# Spring

## 第一章 Spring 概述

### 1.1. 什么是spring

spring就是一个java框架，使用java语言开发的， 轻量级的， 开源的框架。 可以在j2se、j2ee项目中都可以使用。

spring核心技术： ioc， aop

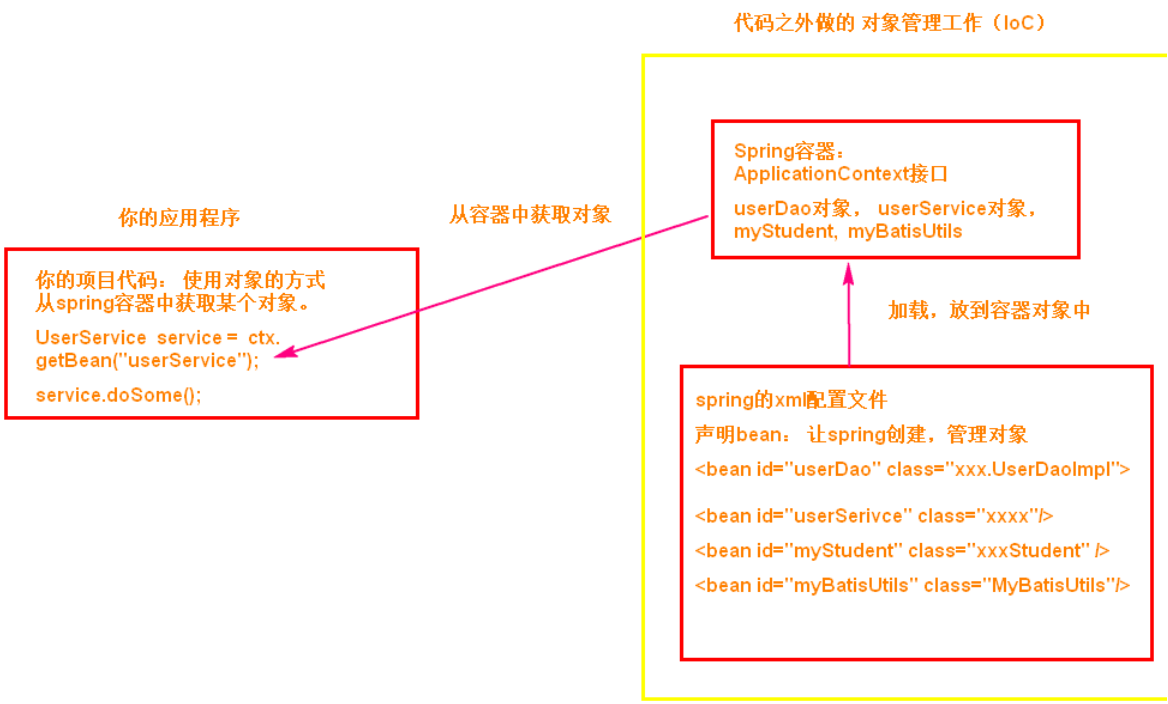
spring又叫做：容器， spring作为容器， 装的是java对象。 可以让spring创建java对象， 给属性赋值。

spring作用： 实现解耦合， 解决java对象之间的耦合， 解决模块之间的耦合。

tomcat也是容器：管理的是servlet， listener， filter等对象。

创建HelloServlet 类， 写web.xml

spring：创建SomeServiceImpl，写spring的配置文件




### 1.2 spring的地址

https://spring.io

## 1.3 Spring优点

Spring 是一个框架，是一个半成品的软件。有 20 个模块组成。它是一个容器管理对象，容器是装东西的，Spring 容器不装文本，数字。装的是对象。Spring 是存储对象的容器。

（1） 轻量

Spring 框架使用的 jar 都比较小，一般在 1M 以下或者几百 kb。Spring核心功能的所需的 jar 总共在 3M 左右。

Spring 框架运行占用的资源少，运行效率高。不依赖其他 jar

（2） 针对接口编程，解耦合

Spring 提供了 Ioc 控制反转，由容器管理对象，对象的依赖关系。原来在程序代码中的对象创建方式，现在由容器完成。对象之间的依赖解耦合。

（3） AOP 编程的支持

通过 Spring 提供的 AOP 功能，方便进行面向切面的编程，许多不容易用传统 OOP 实现的功能可以通过 AOP 轻松应付在 Spring 中，开发人员可以从繁杂的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。

（4） 方便集成各种优秀框架

Spring 不排斥各种优秀的开源框架，相反 Spring 可以降低各种框架的使用难度，Spring 提供了对各种优秀框架（如 Struts,Hibernate、MyBatis）等的直接支持。简化框架的使用。Spring 像插线板一样，其他框架是插头，可以容易的组合到一起。需要使用哪个框架，就把这个插头放入插线板。不需要可以轻易的移除。

# 第二章 IoC 控制反转

## 2.1 IoC 概念

IoC，Inversion of Control： 控制反转， 是一个理论，一个指导思想。 指导开发人员如何使用对象，管理对象的。 把对象的创建，属性赋值，对象的声明周期都交给代码之外的容器管理。

1. IoC分为 控制和反转

控制： 对象创建，属性赋值， 对象声明周期管理

反转：把开发人员管理对象的权限转移给了代码之外的容器实现。 由容器完成对象的管理。

正转：开发人员在代码中， 使用 new 构造方法创建对象。 开发人员掌握了对象的创建，属性赋值，对象从开始到销毁的全部过程。 开发人员有对 对象 全部控制。

通过容器，可以使用容器中的对象（容器已经创建了对象， 对象属性赋值了， 对象也组装好了）。

Spring就是一个容器，可以管理对象，创建对象，给属性赋值。

2. IoC的技术实现

DI （依赖注入）： Dependency Injection， 缩写是DI . 是IoC的一种技术实现。 程序只需要提供要使用的对象的名称就可以了， 对象如何创建， 如何从容器中查找，获取都由容器内部自己实现。

依赖名词： 比如说ClassA类使用了ClassB的属性或者方法，叫做ClassA依赖ClassB.

```
public class ClassB{

    public void createOrder(){}
}

public class ClassA{
    //属性
    private ClassB  b = new ClassB();

    public void buy(){
        b.createOrder();
    }
}

执行ClassA的buy()
ClassA a  = new ClassA();
a.buy();
```

3. Spring框架使用的DI实现IoC.

通过spring框架， 只需要提供要使用的对象名词就可以了。 从容器中获取名称对应的对象。

spring底层使用的 反射机制， 通过反射创建对象，给属性。

## 2.2 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>

spring标准的配置文件：
1）根标签是 beans
2) beans 后面的是约束文件说明
3）beans里面是bean声明。
4）什么是bean： bean就是java对象， spring容器管理的java对象，叫做bean
```

## 2.3 spring容器创建对象的特点

1. 容器对象ApplicationContext：接口

   通过ApplicationContext对象，获取要使用的 其他java对象， 执行getBean(“的id”)

2. spring默认是调用类的无参数构造方法，创建对象
3. spring读取配置文件， 一次创建好所有的java对象， 都放到map中。

## 2.4 DI：给属性赋值。

spring调用类的无参数构造方法，创建对象。 对象创建后给属性赋值。

给属性赋值可以使用 1）xml配置文件中的标签和属性； 2）使用注解。

DI分类： 1 set注入，也叫做设值注入； 2 构造注入。

### 2.4.1 基于xml的DI

在xml配置文件中使用标签和属性，完成对象创建，属性赋值。

1） set注入，也叫做设值注入。

概念： spring调用类中的set方法，在set方法中可以完成属性赋值。 推荐使用。

```
简单类型的设值注入，使用value
<bean id="mySchool" class="com.bjpowernode.ba02.School">
    <property name="name" value="北京大学"/>
    <property name="address" value="北京的海淀区"/>
</bean>
```

```
<!--引用类型set注入-->
<bean id="myStudent" class="com.bjpowernode.ba02.Student">
    <property name="name" value="李四"/>
    <property name="age" value="22" />
    <!--引用类型的赋值-->
    <property name="school" ref="mySchool" /><!--setSchool(mySchool)-->
</bean>
```

2） 构造注入

构造注入：spring调用类中的有参数构造方法， 在创建对象的同时，给属性赋值

```
<!--构造注入，使用name属性-->
<bean id="myStudent" class="com.bjpowernode.ba03.Student">
    <constructor-arg name="myage" value="22" />
    <constructor-arg name="myname" value="李四"/>
    <constructor-arg name="mySchool" ref="mySchool"/>
</bean>

<!--构造注入，使用index，参数的位置，构造方法参数从左往右位置是0,1,2-->
<bean id="myStudent2" class="com.bjpowernode.ba03.Student">
    <constructor-arg index="1" value="28"/>
    <constructor-arg index="0" value="张三"/>
    <constructor-arg index="2" ref="mySchool" />
</bean>


<!--构造注入，省略index属性-->
```

```
<bean id="myStudent3" class="com.bjpowernode.ba03.Student">
    <constructor-arg  value="张峰"/>
    <constructor-arg  value="28"/>
    <constructor-arg  ref="mySchool" />
</bean>
```

3）引用类型的自动注入

概念： spring可以根据某些规则给引用类型完成赋值。 只对引用类型有效。 规则byName, byType.

①：byName（按名称注入）： java类中引用类型属性名称和spring容器中bean的id名称一样的，且数据类型也是一样的，这些bean能够赋值给引用类型。

```
<!-- byName 自动注入 -->
<bean id="myStudent" class="com.bjpowernode.ba04.Student" autowire="byName">
    <property name="name" value="李四"/>
    <property name="age" value="22" />
    <!--引用类型的赋值-->
    <!--<property name="school" ref="mySchool" />-->
</bean>
```

②：byType（按类型注入）：java类中引用类型的数据类型和spring容器中bean的class值是同源关系的，这样的bean赋值给引用类型。

```
注意： 在xml配置文件中， 符合条件的对象，只能有一个。 多余一个是报错的。
```

```
<!-- byType 自动注入 -->
<bean id="myStudent" class="com.bjpowernode.ba05.Student" autowire="byType">
    <property name="name" value="张三"/>
    <property name="age" value="26" />
    <!--引用类型的赋值-->
    <!--<property name="school" ref="mySchool" />-->
</bean>
```

4）作业 IoC

需求：模拟一用户注册操作。

需要定义一个dao接口（UserDao）接口中的方法 insertUser(SysUser user ) , 定义接口的实现类 MySqlUserDao

inserUser()方法里面不需要操作数据， 输出“使用了dao执行insert操作”。

需要定义一个service接口（UserService），定义接口的实现类 UserServiceImpl. 在service的实现类有一个 UserDao类型的属性。 service类中有一个方法 addUser(SysUser user).

操作是 service类中的addUser() { userDao.insertUser() } 完成注册。

定义一个实体类 SysUser， 表示用户的数据。

要求实现：

程序中的UserServiceImpl，MySqlUserDao这些类都有spring容器创建和管理， 同时要给UserServiceImpl类中的 userDao属性赋值。 从spring容器中获取 UserServiceImpl类型的对象， 调用addUser()方法，输出“使用了dao执行insert操作”

5） 项目中使用多个spring配置文件

分多个配置文件的方式： 1）按功能模块分，一个模块一个配置文件。 2） 按类的功能分，数据库操作相关的类在一个文件， service类在一个配置文件， 配置redis，事务等等的一个配置文件。

spring管理多个配置文件： 常用的是包含关系的配置文件。 项目中有一个总的文件， 里面是有impor标签包含其他的多个配置文件。

语法：

```
总的文件（xml）
<import resource="其他的文件的路径1"/>
<import resource="其他的文件的路径2"/>

关键字“classpath:”：表示类路径， 也就是类文件(class文件)所在的目录。 spring到类路径中加载文件
                    什么时候使用classpath： 在一个文件中要使用其他的文件， 需要使用classpath
```

## 2.4.2 基于注解的DI

基于注解的DI：使用spring提供的注解，完成java对象创建，属性赋值。

注解使用的核心步骤：

1.在源代码加入注解，例如@Component

2.在spring的配置文件，加入组件扫描器的标签

```
<context:component-scan base-package="注解所在的包名"/>
```

1.创建对象的注解

@Component 普通java对象

@Respository dao对象，持久层对象， 表示对象能访问数据库。

@Service service对象， 业务层对象， 处理业务逻辑，具有事务能力

@Controller 控制器对象， 接收请求，显示请求的处理结果。 视图层对象

2.简单类型属性赋值

@Value

3.引用类型赋值

@Autowired: spring提供的注解 . 支持byName， byType

@Autowired: 默认就是byType

@Autowired @Qualifier : 使用byName

@Resource： 来自jdk中的注解，给引用类型赋值的，默认是byName

@Resource: 先使用byName, 在byType

@Resource(name="bean的名称")： 只使用byName注入

## 2.5 IoC 总结

IoC:管理对象的，把对象放在容器中，创建，赋值，管理依赖关系。

IoC:通过管理对象，实现解耦合。 IoC解决处理业务逻辑对象之间的耦合关系， 也就是service和dao之间的解耦合。

spring作为容器适合管理什么对象？

1）service对象， dao对象。

2）工具类对象。

不适合交给spring的对象？

1）实体类。

2）servlet， listener， filter等web中的对象。他们是tomcat创建和管理的。

# 第三章 AOP 面向切面编程

## 3.1 增加功能，导致的问题

在源代码中， 业务方法中增加的功能。

1）源代码可能改动的比较多。

2）重复代码比较多。

3）代码难于维护。

# 3.2 AOP 概念

## 3.1 什么是AOP

AOP(Aspect Orient Programming) : 面向切面编程

Aspect： 表示切面， 给业务方法增加的功能，叫做切面。 切面一般都是非业务功能， 而且切面功能一般都是可以复用的。 例如 日志功能， 事务功能， 权限检查，参数检查， 统计信息等等。

Orient：面向，对着

Programming：编程。

怎么理解面向切面编程？ 以切面为核心设计开发你的应用。

1）设计项目时， 找出切面的功能。

2）安排切面的执行时间， 执行的位置。

# 3.2 AOP作用

1）让切面功能复用

2）让开发人员专注业务逻辑。 提高开发效率

3）实现业务功能和其他非业务功能解耦合。

4）给存在的业务方法， 增加功能，不用修改原来的代码

# 3.3 AOP中术语

1）Aspect：切面， 给业务方法增加的功能。

2）JoinPoint:连接点， 连接切面的业务方法。 在这个业务方法执行时，会同时执行切面的功能。

3）Pointcut: 切入点， 是一个或多个连接点集合。 表示这些方法执行时，都能增加切面的功能。

表示切面执行的位置。

4）target: 目标对象， 给那个对象增加切面的功能， 这个对象就是目标对象。

5）Advice：通知（增强），表示切面的执行时间。 在目标方法之前执行切面，还是目标方法之后执行切面。

AOP中重要的三个要素： Aspect， Pointcut , Advice. 这个概念的理解是： 在Advice的时间，在Pointcut的位置， 执行Aspect

AOP是一个动态的思想。 在程序运行期间，创建代理（ServcieProxy），使用代理执行方法时，增加切面的功能。这个代理对象是存在内存中的。

## 3.4 什么时候你想用AOP

你要给某些方法 增加相同的一些功能。 源代码不能改。 给业务方法增加非业务功能，也可以使用AOP

## 3.5 AOP技术思想的实现

使用框架实现AOP。 实现AOP的框架有很多。 有名的两个

1） Spring： Spring框架实现AOP思想中的部分功能。 Spring框架实现AOP的操作比较繁琐，比重。

2） Aspectj： 独立的框架，专门是AOP。 属于Eclipse

## 3.6 使用AspectJ框架实现AOP

AspectJ框架可以使用注解和xml配置文件两种方式 实现 AOP

### 3.6.1 通知

Aspectj表示切面执行时间，用的通知（Advice）。 这个通知可以使用注解表示。

讲5个注解， 表示切面的5个执行时间， 这些注解叫做通知注解。

@Before： 前置通知

@AfterRetunring： 后置通知

@Around: 环绕通知

@AfterThrowing:异常通知

@After:最终通知

### 3.6.2. Pointcut 位置

Pointcut 用来表示切面执行的位置， 使用Aspectj中切入点表达式。

切入点表达式语法： execution(访问权限 方法返回值 方法声明(参数) 异常类型)

### 3.6.3 @Before前置通知

前置通知@Before

```
/**
 * 切面类中的通知方法，可以有参数
 * JoinPoint必须是他。
 *
 * JoinPoint: 表示正在执行的业务方法。 相当于反射中 Method
 *    使用要求：必须是参数列表的第一个
 *    作用：获取方法执行时的信息，例如方法名称， 方法的参数集合
```

```
 */
@Before(value = "execution(* *..SomeServiceImpl.do*(..) )")
public void myBefore2(JoinPoint jp){

    //获取方法的定义
    System.out.println("前置通知中，获取目标方法的定义："+jp.getSignature());
    System.out.println("前置通知中，获取方法名称="+jp.getSignature().getName());
    //获取方法执行时参数
    Object args []= jp.getArgs();// 数组中存放的是 方法的所有参数
    for(Object obj:args){
        System.out.println("前置通知，获取方法的参数："+obj);
    }

    String methodName = jp.getSignature().getName();
    if("doSome".equals(methodName)){
        //切面的代码。
        System.out.println("doSome输出日志=====在目标方法之前先执行==:"+ new
Date());
    } else if("doOther".equals(methodName)){
        System.out.println("doOther前置通知，作为方法名称，参数的记录。");
    }

}
```

### 3.6.4 @AfterReturning后置通知

@AfterReturning: 在目标方法之后执行的

```
 /* 特点：
     *  1.在目标方法之后，执行的。
     *  2.能获取到目标方法的执行结果。
     *  3.不会影响目标方法的执行
     *
     * 方法的参数：
     *   Object res:  表示目标方法的返回值，使用res接收doOther的调用结果。
     *   Object res= doOther();
     *
     *  后置通知的执行顺序
     *  Object res = SomeServiceImpl.doOther(..); Student
     *
     *  myAfterReturning(res);
     *
     *  思考：
     *    1 doOther方法返回是String ， Integer ， Long等基本类型，
     *      在后置通知中，修改返回值，  是不会影响目标方法的最后调用结果的。
     *    2 doOther返回的结果是对象类型，例如Student。
     *      在后置通知方法中，修改这个Student对象的属性值，会不会影响最后调用结果？
     */
    @AfterReturning(value = "execution(* *..SomeServiceImpl.doOther(..))",
                    returning = "res")
    public void myAfterReturning(JoinPoint jp, Object res){

        //修改目标方法的返回值
        if(res != null){
            res = "HELLO Aspectj";
```

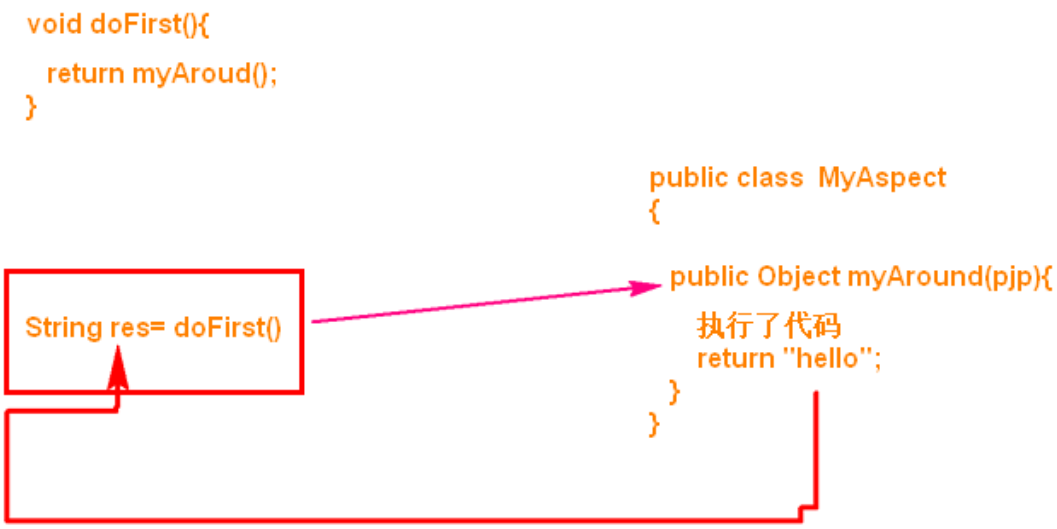
```
        }
        System.out.println("后置通知，在目标方法之后，执行的。能拿到执行结果："+res);
        //Object res有什么用
        if("abcd".equals(res)){
            System.out.println("根据返回值的不同，做不同的增强功能");
        } else if("add".equals(res)){
           System.out.println("doOther做了添加数据库，  我做了备份数据");
        }


    }
```

### 3.6.5 @Around 环绕通知

@Around(value="切入点表达式")

使用环绕通知： 就是调用 切面类中的通知方法。




```
/**
 * @Around：环绕通知
 *     属性：value 切入点表达式
 *     位置：在方法定义的上面
 *
 * 返回值：Object ，表示调用目标方法希望得到执行结果（不一定是目标方法自己的返回值）
 * 参数：   ProceedingJoinPoint，相当于反射中 Method。
 *         作用：执行目标方法的，等于Method.invoke()
 *
 *         public interface ProceedingJoinPoint extends JoinPoint {}
 *
 * 特点：
 *  1.在目标方法的前和后都能增强功能
 *  2.控制目标方法是否执行
 *  3.修改目标方法的执行结果。
```

```
 */
@Around("execution(* *..SomeServiceImpl.doFirst(..))")
public Object myAround(ProceedingJoinPoint pjp) throws Throwable {

    //获取方法执行时的参数值
    String name = "";
    Object args [] = pjp.getArgs();
    if( args != null && args.length > 0){
        Object arg = args[0];
        if(arg !=null){
            name=(String)arg;
        }
    }

    Object methodReturn = null;

    System.out.println("执行了环绕通知，在目标方法之前，输出日志时间=="+ new Date());
    //执行目标方法  ProceedingJoinPoint，表示doFirst

    if("lisi".equals(name)){
        methodReturn = pjp.proceed();//method.invoke(),表示执行doFirst()方法本身
    }

    if( methodReturn != null){
        methodReturn ="环绕通知中，修改目标方法原来的执行结果";
    }

    System.out.println("环绕通知，在目标方法之后，增加了事务提交功能");

    //return "HelloAround,不是目标方法的执行结果";
    //返回目标方法执行结果。没有修改的。
    return methodReturn;

}
```

### 3.6.6 @AfterThrowing 异常通知

语法@AfterThrowing(value="切入点表达式"，throwing="自定义变量")

```
/**
 * @AfterThrowing:异常通知
 *      属性：  value  切入点表达式
 *             throwing 自定义变量，表示目标方法抛出的异常。
 *                      变量名必须和通知方法的形参名一样
 *      位置：在方法的上面
 * 特点：
 *  1. 在目标方法抛出异常后执行的，  没有异常不执行
 *  2. 能获取到目标方法的异常信息。
 *  3. 不是异常处理程序。可以得到发生异常的通知，  可以发送邮件，短信通知开发人员。
 *       看做是目标方法的监控程序。
 *
 *  异常通知的执行
 *  try{
```

```
 *      SomeServiceImpl.doSecond(..)
 *  }catch(Exceptoin e){
 *      myAfterThrowing(e);
 *  }
 */
@AfterThrowing(value = "execution(*
*..SomeServiceImpl.doSecond(..))",throwing = "ex")
public void myAfterThrowing(Exception ex){
    System.out.println("异常通知，在目标方法抛出异常时执行的，异常原因
是："+ex.getMessage());
    /*
       异常发生可以做：
       1.记录异常的时间，位置，等信息。
       2.发送邮件，短信，通知开发人员
     */
}
```

### 3.6.7 @After 最终通知

语法: @After(value="切入点表达式")

```
/**
 * @After：最终通知
 *    属性： value 切入点表达式
 *    位置： 在方法的上面
 * 特点：
 *  1. 在目标方法之后执行的。
 *  2. 总是会被执行。
 *  3. 可以用来做程序最后的收尾工作。例如清除临时数据，变量。 清理内存
 *
 *  最终通知
 *  try{
 *      SomeServiceImpl.doThird(..)
 *  }finally{
 *      myAfter()
 *  }
 */
@After(value = "execution(* *..SomeServiceImpl.doThird(..))")
public void myAfter(){
    System.out.println("最终通知，总是会被执行的");
}
```

### 3.6.8 @Pointcut定义和管理切入点注解

@Pointcut(value="切入点表达式")

```
@After(value = "mypt()")
public void myAfter(){
    System.out.println("最终通知，总是会被执行的");
}

```

```
6
7  /**
8   * @Pointcut: 定义和管理切入点，不是通知注解。
9   *      属性： value 切入点表达式
10  *      位置： 在一个自定义方法的上面， 这个方法看做是切入点表达式的别名。
11  *            其他的通知注解中，可以使用方法名称，就表示使用这个切入点表达式了
12  */
13 @Pointcut("execution(* *..SomeServiceImpl.doThird(..))")
14 private void mypt(){
15     //无需代码
16 }
```

## 3.7 AOP总结

AOP是一种动态的技术思想， 目的是实现业务功能和非业务功能的解耦合。 业务功能是独立的模块， 其他功能也是独立的模块。例如事务功能，日志等等。 让这些事务，日志功能是可以被复用的。

当目标方法需要一些功能时，可以在不修改，不能修改源代码的情况下， 使用aop技术在程序执行期间，生成代理对象， 通过代理执行业务方法，同时增加功能。

# 第四章 Spring集成MyBatis

## 4.1 集成思路

spring能集成很多的框架，是spring一个优势功能。 通过集成功能，让开发人员使用其他框架更方便。

集成使用的是spring ioc 核心技术。

## 4.2 要使用框架，例如mybatis，怎么使用mybatis？

使用mybatis，需要创mybatis框架中的某些对象， 使用这些对象，就能使用mybatis提供的功能了。

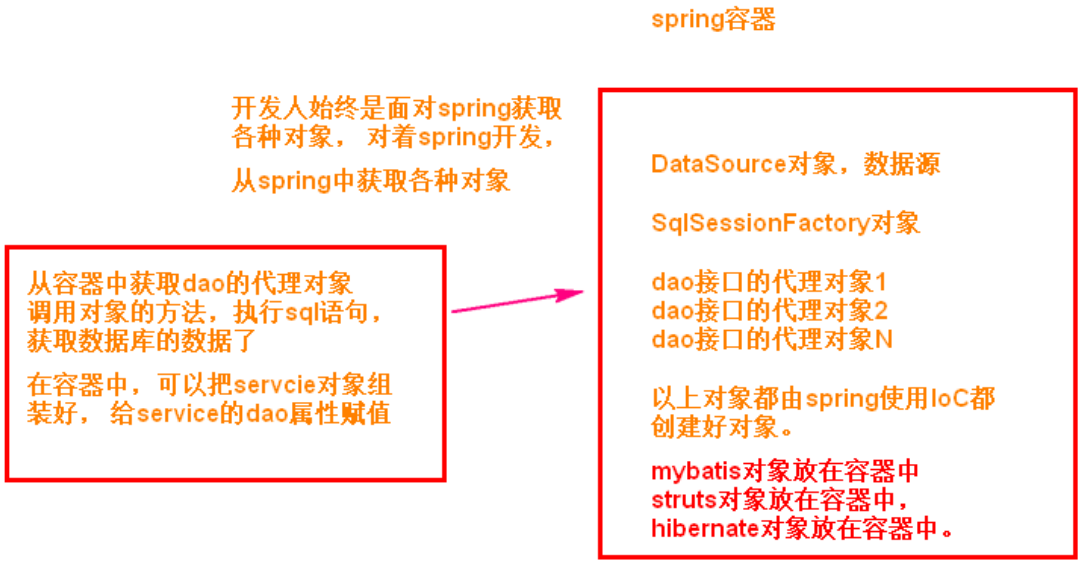
分析： mybatis执行sql语句，需要使用那些对象

1. 需要有Dao接口的代理对象， 例如StudentDao接口, 需要一个它的代理对象

   使用 SqlSession.getMapper(StudentDao.class)，得到dao代理对象

2. 需要有SqlSessionFactory，创建SqlSessionFactory对象，才能使用openSession()得到SqlSession对象

   3.数据源DataSource对象， 使用一个更强大，功能更多的连接池对象代替mybatis自己的PooledDataSource




# 第五章 Spring 事务

## 5.1 事务的概念

什么事务？ 事务是一些sql序列的集合， 是多条sql， 作为一个整体执行。

```
mysql执行事务
beginTransaction 开启事务
insert into student() values.....
select * from student where id=1001
update school set name=xxx where id=1005
endTransaction   事务结束
```

什么情况下需要使用事务?

一个操作需要多条（2条或2条以上的sql）sql语句一起完成， 操作才能成功。

## 5.2 在程序中事务在哪说明

事务：加在业务类的方法上面（public方法上面），表示业务方法执行时，需要事务的支持。

```
public class AccountService{

    private AccountDao dao;
    private MoneyDao dao2;
    // 在service（业务类）的public方法上面，需要说明事务。
    public void trans(String a, String b, Integer money){
        dao.updateA();
        dao.updateB();
        dao2.insertA();
    }
}
```

```
12
13
14 public class AccountDao{
15
16     public void updateA(){}
17     public void updateB(){}
18
19 }
20
21 public class MoneyDao{
22
23     public void insertA(){}
24     public void deleteB(){}
25
26 }
```

## 5.3 事务管理器

### 5.3.1 不同的数据库访问技术，处理事务是不同的

1）使用jdbc访问数据库， 事务处理。

```
1 public void updateAccount(){
2     Connection conn = ...
3     conn.setAutoCommit(false);
4     stat.insert()
5     stat.update();
6     conn.commit();
7     con.setAutoCommit(true)
8 }
```

2. mybatis执行数据库，处理事务

```
1  public void updateAccount(){
2      SqlSession session = SqlSession.openSession(false);
3      try{
4          session.insert("insert into student...");
5          session.update("update school ...");
6          session.commit();
7      }catch(Exception e){
8          session.rollback();
9      }
10 }
```

### 5.3. 2 spring统一管理事务， 把不同的数据库访问技术的事务处理统一起来。

使用spring的事务管理器，管理不同数据库访问技术的事务处理。 开发人员只需要掌握spring的事务处理一个方案， 就可以实现使用不同数据库访问技术的事务管理。

管理事务面向的是spring， 有spring管理事务，做事务提交，事务回顾。

### 5.3.3 Spring事务管理器

Spring框架使用事务管理器对象，管理所有的事务。

事务管理器接口： PlatformTransactionManager

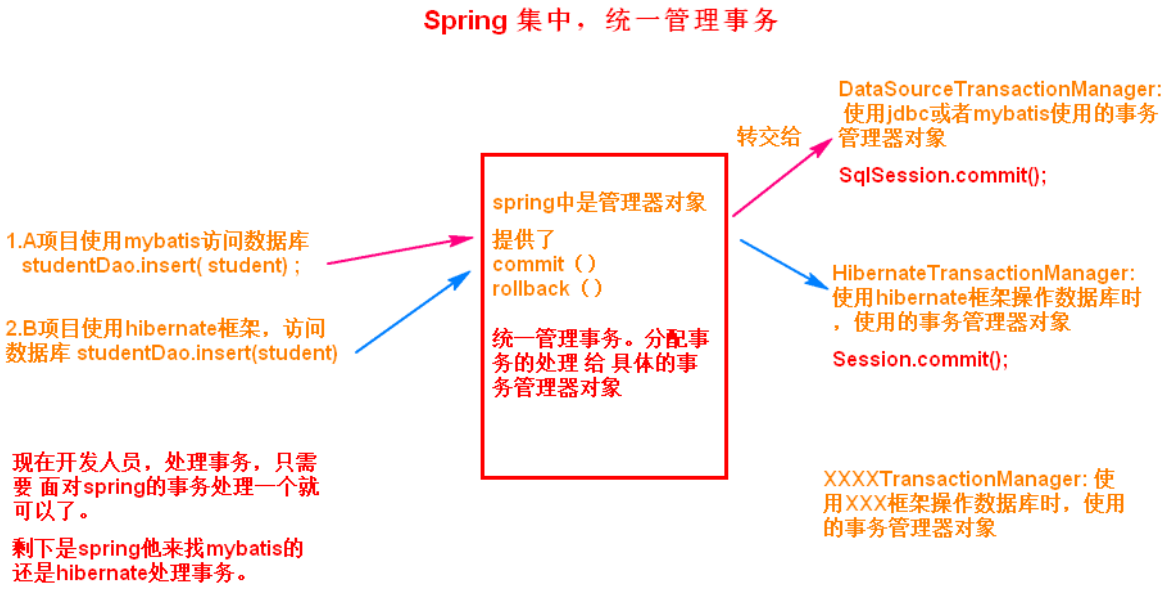
作用： 定义了事务的操作， 主要是commit(), rollback()

事务管理器有很多实现类： 一种数据库的访问技术有一个实现类。 由实现类具体完成事务的提交，回顾。

意味着： jdbc或者mybatis访问数据库有自己的事务管理器实现类： DataSourceTranactionManager

hibernate框架，他的事务管理器实现类： HibernateTransactionManager.

事务管理器工作方式：




### 5.3.4 事务的提交和回顾的时机

什么时候提交事务，回滚事务？

当你的业务方法正常执行时，没有异常，事务是提交的。 如果你的业务方法抛出了运行时异常， 事务是回滚的。

异常分类：

Error： 严重错误。 回滚事务。

Exception：异常类，可以出来的异常情况

1） 运行时异常： RuntimeException和他的子类都是运行时异常， 在程序执行过程中抛出的异常。常见的运行时异常： NullPoinerException，NumberFormatException，ArithmeticException, IndexOutOfBoundsException.

2. 受查异常： 编写java代码的时候，必须出来的异常。 例如IOException，SQLException，FileNotFoundException

怎么记忆：

方法中抛出了运行时异常， 事务回滚，其他情况（正常执行方法，受查异常）就是提交事务。

### 5.3.5 事务使用的AOP的环绕通知

环绕通知：可以在目标方法的前和后都能增强功能，不需要修改代码代码

```
spring给业务方法在执行时，增加上事务的切面功能
@Around("execution(*  所有的业务类中的方法)")
public Object myAround(ProceedingJoinPoint pjp) {
    try{
       PlatformTransactionManager.beginTransaction();//使用spring的事务管理器，开启事务
       pjp.proceed(); //执行目标方法 //doSome()
       PlatformTransactionManager.commit();//业务方法正常执行，提交事务
    }catch(Exception e){
        PlatformTransactionManager.rollback();//业务方法正常执行，回滚事务
    }

}
```

## 5.4 事务定义接口TransactionDefinition

TransactionDefinition接口。定义了三类常量， 定义了有关事务控制的属性。

事务的属性：1） 隔离级别 2）传播行为 3）事务的超时

给业务方法说明事务属性。和ACID不一样。

### 5.4.1 隔离级别

隔离级别：控制事务之间影响的程度。

5个值，只有四个隔离级别

1）DEFAULT： 采用DB默认的事务隔离级别。 MySql的默认为 REPEATABLE_READ;

Oracle 默认为 READ_COMMITTED。

2）READ_UNCOMMITTED：读未提交。未解决任何并发问题。
3）READ_COMMITTED：读已提交。解决脏读，存在不可重复读与幻读。
4）REPEATABLE_READ：可重复读。解决脏读、不可重复读，存在幻读
5）SERIALIZABLE：串行化。不存在并发问题。

### 5.4.2 超时时间

超时时间，以秒为单位 。 整数值。 默认是 -1

超时时间：表示一个业务方法最长的执行时间，没有到达时间没有执行完毕，spring回滚事务。

### 5.4.3 传播行为

传播行为有7个值。

传播行为：业务方法在调用时，事务在方法之间的，传递和使用。

使用传播行为，标识方法有无事务。

PROPAGATION_REQUIRED
PROPAGATION_REQUIRES_NEW
PROPAGATION_SUPPORTS

以上三个需要掌握的。

PROPAGATION_MANDATORY
PROPAGATION_NESTED
PROPAGATION_NEVER
PROPAGATION_NOT_SUPPORTED

1） REQUIRED: spring默认传播行为， 方法在调用的时候，如果存在事务就是使用当前的事务，如果没有事务，则新建事务， 方法在新事务中执行。

2） SUPPORTS：支持， 方法有事务可以正常执行，没有事务也可以正常执行。

3）REQUIRES_NEW：方法需要一个新事务。 如果调用方法时，存在一个事务，则原来的事务暂停。 直到新事务执行完毕。 如果方法调用时，没有事务，则新建一个事务，在新事务执行代码。

## 5.5 Spring框架使用自己的注解@Transactional控制事务

@Transactional注解， 使用注解的属性控制事务（隔离级别，传播行为，超时）

属性：

1. propagation： 事务的传播行为， 他使用的 Propagation类的枚举值。例如 Propagation.REQUIRED

2.isolation： 表示隔离级别， 使用Isolation类的枚举值，表示隔离级别。 默认 Isolation.DEFAULT

3.readOnly: boolean类型的值，表示数据库操作是不是只读的。默认是false

4.timeout： 事务超时，默认是-1， 整数值，单位是秒。 例如 timeout=20

5.rollbackFor：表示回滚的异常类列表， 他的值是一个数组，每个值是异常类型的class。

6.rollbackForClassName：表示回滚的异常类列表,他的值是异常类名称，是String类型的值

7.noRollbackFor： 不需要回滚的异常类列表。 是class类型的。

8.noRollbackForClassName： 不需要回滚的异常类列表，是String类型的值

位置：1）在业务方法的上面， 是在public方法的上面

2） 在类的上面。

注解的使用步骤：

1）在spring的配置文件，声明事务的内容

声明事务管理器，说明使用哪个事务管理器对象

声明使用注解管理事务， 开启是注解驱动

2）在类的源代码中，加入@Transactional.

事务的控制模式： 1. 编程式，在代码中编程控制事务。 2. 声明式事务。不用编码

例子：

```
spring配置文件
<!--声明事务的控制-->
<!--声明事务管理器-->
<bean id="transactionManager"
 class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!--指定数据源DataSource-->
    <property name="dataSource" ref="myDataSource" />
</bean>



<!--开启事务注解驱动：  告诉框架使用注解管理事务
    transaction-manager:指定事务管理器的id
-->
<tx:annotation-driven transaction-manager="transactionManager" />
```

```
java代码
//@Transactional 放在public方法的上面。表示方法有事务功能
    /*
    第一种设置方式
    @Transactional(
            propagation = Propagation.REQUIRED,
            isolation = Isolation.DEFAULT,
            readOnly = false, timeout = 20,
            rollbackFor =
{NullPointerException.class,NotEnougthException.class})
     */


    /*
    第二种设置方式
     @Transactional(
            propagation = Propagation.REQUIRED,
            isolation = Isolation.DEFAULT,
            readOnly = false, timeout = 20
     )
```

```
20
21      解释 rollbackFor 的使用；
22      1）框架首先检查方法抛出的异常是不是在 rollbackFor 的数组中， 如果在一定回滚。
23      2）如果方法抛出的异步不在 rollbackFor 数组， 框架会继续检查 抛出的异常 是不是
RuntimeException.
24          如果是RuntimeException， 一定回滚。
25
26
27      例如 抛出 SQLException , IOException
28      rollbackFor={SQLException.class, IOException.class}
29      */
30
31
32      //第三种方式： 使用默认值 REQUIRED ， 发生运行时异常回滚。
33      @Transactional
34      @Override
35      public void buy(Integer goodsId, Integer num) { }
36
```

@Transactional使用的特点：

1.spring框架自己提供的事务控制

2.适合中小型项目。

3.使用方便，效率高。

## 5.6 使用Aspectj框架在spring的配置文件中，声明事务控制

使用aspectj的aop，声明事务控制叫做声明式事务

使用步骤：

1. pom.xml加入 spring-aspects的依赖
2. 在spring的配置文件声明事务的内容
   1）声明事务管理器
   2）声明业务方法需要的事务属性
   3）声明切入点表达式

声明式事务

```
1   <!--声明式事务： 不用写代码 -->
2       <!--1. 声明事务管理器-->
3       <bean id="transactionManager"
 class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
4           <property name="dataSource" ref="myDataSource" />
5       </bean>
6
7
8       <!--2.声明业务方法的事务属性（隔离级别，传播行为，超时）
9           id:给业务方法配置事务段代码起个名称，唯一值
10          transaction-manager:事务管理器的id
```

```
    -->
    <tx:advice id="serviceAdvice" transaction-manager="transactionManager">
        <!--给具体的业务方法增加事务的说明-->
        <tx:attributes>
            <!--
                给具体的业务方法，说明他需要的事务属性
                name: 业务方法名称。
                      配置name的值： 1. 业务方法的名称； 2. 带有部分通配符(*)的方法名称； 3 使用*
                propagation:指定传播行为的值
                isolation：隔离级别
                read-only：是否只读，默认是false
                timeout：超时时间
                rollback-for：指定回滚的异常类列表，使用的异常全限定名称
            -->
            <tx:method name="buy" propagation="REQUIRED" isolation="DEFAULT"
                       read-only="false" timeout="20"
                       rollback-for="java.lang.NullPointerException,com.bjpowernode.excetion.NotEnougthException"/>


            <!--在业务方法有命名规则后， 可以对一些方法使用事务-->
            <tx:method name="add*" propagation="REQUIRES_NEW"
                rollback-for="java.lang.Exception" />
            <tx:method name="modify*"
                propagation="REQUIRED" rollback-for="java.lang.Exception" />
            <tx:method name="remove*"
                propagation="REQUIRED" rollback-for="java.lang.Exception" />

            <!--以上方法以外的 * :querySale, findSale, searchSale -->
            <tx:method name="*" propagation="SUPPORTS" read-only="true" />
        </tx:attributes>
    </tx:advice>

    <!--声明切入点表达式： 表示那些包中的类，类中的方法参与事务-->
    <aop:config>
       <!--声明切入点表达式
            expression：切入点表达式， 表示那些类和类中的方法要参与事务
            id：切入点表达式的名称，唯一值

            expression怎么写？
       -->
       <aop:pointcut id="servicePointcut" expression="execution(* *..service..*.*(..))" />
       <!--关联切入点表达式和事务通知-->
       <aop:advisor advice-ref="serviceAdvice" pointcut-ref="servicePointcut" />
    </aop:config>
```

声明式事务优缺点：

1. 缺点： 理解难， 配置复杂。
2. 优点： 代码和事务配置是分开的。 控制事务源代码不用修改。
   能快速的了解和掌控项目的全部事务。 适合大型项目。

# 第六章 Spring和Web

## 6.1 现在使用容器对象的问题

1. 创建容器对象次数多
2. 在多个servlet中，分别创建容器对象

## 6.2 需要一个什么样的容器对象

1. 容器对象只有一个， 创建一次就可以了
2. 容器对象应该在整个项目中共享使用。 多个servlet都能使用同一个容器对象

解决问题使用监听器 ServletContextListener (两个方法 初始时执行的，销毁时执行的)

在监听器中，创建好的容器对象，应该放在web应用中的ServletContext作用域中。

## 6.3 ContextLoaderListener

ContextLoaderListener 是一个监听器对象， 是spring框架提供的， 使用这个监听器作用：

1.创建容器对象，一次

2.把容器对象放入到ServletContext作用域。

使用步骤：

1.pom.xml加入依赖spring-web

2.web.xml 声明监听器

依赖

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>5.2.5.RELEASE</version>
</dependency>
```

监听器的设置

```xml
<context-param>
    <!--
      contextConfigLocation：名称是固定的， 表示自定义spring配置文件的路径
    -->
    <param-name>contextConfigLocation</param-name>
    <!--自定义配置文件的路径-->
    <param-value>classpath:spring-beans.xml</param-value>
</context-param>
<listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

## 6.4 ContextLoaderListener 源代码

```
public class ContextLoaderListener extends ContextLoader implements
ServletContextListener {

    //监听器的初始方法
    public void contextInitialized(ServletContextEvent event) {
        this.initWebApplicationContext(event.getServletContext());
    }

}


private WebApplicationContext context;

public WebApplicationContext initWebApplicationContext(ServletContext
servletContext) {


            try {
                if (this.context == null) {
                    //创建srping的容器对象
                    this.context =
this.createWebApplicationContext(servletContext);
                }

                //把容器对象放入的ServletContext作用域

 //key=WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE
                //value=容器对象

                servletContext.setAttribute(

 WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE,
                this.context);


            } catch (Error | RuntimeException var8) {

```

```
            }
        }
    }


//WebApplicationContext是web项目中使用的容器对象
public interface WebApplicationContext extends ApplicationContext

```