# 事务控制

## 事务的概念

事务就是一系列指令的集合。

理解事务之前，先讲一个你日常生活中最常干的事：取钱。

比如你去ATM机取1000块钱，大体有两个步骤：首先输入密码金额，银行卡扣掉1000元钱；然后ATM出1000元钱。这两个步骤必须是要么都执行要么都不执行。如果银行卡扣除了1000块但是ATM出钱失败的话，你将会损失1000元；如果银行卡扣钱失败但是ATM却出了1000块，那么银行将损失1000元。所以，如果一个步骤成功另一个步骤失败对双方都不是好事，如果不管哪一个步骤失败了以后，整个取钱过程都能回滚，也就是完全取消所有操作的话，这对双方都是极好的。

事务就是用来解决类似问题的。事务是一系列的动作，它们综合在一起才是一个完整的工作单元，这些动作必须全部完成，如果有一个失败的话，那么事务就会回滚到最开始的状态，仿佛什么都没发生过一样。

在企业级应用程序开发中，事务管理必不可少的技术，用来确保数据的完整性和一致性。

事务必须满足ACID原则，ACID指的是原子性（atomicity）、一致性（consistency）、隔离性（isolation）和持久性（durability）。

1、原子性：即事务是不可分割的最小工作单元，事务内的指令要么全部执行，要么全不执行。

2、一致性：指事务前后数据的完整性必须保持一致，一旦事务完成（不管成功还是失败），系统必须确保数据处于一致的状态，而不会是部分完成部分失败。比如A转账给B，不能A的钱转出去了，但是B没有收到。

3、隔离性：指多个用户并发访问数据库时，一个用户的事务不能被其他用户的事务所干扰，多个并发事务之间数据要相互隔离

4、持久性：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。

## 事务类型

### 数据库事务类型

1. 本地事务：就是数据库事务类型中的本地事务，通过Connection对象的控制来管理事务；
2. 分布式事务：涉及两个或多个数据库源的事务，即跨越多台同类或异类数据库的事务（由每台数据库的本地事务组成的），分布式事务旨在保证这些本地事务的所有操作的ACID，使事务可以跨越多台数据库；

### Java事务类型

1. JDBC事务：就是数据库事务类型中的本地事务，通过Connection对象的控制来管理事务；
2. JTA事务：JTA指Java事务API(Java Transaction API)，是Java EE数据库事务规范， JTA只提供了事务管理接口，由应用程序服务器厂商（如WebSphere Application Server）提供实现，JTA事务比JDBC更强大，支持分布式事务。

### Java EE事务类型

1. 本地事务：使用JDBC编程实现事务；
2. 全局事务：由应用程序服务器提供，使用JTA事务；

### Spring事务类型

1. 编程式事务：通过编写代码实现事务；
2. 声明式事务：通过注解或XML配置文件指定事务信息；

Spring虽然支持编程式事务，但在实际开发中，即使需要事务控制的地方很少，也建议使用声明式事务。因为声明式事务一是通过配置能够大大减少模板代码的使用；二是声明式事务是无侵入性的。

## 事务属性

事务属性：事务属性可以理解为事务的一些基本配置，描述了事务策略如何应用到方法上。

Spring使用AOP来支持声明式事务，并根据事务属性，自动在方法调用之前决定是否开启一个事务，并在方法执行之后决定事务提交或回滚事务。

### 传播行为

事务的第一个方面是传播行为（propagation behavior）。当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。

在项目开发中，我们一般将事务设置在service层。这样，当我们调用service层方法的时候，就能够保证在这个方法中对数据库的所有更新操作都在同一个事务中(事务是一系列指令的集合)。也就是方法中的对数据库的所有更新操作，一起执行，要么全部成功，要么全部失败，即保证数据的一致性。

如果在一个service方法中，除了调用对应dao层的方法外，还调用了其他service或当前service中的其他方法。此时，为了保证事务的一致性，我们必须保证调用的其他service方法和当前service方法在同一个事务中，那么这个时候事务应该如何控制呢？事实上，事务的传播特性就是用来解决这种问题的。

默认情况下当发生RuntimeException的情况下，事务才会回滚，所以要注意一下 如果你在程序发生错误的情况下，有自己的异常处理机制定义自己的Exception，必须从RuntimeException类继承 这样事务才会回滚！

Spring定义了七种传播行为，在开发中一般使用PROPAGATION\_REQUIRED作为传播行为的值。

#### PROPAGATION\_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。  比如：现在在service层定义方法A、B，A和B的事务传播行为都是PROPAGATION\_REQUIRED |
| *//事务属性 PROPAGATION\_REQUIRED* methodA{  ... } |
| 在方法B中调用A。  *//事务属性 PROPAGATION\_REQUIRED* methodB{  ...  methodA();  ... }  此时，在调用方法B之前，会先检测当前是否存在事务，如果存在事务，则将B加入已存在事务；如果不存在，则开启一个新的事务，将B加入新事务中。然后当方法B中调用方法A时，由于A的事务属性也是PROPAGATION\_REQUIRED，A也会先判断事务是否存在，因为此时B已有事务，所以会将A加入B的事务中。 |
|  |

#### PROPAGATION\_SUPPORTS

表示当前方法不需要事务上下文，但是如果当前存在事务的话，那么该方法会在这个事务中运行；如果不存在，则非事务运行。

也就是对当前方法来说，父级有没有事务它无所谓，你有我就用，没有就算了。不强求。

*//事务属性 PROPAGATION\_SUPPORTS*methodA{  
 ...  
}  
*//事务属性 PROPAGATION\_REQUIRED*methodB{  
 ...  
 methodA();  
 ...  
}

此时，方法A的传播行为是PROPAGATION\_SUPPORTS，当单独调用方法A时，如果不存在事务，方法A会非事务执行；在方法B中调用方法A时，由于B存在事务，方法A会被加入B的事务中，事务执行。

#### PROPAGATION\_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。

这就比较尴尬了，对方法A来说，调用它的地方必须有一个事务，否则就抛异常。类似于你想吃糖，要是别人不给你，你就啥也不管了，大哭大闹。

*//事务属性 PROPAGATION\_MANDATORY*

methodA{  
 ...  
}

#### PROPAGATION\_REQUIRED\_NEW

表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果当前存在事务，在该方法执行期间，当前事务会被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager

*//事务属性 PROPAGATION\_REQUIRED\_NEW*methodA{  
 ...  
}  
*//事务属性 PROPAGATION\_REQUIRED*methodB{  
 ...  
 methodA();  
 ...  
}

调用方法B时，方法B肯定会有一个事务，然后在方法B中调用方法A时，A的传播行为是PROPAGATION\_REQUIRED\_NEW，所以在执行方法A时，B的事务会被暂挂。

#### PROPAGATION\_NOT\_SUPPORTED

表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager。

#### PROPAGATION\_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

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。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务

### 隔离级别

事务的第二个维度就是隔离级别（isolation level）。隔离级别定义了一个事务可能受其他并发事务影响的程度。隔离级别越高，则数据库的并发性越差。

在项目的实际运行过程中，经常会发生多个事务并发执行，操作相同的数据来完成各自任务的情况。并发能够提高数据库的响应效率，但可能会导致以下的问题：

1. 丢失更新：两个事务同时更新一行数据，最后一个事务的更新会覆盖掉第一个事务的更新，从而导致第一个事务更新的数据丢失，这是由于没有加锁造成的；
2. 脏读：脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。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；
3. 不可重复读：在同一事务中，多次读取同一数据却返回不同的结果；也就是有其他事务更改了这些数据；
4. 幻读：一个事务在执行过程中读取到了另一个事务已提交的插入数据；即在第一个事务开始时读取到一批数据，但此后另一个事务又插入了新数据并提交，此时第一个事务又读取这批数据但发现多了一条，即好像发生幻觉一样。

不可重复读和幻读虽然都是两次查询的结果不一样，但侧重点有区别：

1、不可重复读的重点是修改，是多次读取同一条数据，在多次读取的时间间隔内，要读取的数据被修改，导致多次读取得到的数据内容不一致。

例如：

A原本的工资是1000，他听说自己要涨100块钱工资，于是就登陆公司的管理系统，查看自己的工资（开启事务1），结果发现还是1000（财务还没有修改），A觉得不对啊，不是说要涨工资吗？就在这时，财务将A的工资调整为1100（开启事务2），并且点击了确定（提交事务2），此时A又点了一下查询，这时候发现工资已经变成1100了，两次查询的工资不一样。这就是不可重复读。

对于不可重复读，只需要锁住满足条件的记录即可。

1. 幻读的重点在于新增或者删除满足了查询条件的数据。同样的条件, 第1次和第2次读出来的记录数不一样。

例如：

财务部门统计公司内工资在1000元以内的员工数，第一次统计结果发现有10个人满足条件。结果，在第一次统计之后，人事部又招聘了一个新员工，工资只有900，然后呢，财务为了确保统计结果的正确性，又统计了一次，这时候却发现满足条件的员工有11个了。

对于幻读，要锁住满足条件及其相近的记录

不可重复读读取的数据不一致针对的是同一条数据的内容；幻读读取的数据不一致针对的是读取的数据结果数量不一致。

#### ISOLATION\_DEFAULT

使用后端数据库默认的隔离级别

#### ISOLATION\_READ\_UNCOMMITTED

最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读

#### ISOLATION\_READ\_COMMITTED

允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生

#### ISOLATION\_REPEATABLE\_READ

对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生

#### ISOLATION\_SERIALIZABLE

最高的隔离级别，完全服从ACID的隔离级别，确保阻止脏读、不可重复读以及幻读，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的

### 回滚规则

这些规则定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚（这一行为与EJB的回滚行为是一致的）。

但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

### 事务超时

为了使应用程序很好地运行，事务不能运行太长的时间。因为事务可能涉及对后端数据库的锁定，所以长时间的事务会不必要的占用数据库资源。事务超时就是事务的一个定时器，在特定时间内事务如果没有执行完毕，那么就会自动回滚，而不是一直等待其结束。

### 是否只读

事务的第三个特性是它是否为只读事务。如果事务只对后端的数据库进行该操作，数据库可以利用事务的只读特性来进行一些特定的优化。通过将事务设置为只读，你就可以给数据库一个机会，让它应用它认为合适的优化措施。

## Spring提供的事务管理

Spring框架最核心功能之一就是事务管理，而且提供一致的事务管理抽象。Spring框架支持编程式事务和声明式事务。

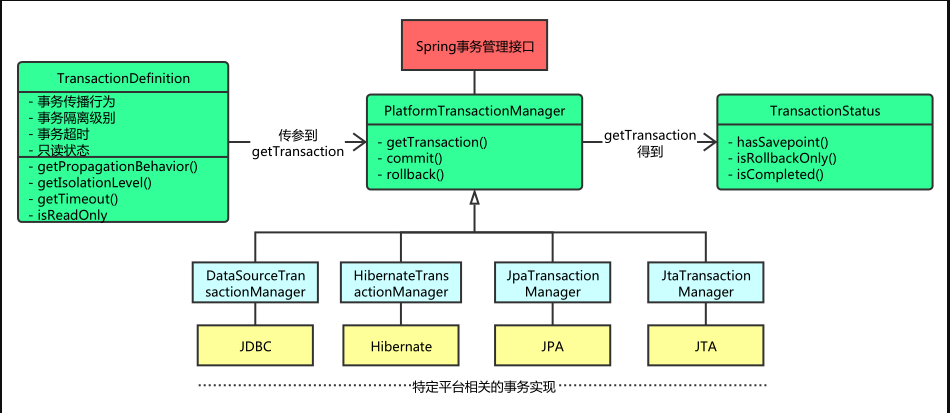
1、提供一致的编程式事务管理API，由应用程序服务器厂商提供具体实现，不管使用Spring JDBC框架还是集成第三方框架都使用该API进行事务编程；

2、无侵入式的声明式事务支持。

## 事务管理器

Spring中事务管理的实现依赖于许多核心接口，了解这些接口之间的结构关系，会对我们理解Spring事务提供很大的帮助。其中，最核心的三个接口分别是：PlatformTransactionManager(事务管理器)、TransactionDefinition(事务定义)、TransactionStatus(事务状态)。

Spring中事务管理相关的接口的关联关系：



### PlatformTransactionManager

Spring框架支持事务管理的核心是事务管理器抽象，即Spring并不直接管理事务，而是提供了一个抽象的事务管理器接口(PlatformTransactionManager)，不同的数据访问框架(如Hibernate)通过实现该抽象接口，从而实现各种数据访问框架的事务管理。

PlatformTransactionManager接口定义：

**public interface** PlatformTransactionManager {  
 TransactionStatus getTransaction(TransactionDefinition var1) **throws** TransactionException;  
 **void** commit(TransactionStatus var1) **throws** TransactionException;  
 **void** rollback(TransactionStatus var1) **throws** TransactionException;  
}

PlatformTransactionManager中提供了三个方法：

1. getTransaction()：返回一个已经激活的事务或创建一个新的事务（根据给定的TransactionDefinition类型参数定义的事务属性），返回的是TransactionStatus对象代表了当前事务的状态，其中该方法抛出TransactionException（未检查异常）表示事务由于某种原因失败。
2. commit()：用于提交TransactionStatus参数代表的事务。
3. rollback()：用于回滚TransactionStatus参数代表的事务。

### TransactionDefinition

**public interface** TransactionDefinition {  
 **int** getPropagationBehavior();  
 **int** getIsolationLevel();  
 **int** getTimeout();  
 **boolean** isReadOnly();

String getName();  
}

TransactionDefinition接口中提供了5个方法：

1. getPropagationBehavior()：返回定义的事务传播行为；
2. getIsolationLevel()：返回定义的事务隔离级别；
3. getTimeout()：返回定义的事务超时时间；
4. isReadOnly()：返回定义的事务是否是只读的；
5. getName()：返回定义的事务名字。

### TransactionStatus

**public interface** TransactionStatus **extends** SavepointManager, Flushable {  
 **boolean** isNewTransaction();  
 **boolean** hasSavepoint();  
 **void** setRollbackOnly();  
 **boolean** isRollbackOnly();  
 **void** flush();  
 **boolean** isCompleted();  
}

TransactionStatus提供了6个方法：

1. isNewTransaction()：返回当前事务状态是否是新事务；
2. hasSavepoint()：返回当前事务是否有保存点；
3. setRollbackOnly()：设置当前事务应该回滚；
4. isRollbackOnly()：返回当前事务是否应该回滚；
5. flush()：用于刷新底层会话中的修改到数据库，一般用于刷新如Hibernate/JPA的会话，可能对如JDBC类型的事务无任何影响；
6. isCompleted()：当前事务否已经完成。

具体的事务管理机制对Spring来说是透明的，它并不关心那些，那些是对应各个平台需要关心的，所以Spring事务管理的一个优点就是为不同的事务API提供一致的编程模型，如JTA、JDBC、Hibernate、JPA。下面分别介绍各个平台框架实现事务管理的机制。

### JDBC事务

如果应用程序中直接使用JDBC来进行持久化，DataSourceTransactionManager会为你处理事务边界。为了使用DataSourceTransactionManager，你需要使用如下的XML将其装配到应用程序的上下文定义中：

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

实际上，DataSourceTransactionManager是通过调用java.sql.Connection来管理事务，而后者是通过DataSource获取到的。通过调用连接的commit()方法来提交事务，同样，事务失败则通过调用rollback()方法进行回滚。

### Hibernate事务

如果应用程序的持久化是通过Hibernate实习的，那么你需要使用HibernateTransactionManager。对于Hibernate3，需要在Spring上下文定义中添加如下的<bean>声明：

<bean id=**"transactionManager" class**=**"org.springframework.orm.hibernate3.HibernateTransactionManager"**>  
 <property name=**"sessionFactory"** ref=**"sessionFactory"** />  
</bean>

sessionFactory属性需要装配一个Hibernate的session工厂，HibernateTransactionManager的实现细节是它将事务管理的职责委托给org.hibernate.Transaction对象，而后者是从Hibernate Session中获取到的。当事务成功完成时，HibernateTransactionManager将会调用Transaction对象的commit()方法，反之，将会调用rollback()方法。

### Java持久化API事务（JPA）

Hibernate多年来一直是事实上的Java持久化标准，但是现在Java持久化API作为真正的Java持久化标准进入大家的视野。如果你计划使用JPA的话，那你需要使用Spring的JpaTransactionManager来处理事务。你需要在Spring中这样配置JpaTransactionManager：

<bean id=**"transactionManager" class**=**"org.springframework.orm.jpa.JpaTransactionManager"**>  
 <property name=**"sessionFactory"** ref=**"sessionFactory"** />  
</bean>

JpaTransactionManager只需要装配一个JPA实体管理工厂（javax.persistence.EntityManagerFactory接口的任意实现）。JpaTransactionManager将与由工厂所产生的JPA EntityManager合作来构建事务。

### Java原生API事务

如果你没有使用以上所述的事务管理，或者是跨越了多个事务管理源（比如两个或者是多个不同的数据源），你就需要使用JtaTransactionManager：

<bean id=**"transactionManager" class**=**"org.springframework.transaction.jta.JtaTransactionManager"**>  
 <property name=**"transactionManagerName"** value=**"java:/TransactionManager"** />  
</bean>

JtaTransactionManager将事务管理的责任委托给javax.transaction.UserTransaction和javax.transaction.TransactionManager对象，其中事务成功完成通过UserTransaction.commit()方法提交，事务失败通过UserTransaction.rollback()方法回滚。

## 编程式事务和声明式事务的区别

Spring提供了对编程式事务和声明式事务的支持，编程式事务允许用户在代码中精确定义事务的边界，而声明式事务（基于AOP）有助于用户将操作与事务规则进行解耦。

简单地说，编程式事务侵入到了业务代码里面，但是提供了更加详细的事务管理；而声明式事务由于基于AOP，所以既能起到事务管理的作用，又可以不影响业务代码的具体实现。

## 编程式事务

Spring提供两种方式的编程式事务管理，分别是：使用TransactionTemplate和直接使用PlatformTransactionManager。

### 使用TransactionTemplate

采用TransactionTemplate和采用其他Spring模板，如JdbcTempalte和HibernateTemplate是一样的方法。它使用回调方法，把应用程序从处理取得和释放资源中解脱出来。如同其他模板，TransactionTemplate是线程安全的。代码片段：

TransactionTemplate **tt** = **new** TransactionTemplate(); *// 新建一个TransactionTemplate*Object **result** = **tt**.execute(  
 **new** TransactionCallback(){  
 **public** Object doTransaction(TransactionStatus status){  
 updateOperation();  
 **return** resultOfUpdateOperation();  
 }  
 }); *// 执行execute方法进行事务管理*

使用TransactionCallback()可以返回一个值。如果使用TransactionCallbackWithoutResult则没有返回值。

### 使用PlatformTransactionManager

示例代码如下：

DataSourceTransactionManager **dataSourceTransactionManager** = **new** DataSourceTransactionManager(); *//定义一个某个框架平台的TransactionManager，如JDBC、Hibernate*dataSourceTransactionManager.setDataSource(**this**.getJdbcTemplate().getDataSource()); *// 设置数据源* DefaultTransactionDefinition transDef = **new** DefaultTransactionDefinition(); *// 定义事务属性* transDef.setPropagationBehavior(DefaultTransactionDefinition.PROPAGATION\_REQUIRED); *// 设置传播行为属性* TransactionStatus status = dataSourceTransactionManager.getTransaction(transDef); *// 获得事务状态* **try** {  
 *// 数据库操作* dataSourceTransactionManager.commit(status);*// 提交* } **catch** (Exception e) {  
 dataSourceTransactionManager.rollback(status);*// 回滚* }

## 声明式事务

Spring配置文件中关于事务配置总是由三个组成部分，分别是DataSource、TransactionManager和代理机制这三部分，无论哪种配置方式，一般变化的只是代理机制这部分。

DataSource、TransactionManager这两部分只是会根据数据访问方式有所变化，比如使用Hibernate进行数据访问时，DataSource实际为SessionFactory，TransactionManager的实现为HibernateTransactionManager。

### 每个Bean都有一个代理

<?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"** xmlns:aop=**"http://www.springframework.org/schema/aop"** xsi:schemaLocation=**"http://www.springframework.org/schema/beans** http:*//www.springframework.org/schema/beans/spring-beans-2.5.xsd* http:*//www.springframework.org/schema/context* http:*//www.springframework.org/schema/context/spring-context-2.5.xsd* http:*//www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">* <bean id=**"sessionFactory"  
 class**=**"org.springframework.orm.hibernate3.LocalSessionFactoryBean"**>  
 <property name=**"configLocation"** value=**"classpath:hibernate.cfg.xml"** />  
 <property name=**"configurationClass"** value=**"org.hibernate.cfg.AnnotationConfiguration"** />  
 </bean>  
  
 <!-- 定义事务管理器（声明式的事务） -->  
 <bean id=**"transactionManager"  
 class**=**"org.springframework.orm.hibernate3.HibernateTransactionManager"**>  
 <property name=**"sessionFactory"** ref=**"sessionFactory"** />  
 </bean>  
   
 <!-- 配置DAO -->  
 <bean id=**"userDaoTarget" class**=**"com.bluesky.spring.dao.UserDaoImpl"**>  
 <property name=**"sessionFactory"** ref=**"sessionFactory"** />  
 </bean>  
   
 <bean id=**"userDao"  
 class**=**"org.springframework.transaction.interceptor.TransactionProxyFactoryBean"**>  
 <!-- 配置事务管理器 -->  
 <property name=**"transactionManager"** ref=**"transactionManager"** />  
 <property name=**"target"** ref=**"userDaoTarget"** />  
 <property name=**"proxyInterfaces"** value=**"com.bluesky.spring.dao.GeneratorDao"** />  
 <!-- 配置事务属性 -->  
 <property name=**"transactionAttributes"**>  
 <props>  
 <prop key=**"\*"**>PROPAGATION\_REQUIRED</prop>  
 </props>  
 </property>  
 </bean>  
</beans>

### 所有Bean共享一个代理基类

<?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"** xmlns:aop=**"http://www.springframework.org/schema/aop"** xsi:schemaLocation=**"http://www.springframework.org/schema/beans** http:*//www.springframework.org/schema/beans/spring-beans-2.5.xsd* http:*//www.springframework.org/schema/context* http:*//www.springframework.org/schema/context/spring-context-2.5.xsd* http:*//www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">* <bean id=**"sessionFactory"  
 class**=**"org.springframework.orm.hibernate3.LocalSessionFactoryBean"**>  
 <property name=**"configLocation"** value=**"classpath:hibernate.cfg.xml"** />  
 <property name=**"configurationClass"** value=**"org.hibernate.cfg.AnnotationConfiguration"** />  
 </bean>  
   
 <!-- 定义事务管理器（声明式的事务） -->  
 <bean id=**"transactionManager"  
 class**=**"org.springframework.orm.hibernate3.HibernateTransactionManager"**>  
 <property name=**"sessionFactory"** ref=**"sessionFactory"** />  
 </bean>  
   
 <bean id=**"transactionBase"  
 class**=**"org.springframework.transaction.interceptor.TransactionProxyFactoryBean"** lazy-init=**"true" abstract**=**"true"**>  
 <!-- 配置事务管理器 -->  
 <property name=**"transactionManager"** ref=**"transactionManager"** />  
 <!-- 配置事务属性 -->  
 <property name=**"transactionAttributes"**>  
 <props>  
 <prop key=**"\*"**>PROPAGATION\_REQUIRED</prop>  
 </props>  
 </property>  
 </bean>  
   
 <!-- 配置DAO -->  
 <bean id=**"userDaoTarget" class**=**"com.bluesky.spring.dao.UserDaoImpl"**>  
 <property name=**"sessionFactory"** ref=**"sessionFactory"** />  
 </bean>  
   
 <bean id=**"userDao"** parent=**"transactionBase"** >  
 <property name=**"target"** ref=**"userDaoTarget"** />  
 </bean>  
</beans>

### 使用拦截器

<?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:context=**"http://www.springframework.org/schema/context"** xmlns:aop=**"http://www.springframework.org/schema/aop"** xsi:schemaLocation=**"http://www.springframework.org/schema/beans** http:*//www.springframework.org/schema/beans/spring-beans-2.5.xsd* http:*//www.springframework.org/schema/context* http:*//www.springframework.org/schema/context/spring-context-2.5.xsd* http:*//www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">* <bean id=**"sessionFactory"  
 class**=**"org.springframework.orm.hibernate3.LocalSessionFactoryBean"**>  
 <property name=**"configLocation"** value=**"classpath:hibernate.cfg.xml"** />  
 <property name=**"configurationClass"** value=**"org.hibernate.cfg.AnnotationConfiguration"** />  
 </bean>  
   
 <!-- 定义事务管理器（声明式的事务） -->  
 <bean id=**"transactionManager"  
 class**=**"org.springframework.orm.hibernate3.HibernateTransactionManager"**>  
 <property name=**"sessionFactory"** ref=**"sessionFactory"** />  
 </bean>  
   
 <bean id=**"transactionInterceptor"  
 class**=**"org.springframework.transaction.interceptor.TransactionInterceptor"**>  
 <property name=**"transactionManager"** ref=**"transactionManager"** />  
 <!-- 配置事务属性 -->  
 <property name=**"transactionAttributes"**>  
 <props>  
 <prop key=**"\*"**>PROPAGATION\_REQUIRED</prop>  
 </props>  
 </property>  
 </bean>  
   
 <bean **class**=**"org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator"**>  
 <property name=**"beanNames"**>  
 <list><value>\*Dao</value></list>  
 </property>  
 <property name=**"interceptorNames"**>  
 <list><value>transactionInterceptor</value></list>  
 </property>  
 </bean>  
   
 <!-- 配置DAO -->  
 <bean id=**"userDao" class**=**"com.bluesky.spring.dao.UserDaoImpl"**>  
 <property name=**"sessionFactory"** ref=**"sessionFactory"** />  
 </bean>  
</beans>

### 使用tx标签配置的拦截器

<?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:context=**"http://www.springframework.org/schema/context"** xmlns:aop=**"http://www.springframework.org/schema/aop"** xmlns:tx=**"http://www.springframework.org/schema/tx"** xsi:schemaLocation=**"http://www.springframework.org/schema/beans** http:*//www.springframework.org/schema/beans/spring-beans-2.5.xsd* http:*//www.springframework.org/schema/context* http:*//www.springframework.org/schema/context/spring-context-2.5.xsd* http:*//www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd* http:*//www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">* <context:annotation-config />  
 <context:component-scan base-**package**=**"com.bluesky"** />  
   
 <bean id=**"sessionFactory"  
 class**=**"org.springframework.orm.hibernate3.LocalSessionFactoryBean"**>  
 <property name=**"configLocation"** value=**"classpath:hibernate.cfg.xml"** />  
 <property name=**"configurationClass"** value=**"org.hibernate.cfg.AnnotationConfiguration"** />  
 </bean>  
   
 <!-- 定义事务管理器（声明式的事务） -->  
 <bean id=**"transactionManager"  
 class**=**"org.springframework.orm.hibernate3.HibernateTransactionManager"**>  
 <property name=**"sessionFactory"** ref=**"sessionFactory"** />  
 </bean>  
   
 <tx:advice id=**"txAdvice"** transaction-manager=**"transactionManager"**>  
 <tx:attributes>  
 <tx:method name=**"\*"** propagation=**"REQUIRED"** />  
 </tx:attributes>  
 </tx:advice>  
   
 <aop:config>  
 <aop:pointcut id=**"interceptorPointCuts"** expression=**"execution(\* com.bluesky.spring.dao.\*.\*(..))"** />  
 <aop:advisor advice-ref=**"txAdvice"** pointcut-ref=**"interceptorPointCuts"** />  
 </aop:config>  
</beans>

### 全注解

<?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"** xmlns:aop=**"http://www.springframework.org/schema/aop"** xmlns:tx=**"http://www.springframework.org/schema/tx"** xsi:schemaLocation=**"http://www.springframework.org/schema/beans** http:*//www.springframework.org/schema/beans/spring-beans-2.5.xsd* http:*//www.springframework.org/schema/context* http:*//www.springframework.org/schema/context/spring-context-2.5.xsd* http:*//www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd* http:*//www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">* <context:annotation-config />  
 <context:component-scan base-**package**=**"com.bluesky"** />  
   
 <tx:annotation-driven transaction-manager=**"transactionManager"**/>  
   
 <bean id=**"sessionFactory"  
 class**=**"org.springframework.orm.hibernate3.LocalSessionFactoryBean"**>  
 <property name=**"configLocation"** value=**"classpath:hibernate.cfg.xml"** />  
 <property name=**"configurationClass"** value=**"org.hibernate.cfg.AnnotationConfiguration"** />  
 </bean>  
   
 <!-- 定义事务管理器（声明式的事务） -->  
 <bean id=**"transactionManager"  
 class**=**"org.springframework.orm.hibernate3.HibernateTransactionManager"**>  
 <property name=**"sessionFactory"** ref=**"sessionFactory"** />  
 </bean>  
  
</beans>

# AOP

## AOP理解

AOP，面向切面编程，是面向对象编程(OOP)的一种有益补充。

通俗的理解，就是在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。

我们知道，面向对象的特点是继承、多态和封装。而封装就要求将功能分散到不同的对象中去，这在软件设计中往往称为职责分配。实际上也就是说，让不同的类设计不同的方法。这样代码就分散到一个个的类中去了。这样做的好处是降低了代码的复杂程度，使类可重用。

  但是人们也发现，在分散代码的同时，也增加了代码的重复性。什么意思呢？比如说，我们在两个类中，可能都需要在每个方法中做日志（为了低耦合，保持类的独立性，在每个类中都写上日志方法）。按面向对象的设计方法，我们就必须在两个类的方法中都加入日志的内容。也许他们是完全相同的，但就是因为面向对象的设计让类与类之间无法联系，而不能将这些重复的代码统一起来。

 也许有人会说，那好办啊，我们可以将这段代码写在一个独立的类独立的方法里，然后再在这两个类中调用。但是，这样一来，这两个类跟我们上面提到的独立的类就有耦合了，它的改变会影响这两个类。那么，有没有什么办法，能让我们在需要的时候，随意地加入代码呢？**这种在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。**

  一般而言，我们管切入到指定类指定方法的代码片段称为切面，而切入到哪些类、哪些方法则叫切入点。有了AOP，我们就可以把几个类共有的代码，抽取到一个切片中，等到需要时再切入对象中去，从而改变其原有的行为。

## AOP应用

1. 降低耦合；在类中引用通用功能类，会导致类之间耦合，AOP可以在指定位置织入切面，类之间没有引用关系，降低耦合。
2. 减少重复代码；AOP可以把通用的功能分离出来，然后在需要使用通用功能的地方使用切面切入。
3. 功能扩展；AOP的通知本身也被译为增强，可以在方法前后加入代码增强原方法功能，起到扩展方法作用。

## AOP术语

1. 通知（Advice）：在连接点上执行的行为，就是你想要的功能，比如安全，事物，日志等。你需要先定义好，然后在想用的地方用一下。
2. 连接点（JoinPoint）：就是spring允许你使用通知的地方，也就是你在哪个地方想使用日志。基本每个方法的前，后（两者都有也行），或抛出异常时都可以是连接点，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，不过那不是咱关注的，只要记住，和方法有关的前前后后（抛出异常），都是连接点。
3. 切入点（PointCut）：切入点是在连接点的基础上定义的。比如一个类有十几个方法，，那就有几十个连接点了对把，但是你并不想在所有方法附近都使用通知（使用叫织入，以后再说），你只想让其中的几个，在调用这几个方法之前，之后或者抛出异常时干点什么。然后你可以将这几个连接点通过配置或注解定义为切入点。
4. 切面（Aspect）：切面是横切关注点的模块化，可以认为是通知和切入点的结合。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before,after，around等就能知道），而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。
5. 引入（introduction）：允许我们向现有的类添加新方法属性。也就是把切面（也就是新方法属性：通知定义的）用到目标类中。
6. 目标（target）：引入中所提到的目标类，也就是要被切面切入的类，也就是真正的业务逻辑，他可以在毫不知情的情况下，被咱们织入切面。而自己专注于业务本身的逻辑。
7. 织入（weaving）：织入是一个过程，是将切面应用到目标对象从而创建出AOP代理对象的过程。织入可以在编译期、类装载期、运行期进行，spring采用的是运行时。
8. 代理（proxy）：AOP框架使用代理模式创建的对象，从而实现在连接点处插入通知（即应用切面），就是通过代理来对目标对象应用切面。在Spring中，AOP代理可以用JDK动态代理或CGLIB代理实现，而通过拦截器模型应用切面。

## 通知类型

1. 前置通知：在目标类中切入点关联的方法执行之前执行
2. 后置通知：在目标类中切入点关联的方法执行之后执行
3. 后置返回通知：在目标类中切入点关联的方法执行之后执行，且要求切入点关联的方法正常执行
4. 后置异常通知：在目标类中切入点关联的方法抛出异常时触发，执行通知的代码
5. 后置最终通知：无论目标类中切入点关联的方法是抛出异常还是正常执行，都会执行。类似于Java中的finally块语句。
6. 环绕通知：在目标类中切入点关联的方法执行之前和执行之后触发，环绕方法首尾

## AOP代理机制

spring用代理类包裹切面，把他们织入到Spring管理的bean中。也就是说代理类伪装成目标类，它会截取对目标类中方法的调用，让调用者对目标类的调用都先变成调用伪装类，伪装类中就先执行了切面，再把调用转发给真正的目标bean。

现在可以自己想一想，怎么搞出来这个伪装类，才不会被调用者发现（过JVM的检查，JAVA是强类型检查，哪里都要检查类型）。

　　1、实现和目标类相同的接口，我也实现和你一样的接口，反正上层都是接口级别的调用，这样我就伪装成了和目标类一样的类（实现了同一接口，咱是兄弟了），也就逃过了类型检查，到java运行期的时候，利用多态的后期绑定（所以spring采用运行时），伪装类（代理类）就变成了接口的真正实现，而他里面包裹了真实的那个目标类，最后实现具体功能的还是目标类，只不过伪装类在之前干了点事情（写日志，安全检查，事物等）。

　　这就好比，一个人让你办件事，每次这个时候，你弟弟就会先出来，当然他分不出来了，以为是你，你这个弟弟虽然办不了这事，但是他知道你能办，所以就答应下来了，并且收了点礼物（写日志），收完礼物了，给把事给人家办了啊，所以你弟弟又找你这个哥哥来了，最后把这是办了的还是你自己。但是你自己并不知道你弟弟已经收礼物了，你只是专心把这件事情做好。

　　顺着这个思路想，要是本身这个类就没实现一个接口呢，你怎么伪装我，我就压根没有机会让你搞出这个双胞胎的弟弟，那么就用第2种代理方式，创建一个目标类的子类，生个儿子，让儿子伪装我

　　2、生成子类调用，这次用子类来做为伪装类，当然这样也能逃过JVM的强类型检查，我继承的吗，当然查不出来了，子类重写了目标类的所有方法，当然在这些重写的方法中，不仅实现了目标类的功能，还在这些功能之前，实现了一些其他的（写日志，安全检查，事物等）。

这次的对比就是，儿子先从爸爸那把本事都学会了，所有人都找儿子办事情，但是儿子每次办和爸爸同样的事之前，都要收点小礼物（写日志），然后才去办真正的事。当然爸爸是不知道儿子这么干的了。这里就有件事情要说，某些本事是爸爸独有的(final的)，儿子学不了，学不了就办不了这件事，办不了这个事情，自然就不能收人家礼了。

前一种兄弟模式，spring会使用JDK的java.lang.reflect.Proxy类，它允许Spring动态生成一个新类来实现必要的接口，织入通知，并且把对这些接口的任何调用都转发到目标类。

　　后一种父子模式，spring使用CGLIB库生成目标类的一个子类，在创建这个子类的时候，spring织入通知，并且把对这个子类的调用委托到目标类。

　　相比之下，还是兄弟模式好些，他能更好的实现松耦合，尤其在今天都高喊着面向接口编程的情况下，父子模式只是在没有实现接口的时候，也能织入通知，应当做一种例外。

## 基于配置文件的AOP

Spring在2.0之后的版本中，提供了在XML配置文件中使用<aop:config>标签定义切面、切入点和通知相关内容的支持。通过配置，我们可以对目标类中切入点关联的方法进行增强。

在Spring配置文件中，所有与AOP相关的定义必须放在<aop:config>标签下，该标签下可以有<aop:pointcut>、<aop:advisor>、<aop:aspect>标签，且这几个标签的配置顺序不可变。

<aop:pointcut>：用来定义切入点，该切入点可以重用；

<aop:advisor>：用来定义只有一个通知和一个切入点的切面；

<aop:aspect>：用来定义切面，该切面可以包含多个切入点和通知，而且标签内部的通知和切入点定义是无序的；和advisor的区别就在此，advisor只包含一个通知和一个切入点。

advisor除了在进行事务控制的情况下，其他情况下一般不建议使用。该方式属于侵入式设计，必须实现通知API

### 切入点表达式

在声明切入点时，需要通过expression表达式匹配指定类中的指定方法作为切人点。

<**aop:pointcut id="mycut" expression="execution(\* \*.\*(..))"**></**aop:pointcut**>

<**aop:pointcut id="mycut" expression="execution(\* com.wei.test.AOPTest.\*.\*(..))"**></**aop:pointcut**>

execution(\* \*.\*(..))

1. execution 声明表达式，不能忽略
2. \* 第一个\*用来匹配任意返回值类型（即切入点的方法返回值为任意类型）
3. 第一个\*后面要有一个空格，否则表达式匹配不到切入点
4. 第二个\*匹配任意包中的任意类
5. 第三个\*表示匹配类中的所有方法
6. (..)表示方法中可以存在任意个参数

以上\*可以用具体的包名、类名、方法名代替

### 切入点声明方式

切入点一般是Java类中的某个方法，在配置文件中有以下3中声明方式：

1. 在<aop:config>标签下使用<aop:pointcut>声明一个切入点。

该切入点可以被多个切面使用，对于需要共享使用的切入点最好使用该方式。该切入点中：

id属性指定切入点所在目标类的名字（应当具有唯一性，在定义通知时通过该ID引用该切入点）；

expression属性指定切入点表达式（expression表达式匹配指定类中的指定方法，也就是决定哪些方法被定义为切入点）

<**aop:config**>  
 <**aop:pointcut id="pointcut" expression="execution(\* cn.javass..\*.\*(..))"**/>  
 <**aop:aspect ref="aspectSupportBean"**>  
 <**aop:before pointcut-ref="pointcut" method="before"**/>  
 </**aop:aspect**>  
</**aop:config**>

1. 在<aop:aspect>标签下使用<aop:pointcut>声明一个切入点。

该切入点也可以被多个切面引用，但一般只用于当前<aop:aspect>标签定义的切面。如果想多个切面共享，应该使用方式1中的切入点声明方式。

<**aop:config**>  
 <**aop:aspect ref="aspectSupportBean"**>  
 <**aop:pointcut id=" pointcut" expression="execution(\* cn.javass..\*.\*(..))"**/>  
 <**aop:before pointcut-ref="pointcut" method="before"**/>  
 </**aop:aspect**>  
</**aop:config**>

1. 在声明通知时通过pointcut属性指定切入点表达式声明一个匿名切入点。

该切入点是匿名切入点，只被声明处的通知中使用。

<**aop:config**>  
 <**aop:aspect ref="aspectSupportBean"**>  
 <**aop:after pointcut="execution(\* cn.javass..\*.\*(..))" method="afterFinallyAdvice"**/>  
 </**aop:aspect**>  
</**aop:config**>

### 前置通知

在切入点关联的方法之前执行，通过<aop:aspect>标签下的<aop:before>标签声明

<**aop:before pointcut="切入点表达式" pointcut-ref="切入点Bean引用"  
 method="前置通知实现方法名"  
 arg-names="前置通知实现方法参数列表参数名字"**/>

pointcut和pointcut-ref：二者选一，指定切入点（pointcut-ref通过已定义切入点的ID引用）

method：指定前置通知关联的方法名，如果是多态需要加上参数类型，多个用“，”隔开，如beforeAdvice(java.lang.String)；

arg-names：指定通知关联方法的参数名字，多个用“，”分隔，可选，类似于构造器注入中的参数名注入限制：在class文件中没生成变量调试信息是获取不到方法参数名字的，因此只有在类没生成变量调试信息时才需要使用arg-names属性来指定参数名，如arg-names="param"表示通知实现方法的参数列表的第一个参数名字为“param”。

<**bean id="targetClass" class="com.wei.test.AOPTest.TargetClass"**></**bean**>  
<**bean id="aspect" class="com.wei.test.AOPTest.HelloWorldAspect"**></**bean**>  
  
<**aop:config**>  
 <**aop:pointcut id="mycut" expression="execution(\* \*.\*(..)) and args(param)"**></**aop:pointcut**>  
 <**aop:aspect ref="aspect"**>  
 <**aop:before method="beforeAdvisor" pointcut-ref="mycut" arg-names="param"**></**aop:before**>  
 </**aop:aspect**>  
</**aop:config**>

1. pointcut表达式后跟的参数名和arg-names后跟的参数名要保持一致；
2. 如果通知方法中有参数，在声明通知时必须在pointcut表达式和arg-names标签中配置参数；
3. 如果目标类中切入点方法中没有参数，而配置文件中声明的切入点中execution表达式带参数时，即使前半部分匹配，通知也不会被调用。可以认为此时execution表达式增加了一个条件，就是切入点关联的方法中必须带有参数；
4. 无论几个参数，只要满足1的条件，切入点方法和通知方法中参数名可以不一致（最好保持一致）
5. 多态？？

### 后置返回通知

在切入点选择的方法正常返回时执行，通过<aop:aspect>标签下的<aop:after-returning>标签声明

<aop:after-returning pointcut=**"切入点表达式"** pointcut-ref=**"切入点Bean引用"** method=**"后置返回通知实现方法名"** arg-names=**"后置返回通知实现方法参数列表参数名字"** returning=**"返回值对应的后置返回通知实现方法参数名"** />

returning：定义一个名字，该名字用于匹配通知实现方法的一个参数名，当目标方法执行正常返回后，将把目标方法返回值传给通知方法；returning限定了只有目标方法返回值匹配与通知方法相应参数类型时才能执行后置返回通知，否则不执行，对于returning对应的通知方法参数为Object类型将匹配任何目标返回值。

### 后置异常通知

在切入点选择的方法抛出异常时执行，通过<aop:aspect>标签下的<aop:after-throwing>标签声明

<aop:after-throwing pointcut=**"切入点表达式"** pointcut-ref=**"切入点Bean引用"** method=**"后置异常通知实现方法名"** arg-names=**"后置异常通知实现方法参数列表参数名字"** throwing=**"将抛出的异常赋值给的通知实现方法参数名"**/>

throwing：定义一个名字，该名字用于匹配通知实现方法的一个参数名，当目标方法抛出异常返回后，将把目标方法抛出的异常传给通知方法；throwing限定了只有目标方法抛出的异常匹配与通知方法相应参数异常类型时才能执行后置异常通知，否则不执行，对于throwing对应的通知方法参数为Throwable类型将匹配任何异常。

### 后置最终通知

在切入点选择的方法返回时执行，不管是正常返回还是抛出异常都执行，通过<aop:aspect>标签下的<aop:after >标签声明

<aop:after pointcut=**"切入点表达式"** pointcut-ref=**"切入点Bean引用"** method=**"后置最终通知实现方法名"** arg-names=**"后置最终通知实现方法参数列表参数名字"**/>

### 环绕通知

环绕着在切入点选择的连接点处的方法所执行的通知，环绕通知非常强大，可以决定目标方法是否执行，什么时候执行，执行时是否需要替换方法参数，执行完毕是否需要替换返回值，可通过<aop:aspect>标签下的<aop:around >标签声明

<aop:around pointcut=**"切入点表达式"** pointcut-ref=**"切入点Bean引用"** method=**"后置最终通知实现方法名"** arg-names=**"后置最终通知实现方法参数列表参数名字"**/>

环绕通知第一个参数必须是org.aspectj.lang.ProceedingJoinPoint类型，在通知实现方法内部使用ProceedingJoinPoint的proceed()方法使目标方法执行，proceed 方法可以传入可选的Object[]数组，该数组的值将被作为目标方法执行时的参数。

### 引入

Spring引入允许为目标对象引入新的接口，通过在< aop:aspect>标签内使用< aop:declare-parents>标签进行引入，定义方式如下

<aop:declare-parents  
 types-matching=**"AspectJ语法类型表达式"** implement-**interface**=引入的接口**"   
 default**-impl=**"引入接口的默认实现"** delegate-ref=**"引入接口的默认实现Bean引用"**/>

types-matching：匹配需要引入接口的目标对象的AspectJ语法类型表达式；

implement-interface：定义需要引入的接口；

default-impl和delegate-ref：定义引入接口的默认实现，二者选一，default-impl是接口的默认实现类全限定名，而delegate-ref是默认的实现的委托Bean名；

## AspectJ实现AOP

Spring除了支持Schema方式配置AOP，还支持注解方式：使用@AspectJ风格的切面声明。

### 启动AspectJ注解支持

Spring默认情况下不支持@AspectJ风格的切面声明。如果要使用@AspectJ注解声明切面，需要在Spring的配置文件中启动注解支持：

<aop:aspectj-autoproxy/>

添加如上配置后，Spring才能够映射@AspectJ注解并将切面应用到目标对象。

使用@AspectJ风格的注解声明切面，可以大大减少AOP相关的配置。

### 声明切面

@AspectJ风格声明切面非常简单。如果我们想将一个类声明为切面，直接在这个类上加一个@Aspect注解即可。但要注意，要确保声明为切面的这个类能够被Spring容器加载。

@Aspect()   
Public **class** Aspect{   
……  
}

### 声明切入点

@AspectJ风格声明切入点时，只需要在方法上加@Pointcut注解即可。另外，使用@Pointcut注解修饰的方法返回类型必须是void。

@Pointcut(value="切入点表达式", argNames = "参数名列表")  
public void pointcutName(……) {}

value：指定切入点表达式；

argNames：指定命名切入点方法参数列表参数名字，可以有多个用“，”分隔，这些参数将传递给通知方法同名的参数，同时比如切入点表达式“args(param)”将匹配参数类型为命名切入点方法同名参数指定的参数类型。

pointcutName：切入点名字，可以使用该名字进行引用该切入点表达式。

### 声明通知

@AspectJ和通过XML配置文件声明AOP一样，也支持5种通知类型。

#### 前置通知

前置通知通过@Before注解声明

@Before(value = "切入点表达式或命名切入点", argNames = "参数列表参数名")

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义

#### 后置返回通知

后置返回通知通过@AfterReturning注解声明

@AfterReturning(  
 value="切入点表达式或命名切入点",  
 pointcut="切入点表达式或命名切入点",  
 argNames="参数列表参数名",  
 returning="返回值对应参数名")

value：指定切入点表达式或命名切入点；

pointcut：同样是指定切入点表达式或命名切入点，如果指定了将覆盖value属性指定的，pointcut具有高优先级；

argNames：与Schema方式配置中的同义；

returning：与Schema方式配置中的同义。

#### 后置异常通知

@AfterThrowing注解声明

@AfterThrowing (  
 value="切入点表达式或命名切入点",  
 pointcut="切入点表达式或命名切入点",  
 argNames="参数列表参数名",  
 throwing="异常对应参数名")

value：指定切入点表达式或命名切入点；

pointcut：同样是指定切入点表达式或命名切入点，如果指定了将覆盖value属性指定的，pointcut具有高优先级；

argNames：与Schema方式配置中的同义；

throwing：与Schema方式配置中的同义

#### 后置最终通知

@After注解声明

@After (  
 value="切入点表达式或命名切入点",  
 argNames="参数列表参数名")

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义；

#### 环绕通知

@Around注解声明

@Around (  
 value="切入点表达式或命名切入点",  
 argNames="参数列表参数名")

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义；

### 声明引入

@DeclareParents声明

@DeclareParents(value=" AspectJ语法类型表达式",

defaultImpl=引入接口的默认实现类)  
private Interface interface;

value：匹配需要引入接口的目标对象的AspectJ语法类型表达式；与Schema方式中的types-matching属性同义；

private Interface interface：指定需要引入的接口；

defaultImpl：指定引入接口的默认实现类，没有与Schema方式中的delegate-ref属性同义的定义方式；

## 通知参数

通知可以对切入点关联的方法进行增强，方法中基本都是有参数的。如何获取方法参数，并将这些参数传给通知进行处理是非常重要的一点。

接下来介绍两种AOP获取切入点关联方法参数的方式：

1. Spring AOP提供使用org.aspectj.lang.JoinPoint类型获取连接点数据，任何通知方法的第一个参数都可以是JoinPoint(环绕通知是ProceedingJoinPoint，JoinPoint子类)，当然第一个参数位置也可以是JoinPoint.StaticPart类型，这个只返回连接点的静态部分。

JoinPoint：提供访问当前被通知方法的目标对象、代理对象、方法参数等数据：

public interface JoinPoint {  
 String toString(); //连接点所在位置的相关信息  
 String toShortString(); //连接点所在位置的简短相关信息  
 String toLongString(); //连接点所在位置的全部相关信息  
 Object getThis(); //返回AOP代理对象  
 Object getTarget(); //返回目标对象  
 Object[] getArgs(); //返回被通知方法参数列表  
 Signature getSignature(); //返回当前连接点签名  
 SourceLocation getSourceLocation();//返回连接点方法所在类文件中的位置  
 String getKind(); //连接点类型  
 StaticPart getStaticPart(); //返回连接点静态部分  
}

ProceedingJoinPoint：用于环绕通知，使用proceed()方法来执行目标方法：

public interface ProceedingJoinPoint extends JoinPoint {  
 public Object proceed() throws Throwable;  
 public Object proceed(Object[] args) throws Throwable;  
}

JoinPoint.StaticPart：提供访问连接点的静态部分，如被通知方法签名、连接点类型等：

public interface StaticPart {  
 Signature getSignature(); //返回当前连接点签名  
 String getKind(); //连接点类型  
 int getId(); //唯一标识  
 String toString(); //连接点所在位置的相关信息  
 String toShortString(); //连接点所在位置的简短相关信息  
 String toLongString(); //连接点所在位置的全部相关信息  
}

1. 自动获取：通过切入点表达式可以将相应的参数自动传递给通知方法，例如前边章节讲过的返回值和异常是如何传递给通知方法的。

在Spring AOP中，除了execution和bean指示符不能传递参数给通知方法，其他指示符都可以将匹配的相应参数或对象自动传递给通知方法。

2.1、如果我们通过“argNames”属性指定了参数名

2.2、如果第一个参数类型是JoinPoint、ProceedingJoinPoint或JoinPoint.StaticPart类型，应该从“argNames”属性省略掉该参数名（可选，写上也对），这些类型对象会自动传入的，但必须作为第一个参数；

2.3、如果“class文件中含有变量调试信息”将使用这些方法签名中的参数名来确定参数名；

2.4、如果没有“class文件中含有变量调试信息”，将尝试自己的参数匹配算法，如果发现参数绑定有二义性将抛出AmbiguousBindingException异常；对于只有一个绑定变量的切入点表达式，而通知方法只接受一个参数，说明绑定参数是明确的，从而能配对成功。

除了上边介绍的普通方式，也可以对使用命名切入点自动获取参数

## 通知顺序

如果我们有多个通知想要在同一连接点执行，那执行顺序如何确定呢？Spring AOP使用AspectJ的优先级规则来确定通知执行顺序。总共有两种情况：同一切面中通知执行顺序、不同切面中的通知执行顺序。

### 同一切面中通知执行顺序

### 不同切面中的通知执行顺序

当定义在不同切面的相同类型的通知需要在同一个连接点执行，如果没指定切面的执行顺序，这两个通知的执行顺序将是未知的。

如果需要他们顺序执行，可以通过指定切面的优先级来控制通知的执行顺序。

Spring中可以通过在切面实现类上实现org.springframework.core.Ordered接口或使用Order注解来指定切面优先级。在多个切面中，Ordered.getValue()方法返回值（或者注解值）较小值的那个切面拥有较高优先级

## 切面实例化模型

所谓切面实例化模型指何时实例化切面。

Spring AOP支持AspectJ的singleton、perthis、pertarget实例化模型（目前不支持percflow、percflowbelow 和pertypewithin）。