

Spring4 课程教案

版本号：JAVAEE1.0

密 级：受控文档

课 程 标 准 化

2018 年 07 月 01 日

1.文档属性

文档属性	内容
项目/任务名称:	
项目/任务编号:	
文档名称:	
文档编号:	
文档状态:	
作 者:	实训部开发组
文档评审通过日期:	
评审负责人签字:	
参考模板:	

2.文档变更过程

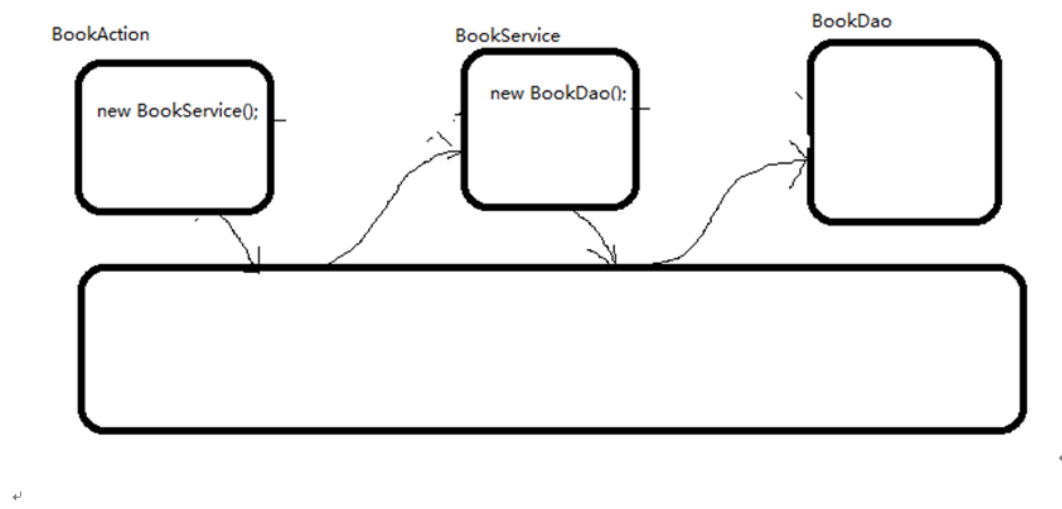
[illegible]

目录

前言	5
Spring 的核心技术	5
1: 下载 spring.jar	5
2: 认识 spring	6
3: 第一个 spring 入门项目	8
4: Spring 的核心类和核心配置文件	8
Spring IOC	10
Spring 的注入	10
1、属性注入--setter 方法注入	10
2、通过构造器注入	11
3、注入其他类	12
4、<bean>元素中的属性	13
**Bean 的实例化	13
1、构造器实例化	13
2、静态工程方式实例化	13
3、实例工厂化方式实例化	14
Bean 的作用域	14
Bean 的生命周期	15
Bean 的装配方式	17
1、 基于 XML 的装配	17
2、 基于注解 (annotation)	17
3、 自动装配	18
高级 Bean	18
1、引用外部资源文件	18
Spring AOP	19
什么是 AOP	19
JDK 动态代理	22
**CGLIB 代理	24
基于代理类的 AOP 实现—java 代码实现	24
**基于代理类的 AOP 实现—XML 配置方式	26
AspectJ 开发	26
1、基于 AspectJ 的切点表达式	26
2、基于 XML 的声明式 AspectJ	28
3、基于注解的声明式 AspectJ	31
Spring+mybatis 整合	34
1、核心配置文件	34
2、传统 DAO 方式的开发整合	36
3、Mapper 接口的方式整合	38
3.1 基于 MapperFactoryBean 的整合	38
3.2 基于 MapperScannerConfigurer 的整合	40
Spring 事务	41
1. 事务是什么?	41

2.	事务管理核心接口	42
3.	声明式事务管理	44
3.1	基于 XML 方式的事务管理	45
3.2	基于注解方式的事务管理	48

前言



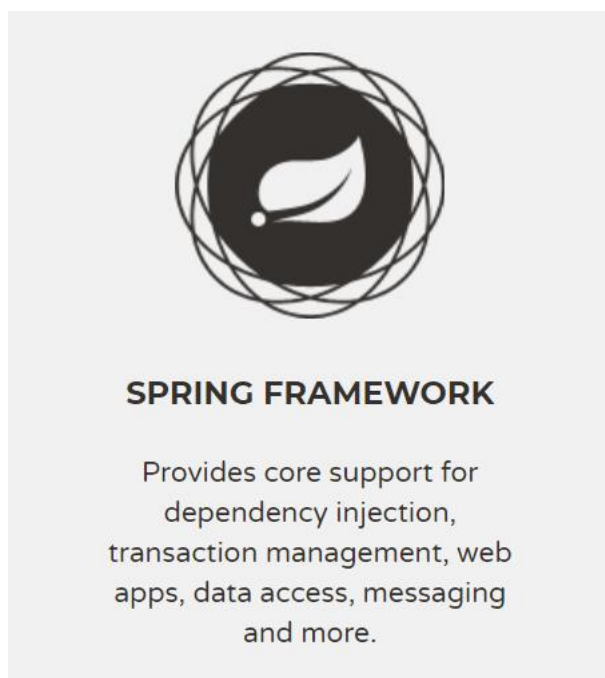
实现完全的注入，可以通过配置文件实现。

传统开发使用一个对象，采用 `new()` 的方式来直接实例化，耦合度高。需要一个容器来提供实例化好的对象，只关注业务。

Spring 的核心技术

1: 下载 spring.jar

<https://spring.io>



DI: 控制反转

TXM 事务管理

Data access : 数据访问

Features

- **Core technologies:** dependency injection, events, resources, i18n, validation, data binding, type conversion, SpEL, AOP.
- **Testing:** mock objects, TestContext framework, Spring MVC Test, `WebTestClient`.
- **Data Access:** transactions, DAO support, JDBC, ORM, Marshalling XML.
- **Spring MVC** and **Spring WebFlux** web frameworks.
- **Integration:** remoting, JMS, JCA, JMX, email, tasks, scheduling, cache.
- **Languages:** Kotlin, Groovy, dynamic languages.

核心技术：依赖注入、事件、资源、I18N、验证、数据绑定、类型转换、SPEL、AOP。

测试：模拟对象，TestFieldFrrices，Spring MVC 测试，WebTestClient。

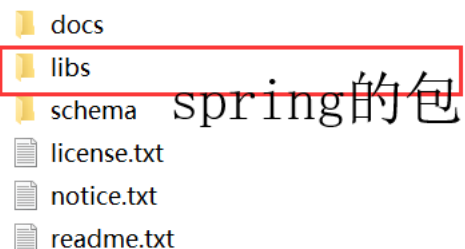
数据访问：事务、DAO 支持、JDBC、ORM、编组 XML。

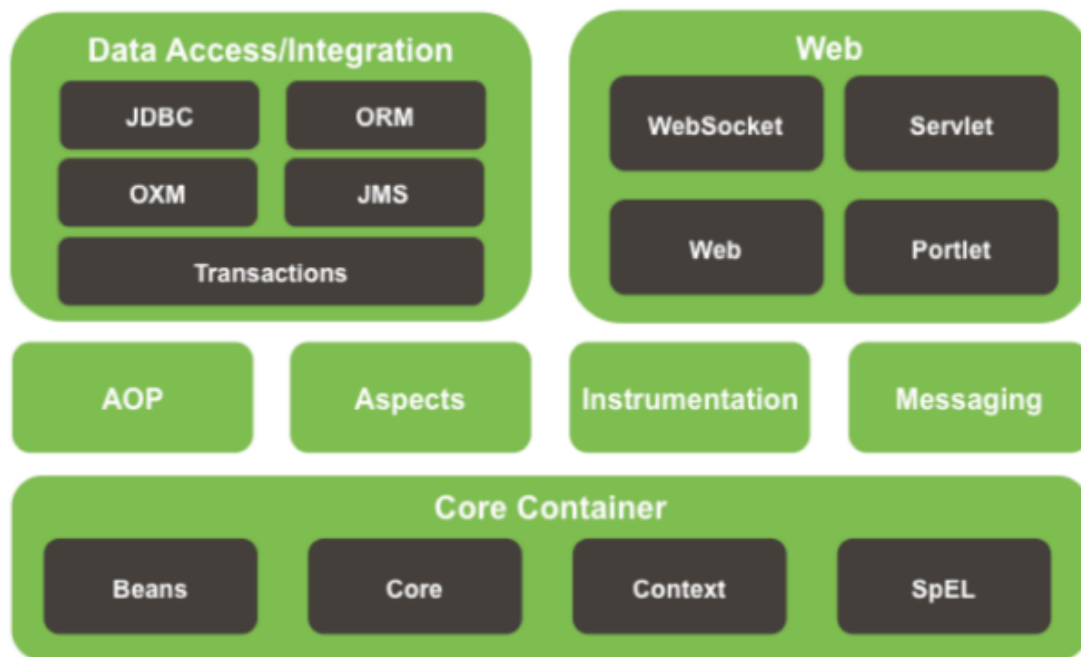
Spring MVC 和 Spring WebFrاند Web 框架。

集成：远程处理、JMS、JCA、JMX、电子邮件、任务、调度、缓存。

语言：Kotlin，Groovy，动态语言。

2: 认识 spring





如上图所示，Spring 由 20 多个模块组成，它们可以分为核心容器（Core Container）、数据访问/集成（Data Access/Integration）、Web、面向切面编程（AOP, Aspect Oriented Programming）、设备（Instrumentation）、消息发送（Messaging）和测试（Test）。

Spring 是一个轻量级的控制反转（IoC）和面向切面编程（AOP）的容器框架。

- 轻量——从大小与开销两方面而言 Spring 都是轻量的。完整的 Spring 框架可以在一个大小只有 1MB 多的 JAR 文件里发布。并且 Spring 所需的处理开销也是微不足道的。
- 非侵入 ——在应用中，一般不需要引用 springjar 包里的类。
- 控制反转——Spring 通过一种称作控制反转（IoC）的技术促进了松耦合。当应用了 IoC，某一接口的具体实现类的选择控制权从调用类中移除，转交给第三方裁决。
将控制权交给接口--面向接口的编程
核心技术就是 DI 容器（依赖注入）
- 面向切面(AOP)——Spring 提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如审计（auditing）和事务（）管理）进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责其它的系统级关注点，例如日志或事务支持。

为什么使用 spring?

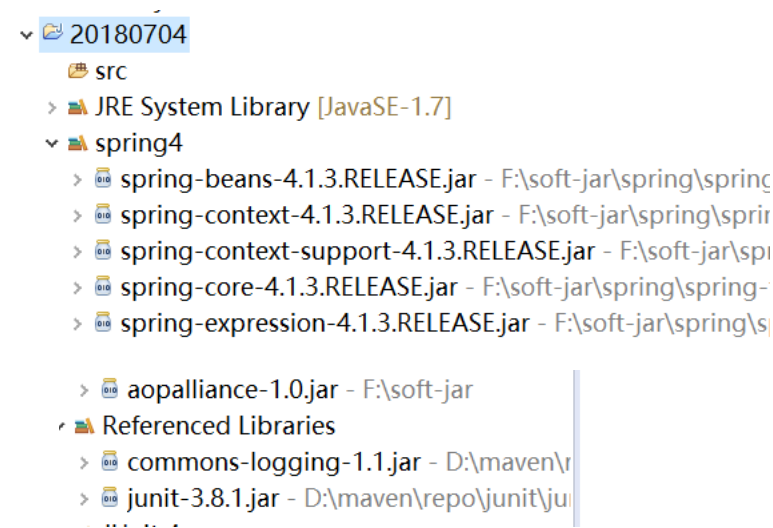
至少在我看来，在项目中引入 spring 立即可以带来下面的好处

- 降低组件之间的耦合度,实现软件各层之间的解耦。

- 可以使用容器提供的众多服务，如：事务管理服务、消息服务等等。当我们使用容器管理事务时，开发人员就不再需要手工控制事务.也不需处理复杂的事务传播。
- 容器提供单例模式支持，开发人员不再需要自己编写实现代码。
- 容器提供了 AOP 技术，利用它很容易实现如权限拦截、运行期监控等功能。
- 容器提供的众多辅作类，使用这些类能够加快应用的开发，如：JdbcTemplate、 HibernateTemplate。
- Spring 对于主流的应用框架提供了集成支持，如：集成 Hibernate、JPA、Struts 等，这样更便于应用的开发。

3：第一个 spring 入门项目

由于 spring 管理的是 service 层，以及它是个容器，故新建 java 项目即可。



4：Spring 的核心类和核心配置文件

类或者配置文件		
*.xml applicationContext.xml(默认)	用于配置所有类， 这些类就叫做 springBean	
BeanFactory(接 口)— 基 本 的 容 器 类	容器的工厂类， 用户创建或者获取 springBean。即获取配置文件中的 类	
ApplicationContext(接口)	它是 BeanFactory 的子类， 叫做应用上下文对象， 功能比 BeanFactory 强大	

4.1 申明一个类

```
package com.hpe;
```



```

public class Person {
    private String name;
    public void eat(){
        System.err.println("eating.....");
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

```

4.2 申明 spring 的配置文件

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

</beans>

```

4.3 将 Persion 配置到配置文件中

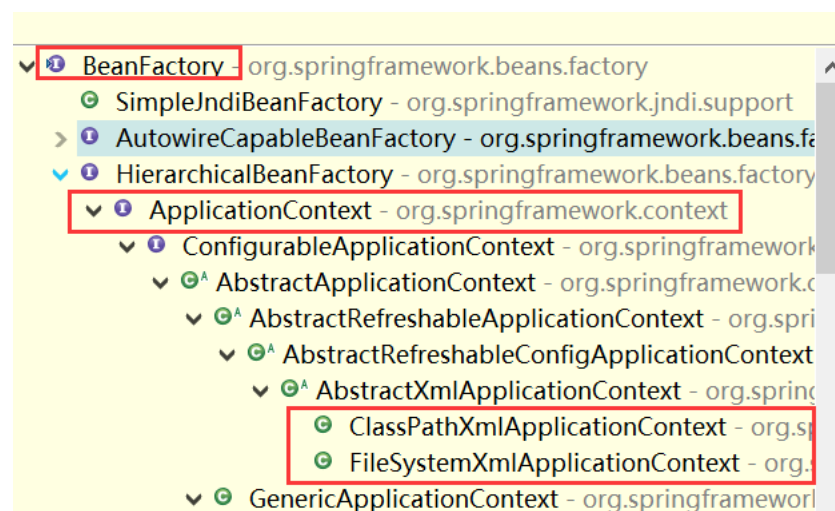
```

<!-- 配置第一个springBean
    Person person = new Person();
    contain.map.put("person",person);
    id必须是唯一的
-->
<bean id="person" class="com.hpe.Person"/>

```

4.4 读取这个文件

Ctrl+T



ClassPathXmlApplicationContext: 用于在 classpath 下加载 xml 配置文件 例 classpath:a.xml

FileSystemXmlApplicationContext: 用于加载文件系统中的 xml 配置文件 例 d:/a.xml

```
@Test
    public void test1(){
        //1.导入jar包
        //2.配置xml文件（spring的配置文件），填写信息-》让spring知道创建什么类的实例
        //3.加载配置文件，初始化并创建实例。
        ApplicationContext ctx = new
        ClassPathXmlApplicationContext("applicationContext.xml");
        //4.取得spring给我们创建的实例
        Person person = ctx.getBean("person",Person.class);
        System.err.println(person);
        ///5.调用方法
        person.eat();
    }
```

BeanFactory 与 ApplicationContext 的区别

BeanFactory:懒加载 lazy=true

ApplicationContext: 非懒加载 lazy=false 常用方式

Spring IOC

控制反转（Inversion of Control，英文缩写为 IoC）把创建对象的权利交给框架,是框架的重要特征，并非面向对象编程的专用术语。它包括依赖注入（Dependency Injection，简称 DI）和依赖查找（Dependency Lookup）。

IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。

Spring 的注入--就是在实例化这个类时，由外部容器来设置这个对象的值

1、属性注入--setter 方法注入

```
public class Person {
    private String name;
    private Integer age;
    private String[] str;
    private List<String> coll;
    private Set<String> sets;
    private Map<String,Object> mm;
```

前提： 要有属性对应的 set 方法

```
<bean id="person" class="com.hpe.Person">
    <property name="name" value="jack"></property>
    <property name="age" value="11"></property>
    <property name="coll">
        <list>
            <value>aa</value>
            <value>bb</value>
        </list>
    </property>
    <property name="mm">
        <map>
            <entry key="name" value="zhangsan"></entry>
            <entry key="tel"><value>10086</value></entry>
        </map>
    </property>
    <property name="sets">
        <set>
            <value>AAAA</value>
            <value>BBBB</value>
        </set>
    </property>
    <property name="strs">
        <array>
            <value>CCCC</value>
            <value>DDDD</value>
        </array>
    </property>
</bean>
```

2、通过构造器注入

```
public class User {
    private String name;
    private Integer age;
    public User(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "User [name=" + name + ", age=" + age + "];"
    }
}
```

```
}  
}
```

配置如下：

指定构造参数的顺序

<!-- 以下是通过构造器注入 -->

```
<bean id="user" class="cn.domain.User">  
    <constructor-arg value="Jack"/>  
    <constructor-arg value="44"/>  
</bean>
```


以下是指定下标的方式：

```
<bean id="user" class="cn.domain.User">  
    <constructor-arg value="44" index="1"/>  
    <constructor-arg value="Jack" index="0"/>  
</bean>
```

指定在构造中的参数名称

```
<bean id="user" class="cn.domain.User">  
    <constructor-arg value="44" name="age"/>  
    <constructor-arg value="Jack" name="nm"/>  
</bean>
```

3、注入其他类

▼  User

- name : String
- age : Integer
- car : Car
- getCar() : Car
- setCar(Car) : void
- User(String, Integer)
- toString() : String

<!-- 通过构造器注入 -->

```
<bean id="user" class="com.hpe.User">  
    <constructor-arg value="Jack"></constructor-arg>  
    <constructor-arg value="44"></constructor-arg>  
    <!-- 引用的方式 设置引用id -->  
    <property name="car" ref="car"></property>  
</bean>
```

<!-- 声明car的实例 -->

```
<bean id="car" class="com.hpe.Car">  
    <property name="name" value="audi"></property>  
</bean>
```

内部声明

```
<property name="car">
    <bean class="cn.hpe.Car">
        <property name="name" value="Buick"></property>
    </bean>
</property>
```

这种声明方式，别的 Bean 不能引用了。

4、<bean>元素中的属性

- id: Bean 的唯一标识符，Spring 对 Bean 的配置、管理都通过该属性来完成；
- name: Spring 同样可以通过 name 对 Bean 进行配置和管理，name 属性可以为 Bean 定义多个名称，每个名称以逗号隔开；
- class: 该属性指定了 Bean 的具体实现类，必须是一个完整的类名，使用类的全限定名；
- scope: 设定 Bean 实例的作用域，其属性有 singleton（单例）、prototype（原型）、request、session、和 global Session，默认值为 singleton，该属性会在下一篇博客中详细讲解；
- constructor-arg: <bean>元素的子元素，可以使用此元素传入构造参数进行实例化（上一篇博客的最后补充就是使用此属性进行实例化的），该元素的 index 属性指定构造参数的序号（从 0 开始）；
- property: <bean>元素的子元素，通过调用 Bean 实例中的 setter 方法完成属性赋值，从而完成依赖注入；
- ref: property、constructor-arg 等元素的子元素，该元素中的 bean 属性用于指定对 Bean 工厂中某个 Bean 实例的引用；
- value: property、constructor-arg 等元素的子元素，用来直接指定一个常量值；
- list: 用于封装 List 或数组类型的依赖注入；
- set: 用于封装 Set 或数组类型的依赖注入；
- map: 用于封装 Map 或数组类型的依赖注入；
- entry: map 元素的子元素，用于设定一个键值对，其 key 属性指定字符串类型的键值，ref 或 value 子元素指定其值。

**Bean 的实例化

1、构造器实例化

Spring 容器通过 Bean 对应的默认的构造函数来实例化 Bean。

2、静态工程方式实例化

通过创建静态工厂的方法来创建 Bean 的实例

```

public class Bean2Factory {
    //使用自己的工厂创建Bean2的实例
    public static Bean2 createBean(){
        return new Bean2();
    }
}

<!-- factory-method指定工厂方法 -->
<bean id="bean2" class="com.bean.Bean2Factory" factory-method="createBean">
</bean>

```

3、实例工厂化方式实例化

不再使用静态方法创建 Bean 实例，而是采用直接创建 Bean 实例的方式。

```

public class Bean3Factory {
    public Bean3Factory() {
        System.err.println("Bean3Factory 工厂实例化");
    }
    //Bean3实例化方法
    public Bean3 createBean(){
        return new Bean3();
    }
}

<!-- 配置工厂 -->
<bean id="bean3Factory" class="com.bean.Bean3Factory"></bean>
<!-- factory-bean 属性指向配置的实例工厂
    factory-method 属性确定使用工厂的哪个方法-->
<bean id="bean3" factory-bean="bean3Factory" factory-method="createBean">
</bean>

```

Bean 的作用域

1. singleton: 单例模式，Spring IoC 容器中只会存在一个共享的 Bean 实例，无论有多少个 Bean 引用它，始终指向同一对象。Singleton 作用域是 Spring 中的缺省作用域，也可以显示的将 Bean 定义为 singleton 模式

```

<bean id="dog" class="com.bean.Dog" scope="singleton"></bean>

public void test4(){
    //3.加载配置文件，并创建实例
}

```

```

ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
//4.取得spring给我们创建的实例
Dog dog1 = (Dog) ctx.getBean("dog");
System.err.println(dog1);
Dog dog2 = (Dog) ctx.getBean("dog");
System.err.println(dog2);
}

```

输出的内存地址一致。

2. prototype:原型模式，每次通过 Spring 容器获取 prototype 定义的 bean 时，容器都将创建一个新的 Bean 实例，每个 Bean 实例都有自己的属性和状态，而 singleton 全局只有一个对象。

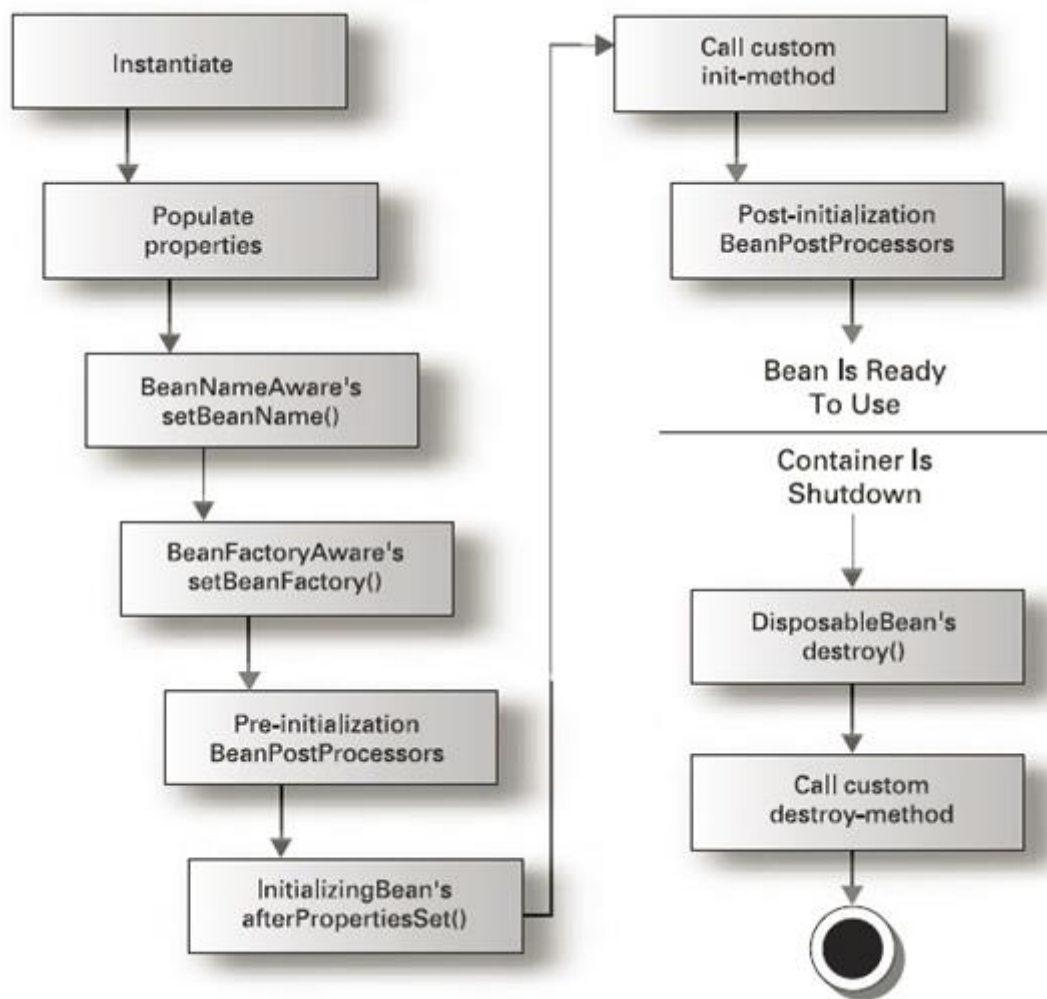
```
<bean id="dog" class="com.bean.Dog" scope="prototype "></bean>
```

输出的内存地址不一致

3. request: 在一次 Http 请求中，容器会返回该 Bean 的同一实例。而对不同的 Http 请求则会产生新的 Bean，而且该 bean 仅在当前 Http Request 内有效。
 - 针对每一次 Http 请求，Spring 容器根据该 bean 的定义创建一个全新的实例，且该实例仅在当前 Http 请求内有效，而其它请求无法看到当前请求中状态的变化，当当前 Http 请求结束，该 bean 实例也将会被销毁。
4. session: 在一次 Http Session 中，容器会返回该 Bean 的同一实例。而对不同的 Session 请求则会创建新的实例，该 bean 实例仅在当前 Session 内有效。
 - 同 Http 请求相同，每一次 session 请求创建新的实例，而不同的实例之间不共享属性，且实例仅在自己的 session 请求内有效，请求结束，则实例将被销毁。

Bean 的生命周期

Spring 容器可以管理 singleton 作用域下 Bean 的生命周期，在此作用域下，Spring 能够精确地知道 Bean 何时被创建，何时初始化完成，以及何时被销毁。而对于 prototype 作用域的 Bean，Spring 只负责创建，当容器创建了 Bean 的实例后，Bean 的实例就交给了客户端的代码管理，Spring 容器将不再跟踪其生命周期，并且不会管理那些被配置成 prototype 作用域的 Bean 的生命周期。



```

public class Dog {
    public Dog() {
        System.err.println("1:初始化了。。。");
    }
    public void setName(String name){
        System.err.println("2:设置属性....");
    }
    public void init(){
        System.err.println("3:初始化了.....");
    }
    public void des(){
        System.err.println("4:销毁方法。。。。");
    }
}

<bean id="dog" class="com.bean.Dog" init-method="init" destroy-method="des">
    <property name="name" value="xiaohuang"></property>
</bean>

```



```
public void test5(){
    ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
    ctx.destroy();
}
```

Bean 的装配方式

1、基于 XML 的装配

两种装配方式：设值注入和构造器注入。以上案例都是这种方式的，略
设值注入的两个要求：

- Bean 类必须提供一个默认的空参构造方法
- Bean 类必须为需要注入的属性提供对应的 setter 方法。
见项目案例 面向结构编程 controller—service—dao

2、基于注解（annotation）

常用注解：

- @Component 是所有受 Spring 管理组件的通用形式，@Component 注解可以放在类的头上，@Component 不推荐使用。
- @Controller 控制器，表示 web 层组件
- @Service 业务类，表示业务层组件
- @Repository 表示持久层的组件
- @Autowired 默认按类型装配
- @Qualifier 与@Autowired 配合使用，存在多个实例配合使用，按照名字匹配
- @Resource 默认按名称装配，当找不到与名称匹配的 bean 才会按类型装配

使用见案例...

配置文件批量扫描

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    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
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
    <context:component-scan base-package="com.hpe.*">
```

```
</context:component-scan>
</beans>
```

注意，项目需要导入 aop 包！

接口多个实现类的按照类型装配报错问题讲解。

3、自动装配

属性值	说明	
default（默认值）	由<beans>的 default-autowire 属性值确定。	default-autowire="default"
byName	根据属性名称自动装配	
byType	根据属性的数据类型自动装配	
constructor	根据构造函数参数的数据类型自动装配	
no	不适用自动装配，必须通过 ref 元素定义	

案例：

高级 Bean

1、引用外部资源文件

核心类：PropertyPlaceholderConfigurer

```
public class Dog {
    private String name;
    private Integer age;
```

新建 dog.properties 资源配置文件

```
name=jack
age=5
```

```
<!-- 加载外部资源文件 -->
<bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>classpath:com/test/dog.properties</value>
        </list>
    </property>
```

```
</bean>
```

```
<bean id="dog" class="com.test.Dog">  
  <property name="name" value="${name}"></property>  
  <property name="age" value="${age}"></property>  
</bean>
```

或者使用<context>便签

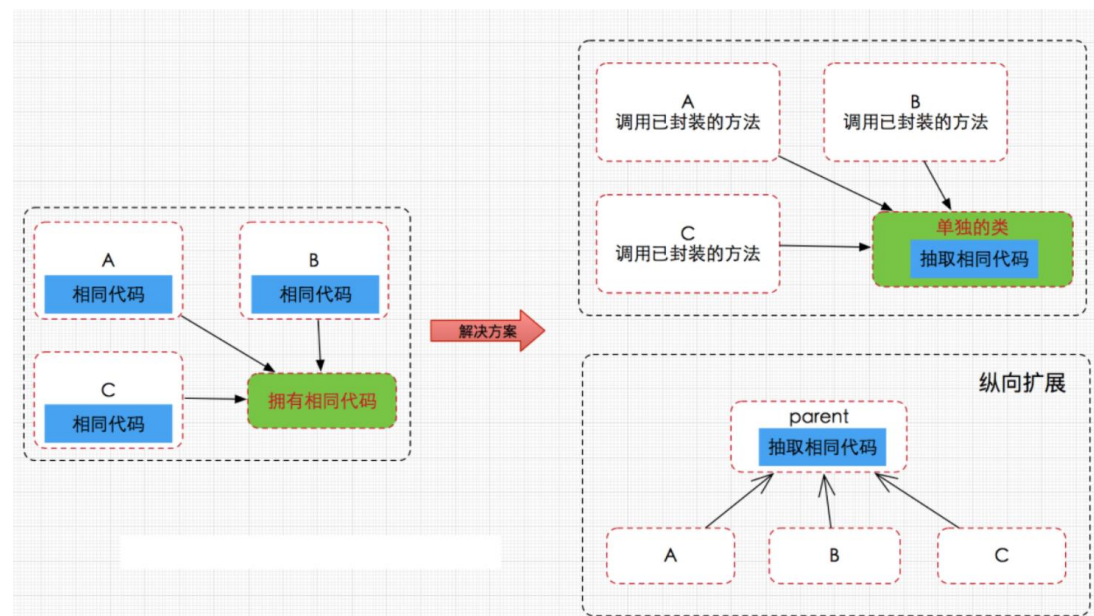
```
<context:property-placeholder location="classpath:com/test/dog.properties"/>
```

Spring AOP

什么是 AOP

引言：项目中如何做日志记录、事务控制？

OOP 方案：



传统的 OOP 程序经常表现出一些不自然的现象，核心业务中总掺杂着一些不相关联的特殊业务，如日志记录，权限验证，事务控制，性能检测，错误信息检测等等，这些特殊业务可以说和核心业务没有根本上的关联而且核心业务也不关心它们。而工程师更希望的是这些模块可以实现热插拔特性而且无需把外围的代码入侵到核心模块中

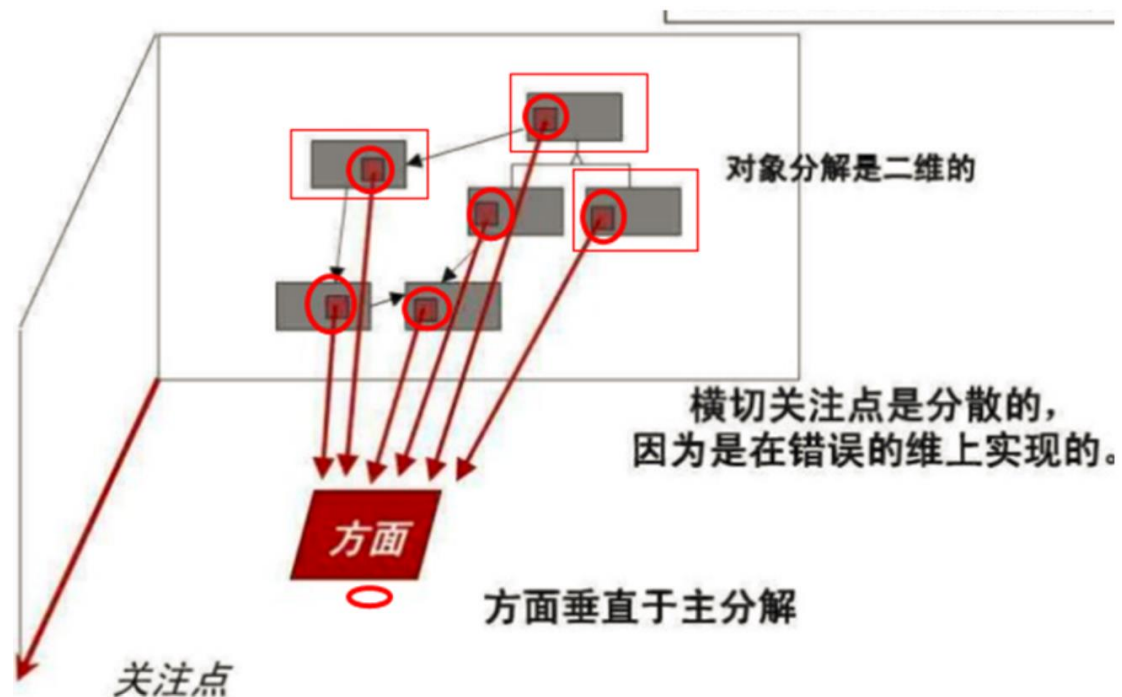


每个关注点与核心业务模块分离，作为单独的功能，横切几个核心业务模块，这样的做的好处是显而易见的，每份功能代码不再单独入侵到核心业务类的代码中，即核心模块只需关注自己相关的业务，当需要外围业务(日志，权限，性能监测、事务控制)时，这些外围业务会通过一种特殊的技术自动应用到核心模块中，这些关注点有个特殊的名称，叫做“横切关注点”，上图也很好的表现出这个概念，另外这种抽象级别的技术也叫 AOP（面向切面编程）。

AOP 为 Aspect Oriented Programming 的缩写，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。

AOP 采取横向抽取机制，将分散在各个方法中的重复代码提取出来，然后在程序编译或者运行时，再将这些提取出来的代码应用到需要执行的地方。

AOP 的本质是拦截方法的，对方法进行增强。如何增强？通过代理----所以 AOP 的本质就是代理。



AOP 的基本概念：

- (1)Aspect(切面):通常是一个类，里面可以定义切入点和通知
- (2)JointPoint(连接点):程序执行过程中明确的点，一般是方法的调用
- (3)Advice(通知):AOP 在特定的切入点上执行的增强处理，有 before,after,afterReturning,afterThrowing,around
- (4)Pointcut(切入点):就是带有通知的连接点，在程序中主要体现为书写切入点表达式
- (5)AOP 代理：AOP 框架创建的对象，代理就是目标对象的加强。Spring 中的 AOP 代理可以使 JDK 动态代理，也可以是 CGLIB 代理，前者基于接口，后者基于子类

AOP 的核心概念：拦截

- 通知 - 定义在哪一个时间点上去做。在类上有方法,执行之前,在方法执行之后,在方法执行的过程中,在方法执行并成功的返回值以后,在方法抛出异常以后。
- 连接点 - 拦截哪一个类，哪一个方法。

AOP（拦截） = 通知（时间） + 切点（在哪个类上）

Advisor = Advice + PointCut

Advice --通知

类 -接口，要求用户实现	功能	
MethodBeforeAdvice	前通知	
AfterAdvice	后通知	

AroundAdvice MethodInterceptor	环绕通知	
AfterReturningAdvice	返回值以后通知	
ThrowsAdvice	抛出错误以后通知	

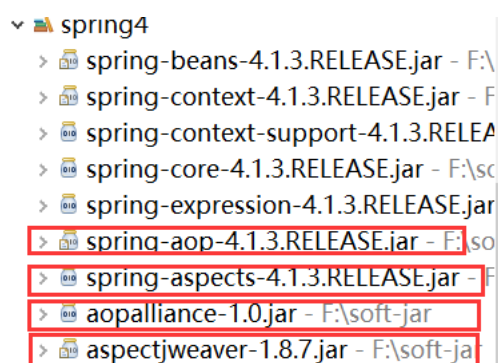
PointCut –切点

类，要用户去实例化就可以了		
JDKRegexpPatternPointCut	基于正则表达式的切点	.* 表示所有 com.domain.*
AspectJExpressionPointCut	基于 AOP 的切点表达式	

Spring 中的 aop 代理，可以是 JDK 动态代理，也可是是 CGLIB 代理。

JDK 动态代理

JDK 动态代理是通过 `java.lang.reflect.Proxy` 类来实现的，我们可以调用 `Proxy` 类的 `newProxyInstance()` 方法来创建代理对象。对于使用业务接口的类，Spring 默认会使用 JDK 动态代理来实现 AOP。



代码演示：

1、在 `com.hpe.jdk` 下，新建 `UserDao` 接口

```
public interface UserDao {
    public void addUser();
    public void deleteUser();
}
```

2、新建 `UserDao` 接口的实现类 `UserDaoImpl`

```
public class UserDaoImpl implements UserDao{
    public void addUser() {
        System.err.println("添加用户...");
    }
    public void deleteUser() {
        System.err.println("删除用户...");
    }
}
```

```
}  
}
```

3、新建切面类 MyAspect,模拟权限检查和记录日志

//切面类，可以存放多个通知advice（即增强方法）

```
public class MyAspect {  
    public void check(){  
        System.err.println("模拟检查权限。。。");  
    }  
    public void log(){  
        System.err.println("模拟记录日志.....");  
    }  
}
```

4、创建代理类 JdkProxy，该类需要实现 InvocationHandler 接口，变编写代理方法

//jdk代理类

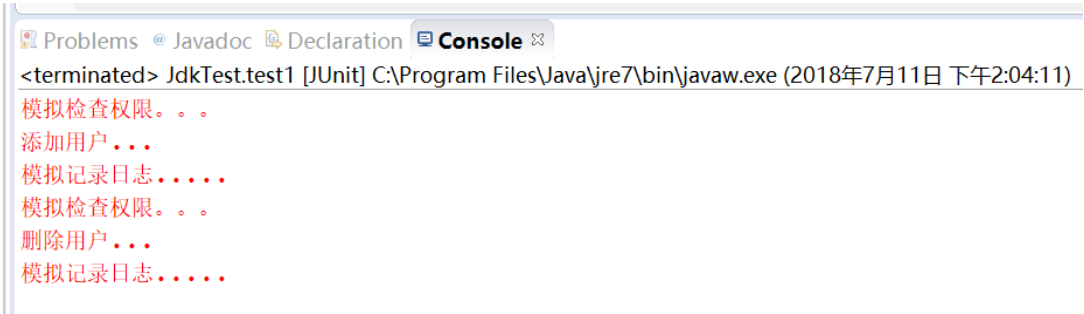
```
public class JdkProxy implements InvocationHandler{  
    //声明目标类接口  
    private UserDao userDao;  
    //创建一个代理方法  
    public Object createProxy(UserDao userDao){  
        this.userDao = userDao;  
        //1、类加载器  
        ClassLoader classLoader = JdkProxy.class.getClassLoader();  
        //2、被代理对象实现的所有接口  
        Class[] clazz = userDao.getClass().getInterfaces();  
        //3、使用代理类，进行增强，返回的是代理后的对象  
        Object obj = Proxy.newProxyInstance(classLoader,clazz,this);  
        return obj;  
    }  
    /*  
    * 所有动态代理类的方法调用，都交给invoke()方法去处理  
    * proxy是被代理后的对象  
    * method 将被执行的方法(反射)  
    * args 执行方法时需要的参数  
    */  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws  
    Throwable { //声明切面  
        MyAspect myAspect = new MyAspect();  
        //强增强  
        myAspect.check();  
        //在目标上调用方法  
        Object obj = method.invoke(userDao, args);  
        //后增强  
        myAspect.log();  
    }  
}
```

```
        return obj;
    }
}
```

5、创建测试类

```
@Test
    public void test1(){
        //1、创建代理对象
        JdkProxy jdkProxy = new JdkProxy();
        //2、创建目标对象
        UserDao userDao = new UserDaoImpl();
        //3、从代理对象中获取增强后的 目标对象
        UserDao userDaoProxy = (UserDao) jdkProxy.createProxy(userDao);
        //4、执行方法
        userDaoProxy.addUser();
        userDaoProxy.deleteUser();
    }
```

打印结果：



****CGLIB 代理**

JDK 动态代理的局限性—使用动态代理的对象必须实现一个或者多个接口。如果要对没有实现接口的对象进行代理，那么可使用 CGLIB 代理。

CGLIB(Code Generation Library)是一个高性能开源的代码生成包，他采用非常底层的字节码技术，对指定的木匾类生成一个子类，并对子类进行增强。

了解即可。代码演示略。

基于代理类的 AOP 实现—java 代码实现

类 –接口， 要求用户实现	功能	
MethodBeforeAdvice	前通知	
AfterAdvice	后通知	
AroundAdvice MethodInterceptor	环绕通知	

AfterReturningAdvice	返回值以后通知	
ThrowsAdvice	抛出错误以后通知	

1、创建一个通知

```
public class MyAdvice implements MethodBeforeAdvice{
    @Override
    public void before(Method arg0, Object[] arg1, Object arg2) throws Throwable
    {
        // TODO Auto-generated method stub
        System.err.println("方法前:"+arg0.getName()+" ,拦截的是哪一个类:"+arg2);
    }
}
```

2、创建一个被代理对象

```
public class Person {
    public void say(){
        System.err.println(" ----- hello every body..");
    }
    public void run(){
        System.err.println("i am running.....");
    }
    public void geeting(){
        System.err.println("大家好同学们");
    }
}
```

3、aop 的测试类

```
@Test
    public void test1() throws Exception {
        //1:声明通知
        Advice advice = new MyAdvice();
        //2:声明切入点
        JdkRegexpMethodPointcut cut =
            new JdkRegexpMethodPointcut();
        //设置对这个类切入,以下切点是指在下com包下的,不管是多么深入的包,只要是peron的方法都拦截
        cut.setPattern("com.*.Person.*");
        //3:将通知+切点=AOP
        Advisor advisor = new DefaultPointcutAdvisor(cut, advice);
        //4:声明代理的Bean
        ProxyFactory proxy = new ProxyFactory(new Person());
        //4.2:设置aop
        proxy.addAdvisor(advisor);
        //5:从代理中获取代理的对象
        Object obj = proxy.getProxy();
        Person pp = (Person) obj;
```

```

        pp.say();
        System.err.println("-----");
        pp.run();
        System.err.println("-----");
        pp.geeting();
    }

```

作业：实现前通知+后通知

**基于代理类的 AOP 实现—XML 配置方式

AspectJ 开发

AspectJ 是一个基于 Java 语言的 AOP 框架，它提供了强大的 AOP 功能。

使用 AspectJ 实现 AOP 有两种方式：一种是基于 XML 的声明式 AspectJ，一种是基于注解的声明式 AspectJ

1、基于 AspectJ 的切点表达式

类		问题
JDKRegexpPatternPointCut	基于正则表达式的切点	1、限制方法的参数 2、不能限制返回值
AspectJExpressionPointCut	基于 AOP 的切点表达式	可以解决上述问题

java 案例：

1、新建 com.bean.User 对象

```

public class User {
    public String login(String name){
        System.err.println("登录名称为: "+name);
        return "name is "+name;
    }
    public String login(String name,int pwd){
        System.err.println("登录名称为: "+name+",密码为: "+pwd);
        return "name is "+name+", pwd is "+pwd;
    }
}

```

2、新建一个环绕通知

```

public class MyArounAdvice implements MethodInterceptor{

```

```
@Override
public Object invoke(MethodInvocation arg0) throws Throwable {
    System.err.println("方法执行前:
"+arg0.getMethod().getName()+", "+arg0.getArguments()[0]);
    Object obj = arg0.proceed();
    System.err.println("执行完成之后...");
    return obj;
}
}
```

3、新建基于 AspectJ 切点的测试类

```
@Test
public void test01(){
    //1、声明通知
    Advice advice = new MyArounAdivice();
    //2、基于AspectJ声明切点对象
    AspectJExpressionPointcut cut = new AspectJExpressionPointcut();
    //3、设置表达式
    /*返回类型为String类型 com.bean.User的login方法 参数为String类型*/
    //cut.setExpression("execution (String com.bean.User.login(String))");
    //任意放回类型 com包下包括com子包下 任意方法 任意的参数0~n
    cut.setExpression("execution(* com..*(..))");
    //4、切点+通知 =AOP
    Advisor advisor = new DefaultPointcutAdvisor(cut, advice);
    //5、声明代理bean
    ProxyFactory proxy = new ProxyFactory(new User());
    //6、设置aop
    proxy.addAdvisor(advisor);
    //7、从代理中获取代理的对象
    User user = (User) proxy.getProxy();
    user.login("rose");
    System.err.println("-----");
    user.login("jack",123);
}
```

功能：

- 1：可以控制返回值。
- 2：可以控制参数。
- 3：可以控制类的深度。
- 4：面向接口的。（创建 IUser 接口，切 IUser 接口，则实现类也会被切）

表达式	含义	
*	这个目录下的，或是类上的所有	
..	任意的多个。0~N 个	
execution (* com.bean.User.login(String,int))	对 login 方法，必须要接收两个参数，且参数的类型必须是 Strig,int 的，且返	

	返回值无限制。且这个方法必须是 User 这个类的	
Execution (* com.*.*(..))	返回所有的类型 所有在 com 包下的类，不包含子包类的所有方法 接收任意多个参数 0~N	
Execution (* com..*.*(..))	在 com 包下的，包含子包下的所有类的所有方法，所有参数都切入	
execution(* com.*.*(String)) execution(* com..*.*(*,int))	逻辑或	

2、基于 XML 的声明式 AspectJ

applicationContext.xml 配置文件命名空间加入以下内容

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

</beans>
```

```
<aop:config></aop:config>
  <aop:advisor
  <aop:aspect
  <aop:pointcut
```

所有的切面、切点、通知都配置到<aop:config>里。其子元素包括<aop:advisor>(配置通知器)、<aop:aspect> (配置切面)、<aop:pointcut>(配置全局切入点)。

注意，这三个元素的顺序必须按照此顺序来定义。

核心：<aop:aspect>元素下配置切面、切点和通知。

案例：

第一步：自定义一个通知

```
public class MyAdvice {
    public void doBefore(JoinPoint joinPoint)
    {
        System.out.println("前置通知，方法开始..."+"：目标类是
"+joinPoint.getTarget()+"，被植入的增强方法是：
```

```

"+joinPoint.getSignature().getName());
    }
    public void doAfter()
    {
        System.out.println("后置通知, 方法结束...");
    }
    public void doReturnAfter()
    {
        System.out.println("返回通知, 方法最终结束...");
    }
    /**
     * 环绕通知 ProceedingJoinPoint用来调用被增强的方法
     * 必须是Object的返回类型
     * 必须接受一个参数, 类型为ProceedingJoinPoint
     * 必须throws Throwable
     */
    public Object doAround(ProceedingJoinPoint joinPoint) throws Throwable
    {
        System.out.println("环绕通知: begin...");
        //执行被增强的方法
        Object obj = joinPoint.proceed();
        System.out.println("环绕通知:end.....");
        return obj;
    }
    public void doThrowing(Throwable e){
        System.out.println("异常通知....."+e.getMessage());
    }
}

```

第二步: 创建一个被代理对象

```

public class User {
    public String login(String name){
        System.err.println("登录名称为: "+name);
        return "name is "+name;
    }
    public String login(String name,int pwd){
        System.err.println("登录名称为: "+name+",密码为: "+pwd);
        return "name is "+name+", pwd is "+pwd;
    }
    public void say(){
        System.err.println("say.....");
    }
    public void run(){
        System.err.println("running.....");
        int a = 1/0;
    }
}

```

```
}  
}
```

第三步: xml 配置文件

```
<bean id="user" class="com.bean.User"></bean>  
  
<!-- 定义一个切面 -->  
<bean id="myBeforeAdvice" class="com.demo03.MyAdvice"></bean>  
  
<aop:config>  
    <!-- 配置切入点 -->  
    <!-- 表达式 (用来表示方法) -->  
    <!-- execution(<访问修饰符>?<返回类型><方法名>(<参数>)<异常>), 返回值, 方法  
    名, 参数不能少 -->  
    <!-- *代表: 任意值 方法名: 全类名.方法名 参数中的..: 任意个数, 任意类型 -->  
    <aop:pointcut expression="execution(* com..*(..))" id="myPointcut" />  
    <!-- 切面配置 -->  
    <aop:aspect ref="myBeforeAdvice">  
        <!-- 配置前通知 -->  
        <aop:before method="doBefore" pointcut-ref="myPointcut" />  
        <!-- 配置后通知 -->  
        <aop:after method="doAfter" pointcut-ref="myPointcut" />  
        <!-- 配置返回通知 -->  
        <aop:after-returning method="doReturnAfter"  
            pointcut-ref="myPointcut" />  
        <!-- 配置环绕通知 -->  
        <aop:around method="doAround" pointcut-ref="myPointcut" />  
        <!-- 异常通知 -->  
        <aop:after-throwing method="doThrowing"  
            pointcut-ref="myPointcut" throwing="e" />  
    </aop:aspect>  
</aop:config>
```

第四步: 测试方法

```
@Test  
public void test1() {  
    ApplicationContext ctx =  
    new ClassPathXmlApplicationContext("com/demo03/applicationContext.xml");  
    User user = ctx.getBean("user", User.class);  
    user.say();  
}
```

Problems @ Javadoc Declaration Console Servers

<terminated> demo3.test1 [JUnit] C:\Program Files\Java\jre7\bin\javaw.exe (2018年7月12日 下午3:51:02)

七月 12, 2018 3:51:02 下午 org.springframework.context.support.AbstractApplicationContext
 信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext
 七月 12, 2018 3:51:02 下午 org.springframework.beans.factory.xml.XmlBeanDefinition
 信息: Loading XML bean definitions from class path resource [com/demo03/applicat
 前置通知, 方法开始...: 目标类是com.bean.User@8c83f72, 被植入的增强方法是: say
 环绕通知: begin...
 say.....
 环绕通知: end.....
 返回通知, 方法最终结束...
 后置通知, 方法结束...

```
@Test
public void test1() {
    ApplicationContext ctx =
    new ClassPathXmlApplicationContext("com/demo03/applicationContext.xml");
    User user = ctx.getBean("user", User.class);
    user.run();
}
```

Problems @ Javadoc Declaration Console Servers

<terminated> demo3.test1 [JUnit] C:\Program Files\Java\jre7\bin\javaw.exe (2018年7月12日 下午3:09:53)

七月 12, 2018 3:09:53 下午 org.springframework.context.support.AbstractApplicationCo
 信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext
 七月 12, 2018 3:09:53 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionRe
 信息: Loading XML bean definitions from class path resource [com/demo03/applicatio
 前置通知, 方法开始...
 环绕通知: begin...
 running.....
 异常通知...../ by zero
 后置通知, 方法结束...

注意: 前通知和后通知配合使用, 效果和环绕通知一致, 二者不一起使用。

在拦截到方法时, 可以接收一个参数: - 可以获取拦截的方法, 参数, 拦截的对象

参数类	用在哪儿
JoinPoint	连接点 - 用在前后抛出这样通知中
ProceedingJoinPont	必须用的环绕通知中。

3、基于注解的声明式 AspectJ

注解	功能
@Aspect	注解在类上, 声明这是一个切面
@Pointcut	注解在方法上声明是一个切点, 且要声明 aspectj 的切点表达式

@Before	前通知
@After @AfterReturning - 正确执行成功返回值 @AfterThrow - 错误的执行，抛出异常	后通知
@Around	环绕通知

第一步：新建注解的切面类

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

/**
 * 切面类
 */
@Aspect
@Component
public class MyAspect {
    //定义一个切入点表达式 使用一个返回值为void,方法体为空的方法来命名切点
    @Pointcut("execution(* com..*(..))")
    public void myPointCut(){}

    //前置通知
    @Before("myPointCut()")
    public void doBefore(JoinPoint joinPoint)
    {
        System.out.println("前置通知，方法开始..."+"目标类是"+joinPoint.getTarget()+"，被植入的增强方法是："+joinPoint.getSignature().getName());
    }

    //后置通知
    @After("myPointCut()")
    public void doAfter()
    {
        System.out.println("后置通知，方法结束...");
    }

    //返回通知
    @AfterReturning("myPointCut()")
```



```

    public void doReturnAfter()
    {
        System.out.println("返回通知，方法最终结束...");
    }
    /**
     * 环绕通知 ProceedingJoinPoint用来调用被增强的方法
     * 必须是Object的返回类型
     * 必须接受一个参数，类型为ProceedingJoinPoint
     * 必须throws Throwable
     */
    @Around("myPointCut()")
    public Object doAround(ProceedingJoinPoint joinPoint) throws Throwable
    {
        System.out.println("环绕通知: begin...");
        //执行被增强的方法
        Object obj = joinPoint.proceed();
        System.out.println("环绕通知:end....");
        return obj;
    }
    @AfterThrowing(value="myPointCut()",throwing="e")
    public void doThrowing(Throwable e){
        System.out.println("异常通知....."+e.getMessage());
    }
}

```

第二步：新建 xml 配置文件

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       default-autowire="byName"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd"
       >
    <bean id="user" class="com.bean.User"></bean>
    <context:component-scan base-package="com.demo04"></context:component-scan>
    <!-- 启动基于注解的声明式AspectJ支持 -->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
</beans>

```

第三步：测试方法

```

@Test
public void test1() {
    ApplicationContext ctx =
    new ClassPathXmlApplicationContext("com/demo04/applicationContext.xml");
    User user = ctx.getBean("user", User.class);
    user.say();
    //user.run();
}

```

Problems Javadoc Declaration Console Servers

<terminated> demo4.test1 [JUnit] C:\Program Files\Java\jre7\bin\javaw.exe (2018年7月12日 下午4:12:14)

七月 12, 2018 4:12:14 下午 org.springframework.context.support.AbstractApplica
 信息: Refreshing org.springframework.context.support.ClassPathXmlApplication
 七月 12, 2018 4:12:14 下午 org.springframework.beans.factory.xml.XmlBeanDefini
 信息: Loading XML bean definitions from class path resource [com/demo04/appl
 环绕通知: begin...
 前置通知, 方法开始...: 目标类是com.bean.User@3098cc00, 被植入的增强方法是: say
 say.....
 环绕通知: end.....
 后置通知, 方法结束...
 返回通知, 方法最终结束...

Spring+mybatis 整合

实现 mybatis 与 spring 进行整合, 通过 spring 容器来管理 SqlSessionFactory、mapper 接口
 核心包: mybatis-spring-1.*.*.jar

LICENSE
 mybatis-spring-1.2.2.jar
 NOTICE

还包括其它 jar:

spring 4.1.3
 mybatis3.2.7
 c3p0 连接池
 数据库驱动

1、核心配置文件

新建项目, 导入 jar 包, 在 src 下分别创建 db.properties 资源文件、日志文件 log4j.properties、
 Spring 的配置文件 applicationContext.xml、Mybatis 的配置文件 SqlMapConfig.xml。

➤ db.properties:

```
jdbc.driver=com.mysql.jdbc.Driver
```

```
jdbc.url=jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8
jdbc.username=root
jdbc.password=root
```

➤ log4j.properties:

```
# Global logging configuration
log4j.rootLogger=DEBUG, stdout
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

➤ applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       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
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd"
>
    <!-- 1、加载外部资源文件 -->
    <context:property-placeholder location="classpath:db.properties"/>
    <!-- 2、配置数据库连接池 -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${jdbc.driver}"></property>
        <property name="jdbcUrl" value="${jdbc.url}"></property>
        <property name="user" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
    </bean>
    <!-- 3、配置sqlSessionFactory
        让spring来管理sqlSessionFactory,使用mybatis-spring包中的
        SqlSessionFactoryBean -->
    <bean id="sqlSessionFactory"
        class="org.mybatis.spring.SqlSessionFactoryBean">
        <!-- 3.1 配置数据源 -->
        <property name="dataSource" ref="dataSource"></property>
        <!-- 3.2 加载mybatis全局配置文件 -->
        <property name="configLocation"
            value="classpath:SqlMapConfig.xml"></property>
```

```
</bean>
</beans>
```

注：在 spring 中配置数据源和 sqlSessionFatory。Mybatis 的工厂 sqlSessionFatory 的创建时通过 mybatis-spring 包中提供的 SqlSessionFactoryBean 来配置的。通常，在配置的时候需要两个参数：一个是数据源，一个是 mybatis 的配置文件路径。这样 spring 的 IOC 容器就会在初始化 id 为 sqlSessionFatory 的 bean 时解析 mybatis 的配置文件，并且与数据源一同保存到 spring 的 Bean 中。

➤ SqlMapConfig.xml

```
<?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>
    <typeAliases>
        <package name="com.hpe.po"/>
    </typeAliases>
    <!-- 配置 mapper -->
    <mappers>

    </mappers>
</configuration>
```

注：由于 spring 中已经配置了数据源信息，所以 mybatis 中的配置文件中就不需要配置数据源信息。这里只需要使用<typeAliases>和<mappers>元素来配置别名和 mapper 位置即可。

2、传统 DAO 方式的开发整合

采用传统 Dao 开发方式进行 mybtis 和 spring 框架的整合，需要编写 DAO 接口以及接口的实现类，并且向 DAO 的实现类中注入 SqlSessionFactory，然后在方法体内通过 SqlSessionFactory 创建 SqlSession。为此，Dao 实现类需要继承 SqlSessionDaoSupport 类。

SqlSessionDaoSupport：是一个抽象类，可以通过调用 getSession()方法来获取所需的 SqlSession。

第一步：实现持久层类。

在 com.hpe.po 包下新建实体类 User.java

```
public class User {
    private int id;
    private String username; // 用户姓名
    private String sex; // 性别
    private Date birthday; // 生日
    private String address; // 地址
```

```
    get**()  set**()  toString()
}
```

第二步：mapper.xml 配置

在 src 下新建 UserMapper.xml

```
<?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="UserMapper">
    <select id="findUserById" parameterType="Integer" resultType="user">
        select * from user where id = #{id}
    </select>
</mapper>
```

修改 mybatis 的配置文件 SqlMapConfig.xml:

```
<mappers>
    <mapper resource="UserMapper.xml"/>
</mappers>
```

第三步：实现 DAO 层

在 com.hpe.dao 路径下新建 UserDao 接口

```
public interface UserDao {
    //根据id查询用户信息
    public User findUserById(int id);
}
```

在 com.hpe.dao.impl 路径下新建 UserDao 接口的实现类 UserDaoImpl:

```
public class UserDaoImpl extends SqlSessionDaoSupport implements UserDao{
    @Override
    public User findUserById(int id) {
        return this.getSqlSession().selectOne("UserMapper.findUserById", id);
    }
}
```

第四步：spring 配置文件

```
<!-- 配置bean -->
<bean id="userDao" class="com.hpe.dao.impl.UserDaoImpl">
    <!-- 注入SqlSessionFactory对象实例 -->
    <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
</bean>
```

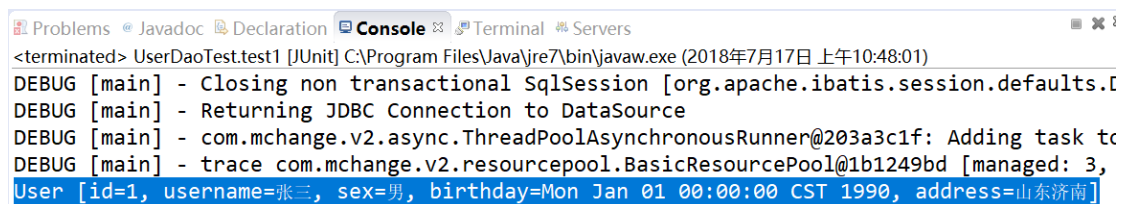
注：由于 SqlSessionDaoSupport 类在使用时需要一个 SqlSessionFactory 对象，所以需要 spring 给 SqlSessionDaoSupport 的子类对象注入一个 SqlSessionFactory。

第五步：整合测试：

在 com.hpe.test 路径下新建 UserDaoTest.java

```
public class UserDaoTest {  
    private ApplicationContext ctx ;  
    @Before  
    public void init(){  
        //spring容器运行环境  
        ctx = new ClassPathXmlApplicationContext("applicationContext.xml");  
    }  
    @Test  
    public void test1(){  
        UserDao userDao = ctx.getBean("userDao",UserDao.class);  
        User user = userDao.findUserById(1);  
        System.err.println(user);  
    }  
}
```

运行结果如下：



```
<terminated> UserDaoTest.test1 [JUnit] C:\Program Files\Java\jre7\bin\javaw.exe (2018年7月17日 上午10:48:01)  
DEBUG [main] - Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@1a2b3c4d]  
DEBUG [main] - Returning JDBC Connection to DataSource  
DEBUG [main] - com.mchange.v2.async.ThreadPoolAsynchronousRunner@203a3c1f: Adding task to  
DEBUG [main] - trace com.mchange.v2.resourcepool.BasicResourcePool@1b1249bd [managed: 3,  
User [id=1, username=张三, sex=男, birthday=Mon Jan 01 00:00:00 CST 1990, address=山东济南]
```

总结：采用传统的 DAO 开发方式虽然可以实现所需功能，但是采用这种方式在实现类中会出现大量的重复代码，在方法中也需要指定映射文件中执行的语句的 id，并且不能保证编写时 id 的正确性。

3、Mapper 接口的方式整合

3.1 基于 MapperFactoryBean 的整合

MapperFactoryBean 是一个根据 Mapper 接口生成 Mapper 对象的类，该类在 spring 的配置文件中使用时可以配置一下参数：

- mapperInterface: 用于指定接口
- sqlSessionFacotry: 用于指定 SqlSessionFactory

第一步：在 com.hpe.mapper 下新建 UserMapper 接口以及其对应的映射文件。

UserMapper.java:

```
public interface UserMapper {  
    //根据id查询用户信息
```

```
    public User findUserById(int id);  
}
```

UserMapper.xml:

```
<?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.hpe.mapper.UserMapper">  
    <select id="findUserById" parameterType="Integer" resultType="user">  
        select * from user where id = #{id}  
    </select>  
</mapper>
```

第二步: spring 配置文件, 创建 userMapper 的 Bean

```
<!-- 配置bean Mapper代理开发 -->  
<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">  
    <!-- 创建哪个接口所对应的bean -->  
    <!-- name: mapperInterface, value: 接口 (创建哪个接口所对应的bean) -->  
    <property name="mapperInterface"  
value="com.hpe.mapper.UserMapper"></property>  
    <!-- 注入sqlSessionFactory -->  
    <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>  
</bean>
```

第三步: 测试

```
public class UserMapperTest {  
    private ApplicationContext ctx;  
    @Before  
    public void init() {  
        // spring容器运行环境  
        ctx = new ClassPathXmlApplicationContext("applicationContext.xml");  
    }  
    @Test  
    public void test1() {  
        UserMapper userMapper = (UserMapper) ctx.getBean("userMapper");  
        User user = userMapper.findUserById(1);  
        System.err.println(user);  
    }  
}
```

运行结果:

```

Problems @ Javadoc Declaration Console Terminal Servers
<terminated> UserMapperTest.test1 [JUnit] C:\Program Files\Java\jre7\bin\javaw.exe (2018年7月17日 上午11:56:31)
DEBUG [main] - Closing non transactional SqlSession [org.apache.ibatis.session.defaults.
DEBUG [main] - Returning JDBC Connection to DataSource
DEBUG [main] - com.mchange.v2.async.ThreadPoolAsynchronousRunner@dc50766: Adding task to
DEBUG [main] - trace com.mchange.v2.resourcepool.BasicResourcePool@5c1ac001 [managed: 3,
User [id=1, username=张三, sex=男, birthday=Mon Jan 01 00:00:00 CST 1990, address=山东济南]

```

3.2 基于 MapperScannerConfigurer 的整合

在实际的项目中，DAO 层会包含很多接口，如果每个都在 spring 的配置文件中配置的话，工作量太大，而且 spring 的配置文件也非常臃肿。为此，可以采用自动扫描的形式来配置 mybatis 的映射器——MapperScannerConfigurer 类。

第一步：spring 的配置文件

```

<!-- 使用包扫描的方式 批量创建mapper的bean，bean的id就是接口名称，首字母小写 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!-- 指定Mapper接口所在的包 basePackage: 扫描包路径，中间可以用逗号或分号分隔
定义多个包 -->
    <property name="basePackage" value="com.hpe.mapper"></property>
</bean>

```

第二步：测试

结果无问题。

此时 mybatis 的配置文件里只配置了别名，是否可以去掉，在 spring 中配置？

--可以 spring 配置文件如下：

```

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <!-- 3.1 配置数据源 -->
    <property name="dataSource" ref="dataSource"></property>
    <!-- 3.2 加载mybatis全局配置文件 -->
    <property name="configLocation"
value="classpath:SqlMapConfig.xml"></property>
    <!-- 3.3 指定别名包 -->
    <property name="typeAliasesPackage" value="com.hpe.po"></property>
</bean>

```


Spring 事务

1. 事务是什么？

事务是逻辑上的一组操作,组成这组操作的各个逻辑单元,要么一起成功,要么一起失败.

事务四个特性：ACID

- 原子性（Atomicity）：事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。
- 一致性（Consistency）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。在现实中的数据不应该被破坏。
- 隔离性（Isolation）：可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。
- 持久性（Durability）：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

如果不考虑隔离级别引发的安全性问题

- 脏读：一个事务读到了另一个事务的未提交的数据，如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。
- 不可重复读：一个事务读到了另一个事务已经提交的 `update` 的数据导致多次查询结果不一致。这通常是因为另一个并发事务在两次查询期间进行了更新。
- 幻读：一个事务读到了另一个事务已经提交的 `insert` 的数据导致多次查询结果不一致。

解决读问题:设置事务隔离级别

隔离级别	描述
DEFAULT	使用底层数据库的默认隔离级别。对于大多数数据库来说，默认隔离级别都是 READ_COMMITTED
READ_UNCOMMITTED	允许事务读取未被其他事物提交的变更。脏读，不可重复读和幻读的问题都会出现
READ_COMMITTED	只允许事务读取已经被其它事务提交的变更。可以避免脏读，但不可重复读和幻读问题仍然可能出现
REPEATABLE_READ	确保事务可以多次从一个字段中读取相同的值。在这个事务持续期间，禁止其他事物对这个字段进行更新。可以避免脏读和不可重复读，但幻读的问题仍然存在。
SERIALIZABLE	确保事务可以从一个表中读取相同的行。在这个事务持续期间，禁止其他事务对该表执行插入，更新和删除操作。所有并发问题都可以避免，但性能十分低下。

2. 事务管理核心接口

依赖包：**spring-tx-4.*.RELEASE**

◆ 事务管理器：PlatformTransactionManager

Spring 并不直接管理事务，而是提供了多种事务管理器，他们将事务管理的职责委托给 Hibernate、Mybatis 或者 JTA 等持久化机制所提供的相关平台框架的事务来实现。

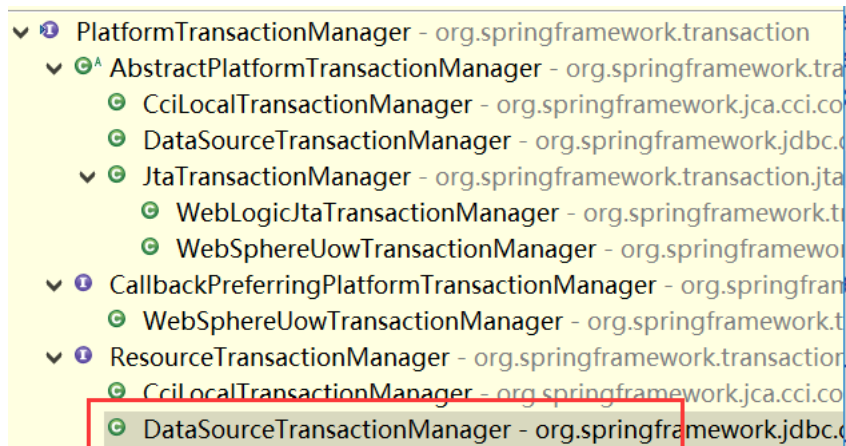
Spring 事务管理器的接口是

org.springframework.transaction.PlatformTransactionManager

① PlatformTransactionManager

- [^] getTransaction(TransactionDefinition) : TransactionStatus
- [^] commit(TransactionStatus) : void
- [^] rollback(TransactionStatus) : void

- getTransaction：用户获取事务状态信息
- commit：用于提交事务
- rollback：用于回滚事务



重要的一个实现类为

org.springframework.jdbc.datasource.DataSourceTransactionManager，用于配置 JDBC

◆ 事务属性 **TransactionDefinition**

事务属性可以理解成事务的一些基本配置，描述了事务策略如何应用到方法上。事务属性包含了 5 个方面

TransactionDefinition

PROPAGATION_REQUIRED : int

PROPAGATION_SUPPORTS : int

PROPAGATION_MANDATORY : int

PROPAGATION_REQUIRES_NEW : int

PROPAGATION_NOT_SUPPORTED : int

PROPAGATION_NEVER : int

PROPAGATION_NESTED : int

ISOLATION_DEFAULT : int

ISOLATION_READ_UNCOMMITTED : int

ISOLATION_READ_COMMITTED : int

ISOLATION_REPEATABLE_READ : int

ISOLATION_SERIALIZABLE : int

TIMEOUT_DEFAULT : int

getPropagationBehavior() : int

getIsolationLevel() : int

getTimeout() : int

isReadOnly() : boolean

getName() : String

- getPropagationBehavior: 获取事务的传播行为
- getIsolationLevel: 获取事务的隔离级别，事务管理器根据它来控制另外一个事务可以看到本事务内的哪些数据
- getTimeout: 获取事务的超时时间
- isReadOnlys: 获取是否只读
- getName: 获取事务对象名称

传播行为:

传播行为	值	描述
PROPAGATION_REQUIRED	REQUIRED	表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务
PROPAGATION_SUPPORTS	SUPPORTS	表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行，否则不使用事务
PROPAGATION_MANDATORY	MANDATORY	表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常
PROPAGATION_REQUIRED_NEW	REQUIRED_NEW	表示当前方法必须运行在它自己的事务中。如果存在当前事务，在该方法执行期间，当前事务会被挂起，再启动新的事务执行该方法；如果当前没有事务，这开启新的事务。
PROPAGATION_NOT_SUPPORTED	NOT_SUPPORTED	表示该方法不应该运行在事务中。如果存在当前事务，在

NOT_SUPPORTED		该方法运行期间，当前事务将被挂起。
PROPAGATION_NEVER	NEVER	表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常
PROPAGATION_NESTED	NESTED	表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与 PROPAGATION_REQUIRED 一样

通常情况下，数据的查询不会影响元数据的改变，所以不需要进行事务管理，而对于数据的插入、更新和删除操作，则必须进行事务管理。

如果没有指定事务的传播行为，spring 默认的传播行为是 REQUIRED。

◆ 事务状态 **TransactionStatus**

TransactionStatus 接口是事务的状态，它描述了某一时间点上事务的状态信息。

① TransactionStatus	
• ^A isNewTransaction() : boolean	→ 获取是否新事物
• ^A hasSavepoint() : boolean	→ 获取是否存在保存点
• ^A setRollbackOnly() : void	→ 设置事务回滚
• ^A isRollbackOnly() : boolean	→ 获取是否回滚
• ^A flush() : void	→ 刷新事务
• ^A isCompleted() : boolean	→ 获取事务是否完成

◆ 事务管理的方式

Spring 中的事务管理分为两种方式：一种是传统的的编程式事务管理，另一种是声明式事务管理。

- 编程式事务管理：通过编写代码实现事务管理，包括定义事务的开始、正常执行后的事务提交和异常时的事务回滚。
- 声明式事务管理：通过 AOP 技术实现事务管理，其主要的思想是讲事务管理作为一个‘切面’代码单独编写，然后通过 AOP 技术将事务管理的‘切面’代码植入到业务目标类中。

显而易见，声明式事务管理最大的优点在于开发者无须通过编程的方式来管理事务，只需在配置文件中进行相关的事务规则声明，就可以将事务规则应用到业务逻辑中。这样的开发效率高，所以在实际开发中，推荐使用声明式事务管理。

3. 声明式事务管理

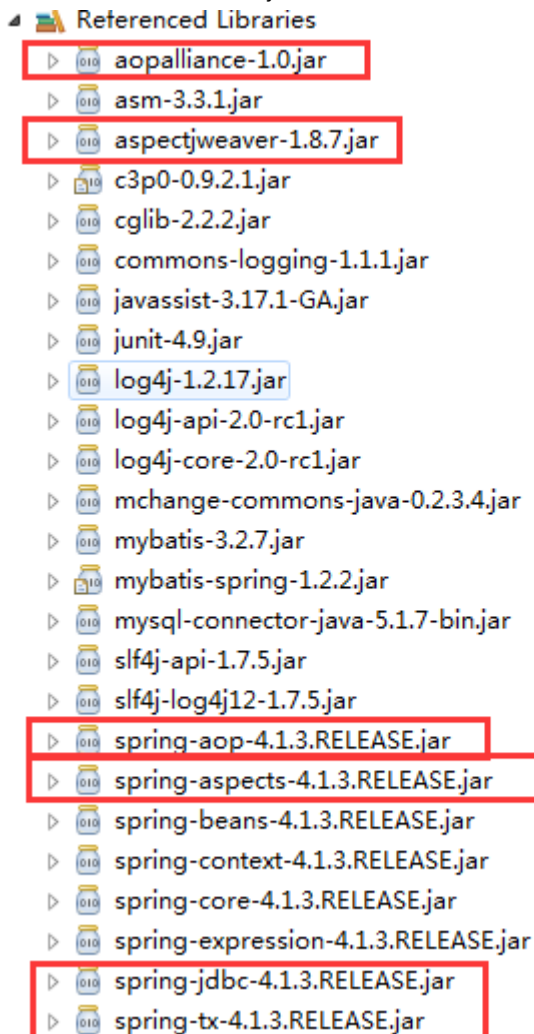
Spring 的声明式事务管理可以通过两种方式来实现，一种是基于 XML 的方式，另一种是基

于 Annotation 的方式。

3.1 基于 XML 方式的事务管理

基于 XML 方式的事务管理是通过在配置文件中配置事务规则的相关声明来实现的。Spring 提供了 tx 命名空间来配置事务,<tx:advice>元素来配置事务的通知。

第一步：新建项目，导入 jar 包



第二步：根据 spring 和 mybatis 的整合新建配置文件。

在 src 下分别创建 db.properties 资源文件、日志文件 log4j.properties、Spring 的配置文件 applicationContext.xml、Mybatis 的配置文件 SqlMapConfig.xml

Spring 配置文件，扫描注解加载 bean，如下

```
<!-- 5、注解扫描 -->
<context:component-scan base-package="com.hpe.service"/></context:component-scan>
```

第三步：新建 User 对象、Mapper 接口和映射文件、Service 接口和实现类

UserServiceImpl 代码片段

```

@Override
    public int addUser() {
        // TODO Auto-generated method stub
        User user = new User(0, "jack1", "男", new Date(), "山东济宁");
        User user2 = new User(0, "jack2", "男", new Date(), "山东济南");
        userMapper.addUser(user);
        int a = 1/0;
        userMapper.addUser(user2);
        return 0;
    }

```

其中 int a = 1/0，是为了代码执行到当前行往外抛异常。
其他代码略，见项目案例。

第四步：测试代码

```

public class UserTest {
    private ApplicationContext ctx;
    @Before
    public void init() {
        // spring容器运行环境
        ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
    }
    @Test
    public void test() {
        UserService userService = (UserService) ctx.getBean("userService");
        int res = userService.addUser();
        System.err.println();
    }
}

```

运行结果：

Jack1 成功插入数据库，而 jack2 没有插入。因为当前是没有事务的

第五步：XML 配置事务

Spring 配置文件 加入 tx 命名空间：

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    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
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd

```

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

Spring 配置文件，添加以下配置：

```
<!-- 5、注解扫描 -->
<context:component-scan base-package="com.hpe.service"></context:component-scan>
<!-- 6.1、spring的声明式事务 事务管理器-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- 注入数据源 -->
    <property name="dataSource" ref="dataSource"></property>
</bean>
<!-- 6.2 编写通知，配置事务增强，需要编写切入点和具体执行事务细节-->
<tx:advice id="txAdvice" transaction-manager="transactionManager" >
    <!--
        isolation="DEFAULT" 隔离级别
        propagation="REQUIRED" 传播行为
        read-only="false" 只读
        timeout="-1" 过期时间
        rollback-for="" 对哪些异常回滚
        no-rollback-for="" 对哪些异常不会滚
    -->
    <tx:attributes>
        <tx:method name="add*" propagation="REQUIRED" />
        <tx:method name="update*" propagation="REQUIRED" />
        <tx:method name="delete*" propagation="REQUIRED" />
        <tx:method name="select*" propagation="SUPPORTS" />
    </tx:attributes>
</tx:advice>
<!-- 6.3 配置AOP事务 AspectJ表达式方式 -->
<aop:config>
    <aop:pointcut expression="execution(* com.hpe.service..*(..))"
id="pointCut1"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="pointCut1"/>
</aop:config>
```

第六步：再次测试

测试结果：

Jack1 和 jack2 都没有插入数据库。当前开启了事务，异常引起了事务的回滚。

扩展：

- 1、把 add*方法的传播行为修改为 SUPPORTS，结果是插入一条记录，未开启事务。
- 2、给 add*方法 加上配置 no-rollback-for="java.lang.ArithmeticException"，这对 1/0 的异常不会滚

3.2 基于注解方式的事务管理

注解为@Transactional，具体使用方式如下

第一步：注册事务注解驱动

Spring 配置文件

```
<!-- 5、注解扫描 -->
<context:component-scan base-package="com.hpe.service"></context:component-scan>
<!-- 6.1、spring的声明式事务 事务管理器-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- 注入数据源 -->
    <property name="dataSource" ref="dataSource"></property>
</bean>
<!-- 6.2 注册事务注解驱动 -->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

- 使用@Transactional 注解声明时事务，必要要配置<tx:annotation-driven>。
- 事务处理器的名称是 transactionManager，就可以在<tx:annotation-driven> 元素中省略 transaction-manager 属性。这个元素会自动检测该名称的事务处理器。

第二步：使用@Transactional 注解

```
@Service("userService")
@Transactional
public class UserServiceImpl implements UserService{
```

- 把@Transactional 注解到类上，表示事务的设置对这个类所有的方法都起作用。
- 把@Transactional 注解到类的方法上，表示事务的设置只对该方法起作用。

@Transactional 注解也可以加入以下属性信息，也 XML 方式含义相同。


```
• isolation : Isolation - Transactional
• noRollbackFor : Class<? extends java.lang.Throwable>[] - Transactional
• noRollbackForClassName : String[] - Transactional
• propagation : Propagation - Transactional
• readOnly : boolean - Transactional
• rollbackFor : Class<? extends java.lang.Throwable>[] - Transactional
• rollbackForClassName : String[] - Transactional
• timeout : int - Transactional
• value : String - Transactional
```

第三步：测试

**Scheduler 任务调度