IOC

Ioc：控制反转容器

概念：系统将创建bean对象的权利转移给Spring框架

作用：降低程序间的耦合

使用ioc

1:创建业务层接口和实现类

2:创建持久层接口和实现类

3创建一个读取的xml

BeanFactory 和 ApplicationContext 的区别

BeanFactory 才是 Spring 容器中的顶层接口。

ApplicationContext 是它的子接口。

BeanFactory 和 ApplicationContext 的区别:
创建对象的时间点不一样。 ApplicationContext:只要一读取配置文件，默认情况下就会创建对象。 BeanFactory:什么使用什么时候创建对象。

ApplicationContext 提供了更多强大的功能

spring默认是单例的因为service这个类中没有啥别的属性，是线程安全的

创建bean的三种方式：

-

第二种方式:spring 管理静态工厂-使用静态工厂的方法创建对象 /**

- 模拟一个静态工厂，创建业务层实现类
  */

---

public class StaticFactory {
public static IAccountService createAccountService(){ return new AccountServiceImpl();
} }
<!-- 此种方式是:
使用 StaticFactory 类中的静态方法 createAccountService 创建对象，并存入 spring 容器

id 属性:指定 bean 的 id，用于从容器中获取 class 属性:指定静态工厂的全限定类名 factory-method 属性:指定生产对象的静态方法
-->

第三种方式:spring 管理实例工厂-使用实例工厂的方法创建对象 /**

- 模拟一个实例工厂，创建业务层实现类
- 此工厂创建对象，必须现有工厂实例对象，再调用方法 */

public class InstanceFactory {
public IAccountService createAccountService(){
return new AccountServiceImpl(); }
}

后面两种创建bean的方式：一般用于创建的bean是在jar包中我们无法修改情况

bean的作用范围

```
<bean class="" scope="prototype"></bean>
<!--  作用范围 scope  , singleton代表单例   prototype代表多例-->

<!--singleton代表单例,容器创建时即创建该bean   prototype代表多例  当用到该bean才创建-->

<!--通过创建单例或多例的方式来控制容器是延时加载还是立即加载-->

  <bean class="" scope="singleton"></bean>
```

bean的生命周期

单例和多例的生命周期区别

单例

- 出生： 和spring的容器生命周期基本一样
- 活着： 和spring的容器生命周期基本一样
- 死亡：当关闭容器时，bean死亡
  - 多例
- 出生：当使用该bean的时候才创建
- 活着：对象使用过程中一直活着
- 死亡：对象在内存中一直没有被使用，通过垃圾回收给回收

spring依赖注入

为啥需要这个?

只适用于默认构造函数，当构造函数中有其他属性时，就没有办法使用这个了。

1构造方法注入（不常用）

```
public userdao(String name,int age,String sex){

//会自动将String 转为int类型

}
```

```
  <bean id="userdao" class="com.baidu.dao.userdao">
    <constructor-arg name="name" value="张三"></constructor-arg>
    <constructor-arg name="age" value="18"></constructor-arg>
    <constructor-arg name="sex" value="男"></constructor-arg>
    <!-- collaborators and configuration for this bean go here -->
  </bean>
```

2set注入

编写set方法

```
public void setName(String name) {

this.name = name;

 }
public void setAge(Integer age) {
this.age = age; }

 public void setBirthday(Date birthday) {

 this.birthday = birthday;
}
```

xml配置:

```
<bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">
<property name="name" value="test"></property>

 <property name="age" value="21"></property> <property name="birthday" ref="now">
</property> </bean>
```

- 复杂类型的注入
- 1.集合.

```
<property name="list">
            <list value-type="java.lang.String">
                <value>aaa</value>
                <value>bbb</value>
                <value>ccc</value>
            </list>
        </property>
```

- 2.map

```
<property name="map">

            <map>
                <entry key="iii"  value="ppp"></entry>

            </map>
        </property>
```

常用的注解:

1:bean对象的类型注入

@Component:用于所有 bean对象的类型注入

@Reponsitory：用于持久层的注入

@Controller：表现层

@Service：业务层

2用于注入数据

@Autowired

相当于:

ref 是指向的对象，value是具体的值
作用:

自动按照类型注入。当使用注解注入属性时，set 方法可以省略。它只能注入其他 bean 类型。注入bean类型时，该注解会寻找实现类，当有多个 类型匹配时，使用要注入的对象变量名称作为 bean 的 id，在 spring 容器查找，找到了也可以注入成功。找不到 就报错。

```
@Autowiredprivate IAccountDao accountDao;``
  会找类 AccountDaoImpl 并实现IAccountDao的类，并实例下面的类public class AccountDaoImpl
 implements IAuntDao{}
当有多个类实现该类时会报错，当没有实现IAuntDao的类也报错
```

@Qualifier

作用:
在自动按照类型注入的基础之上，再按照 Bean 的 id 注入。它在给字段注入时不能独立使用，必须和 @Autowire 一起使用;但是给方法参数注入时，可以独立使用。 属性:
value:指定 bean 的 id。（当@Autowire注入时有多个类实现时，可以指定id，这样就不会有冲突，这时必须跟@Autowire一起用）

```
@Autowired
    @Qualifier("userService1")
    private IUserDao iUserDao;//这两个注解要一起使用
```

当有多个实现没有指定实现类是报错:

```
org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean
with name 'IUserServiceImpl2': Unsatisfied dependency expressed through field
'iUserDao'; nested exception is
org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean
of type 'com.baidu.dao.IUserDao' available: expected single matching bean but found
2:
```

expected single matching bean but found 2:

@Value

用于注入基本类型和string类型

属性 value：用于指定值

简单使用xml配置动态代理

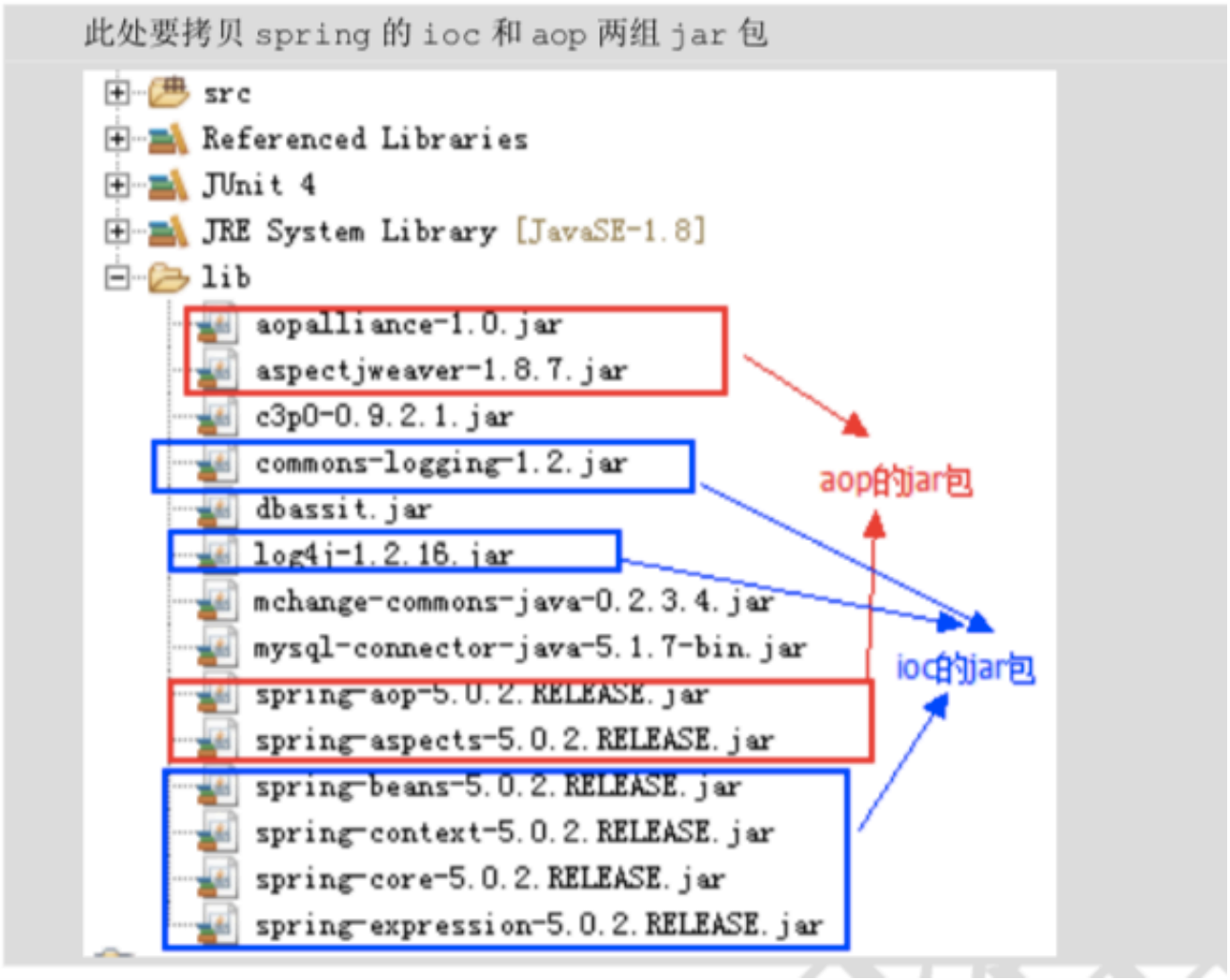
aop动态代理：

为了强化方法的，spring 提供一个增强的方法，用的动态代理，

会在要增强的方法执行前，执行后，进行增强，比如打印日志，可以在方法执行前

打印日志。前提是你要使用该方法。

1:导入需要的包




```
<dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.2.8.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>5.2.8.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.9.5</version>
        </dependency>
```

```
        <dependency>
            <groupId>aopalliance</groupId>
            <artifactId>aopalliance</artifactId>
            <version>1.0</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>5.0.10.RELEASE</version>
        </dependency>

  <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.2.8.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13</version>
            <scope>test</scope>
        </dependency>
```

创建 spring 的配置文件并导入约束

```
<?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.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">

</beans>
```

3创建一个logger类用于打印日志

```
package com.baidu.com.baidu.utils;
public class Logger {
  public void printlog(){
    System.out.println("Logger类中的pringLog方法开始记录日志了。。。");
  }}
```

4:在bean.xml中配置

```
2、使用aop:config标签表明开始AOP的配置
    3、使用aop:aspect标签表明配置切面
            id属性：是给切面提供一个唯一标识
            ref属性：是指定通知类bean的Id。
    4、在aop:aspect标签的内部使用对应标签来配置通知的类型
           我们现在示例是让printLog方法在切入点方法执行之前之前：所以是前置通知
           aop:before：表示配置前置通知
                method属性：用于指定Logger类中哪个方法是前置通知
                pointcut属性：用于指定切入点表达式，该表达式的含义指的是对业务层中哪些方法增强

         切入点表达式的写法：
            关键字：execution(表达式)
            表达式：
                访问修饰符  返回值  包名.包名.包名...类名.方法名(参数列表)
            标准的表达式写法：
                public void com.itheima.service.impl.AccountServiceImpl.saveAccount()
            访问修饰符可以省略
                void com.itheima.service.impl.AccountServiceImpl.saveAccount()
            返回值可以使用通配符，表示任意返回值
                * com.itheima.service.impl.AccountServiceImpl.saveAccount()
            包名可以使用通配符，表示任意包。但是有几级包，就需要写几个*.
                * *.*.*.*.AccountServiceImpl.saveAccount())
            包名可以使用..表示当前包及其子包
                * *..AccountServiceImpl.saveAccount()
            类名和方法名都可以使用*来实现通配
                * *..*.*()
            参数列表：
                可以直接写数据类型：
                    基本类型直接写名称           int
                    引用类型写包名.类名的方式   java.lang.String
                可以使用通配符表示任意类型，但是必须有参数
                可以使用..表示有无参数均可，有参数可以是任意类型
            全通配写法：
                * *..*.*(..)

            实际开发中切入点表达式的通常写法：
                切到业务层实现类下的所有方法
                    * com.itheima.service.impl.*.*(..)
-->
```

```
<!-- 配置Logger类 -->
    <bean id="logger" class="com.baidu.com.baidu.utils.Logger"></bean>
<!--配置aop-->
    <aop:config>
        <!--配置切面 -->
        <aop:aspect id="loggete" ref="logger">
            <aop:before method="printlog" pointcut="execution(* *..*.*(..))">
</aop:before>
            </aop:aspect>
    </aop:config>
```

通配符

xml配置aop的四种通知

xml中的配置如上，

前置通知：在方法执行前的通知 类似于开启事务

后置通知：方法执行后的通知，与异常通知只能出现一个，类似于提交事务

异常通知：方法出现了异常后调用，类似于rollback

最终通知：方法始终会调用，类似于finally

简化环绕通知：

<aop:pointcut id="prt1" expression="execution(* com.baidu.service..(..))"/>

aop:before/aop:before

aop:after/aop:before 等环绕通知中的pointcut="execution(* com.baidu.service..(..))"可以省略。使用point-ref 引用该pointcut。如下：

```
<aop:before method="beforeprintlog" pointcut-ref="prt1"></aop:before>
```

spring的aop日志

回顾用注解实现aop

第一步:准备必要的代码和 jar 包

第二步:在配置文件中导入 context 的名称空间

2.3.1.3 第三步:把资源使用注解配置

第四步:在配置文件中指定 spring 要扫描的包

把通知类也使用注解配置（@component）

在通知类上使用@Aspect 注解声明为切面

在增强的方法上添加注解

2.3.2.4 第四步:在 spring 配置文件中开启 spring 对注解 AOP 的支持

```
<aop:aspectj-autoproxy/>
```

通知类代码：

```
package com.itcast.controller;

import com.itcast.domain.Syslog;
import com.itcast.service.ISyslogService;
import org.aspectj.lang.JoinPoint;
```

```java
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.context.SecurityContext;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.User;
import org.springframework.stereotype.Component;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.http.HttpServletRequest;
import java.lang.reflect.Method;
import java.util.Date;
import java.util.UUID;

@Component
@Aspect
public class Aoplog {

@Autowired
private ISyslogService iSyslogService;
    private Date startTime; // 访问时间
    private Class executionClass;// 访问的类
    //
    private Method executionMethod; // 访问的方法 // 主要获取访问时间、访问的类、访问的方法
    @Autowired
    private HttpServletRequest request;

//    @Before("execution(* com.itcast.controller.*.*(..))")
    @Before("execution(* com.itcast.controller.*.*(..))")
    public void dobefore(JoinPoint jp) throws NoSuchMethodException {
        //获取当前时间
    startTime=new Date();
    Object[] args=jp.getArgs();
        executionClass= jp.getTarget().getClass();
        //获取方法名称
        String methodname=jp.getSignature().getName();
        if(args==null||args.length==0){
            executionMethod=executionClass.getMethod(methodname);

        }else {
            Class [] clazz  =new Class[args.length];
            for(int i=0;i<args.length;i++){
                clazz[i]=args.getClass();

            }
           executionMethod= executionClass.getMethod(methodname,clazz);

        }

    }
```

```
        @After("execution(* com.itcast.controller.*.*(..))")
    public void after(){
            // 主要获取日志中其它信息，时长、ip、url...

            long time = new Date().getTime()-startTime.getTime();
    //获取注解中的url

if(executionClass!=null&&executionMethod!=null&&executionClass!=Aoplog.class){
             //1.获取类上的@RequestMapping("/orders")
             RequestMapping classannotion= (RequestMapping)
executionClass.getAnnotation(RequestMapping.class);
             String[] params=new String[1];
             if(classannotion!=null){
                 //获取类的@requestmapping的参数
                  params = classannotion.value();
//
                 if(executionMethod!=null){
                     //获取方法上的注解@RequestMapping(xxx)
                     RequestMapping
requestMapping=executionMethod.getAnnotation(RequestMapping.class);
                     // RequestMapping requestMapping= annotation;

                     if(requestMapping!=null){
                         String[] value = requestMapping.value();
                         String url=params[0]+value[0];

                         //获取访问的ip
                         String ip = request.getRemoteAddr();

                         //获取当前操作的用户
                         //获取当前操作的用户
                         SecurityContext context =
SecurityContextHolder.getContext();//从上下文中获了当前登录的用户
                         User user = (User)
context.getAuthentication().getPrincipal();
                         String username = user.getUsername();


                         //将日志相关信息封装到Syslog对象
                         String s = UUID.randomUUID().toString();
                         StringBuffer stringBuffer=new StringBuffer(s);
                         String s2= stringBuffer.substring(1,25);
                         Syslog Syslog = new Syslog();
                         Syslog.setExecutionTime(time); //执行时长
                         Syslog.setIp(ip);
                         Syslog.setMethod("[类名] " + executionClass.getName() + "[方法
名] " + executionMethod.getName());
                         Syslog.setUrl(url);
                         Syslog.setUsername(username);
                         Syslog.setVisitTime(startTime);
                         Syslog.setId(s2);
                         iSyslogService.save(Syslog);
```

```
                    }
                }
            }else {
                params[0]="";
                if(executionMethod!=null){
                    //获取方法上的注解@RequestMapping(xxx)
                    RequestMapping
requestMapping=executionMethod.getAnnotation(RequestMapping.class);
                    // RequestMapping requestMapping= annotation;

                    if(requestMapping!=null){
                        String[] value = requestMapping.value();
                        String url=value[0];

                        //获取访问的ip
                        String ip = request.getRemoteAddr();

                        //获取当前操作的用户
                        //获取当前操作的用户
                        SecurityContext context =
SecurityContextHolder.getContext();//从上下文中获了当前登录的用户
                        User user = (User)
context.getAuthentication().getPrincipal();
                        String username = user.getUsername();


                        //将日志相关信息封装到Syslog对象
                        String s = UUID.randomUUID().toString();
                        StringBuffer stringBuffer=new StringBuffer(s);
                        String s2= stringBuffer.substring(1,25);
                        Syslog Syslog = new Syslog();
                        Syslog.setExecutionTime(time); //执行时长
                        Syslog.setIp(ip);
                        Syslog.setMethod("[类名] " + executionClass.getName() + "[方法
名] " + executionMethod.getName());
                        Syslog.setUrl(url);
                        Syslog.setUsername(username);
                        Syslog.setVisitTime(startTime);
                        Syslog.setId(s2);
                        iSyslogService.save(Syslog);

                    }
                }

            }



        }
    }
```

```
}
```

上面的就是通知类，可以实现aop通知的功能，

joinpoint

Joinpoint(连接点): 所谓连接点是指那些被拦截到的点。在 spring 中,这些点指的是方法,因为 spring 只支持方法类型的连接点。

总结：aop类主要是在调用完controller方法获取我们需要的内容：访问的开始时间，访问的用时，IP，用户名，等等信息，

aop

== 简单的说它就是把我们程序重复的代码抽取出来，在需要执行的时候，使用动态代理的技术，在不修改源码的基础上，对我们的已有方法进行增强

AOP 相关术语

Advice(通知/增强):
所谓通知是指拦截到 Joinpoint 之后所要做的事情就是通知。 通知的类型:前置通知,后置通知,异常通知,最终通知,环绕通知。
Introduction(引介):
引介是一种特殊的通知在不修改类代码的前提下, Introduction 可以在运行期为类动态地添加一些方法或 Field。

Target(目标对象):
代理的目标对象。
Weaving(织入): 是指把增强应用到目标对象来创建新的代理对象的过程。
spring 采用动态代理织入，而 AspectJ 采用编译期织入和类装载期织入。

Proxy(代理) :
一个类被 AOP 织入增强后，就产生一个结果代理类。

Aspect(切面):
是切入点和通知(引介)的结合。

Spring基于事务的xml配置（简单，代码少）

问题，实现银行转账功能，发现拓展性不好，一旦TransactionManager中某个方法改个名字时，service层中要修改的代码就太多了，

解决方案：使用代理者模式，可以在不修改原来代码的基础上，对方法的增强。

Spring中用了面向切面的方式（aop），我们只要在代码中配置。

bean.xml中的代码

```
<!--    首先配置加强的bean-->
    <bean id="TransactionManager"
class="com.baidu.com.baidu.utils.TransactionManager">
        <property name="cu" ref="connectionUtils"></property>
    </bean>
<!--    1配置aop-->


    <aop:config>
        <!--配置切面 -->
        <aop:pointcut id="uuu" expression="execution(* com.baidu.service.*.*(..))"/>
        <aop:aspect id="transactionManager" ref="TransactionManager">
        <aop:before method="begintrasation" pointcut-ref="uuu"></aop:before>
            <aop:after-returning method="commit" pointcut-ref="uuu"></aop:after-
returning>
            <aop:after-throwing method="rollback" pointcut-ref="uuu"></aop:after-
throwing>
            <aop:after method="closeconn" pointcut-ref="uuu"></aop:after>
        </aop:aspect>


    </aop:config>
```

TransactionManager类中的代码直接复制即可）

```
package com.baidu.com.baidu.utils;

import java.sql.Connection;

public class TransactionManager {
    private ConnectionUtils cu;
    public void setCu(ConnectionUtils cu) {
```

```
        this.cu = cu;
    }

    public void commit(){
        try{
            cu.getconnection().commit();
            System.out.println("后置通知****");
        }catch (Exception e){

            System.out.println("异常");
        }


    }

    public void closeconn(){
        try{
            cu.getconnection().close();
            cu.removeconn();
            System.out.println("最终通知");
        }catch (Exception e){

            System.out.println("异常");
        }


    }

    public void rollback(){
        try{

            cu.getconnection().rollback();
            System.out.println("异常通知****");
        }catch (Exception e){

            System.out.println("异常");
        }

    }

    public void begintrasation(){
    Connection conn= cu.getconnection();
    try{

        conn.setAutoCommit(false);
        System.out.println("前置通知");
    }catch (Exception e){
        System.out.println("异常");

    }
```

```
    }


}
```

Connectionutil类的代码（直接复制即可）

```
package com.baidu.com.baidu.utils;

import javax.sql.DataSource;
import java.sql.Connection;

public class ConnectionUtils {
    private  ThreadLocal<Connection> tl=new ThreadLocal<Connection>();

    private DataSource ds;

    public void setDs(DataSource ds) {
        this.ds = ds;
    }

public Connection getconnection(){
    Connection conn=   tl.get();
        try {

         if(conn==null){
             conn=ds.getConnection();
             tl.set(conn);

         }

        }catch (Exception e){

            System.out.println(e);
        }


    return  conn;
}

public void removeconn(){
      tl.remove();

}
}
```

aop环绕通知两种配置方法

xml配置环绕通知

1bean.xml中配置dtd

```
 <!-- 告知 spring，在创建容器时要扫描的包 -->                <context:component-scan base-package="com.itheima"></context:component-scan>
```

bean.xml中的代码

```
<!--配置切面-->
<aop:config>
    <!--配置切入点表达式-->
    <aop:pointcut id="ptl" expression="execution(* com.baidu.service.*.*(..))"/>
    <aop:aspect id="kk" ref="logger">
        <!--配置环绕表达式-->
      <aop:around method="aroundlog" pointcut-ref="ptl"></aop:around>
    </aop:aspect>

</aop:config>
```

logger 类中的方法

aroundlog是环绕方法。一定要加ProceedingJoinPoint参数否则方法不能执行

```
public Object aroundlog(ProceedingJoinPoint proceedingJoinPoint)  {


    //获取方法执行所需的参数
    Object[]args= proceedingJoinPoint.getArgs();

    Object retv=null;

    try{

        //前置通知
        System.out.println("前置通知>>>>>>>>>>>>>");
        retv= proceedingJoinPoint.proceed(args);
        // 后置通知
        System.out.println("后置通知>>>>>>>>>>>>");

    }catch (Throwable throwable){

        //异常通知 事务回滚
        System.out.println("异常通知 事务回滚  》》》》》》》》》》》");
    }finally {
```

```
        //最终通知,
        System.out.println("最终通知, >>>>>>>>>>>>");
    }


   return  retv;
}
```

注解配置环绕通知

logger类

```
package com.baidu.com.baidu.utils;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Component(value = "logger")//注意value值一定要是类名的首字母小写
@Aspect
public class Logger {
    @Pointcut(value = "execution(* com.baidu.service.*.*(..))")
    public  void prtll(){}

/*
    public void beforeprintlog(){
        System.out.println("前置通知  Logger类中的pringLog方法开始记录日志了。。。");
    }



    public void afterprintlog(){
        System.out.println("后置通知 Logger类中的pringLog方法开始记录日志了。。。");
    }


    public void afterthrowingprintlog(){
        System.out.println("抛出异常通知 Logger类中的pringLog方法开始记录日志了。。。");
    }


    public void finallyintlog(){
        System.out.println("最终通知  Logger类中的pringLog方法开始记录日志了。。。");
    }
```

```
*/
    @Around("prtll()")//注意一定要加括号
    public Object aroundlog(ProceedingJoinPoint proceedingJoinPoint)  {


        //获取方法执行所需的参数
        Object[]args= proceedingJoinPoint.getArgs();

        Object retv=null;

        try{

            //前置通知
            System.out.println("前置通知>>>>>>>>>>>>>");
            retv= proceedingJoinPoint.proceed(args);
            // 后置通知
            System.out.println("后置通知>>>>>>>>>>>>>");

        }catch (Throwable throwable){

            //异常通知 事务回滚
            System.out.println("异常通知 事务回滚  》》》》》》》》》》》");
        }finally {
            //最终通知,
            System.out.println("最终通知, >>>>>>>>>>>>>");
        }


       return  retv;
    }

}
```

2bean.xml的配置

```
<!-- 开启 spring 对注解 AOP 的支持 -->
<aop:aspectj-autoproxy/>
```

step3:

```
<!-- 告知 spring, 在创建容器时要扫描的包 -->
<context:component-scan base-package="com.itheima"></context:component-scan>
```

Spring自带的声明式事务管理器

JavaEE 体系进行分层开发，事务处理位于业务层，Spring 提供了分层设计业务层的事务处理解决方 案。

Spring 中事务控制的 API 介绍

PlatformTransactionManager 接口

真正管理事务的对象

org.springframework.jdbc.datasource.DataSourceTransactionManager 使用 Spring JDBC 或 iBatis 进行持久化数据时使用

2.2.2.1 事务的隔离级别

事务隔离级反映事务提交并发访问时的处理态度
- ISOLATION_DEFAULT
  - 默认级别，归属下列某一种
- ISOLATION_READ_UNCOMMITTED
  - 可以读取未提交数据
- ISOLATION_READ_COMMITTED
  - 只能读取已提交数据，解决脏读问题(Oracle默认级别)
- ISOLATION_REPEATABLE_READ
  - 是否读取其他事务提交修改后的数据，解决不可重复读问题(MySQL默认级别)
- ISOLATION_SERIALIZABLE
  - 是否读取其他事务提交添加后的数据，解决幻影读问题

xml配置事务管理器：

```
   <!-- 配置一个事务管理器 -->
<bean id="dataSourceTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!--注入datasource-->
    <property name="dataSource" ref="dataSource"></property>
</bean>



    <!--配置事务通知-->
    <tx:advice id="tad" transaction-manager="dataSourceTransactionManager">
        <!--配置事务的属性-->
        <tx:attributes>
            <!-- 指定方法名称:是业务核心方法
read-only:是否是只读事务。默认 false，不只读。 isolation:指定事务的隔离级别。默认值是使用数据库的默认隔离级别。
propagation:指定事务的传播行为。
timeout:指定超时时间。默认值为:-1。永不超时。
rollback-for:用于指定一 个异常，当 执行产生该 异常时，事 务回滚。产 生其他异常 ，事务不回 滚。没有默认值，任何异常都回滚。 no-rollback-for:用于指定一个异常，当产生该异常时，事务不回滚，产生其他异常时，事务回

滚。没有默认值，任何异常都回滚。
-->
            <tx:method name="*" read-only="false" propagation="REQUIRED"></tx:method>
            <tx:method name="find*" read-only="true" propagation="SUPPORTS">
</tx:method>
```

```
        </tx:attributes>

    </tx:advice>

<aop:config>
    <aop:pointcut id="ptl" expression="execution(* com.baidu.service.*.*(..))"/>
    <!--建立事务与表达式的关系-->
    <aop:advisor advice-ref="tad" pointcut-ref="ptl"></aop:advisor>


</aop:config>
```

aop面试题

aop的底层用的什么实现的?

jdk动态代理 和cglib代理

---

spring AOP 默认使用jdk动态代理还是cglib?

要看条件，如果实现了接口的类，是使用jdk。如果没实现接口，就使用cglib。