second">java开发工程师</prop>
                <prop key="third">java讲师</prop>
            </props>
        </property>
        <!-- null注入 -->
        <property name="daughter"><null /></property>
    </bean>
</beans>
```

测试

```
@Test
public void testDI(){
    ApplicationContext applicationContext =
        new ClassPathXmlApplicationContext("applicationContext.xml");
    User user = applicationContext.getBean(User.class);
    System.out.println(user);
}
```

结果

```
User{name='楠哥', address=Address{addressInfo='珠琳旺角大厦'}, hobbies=[写程序, 吃, 喝, 漂亮姑娘, 赢钱], duties=[IT老师, 我儿子的爸爸], familyTies={father=张某某, mather=钟某某}, carts=[纸尿裤, 玩具], workExperience={second=java开发工程师, third=java讲师, first=电厂职工}, daughter='null'}
```

成功!

4、Bean的作用域

Spring IOC容器创建一个Bean实例时，可以为Bean指定实例的作用域，作用域包括singleton（单例模式）、prototype（原型模式）、request（HTTP请求）、session（会话）、global-session（全局会话）。

（1）Singleton

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：

```
<bean id="userServiceImpl" class=""com.xinzhi.service.UserServiceImpl">
```

（2）Prototype

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="account" class="com.xinzhi.entity.User" scope="prototype"/>
```

（3）Request

当一个bean的作用域为Request

表示在一次HTTP请求中，一个bean定义对应一个实例；即每个HTTP请求都会有各自的bean实例，它们依据某个bean定义创建而成。该作用域仅在基于web的Spring ApplicationContext情形下有效。考虑下面bean定义：

```
<bean id="loginAction" class="com.xinzhi.entity.User" scope="request"/>
```

针对每次HTTP请求，Spring容器会根据loginAction bean的定义创建一个全新的LoginAction bean实例，且该loginAction bean实例仅在当前HTTP request内有效，因此可以根据需要放心的更改所建实例的内部状态，而其他请求中根据loginAction bean定义创建的实例，将不会看到这些特定于某个请求的状态变化。当处理请求结束，request作用域的bean实例将被销毁。

（4）Session

(4)当一个bean的作用域为Session，表示在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。考虑下面bean定义：

```
<bean id="userPreferences" class="com.xinzhi.entity.User" scope="session"/>
```

针对某个HTTP Session，Spring容器会根据userPreferences bean定义创建一个全新的userPreferences bean实例，且该userPreferences bean仅在当前HTTP Session内有效。与request作用域一样，可以根据需要放心的更改所创建实例的内部状态，而别的HTTP Session中根据userPreferences创建的实例，将不会看到这些特定于某个HTTP Session的状态变化。当HTTP Session最终被废弃的时候，在该HTTP Session作用域内的bean也会被废弃掉。

5、自动装配

spring作为长期以来java最火的框架，其IOC做的十分的健全，以上情况都是我们手动装配，但是我们也说了spring及其灵活的帮助我们完成了解耦工作，如果所以的类都是自己手动完成注入，他的解耦能力就不会体现的那么强烈了，于是spring还为我们提供了自动装配的能力

只要我们的Beans满足bean之间的依赖，且这些bean都存在于容器，且唯一，那么我么就可以按照约定进行bean的自动装配。同时还能大量的减少配置文件的数量

本章使用配置文件的方式进行自动注入，下章会开启全新的注解方式注入。

测试环境搭建

删除user配置文件中的address注入，尝试让程序自动装配

```
<!-- 引用类型注入 -->
<property name="address" ref="address"/>
```

（1）按名称自动装配

1、修改bean配置，增加一个属性 autowire="byName"

```
<bean id="user" class="com.xinzhi.entity.User"  autowire="byName">
    ...中间省略
</bean>
```

2、再次测试，成功输出！

```
User{name='楠哥', address=Address{addressInfo='珠琳旺角大厦'}, hobbies=[写程序, 吃, 喝, 漂亮姑娘, 赢钱], duties=[IT老师, 我儿子的爸爸], familyTies={father=张某某, mather=钟某某}, carts=[纸尿裤, 玩具], workExperience={second=java开发工程师, third=java讲师, first=电厂职工}, daughter='null'}
```

3、修改address的名字

```
<bean id="address2" class="com.xinzhi.entity.Address" >
    <property name="addressInfo" value="珠琳旺角大厦"/>
</bean>
```

4、再次测试，注入失败，为空

```
User{name='楠哥', address=null, hobbies=[写程序, 吃, 喝, 漂亮姑娘, 赢钱], duties=[IT老师, 我儿子的爸爸], familyTies={father=张某某, mather=钟某某}, carts=[纸尿裤, 玩具], workExperience={second=java开发工程师, third=java讲师, first=电厂职工}, daughter='null'}
```

过程

1. 冲javabena的set方法获取名字，如setTeacher，就去spring容器中找名字为teacher的bean。
2. 如果有，就取出注入；如果没有，就不注入。

（2）按类型自动装配

1、将user的bean配置修改一下 ,增加 autowire="byType"

```
<bean id="user" class="com.xinzhi.entity.User"  autowire="byType">
    ...中间省略
</bean>
```

2、测试，正常输出

```
User{name='楠哥', address=Address{addressInfo='珠琳旺角大厦'}, hobbies=[写程序, 吃, 喝, 漂亮姑娘, 赢钱], duties=[IT老师, 我儿子的爸爸], familyTies={father=张某某, mather=钟某某}, carts=[纸尿裤, 玩具], workExperience={second=java开发工程师, third=java讲师, first=电厂职工}, daughter='null'}
```

3、再注册一个address对象

```
<bean id="address" class="com.xinzhi.entity.Address" >
    <property name="addressInfo" value="珠琳旺角大厦"/>
</bean>
<bean id="address2" class="com.xinzhi.entity.Address" >
    <property name="addressInfo" value="恒大名都"/>
</bean>
<bean id="user" class="com.xinzhi.entity.User"  autowire="byType"></bean>
```

4、测试

```
org.springframework.beans.factory.UnsatisfiedDependencyException:
Error creating bean with name 'user' defined in class path resource
[applicationContext.xml]: Unsatisfied dependency expressed through bean property
'address'; nested exception is
org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of
type 'com.xinzhi.entity.Address' available: expected single matching bean but found 2:
address,address2
```

报错了：期望是一个被匹配的bean但是发现了两个。

> **按照类型注入，容器内的bean必须是单例**

6、注解开发

修改文件，在user的address属性上加注解@Autowired

```
@Autowired
private Address address;
```

删除user配置文件中的address注入

```
<!-- 引用类型注入 -->
<property name="address" ref="address"/>
```

测试看结果：

```
User{name='楠哥', address=null, hobbies=[写程序, 吃, 喝, 漂亮姑娘, 赢钱], duties=[IT老师, 我儿子的爸爸], familyTies={father=张某某, mather=钟某某}, carts=[纸尿裤, 玩具], workExperience={second=java开发工程师, third=java讲师, first=电厂职工}, daughter='null'}
```

发现注解没有生效，address是null。

> **思考**

- 注解只是个标识，我们需要去解析他，但是解析就要在运行前看一看。
- 所以要让注解生效，还需要进行一个配置，在运行前，扫描一下所有的文件，看看哪些类头上戴了我的注解。
- 自己写扫描文件的方法太麻烦，spring当然提供了方式。

1、配置文件需要加头文件

```
<?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
       https://www.springframework.org/schema/context/spring-context.xsd
>
```

2、在配置文件中加次注解，就是告诉spring【com.xinzhi】下的类都扫描一下看看

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

3、测试搞定

```
User{name='楠哥', address=Address{addressInfo='珠琳旺角大厦'}, hobbies=[写程序, 吃, 喝, 漂亮姑娘, 赢钱], duties=[IT老师, 我儿子的爸爸], familyTies={father=张某某, mather=钟某某}, carts=[纸尿裤, 玩具], workExperience={second=java开发工程师, third=java讲师, first=电厂职工}, daughter='null'}
```

> **接下来就谈谈spring为我们提供了哪些好用的注解**

（1）第一组，自动装配

> **@Autowired**

- @Autowired是按类型自动转配的，不支持id匹配。
- 需要导入 spring-aop的包！

> **上一个例子已经说了，这里不再赘述**

> **@Qualifier**

@Qualifier不能单独使用，它和@Autowired配合可以根据名称进行自动装配

测试实验步骤：

1、配置文件中的address名字为address2

```
<bean id="dog1" class="com.xinzhi.entity.Dog"/>
<bean id="dog2" class="com.xinzhi.entity.Dog"/>
<bean id="cat1" class="com.xinzhi.entity.Cat"/>
<bean id="cat2" class="com.xinzhi.entity.Cat"/>
```

2、没有加Qualifier测试，直接报错

3、在属性上添加Qualifier注解

```
@Autowired
@Qualifier(value = "cat2")
private Cat cat;
@Autowired
@Qualifier(value = "dog2")
private Dog dog;
```

测试，成功输出！

@Resource

- @Resource如有指定的name属性，先按该属性进行byName方式查找装配，写了啥名字就找哪个；
- 其次再进行默认的byName方式进行装配；
- 如果以上都不成功，则按byType的方式自动装配。
- 都不成功，则报异常。

实体类：

```
public class User {
   //如果允许对象为null，设置required = false,默认为true
   @Resource(name = "cat2")
   private Cat cat;
   @Resource
   private Dog dog;
   private String str;
}
```

beans.xml

```
<bean id="dog" class="com.xinzhi.entity.Dog"/>
<bean id="cat1" class="com.xinzhi.entity.Cat"/>
<bean id="cat2" class="com.xinzhi.entity.Cat"/>

<bean id="user" class="com.xinzhi.entity.User"/>
```

测试：结果OK

配置文件2：beans.xml， 删掉cat2

```
<bean id="dog" class="com.xinzhi.entity.Dog"/>
<bean id="cat1" class="com.xinzhi.entity.Cat"/>
```

实体类上只保留注解

```
@Resource
private Cat cat;
@Resource
private Dog dog;
```

结果：OK

结论：先进行byName查找，失败；再进行byType查找，成功

> **@Autowired与@Resource异同：**

1、@Autowired与@Resource都可以用来装配bean。都可以写在字段上，或写在setter方法上。

2、@Autowired默认按类型装配（属于spring规范），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

3、@Resource（属于J2EE复返），

1、按照name的值自行装配

2、byName

3、byType

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。

> **为了减少配置文件数量，spring提供了大量的注解进行开发**

（2）第二组，分层开发

@Controller

@Service

@Repository

> **领略一下风采**

1、dao层，注意@Repository要加在实现类上

```
/**
 * author IT楠老师
 * date 2020/5/13
 */
public interface IUserDao {

    /**
     * 保存用户
     * @param user
     */
```

```
    int saveUser(User user);
}


/**
 * author IT楠老师
 * date 2020/5/13
 */
@Repository
public class UserOracleDao implements IUserDao {
    public int saveUser(User user) {

        System.out.println("我是将数据保存至oracle");
        return 1;
    }
}
```

2、service层，注意@Service要加在实现类上

```
/**
 * author IT楠老师
 * date 2020/5/13
 */
public interface IUserService {

    /**
     * 注册业务
     * @param user
     */
    void register(User user);

}

/**
 * author IT楠老师
 * date 2020/5/13
 */
//表明是一个service
@Service
public class UserServiceImpl implements IUserService {

    //自动注入该类型的dao的实现
    @Autowired
    private IUserDao userDao;

    public void register(User user) {
        System.out.println(user.getName()+"注册成功！");
        userDao.saveUser(user);
    }
}
```

@Controller等咱们学了springmvc再演示。

其实一个注解节能搞定，@Bean能代替所有，但是应为分层的原因，使用不同的注解可读性更高。

这些注解就是在容器启动时生成bean，和配置文件的作用一样。

3、测试

```
    @Test
    public void testAnnotation(){
        ApplicationContext applicationContext =
                new ClassPathXmlApplicationContext("applicationContext.xml");
        IUserService bean = applicationContext.getBean(IUserService.class);
        User user = applicationContext.getBean(User.class);
        bean.register(user);
    }
```

结果：

```
楠哥注册成功!
我是将数据保存至oracle
```

我们以后写其实是都是这么写的。

使用注解注入一个

（3）第三组，组件开发

@Component

@Value

```
/**
 * author IT楠老师
 * date 2020/5/13
 */
@Component
public class Student {

    @Value("张三")
    private String name;
    @Autowired
    private Address address;
}
```

知识开启了包扫描，并没有配置

```
@Test
public void testStudent(){
    ApplicationContext applicationContext =
        new ClassPathXmlApplicationContext("applicationContext.xml");
    Student user = applicationContext.getBean(Student.class);
    System.out.println(user);
}
```

结果

```
Student{name='张三', address=Address{addressInfo='珠琳旺角大厦'}}
```

关于@Value的高级用法，学了springboot继续研究。

（4）第四组，配置文件

@Bean

@Configuration

```
/**
 * author IT楠老师
 * date 2020/5/18
 */
@Configuration
public class AnimalConfig {

    @Bean
    public Mouse mouse(){
        Mouse mouse = new Mouse();
        mouse.setAge(12);
        mouse.setName("jerry");
        return mouse;
    }

    //参数是你要注入的其他的bean
    @Bean
    public Cat cat(Mouse mouse){
        Cat cat = new Cat();
        cat.setAge(5);
        cat.setName("Tom");
        cat.setFood(mouse);
        return cat;
    }
}
```

测试

```
@Test
public void testCatAndMouse(){
    ApplicationContext applicationContext =
        new ClassPathXmlApplicationContext("applicationContext.xml");
    Mouse mouse = applicationContext.getBean(Mouse.class);
    System.out.println(mouse);
    Cat cat = applicationContext.getBean(Cat.class);
    System.out.println(cat);
}
```

结果，正确输出

```
Dog{name='jerry', age=12}
Cat{name='Tom', age=5, food=Dog{name='jerry', age=12}}
```

小结

很多注解的功能一样，都是讲javabean注入到容器，知识多种多样的注解使得我们的语义化更加友好，不要不自己纠结于为啥这么多注解，好难啊。

更多的注解会在学习springboot的时候接触

7、集成spring测试环境

添加依赖

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.2.6.RELEASE</version>
</dependency>
```

测试用例

```
/**
 * @author IT楠老师
 * @date 2020/5/28
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:beans.xml")
public class Test2 {

    @Resource
    private MaleActor maleActor;

    @Test
    public void testLiftCircle(){
        maleActor.work();
```

```
    }
}
```

能够更加简单的帮助我们完成spring的测试工作。

8、jdbcTemplate感受依赖注入的好处

配置数据源

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.18</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.6.RELEASE</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>
```

spring为我们提供了一个jdbc操作模板，只需要注入一下就行了

```
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="username" value="root"/>
    <property name="password" value="root"/>
    <property name="url" value="jdbc:mysql://localhost:3306/ssm?
useSSL=false&amp;useUnicode=true&amp;characterEncoding=utf8"/>
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
</bean>

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

思考？能不能用@Configurable注解注入这个bean

```
/**
 * @author IT楠老师
 * @date 2020/5/28
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:beans.xml")
public class Test2 {
```

```
    @Resource
    private JdbcTemplate template;

    @Test
    public void testLiftCircle(){
//        template.execute("insert into user values(100,'aa','bbb')");

        //这玩意就是建立映射关系，避免不了，这里是测试，实际上还是新建一个类好。
        RowMapper<User> rowMapper = (ResultSet resultSet, int i) -> {
            User user = new User();
            user.setId(resultSet.getInt("u_id"));
            user.setUsername(resultSet.getString("u_user_name"));
            user.setPassword(resultSet.getString("u_password"));
            return user;
        };

        List<User> query = template.query("select * from user", rowMapper);
        System.out.println(query);
    }
}
```

> **总结：当第三方的插件和spring集成的时候，我们只需要注入一个bean，他所有的功能我们就都能使用了，是不是很爽。**

三、Bean的生命周期（面试老问）

> **这玩意是面试常问的一个问题**

生命周期不管在哪个概念里都是说明了一个 `东西` 从生到死的一个过程。

咱们学了以上的内容分析一下

> **在这个bean的各个阶段加入一些内容**

```
/**
 * @author IT楠老师
 * @date 2020/5/28
 */

public class User implements Serializable {
    private int id;
    private String username;
    private String password;

    public User() {
        System.out.println("--------构造--------");
    }

    public int getId() {
```

```
        return id;
    }

    public void setId(int id) {
        System.out.println("--------注入--------");
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public void init(){
        System.out.println("--------init初始化--------");
    }

    public void destroy(){
        System.out.println("--------destroy销毁--------");
    }

    @Override
    public String toString() {
        System.out.println("--------使用了--------");
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                '}';
    }
}
```

注入bean

```
<bean id="user" class="com.entity.User" init-method="init" destroy-method="destroy">
    <property name="id" value="1"/>
    <property name="username" value="zhangsna"/>
    <property name="password" value="123"/>
</bean>
```

测试

```
@Test
public void testLiftCircle(){
    ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:beans.xml");
    User bean = applicationContext.getBean(User.class);
    System.out.println(bean);
    applicationContext.close();
}
```

结果

```
--------构造--------
--------注入--------
--------init初始化--------
--------使用了--------
User{id=1, username='zhangsna', password='123'}
10:16:45.035 [main] DEBUG
org.springframework.context.support.ClassPathXmlApplicationContext - Closing
org.springframework.context.support.ClassPathXmlApplicationContext@2ff4f00f, started on
Sat May 30 10:16:44 CST 2020
--------destroy销毁--------
```

扩展

```
/**
 * @author IT楠老师
 * @date 2020/5/30
 */
public class BeanHandler implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
        System.out.println("--------初始化前--------");
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        System.out.println("--------初始化后--------");
        return bean;
    }

}
```

注入

```
<bean class="com.handler.BeanHandler"/>
```

结果

```
--------构造--------
--------注入--------
--------初始化前--------
--------init初始化--------
--------初始化后--------
--------使用了--------
User{id=1, username='zhangsna', password='123'}
10:18:33.163 [main] DEBUG
org.springframework.context.support.ClassPathXmlApplicationContext - Closing
org.springframework.context.support.ClassPathXmlApplicationContext@2ff4f00f, started on
Sat May 30 10:18:32 CST 2020
--------destroy销毁--------
```

看一个博客：https://www.cnblogs.com/zrtqsk/p/3735273.html

四、AOP

1、代理模式回顾

代理模式，使得在IOC中实现AOP变得可能。

代理模式分为静态代理和动态代理。代理的核心功能是方法增强。

（1）静态代理

静态代理角色分析

- 抽象角色 : 一般使用接口或者抽象类来实现
- 真实角色 : 被代理的角色
- 代理角色 : 代理真实角色 ; 代理真实角色后 , 一般会做一些附属的操作 .
- 客户 : 使用代理角色来进行一些操作 .

代码实现

写一个接口

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public interface Singer {
    /**
     * 歌手的统一方法
     */
    void sing();
}
```

MaleSinger. java 即真实角色

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class MaleSinger implements Singer {
    @Override
    public void sing() {
        System.out.println("我要唱歌了！！！");
    }
}
```

Agent. java 即代理角色

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 * 经纪人
 */
public class Agent implements Singer {

    private Singer singer;

    /**
     * 经纪人必须代理一个歌手
     * @param singer
     */
    Agent(Singer singer){
        this.singer = singer;
    }

    @Override
    public void sing() {
        System.out.println("经纪人要把把关，看看这个演出合适不合适！");
        singer.sing();
        System.out.println("歌手唱完歌，经纪人去收钱！");
    }
}
```

Client . java 即客户

```java
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Client {
    public static void main(String[] args) {
        Singer luhan = new MaleSinger();
        Agent zhangsan = new Agent(luhan);
        zhangsan.sing();
    }
}
```

分析：在这个过程中，你直接接触的就是鹿晗的经济人，经纪人在鹿晗演出的前后跑前跑后发挥了巨大的作用。

优点

- 鹿晗还是鹿晗，没有必要为了一下前置后置工作改变鹿晗这个类
- 公共的统一问题交给代理处理，这不就是
- 公共业务进行扩展或变更时，可以更加方便
- 这不就是更加符合开闭原则，单一原则吗?

缺点：

- 每个类都写个代理，麻烦死了。

（2）动态代理

- 动态代理的角色和静态代理的一样 .
- 动态代理的代理类是动态生成的 . 静态代理的代理类是我们提前写好的
- 动态代理分为两类 : 一类是基于接口动态代理 , 一类是基于类的动态代理
-
  - 基于接口的动态代理----JDK动态代理
  - 基于类的动态代理--cglib
  - 现在用的比较多的是 javasist 来生成动态代理 . 百度一下javasist
  - 我们这里使用JDK的原生代码来实现，其余的道理都是一样的！、

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(),
                                 singger.getClass().getInterfaces(),this);
}
```

代码实现

抽象角色和真实角色和之前的一样!

还是歌星和男歌星

Agent. java 即代理角色

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 * 经纪人
 */
public class Agent implements InvocationHandler {

    private Singer singer;

    /**
     * 设置代理的经济人
     * @param singer
     */
    public void setSinger(Singer singer) {
        this.singer = singer;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("---------------经纪人把把关--------------");
        Object returnObj = method.invoke(singer, args);
        System.out.println("---------------唱完了收收钱-----------------------");
        return returnObj;
    }

    /**
     * 获取一个代理对象
```

```
     * @return
     */
    public Object getProxy(){
        return Proxy.newProxyInstance(this.getClass().getClassLoader(),
                new Class[]{Singer.class},this);
    }
}
```

Client . java

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Client {
    public static void main(String[] args) {

        MaleSinger luhan = new MaleSinger();

        Agent agent = new Agent();
        agent.setSinger(luhan);
        Singer singer = (Singer)agent.getProxy();

        singer.sing();
    }
}
```

核心：**一个动态代理，一般代理某一类业务，一个动态代理可以代理多个类，代理的是接口！、**

（3）万能代理

批量做代理，不用一个一个写

我们来使用动态代理实现代理我们后面写的UserService！

我们也可以编写一个通用的动态代理实现的类！所有的代理对象设置为Object即可！

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class ProxyAll implements InvocationHandler {
    private Object target;

    /**
     * 设置代理的经济人
     * @param target
     */
    public void setSinger(Object target) {
        this.target = target;
```

```
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("---------------之前干点啥！比如打印日志，开启事务？--------------
");
        Object returnObj = method.invoke(target, args);
        System.out.println("---------------之后干点啥！比如打印日志，关闭事务？--------------
----------");
        return returnObj;
    }

    /**
     * 获取一个代理对象
     * @return
     */
    public Object getProxy(){
        return Proxy.newProxyInstance(this.getClass().getClassLoader(),
                target.getClass().getInterfaces(),this);
    }
}
```

测试！

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Client {
    public static void main(String[] args) {

        MaleSinger luhan = new MaleSinger();

        ProxyAll agent = new ProxyAll();
        agent.setSinger(luhan);
        Singer singer = (Singer)agent.getProxy();

        singer.sing();
    }
}
```

测试，增删改查，查看结果，依然可以

```
---------------之前干点啥！比如打印日志，开启事务？--------------
我要唱歌了！！！
---------------之后干点啥！比如打印日志，关闭事务？---------------
```

> **思考那我们其他所有的类是不是都可以了？**

比如讲IOC的Bean遍历一下全部搞成代理，是不是就能在方法执行前后搞事情了！

（4）cglib生成代理

CGLIB（Code Generator Library）是一个**强大的、高性能**的代码生成库。其被广泛应用于AOP框架（Spring、dynaop）中，用以提供方法拦截操作。

为什么使用CGLIB

CGLIB代理主要通过对字节码的操作，为对象引入间接级别，以控制对象的访问。我们知道Java中有一个动态代理也是做这个事情的，那我们为什么不直接使用Java动态代理，而要使用CGLIB呢？答案是CGLIB相比于JDK动态代理更加强大，JDK动态代理虽然简单易用，但是其有一个致命缺陷是，只能对接口进行代理。如果要代理的类为一个普通类、没有接口，那么Java动态代理就没法使用了。

CGLIB组成结构

CGLIB底层使用了ASM（一个短小精悍的字节码操作框架）来操作字节码生成新的类。除了CGLIB库外，脚本语言（如Groovy何BeanShell）也使用ASM生成字节码。ASM使用类似SAX的解析器来实现高性能。我们不鼓励直接使用ASM，因为它需要对Java字节码的格式足够的了解。

说了这么多，可能大家还是不知道CGLIB是干什么用的。下面我们将使用一个简单的例子来演示如何使用CGLIB对一个方法进行拦截。 首先，我们需要在工程的POM文件中引入cglib的dependency，这里我们使用的是2.2.2版本

```
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>2.2.2</version>
</dependency>
```

依赖包下载后，我们就可以干活了，按照国际惯例，写个hello world

```
/**
 * @author IT楠老师
 * @date 2020/5/30
 */
public class MaleActor {

    private String name;

    public MaleActor() {
    }

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

    public void work(){
        System.out.println(this.name + "开始工作! ");
    }
}
```

实现

```
/**
 * @author IT楠老师
```

```
 * @date 2020/5/30
 */
public class Client {
    public static void main(String[] args) {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(MaleActor.class);
        enhancer.setCallback(new InvocationHandler() {
            @Override
            public Object invoke(Object o, Method method, Object[] objects) throws
Throwable {
                System.out.println("经济人和马蓉乱搞！");
                Object invoke = method.invoke(new MaleActor("王宝强"), objects);
                System.out.println("经纪人和马蓉转移费用！");
                return invoke;
            }
        });
        MaleActor agent = (MaleActor) enhancer.create();
        agent.work();
    }
}
```

在mian函数中，我们通过一个`Enhancer`和一个`MethodInterceptor`来实现对方法的拦截，运行程序后输出为：

```
经济人和马蓉乱搞！
王宝强开始工作！
经纪人和马蓉转移费用！
```

Enhancer

Enhancer可能是CGLIB中最常用的一个类，和Java1.3动态代理中引入的Proxy类差不多(如果对Proxy不懂，可以参考这里)。和Proxy不同的是，Enhancer既能够代理普通的class，也能够代理接口。Enhancer创建一个被代理对象的子类并且拦截所有的方法调用（包括从Object中继承的toString和hashCode方法）。Enhancer不能够拦截final方法，例如Object.getClass()方法，这是由于Java final方法语义决定的。基于同样的道理，Enhancer也不能对fianl类进行代理操作。这也是Hibernate为什么不能持久化final class的原因。

区别

java原生的是实现同一个接口代理和被代理是兄弟类。

cglib是集成被代理对象，是父子类。

2、面向切面编程

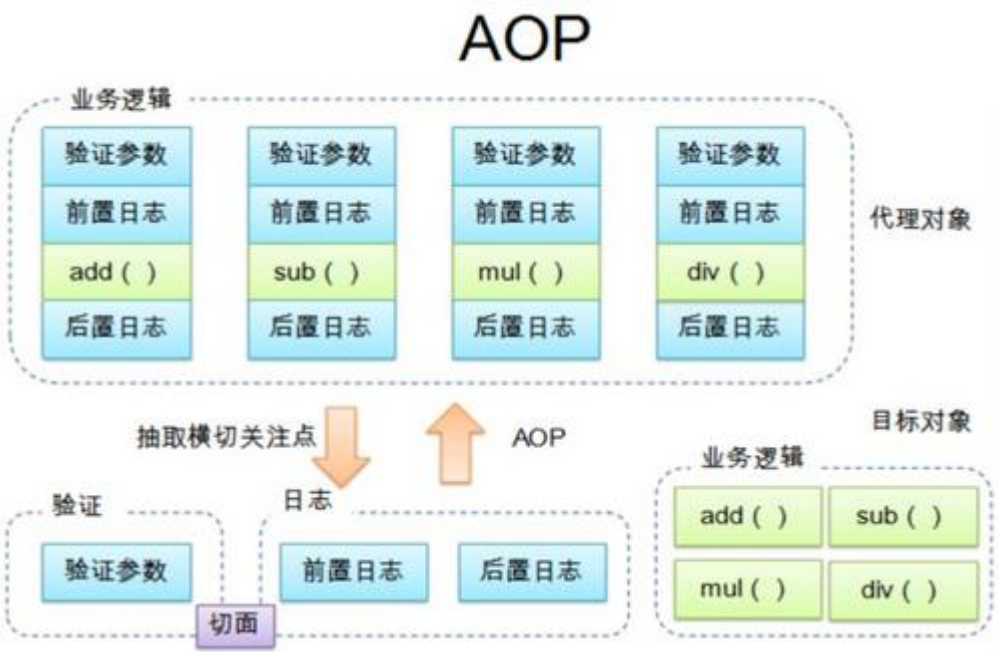
什么是AOP

- AOP（Aspect Oriented Programming）称为面向切面编程，在程序开发中主要用来解决一些系统层面上的问题，比如**日志，事务，权限**等待。
- 在不改变原有的逻辑的基础上，增加一些额外的功能。代理也是这个功能、。
- AOP可以说是OOP（Object Oriented Programming，面向对象编程）的补充和完善。OOP引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过OOP允许开发者定义纵向的关系，但并不适合定义横向的关系，例如日志功能。日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，这种散布在各处的无关的代码被称为横切（cross cutting），在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。
- AOP技术恰恰相反，它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。
- 使用"横切"技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志、事物。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

长话短说

就是有这样的需求，需要在某一些类中，增加很多统一的代码，比如，日志，事务，权限等。

- 一种方案是，每个类里自己手写，当有新的类是要注意不能忘记，重复繁琐。
- 另一种方案是，让程序自动加上需要的统一代码。说是加，其实就是方法增强，增强后的对象是一个代理对象而非原对象，正是有了bean工厂我们可以对bean进行统一处理，才能更好的实现统一生成代理的策略。
- 由此也告诉大家一个结论，spring的bean容器内方的都是代理对象而非原对象，我们可以任意定制代理的内容，对原生bean进行功能的扩展。



和AOP相关的名词：

通知、增强处理（Advice）

通知、增强处理（Advice） 就是你想要的功能，也就是上说的安全、事物、日子等。你给先定义好，然后再想用的地方用一下。包含Aspect的一段处理代码

连接点（JoinPoint）

连接点（JoinPoint） 这个就更好解释了，就是spring允许你是通知（Advice）的地方，那可就真多了，基本每个方法的钱、后（两者都有也行），或抛出异常是时都可以是连接点，spring只支持方法连接点。其他如AspectJ还可以让你在构造器或属性注入时都行，不过那不是咱们关注的，只要记住，和方法有关的前前后后都是连接点。

切入点（Pointcut）

切入点（Pointcut） 上面说的连接点的基础上，来定义切入点，你的一个类里，有15个方法，那就有十几个连接点了对吧，但是你并不想在所有方法附件都使用通知（使用叫织入，下面再说），你只是想让其中几个，在调用这几个方法之前、之后或者抛出异常时干点什么，那么就用切入点来定义这几个方法，让切点来筛选连接点，选中那几个你想要的方法。

切面（Aspect）

切面（Aspect） 切面是通知和切入点的结合。现在发现了吧，没连接点什么事，链接点就是为了让你好理解切点搞出来的，明白这个概念就行了。通知说明了干什么和什么时候干（什么时候通过方法名中的befor，after，around等就能知道），二切入点说明了在哪干（指定到底是哪个方法），这就是一个完整的切面定义。

引入（introduction）

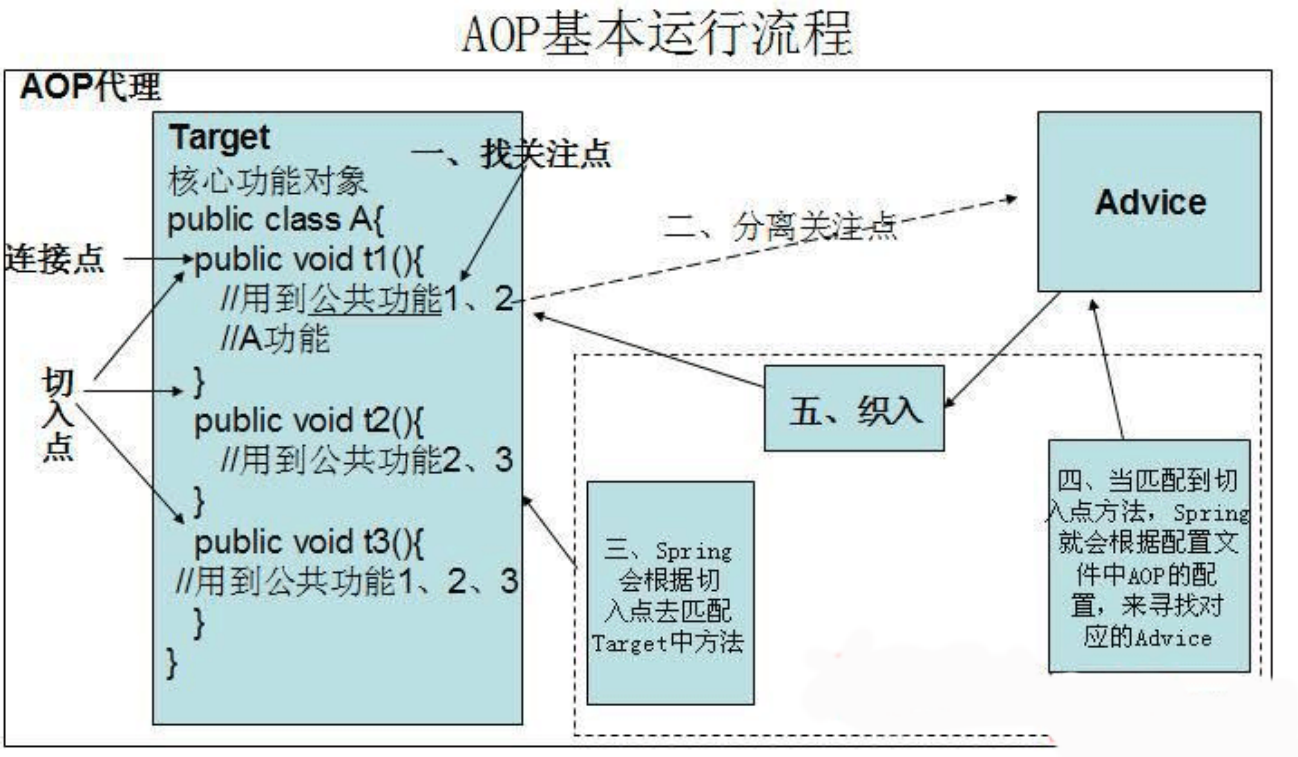
引入（introduction） 允许我们向现有的类添加新方法属性。这不就是把切面（也就是新方法属性：通知定义的）用到目标类中吗

目标（target）

目标（target） 引入中所提到的目标类，也就是要被通知的对象，也就是真正的业务逻辑，他可以在毫不知情的情况下，被咋们织入切面。二自己专注于业务本身的逻辑。

织入（weaving）

织入（weaving） 把切面应用到目标对象来创建新的代理对象的过程。



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

即 Aop 在 不改变原有代码的情况下 , 去增加新的功能 .

使用Spring实现Aop

【重点】使用AOP织入，需要导入一个依赖包!

```
<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
<dependency>
   <groupId>org.aspectj</groupId>
   <artifactId>aspectjweaver</artifactId>
   <version>1.9.4</version>
</dependency>
```

（1）spring内置的接口

首先编写我们的业务接口和实现类

加入头文件

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

http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd
```

```java
/**
 * author IT楠老师
 * date 2020/5/13
 */
public interface IUserService {

    /**
     * 注册业务
     * @param user
     */
    void register(User user);

    /**
     * 登录业务
     * @param username
     * @param password
     */
    void login(String username,String password);

}
/**
 * author IT楠老师
 * date 2020/5/13
 */
@Service
public class UserServiceImpl implements IUserService {

    @Autowired
    private IUserDao userDao;

    public void register(User user) {
        System.out.println(user.getName()+"注册成功！");
        userDao.saveUser(user);
    }

    public void login(String username, String password) {
        System.out.println("登录成功！");
    }
}
```

然后去写我们的增强类，我们编写两个，一个前置增强 一个后置增强

```java
/**
 * author IT楠老师
 * date 2020/5/18
 */
public class LogAfter implements AfterReturningAdvice {
    public void afterReturning(Object o, Method method, Object[] objects, Object o1)
throws Throwable {
        System.out.println(method.getName()+"：之后打印一条日志！  -- 内置接口形式");
    }
}
```

```java
/**
 * author IT楠老师
 * date 2020/5/18
 */
public class LogBefore implements MethodBeforeAdvice {
    public void before(Method method, Object[] objects, Object o) throws Throwable {
        System.out.println(method.getName()+": 之前打印一条日志! -- 内置接口形式");
    }
}
```

最后去spring的文件中注册，并实现aop切入实现，注意导入约束 .

```xml
<bean id="logBefore" class="com.xinzhi.aop.LogBefore"/>
<bean id="logAfter" class="com.xinzhi.aop.LogAfter"/>
<!--aop的配置-->
<aop:config>
    <!--切入点 expression:表达式匹配要执行的方法-->
    <aop:pointcut id="pointcut" expression="execution(* com.xinzhi.service.impl.*.*
(..))"/>
    <!--执行环绕; advice-ref执行方法 . pointcut-ref切入点-->
    <aop:advisor advice-ref="logBefore" pointcut-ref="pointcut"/>
    <aop:advisor advice-ref="logAfter" pointcut-ref="pointcut"/>
</aop:config>
```

测试

```java
@Test
public void testAOP1(){
    ApplicationContext applicationContext =
        new ClassPathXmlApplicationContext("applicationContext.xml");
    IUserService userService = applicationContext.getBean(IUserService.class);
    User user = applicationContext.getBean(User.class);
    userService.register(user);
    System.out.println("------开始登录--------");
    userService.login("","");
}
```

结果

```
register: 之前打印一条日志! -- 内置接口形式
楠哥注册成功!
我是将数据保存至oracle
register: 之后打印一条日志!  -- 内置接口形式
------开始登录--------
login: 之前打印一条日志! -- 内置接口形式
登录成功!
login: 之后打印一条日志!  -- 内置接口形式
```

Aop的重要性 : 很重要 . 一定要理解其中的思路 , 主要是思想的理解这一块 .

Spring的Aop就是将公共的业务 (日志 , 安全等) 和领域业务结合起来 , 当执行领域业务时 , 将会把公共业务加进来 . 实现公共业务的重复利用 . 领域业务更纯粹 , 程序猿专注领域业务 , 其本质还是动态代理 .

(2) 自定义类和方法实现

目标业务类不变依旧是userServiceImpl

第一步 : 写我们自己的一个切入类

```
/**
 * author IT楠老师
 * date 2020/5/18
 */
public class MyAop {
    public void before(){
        System.out.println("执行方法之前打印一条日志!  -- 自定义形式");
    }

    public void after(){
        System.out.println("执行方法之后打印一条日志!  -- 自定义形式");
    }
}
```

去spring中配置

```
<!--注册bean-->
<bean id="myAop" class="com.xinzhi.aop.MyAop"/>

<!--aop的配置-->
<aop:config>
    <!--第二种方式：使用AOP的标签实现-->
    <aop:aspect ref="myAop">
        <aop:pointcut id="pointcut" expression="execution(* com.xinzhi.service.impl.*.*
(..))"/>
        <aop:before pointcut-ref="pointcut" method="before"/>
        <aop:after pointcut-ref="pointcut" method="after"/>
    </aop:aspect>
</aop:config>
```

测试:

```
@Test
public void testAOP2(){
    ApplicationContext applicationContext =
        new ClassPathXmlApplicationContext("applicationContext.xml");
    IUserService userService = applicationContext.getBean(IUserService.class);
    User user = applicationContext.getBean(User.class);
    userService.register(user);
    System.out.println("------开始登录--------");
    userService.login("","");
}
```

> **自定义的方式无法拿到对象和方法，灵活性不如spring内置接口**

（3）注解实现（当下主流）

第一步：编写一个注解实现的增强类

```
/**
 * author IT楠老师
 * date 2020/5/18
 */
@Component
@Aspect
public class AnnotationAOP {

    //springaop自动的5种aop这里全部列出
    @Before("execution(* com.xinzhi.service.impl.*.*(..))")
    public void before(){
        System.out.println("---------方法执行前before()---------");
    }

    @After("execution(* com.xinzhi.service.impl.*.*(..))")
    public void after(){
        System.out.println("---------方法执行后after()---------");
    }

    @AfterReturning("execution(* com.xinzhi.service.impl.*.*(..))")
    public void afterReturning(){
        System.out.println("---------方法返回后afterReturning()---------");
    }

    @Around("execution(* com.xinzhi.service.impl.*.*(..))")
    public void around(ProceedingJoinPoint jp) throws Throwable {
        System.out.println("-------环绕前-------");
        System.out.println("签名（拿到方法名）:"+jp.getSignature());
        //执行目标方法proceed
        Object proceed = jp.proceed();
        System.out.println("-------环绕后------");
        System.out.println(proceed);
    }
```

```
    @AfterThrowing("execution(* com.xinzhi.service.impl.*.*(..))")
    public void afterThrow() {
        System.out.println("--------------有异常发生-----------------" + new Date());
    }
}
```

第二步：在Spring配置文件中，注册bean，并增加支持注解的配置

```
<!--第三种方式:注解实现-->
<aop:aspectj-autoproxy/>
```

aop:aspectj-autoproxy：说明

1. 通过aop命名空间的<aop:aspectj-autoproxy />声明自动为spring容器中那些配置@aspectJ切面的bean创建代理，织入切面。当然，spring 在内部依旧采用AnnotationAwareAspectJAutoProxyCreator进行自动代理的创建工作，但具体实现的细节已经被<aop:aspectj-autoproxy />隐藏起来了
2. <aop:aspectj-autoproxy />有一个proxy-target-class属性，默认为false，表示使用jdk动态代理织入增强，当配为<aop:aspectj-autoproxy poxy-target-class="true"/>时，表示使用CGLib动态代理技术织入增强。不过即使proxy-target-class设置为false，如果目标类没有声明接口，则spring将自动使用CGLib动态代理。

正常测试：

```
-------环绕前-------
签名（拿到方法名）:void com.xinzhi.service.IUserService.register(User)
---------方法执行前before()---------
楠哥注册成功!
我是将数据保存至oracle
-------环绕后------
null
---------方法执行后after()---------
---------方法返回后afterReturning()---------
```

异常测试：

在service中加入 int i=1/0;

结果

```
-------环绕前-------
签名（拿到方法名）:void com.xinzhi.service.IUserService.register(User)
---------方法执行前before()---------
---------方法执行后after()---------
--------------有异常发生-----------------Mon May 18 14:52:57 CST 2020

java.lang.ArithmeticException: / by zero
```

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。

五、整合MyBatis

> **MyBatis-Spring 可以将 MyBatis 代码无缝地整合到 Spring 中。**

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

如果使用 Maven 作为构建工具，仅需要在 pom.xml 中加入以下代码即可：

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

mybatis和spring的整合其实就是让spring管理我们的**SqlSessionFactory** ，但是**SqlSessionFactory** 需要一个数据源，给他一个就行：

```
 <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
 </bean>
```

在原生的 MyBatis 用法中，是通过 SqlSessionFactoryBuilder 来创建 SqlSessionFactory 的。而在 MyBatis-Spring 中，则使用 SqlSessionFactoryBean 来创建。

在 MyBatis 中，你可以使用 SqlSessionFactory 来创建 SqlSession。一旦你获得一个 session 之后，你可以使用它来执行映射了的语句，提交或回滚连接，最后，当不再需要它的时候，你可以关闭 session。

SqlSessionFactory有一个唯一的必要属性：用于 JDBC 的 DataSource。这可以是任意的 DataSource 对象，它的配置方法和其它 Spring 数据库连接是一样的。

SqlSessionTemplate 是 MyBatis-Spring 的核心。作为 SqlSession 的一个实现，这意味着可以使用它无缝代替你代码中已经在使用的 SqlSession。

模板可以参与到 Spring 的事务管理中，并且由于其是线程安全的，可以供多个映射器类使用，你应该总是用 SqlSessionTemplate 来替换 MyBatis 默认的 DefaultSqlSession 实现。在同一应用程序中的不同类之间混杂使用可能会引起数据一致性的问题。

整合MyBatis

步骤

1、导入相关jar包

```
<dependencies>
    <!-- 单元测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.7</version>
        <scope>test</scope>
    </dependency>
    <!-- spring 相关 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.1.10.RELEASE</version>
    </dependency>
    <!-- mybatis 相关 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.2</version>
    </dependency>
    <!-- 数据库连接驱动 相关 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>

    <!-- 提供了对JDBC操作的完整封装 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.1.10.RELEASE</version>
    </dependency>
    <!-- 织入 相关 -->
    <dependency>
        <groupId>org.aspectj</groupId>
```

```
        <artifactId>aspectjweaver</artifactId>
        <version>1.9.4</version>
    </dependency>
    <!-- spring, mybatis整合包 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>2.0.2</version>
    </dependency>

</dependencies>

<!-- 资源文件过滤 -->
<build>
    <resources>
        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>false</filtering>
        </resource>
        <resource>
            <directory>src/main/resources</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>false</filtering>
        </resource>
    </resources>
</build>
```

编写entity实体类

```
/**
 * author IT楠老师
 * date 2020/5/12
 */
public class User implements Serializable {

    private int id;
    private String username;
    private String password;
    ...
}
```

实现mybatis的配置文件

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

</configuration>
```

UserDao接口编写

```
/**
 * author IT楠老师
 * date 2020/5/12
 */
public interface UserMapper {
    /**
     * 获取所有的用户
     * @return
     */
    List<User> getAllUsers();
}
```

接口对应的Mapper映射文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xinzhi.dao.UserMapper">
    <select id="getAllUsers" resultType="com.xinzhi.entity.User">
      select id,username,password from user
    </select>
</mapper>
```

1、普通整合

1、独立数据库配置文件（db.properties）

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/ssm?
useSSL=true&useUnicode=true&characterEncoding=utf8
jdbc.username=root
jdbc.password=root
```

2、spring配置文件

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

    <!-- 加载外部的数据库信息 -->
    <context:property-placeholder location="db.properties"/>

    <!-- Mapper 扫描器 -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <!-- 扫描 cn.wmyskxz.mapper 包下的组件 -->
        <property name="basePackage" value="com.xinzhi.dao"/>
    </bean>

    <!--配置数据源：数据源有非常多，可以使用第三方的，也可使使用Spring的-->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <!--配置SqlSessionFactory-->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <!--关联Mybatis-->
        <property name="configLocation" value="classpath:mybatis-config.xml"/>
        <property name="mapperLocations" value="classpath:com/xinzhi/dao/*.xml"/>
    </bean>

    <!--注册sqlSessionTemplate ， 关联sqlSessionFactory-->
    <bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
        <!--利用构造器注入-->
        <constructor-arg index="0" ref="sqlSessionFactory"/>
    </bean>

    <bean id="userMapper" class="com.xinzhi.dao.impl.UserMapperImpl">
        <property name="sqlSession" ref="sqlSession"/>
    </bean>
</beans>
```

```
/**
 * author IT楠老师
 * date 2020/5/12
 */
```

```java
public interface UserMapper {

    /**
     * 删除一个用户
     * @param id
     * @return
     */
    int deleteUser(int id);

    /**
     * 修改用户
     * @param user
     * @return
     */
    int updateUser(User user);

    /**
     * 新增user
     * @param user
     * @return
     */
    int addUser(User user);

    /**
     * 获取所有的用户
     * @return
     */
    List<User> getAllUsers();

    User findUserById(int id);

}
```

```java
@Test
public void selectUser() throws IOException {

   String resource = "mybatis-config.xml";
   InputStream inputStream = Resources.getResourceAsStream(resource);
   SqlSessionFactory sqlSessionFactory =
newSqlSessionFactoryBuilder().build(inputStream);
   SqlSession sqlSession = sqlSessionFactory.openSession();

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

   List<User> userList = mapper.selectUser();
   for (User user: userList){
       System.out.println(user);
  }

   sqlSession.close();
}
```

3、测试

```
@Test
public void testFindAll(){
    ApplicationContext context = new
        ClassPathXmlApplicationContext("applicationContext.xml");
    UserMapper mapper = (UserMapper) context.getBean("userMapper");
    List<User> users = mapper.getAllUsers();
    for (User user : users) {
        System.out.println(user);
    }
}
```

结果成功输出！现在我们的Mybatis配置文件的状态！发现都可以被Spring整合！

```
User{id=1, username='楠哥', password='123456'}
User{id=3, username='磊哥', password='987654'}
User{id=5, username='微微姐', password='12345678'}
User{id=6, username='微微姐', password='12345678'}
```

2、使用SqlSessionDaoSupport整合

mybatis-spring1.2.3版以上的才有这个 .

官方文档截图 :

dao继承Support类 , 直接利用 getSqlSession() 获得 , 然后直接注入SqlSessionFactory . 比起方式1 , 不需要管理 SqlSessionTemplate , 而且对事务的支持更加友好 . 可跟踪源码查看

测试:

1、将我们上面写的UserDaoImpl修改一下

```
/**
 * author IT楠老师
 * date 2020/5/13
 */
public class UserMapperImpl extends SqlSessionDaoSupport implements UserMapper {
    public List<User> getAllUsers() {
        return getSqlSession().getMapper(UserMapper.class).getAllUsers();
    }
}
```

2、修改bean的配置

```
<bean id="userMapper" class="com.xinzhi.dao.impl.UserMapperImpl">
    <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>
```

3、测试

```
@Test
public void testFindAll(){
    ApplicationContext context = new
        ClassPathXmlApplicationContext("applicationContext.xml");
    UserMapper mapper = (UserMapper) context.getBean("userMapper");
    List<User> users = mapper.getAllUsers();
    for (User user : users) {
        System.out.println(user);
    }
}
```

4、结果

```
User{id=1, username='楠哥', password='123456'}
User{id=3, username='磊哥', password='987654'}
User{id=5, username='微微姐', password='12345678'}
User{id=6, username='微微姐', password='12345678'}
```

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！**

3、动态Mapper（常用）

上面的实例程序并没有使用 Mapper 动态代理和注解来完成，下面我们就来试试如何用动态代理和注解：

配置文件：

```
<!-- Mapper 扫描器 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!-- 扫描 cn.wmyskxz.mapper 包下的组件 -->
    <property name="basePackage" value="com.xinzhi.dao"/>
</bean>
<!--配置SqlSessionFactory-->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <!--关联Mybatis-->
    <property name="configLocation" value="classpath:mybatis-config.xml"/>
    <property name="mapperLocations" value="classpath:com/xinzhi/dao/*.xml"/>
</bean>
```

在【mapper】下新建一个【UserQueryMapper】代理接口，并使用注解：

```
/**
 * author IT楠老师
 * date 2020/5/12
 */
@Mapper
public interface UserMapper {

    /**
     * 获取所有的用户
     * @return
     */
    List<User> getAllUsers();
    ...

}
```

不需要实现，只要一个接口一个xml即可

测试

```
@Test
public void testFindAll(){
    UserMapper mapper = (UserMapper) context.getBean("userMapper");
    List<User> users = mapper.getAllUsers();
    for (User user : users) {
        System.out.println(user);
    }
}
```

结果：

```
User{id=1, username='楠哥', password='123456'}
User{id=3, username='磊哥', password='987654'}
User{id=5, username='微微姐', password='12345678'}
User{id=6, username='微微姐', password='12345678'}

可以看到，查询结果和之前非 Mapper 代理的查询结果一样
```

当然有的人连配置文件也不想要

```
/**
 * author IT楠老师
 * date 2020/5/12
 */
@Mapper
public interface AdminMapper {

    /**
```

```
     * 保存管理员
     * @param admin
     * @return
     */
    @Insert("insert into admin (username,password) values (#{username},#{password})")
    int saveAdmin(Admin admin);

    /**
     * 跟新管理员
     * @param admin
     * @return
     */
    @Update("update admin set username=#{username} , password=#{password} where id = #
{id}")
    int updateAdmin(Admin admin);

    /**
     * 删除管理员
     * @param id
     * @return
     */
    @Delete("delete from admin where id=#{id}")
    int deleteAdmin(int id);

    /**
     * 根据id查找管理员
     * @param id
     * @return
     */
    @Select("select id,username,password from admin where id=#{id}")
    Admin findAdminById(@Param("id") int id);

    /**
     * 查询所有的管理员
     * @return
     */
    @Select("select id,username,password from admin")
    List<Admin> findAllAdmins();

}
```

这样也行。

> **企业的做法**

一般使用动态mapper，使用动态代理完成工作，一般会使用xml和mappeer配合使用。这才是最佳实践。

六、声明式事务

1、事务特性

1. 事务特性分为四个：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持续性（Durability）简称ACID。
    1. **原子性（Atomicity）**:事务是数据库逻辑工作单元，事务中包含的操作要么都执行成功，要么都执行失败。
    2. **一致性（Consistency）**：事务执行的结果必须是使数据库数据从一个一致性状态变到另外一种一致性状态。当事务执行成功后就说数据库处于一致性状态。如果在执行过程中发生错误，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这是数据库就处于不一致状态。
    3. **隔离性（Isolation）**：一个事务的执行过程中不能影响到其他事务的执行，即一个事务内部的操作及使用的数据对其他事务是隔离的，并发执行各个事务之间无不干扰。
    4. **持续性（Durability）**：即一个事务执一旦提交，它对数据库数据的改变是永久性的。之后的其它操作不应该对其执行结果有任何影响。

2、测试事务存在否

将上面的代码拷贝到一个新项目中

在之前的案例中，我们给userDao接口新增两个方法，删除和增加用户；

```
/**
 * author IT楠老师
 * date 2020/5/12
 */
@Mapper
public interface UserMapper {
     /**
      * 修改用户
      * @param user
      * @return
      */
     int updateUser(User user);

     /**
      * 新增user
      * @param user
      * @return
      */
     int addUser(User user);

     /**
      * 获取所有的用户
      * @return
      */
     List<User> getAllUsers();
     ...
}
```

mapper文件，我们故意把 deletes 写错，测试！

```
<insert id="addUser" parameterType="com.xinzhi.entity.User">
    insert into user (id,username,password) values (#{id},#{username},#{password})
</insert>

<update id="updateUser" parameterType="com.xinzhi.entity.User">
    update user set username=#{username},password=#{password} where id = #{id}
</update>

<select id="getAllUsers" resultType="com.xinzhi.entity.User">
    select id,username,password from user
</select>
```

编写接口的实现类，在实现类中，我们去操作一波

```
/**
 * author IT楠老师
 * date 2020/5/12
 */
@Mapper
public interface UserMapper {
    /**
     * 修改用户
     * @param user
     * @return
     */
    int updateUser(User user);

    /**
     * 新增user
     * @param user
     * @return
     */
    int addUser(User user);

    /**
     * 获取所有的用户
     * @return
     */
    List<User> getAllUsers();

}
```

```
/**
 * author IT楠老师
 * date 2020/5/18
 */
public interface IUserService {

    /**
```

```
     * 测试事务使用，工作中我们的事务一般是定义在service层中
     */
    void transaction();
}

/**
 * author IT楠老师
 * date 2020/5/18
 */
@Service
public class UserServiceImpl implements IUserService{

    @Autowired
    private UserMapper userMapper;


    public void transaction() {
        userMapper.addUser(new User(100,"IT楠老师","123"));
        int i = 1/0;
        userMapper.updateUser(new User(100,"IT楠老师","234"));
    }
}
```

测试

```
@Test
public void testNoTransaction(){
    IUserService userService = context.getBean(IUserService.class);
    userService.transaction();
}
```

结果： 插入成功了。

```
1       楠哥     123456
3       磊哥     987654
5       微微姐   12345678
6       微微姐   12345678
100     IT楠老师 123
```

显然不是我们想要的，接下来的时间我们需要看看事务在spring中怎么处理。

3、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

```
<bean
id="transactionManager"class="org.springframework.jdbc.datasource.DataSourceTransaction
Manager">
       <property name="dataSource" ref="dataSource" />
</bean>
```

这次我们先看看注解形式的事务

```
<!-- 开启事务注解，并配置一个事务管理器 -->
<tx:annotation-driven transaction-manager="transactionManager" />
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>
```

加注解

```
/**
 * author IT楠老师
 * date 2020/5/18
 */
@Service
@Transactional
public class UserServiceImpl implements IUserService{

    @Autowired
    private UserMapper userMapper;

    public void transaction() {
        userMapper.addUser(new User(100,"IT楠老师","123"));
```

```
        int i = 1/0;
        userMapper.updateUser(new User(100,"IT楠老师","234"));
    }
}
```

测试：

```
@Test
public void testNoTransaction(){
    IUserService userService = context.getBean(IUserService.class);
    userService.transaction();
}
```

```
java.lang.ArithmeticException: / by zero
异常抛出异常
```

```
1       楠哥     123456
3       磊哥     987654
5       微微姐   12345678
6       微微姐   12345678
```

数据库并没有插入，完美，一个注解搞定。

并不意外，除了注解的形式，我们还可以使用配置文件进行声名式事务的配置

声明式事务，其实是AOP的典型应用。都是在业务方法中将事务的开启关闭织入。

```
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>


<!--配置事务通知-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <!--配置哪些方法使用什么样的事务,配置事务的传播特性-->
        <tx:method name="add*" propagation="REQUIRED"/>
        <tx:method name="delete*" propagation="REQUIRED"/>
        <tx:method name="update*" propagation="REQUIRED"/>
        <tx:method name="search*" propagation="REQUIRED"/>
        <tx:method name="get*" read-only="true"/>
        <tx:method name="find*" read-only="true"/>
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

```
<!--配置aop织入事务-->
<aop:config>
    <aop:pointcut id="txPointcut" expression="execution(* com.xinzhi.service.impl.*.*
(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
</aop:config>
```

删掉刚才插入的数据，再次测试！

```
@Test
public void test2(){
   ApplicationContext context = newClassPathXmlApplicationContext("beans.xml");
   UserMapper mapper = (UserMapper) context.getBean("userDao");
   List<User> user = mapper.selectUser();
   System.out.println(user);
}
```

测试

```
@Test
public void testNoTransaction(){
    IUserService userService = context.getBean(IUserService.class);
    userService.transaction();
}
```

```
java.lang.ArithmeticException: / by zero
异常抛出异常
```

```
1       楠哥     123456
3       磊哥     987654
5       微微姐   12345678
6       微微姐   12345678
```

数据库并没有插入，完美，一个注解搞定。

4、spring事务传播特性

事务传播行为就是多个事务方法相互调用时，事务如何在这些方法间传播。

spring支持7种事务传播行为：

- propagation_requierd：如果当前没有事务，就新建一个事务，如果已存在一个事务中，加入到这个事务中，这是最常见的选择。
- propagation_supports：支持当前事务，如果没有当前事务，就以非事务方法执行。
- propagation_mandatory：使用当前事务，如果没有当前事务，就抛出异常。
- propagation_required_new：新建事务，如果当前存在事务，把当前事务挂起。
- propagation_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。

- propagation_never：以非事务方式执行操作，如果当前事务存在则抛出异常。
- propagation_nested：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与propagation_required类似的操作

Spring 默认的事务传播行为是 PROPAGATION_REQUIRED，它适合于绝大多数的情况。