# 更新

7月26日重新排整

# spring概念

1、spring是开源的轻量级框架

2、spring核心主要两部分

aop:面向切面编程，扩展功能：可以不修改源代码实现功能

ioc 控制反转：把对象的创建不是通过new方式实现，而是交给spring配置创建类对象

3、spring是一站式框架

(1)spring在javaee三层结构中，每一层都提供不同的解决技术

web层：springmvc，service层：spring的ioc，dao层：spring的jdbcTemplate

## 原理

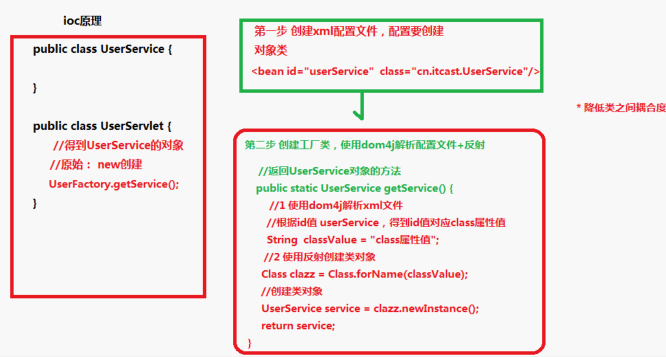
Ioc底层原理使用到的技术

(1)xml配置文件

(2)dom4j解决xml

(3)工厂设计模式 :高内聚解耦合

(4)反射



## Spring需要掌握

1、原理 spring使用web8大监听器的ServletContextListener实现加载

使用ContextLoaderListener 继承ServletContextListener监听器初始化方法来加载spring配置文件，把配置文件配置对象创建，把创建出来的对象放到application域对象里面（setAttribute方法），获取对象时候，到SetvletContext域得到（getArrribute方法）

2、IOC控制反转：把对象的创建不是通过new方式实现，而是交给spring配置创建类对象

3、AOP :面向切面编程，扩展功能：可以不修改源代码实现功能

4、事务的配置

# Spring注入简单步骤

注解方式(包括web开发)

第一步 导入jar包：注意还需要别的jar包： Commons-logging-1.2

commons-logging-1.2.jar 还有spring的：beans、context、core、expression(表达式)、aop

第二步 创建spring配置文件，引入约束

<?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"

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">

开启注解扫描：到包里面扫描类，方法、属性上面是否有注解

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

<context:annotation-config></context:annotation-config>只扫描属性上面的注解：resource，配置了上面的不用配置这个

第三步 注解创建对象：可让此类变成多实例，

类进行写注解service 、controller、Component（通用）、Repository

@Repository

@Scope(value="prototype") //变成多实例操作

public class UsersServlet {

@Resource

private Users users;

public void add() {

System.out.println("我被调用了");

Users.save();

}

}

第四步 加载配置文件：调用操作

第一种：不是web项目加载

ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");

UsersServlet u=( UsersServlet) ac.getBean("usersServlet);

u.add();

第二种：web.xml配置加载配置文件：

1、导包：spring-web-4.3.17.RELEASE.jar

2、寻找： applicationContext.xml文件

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

3、把加载资源放到监听器ServletContext域中

<listener><listener-class>org.springframework.web.context.ContextLoaderListener</listener-class></listener>

常用注解

Scope：等同于xml<bean>中的属性Scope，在类上面使用 需要多实例对象使用

PostConstruct：等同于xml<bean>中的属性init-method，在要初始化的方法上使用

preDestroy：等同于xml<bean>中的属性destroy-method，bean销毁前执行方法

配置方式

第一步 导入jar包： 可以不要aop包

commons-logging-1.2.jar 还有spring的：beans、context、core、expression

expression：在配置文件中，出现了#｛｝形式的表达式，我们就称为Spel表达式。

第二步 自定义类

第一 创建A类

public class A {

public void method() {

System.out.println("我是A类的方法被调用了");

}

}

第二 创建B类 B类中有A类的属性需要调用A类的方法

public class B {

private A a; //维护属性A类，这里不用进行new了，第三步配置文件进行配置操作

public void setA(A a) { // 不是注解方式需要set方法

this.a = a;

}

public void b() {

a.method(); //调用 A类中的方法

}

}

第三步 在类根路径创建核心配置文件application-config.xml

第一 把A类交给spring管理：等同于它帮你进行实例化对象：A a=new A()

<bean id="a" class="com.wwl.A"/>

第二 B类如果有维护A类的操作：官方叫：依赖注入

<bean id="b" class="com.wwl.B">

下面操作等同于B类中的属性：A类进行实例化：A a=new A()

<property name="a" ref="a"></property> name为B类的属性名称，ref调用上面的对象A类的id

</bean>

</beans>

第四步 调用操作

public static void main(String[] args) {

// ClassPathXmlApplicationContext对象寻找配置文件就是在类根路径

ApplicationContext ac = new ClassPathXmlApplicationContext("application-config.xml");

B b=(B) ac.getBean("b");

b.b();

}

# IOC

## 控制反转(创建对象)

控制反转(IOC) (把类的创建对象操作交给Spring管理)

### Xml方式

Xml方式 类中维护的对象需要 set方法才行

第一种正常情况：用无参构造方法创建对象(也是最常用的)

<bean id="a" class="com.wwl.entity.Users" scope="prototype"></bean>

第二种带参构造方法创建对象

<bean id="mservice" class="wwl.service.Mservice">

//参数：name：参数的名字value=八大基本数据类型的值，ref=引用注册的bean，type=类型

<constructor-arg name="wname" value="你好啊" type=”java.lang.String”></constructor-arg>

</bean>

第三种构造方法被私有化的情况：用静态方法来获取本类的对象，

例子：获取A类对象

A类中写法：

public class A {

private A() {}; //私有化构造方法

public static Mservice getMservice() { //使用静态对象获取对象

return new Mservice();

}

public void a() {

System.out.println("呵呵");

}

}

xml

<!-- 参数 factory-method： 类中的方法名-->

<bean id="a" class="wwl.service.A" factory-method="getMservice"/>//这样就获取了a类的对象

第四种情况：一个类中有方法：用来获取别的类的对象（了解）

例子 创建B对象

A类中方法来获取另一个类的对象B类对象

public class A {

private A() {};

public B getMservice() {

return new B(); //第一步 本类中的方法创建另一个对象B类对象

}

}

xml

调用类

<bean id="mcontroller" class="wwl.controller.Mcontroller">

<property name="B" ref="b"></property> //此调用类维护着B类，而不是A类

</bean>

<!-- A类对象需要创建-->

<bean id="a" class="wwl.service.A"/>

第二步 参数：factory-bean：a类的对象，factory-method:a类的方法名

<bean id="b" factory-bean="a" factory-method="getMservice"></bean>

第三步 这样就创建了一个B类的对象

### 注解方式

注解方式 需先在核心配置文件配置<context:component-scan base-package="com.jadyer"/>启动自动扫描

@Service用于标注业务层组件

@Repository用于标注数据访问组件，即DAO组件

@Controller用于标注控制层组件，如Struts中的Action

@Component泛指组件，当组件不要好归类时，可以使用这个注解进行标注

@Service //等同于在xml配置了：<bean id="a" class="com.wwl.dao.impl.A"/>

Public class A{

}

## 依赖注入(成员变量)

依赖注入(DI) 为类中的属性进行注入值

xml配置方式

第一种方式：使用set方法注入属性

步骤有两步：

1、类中需要有set方法

2、<bean id="a" class="com.wwl.entity.Users" scope="prototype">

<property name="属性名称" value="属性值"></property>

<property name="属性名称" ref="对象bean名称引用"></property>

</bean>

第二种方式：带参构造方法为属性注入值：有几个参数写几个标签

<bean id="a" class="com.wwl.entity.Users" scope="prototype">

<constructor-arg name="第一个构造中的参数名称" value="普通属性值" ref="引用bean的id名称"></constructor-arg>

<constructor-arg name="第二个构造中的参数名称" value="普通属性值" ref="引用bean的id名称"></constructor-arg>

</bean>

注解方式 在类中的属性上添加@resource会自动寻找

1、context:component-scan配置

隐式注册了多个对注解进行解析的处理器：

AutowiredAnnotationBeanPostProcessor \CommonAnnotationBeanPostProcessor

PersistenceAnnotationBeanPostProcessor \RequiredAnnotationBeanPostProcessor ………….

其实，注解本身做不了任何事情，和XML一样，只起到配置的作用，主要在于背后强大的处理器

此配置就包括了<context:annotation-config/>的注解所使用的处理器 ，所以配置了<context:component-scan>之后，便无需再配置<context:annotation-config> 了。

2、<context:annotation-config>：注册处理器用的，一般只用于@Resource和@Autowired

@Resource先找id名称，再找类型装备：可以标注在字段或属性的setter方法上，但它默认按名称装备名称可以通过name属性指定，如果没有指定：当注解标注在字段上，即默认取字段的名称作为bean名称寻找依赖对象，当注解标注在属性的setter方法上，即默认去属性名作为名称寻找;

注意：如果没有指定name属性，并且按照默认的名称仍然找不到依赖对象时，@Resource会回退到按类型装配，但一旦指定了name属性，就只能按名称装配了

@Autowired spring提供的，默认类型装配:要求依赖对象必须存在，如允许null值，可以设置它requlred属性为false，如果想用名称装配，可以结合@Qualifier注解一起使用，如下：@Autowired @Qualifier(“personDaoBean”)

IOC和DI区别

IOC：控制反转：把对象创建交给spring进行配置

DI：依赖注入：向类里面的属性中设置值

关系：依赖注入不能单独存在，需要在IOC基础上完成操作

# AOP

Aop控制权限就是基于两个代理实现的：

①当自创类有实现接口调用JDk提供的使用Proxy代理②被代理对象没有实现接口就调用cglib代理

作用：Aop面向切面变成，扩展功能不修改源代码来实现

aop-代理对象分为：静态代理（一般不用），和动态代理

AOP开发步骤

第一步 导包：注意 还需要Aspectj框架的包：aopalliance-1.0.jar、aspectjweaver-1.8.7.jar

commons-logging-1.2.jar 还有spring的：beans、context、core、expression、aop

第二步 导入AOP的xml约束：使用表达式配置切入点

切入点：就是配置要增强的方法

常用的表达式

execution(<访问修饰符>？<返回类型><方法名>(参数)<异常>)

execution(\* com.wwl..\*(..)) 任意修饰符wwl包下的全部类中的全部方法拦截

execution(\* com.wwl.Book.\*(..))：任意修饰符Book类中的全部方法拦截

execution(\* com.wwl.Book.add(..)): 任意修饰符Book类中的add方法拦截

execution(\* save\*(..))：所有save开头的方法都拦截

execution(\* \*.\*(..))：所有的

xml方式操作

第一步：定义目标类（被代理的类）A，A类中的add方法进行(增强、通知)

public class A {

public void add() {

System.out.println("bool的add的方法");

}

}

第二步：定义增强类（执行A类add方法前的逻辑代码方法）

public class B {

public void b() {

System.out.println("前置通知");

}

public void bb() {

System.out.println("后置通知");

}

//环绕通知

public void huanRao(ProceedingJoinPoint pjp) throws Throwable {

//方法之前

System.out.println("方法之前");

//被增强的方法

pjp.proceed();

//方法之后

System.out.println("方法之后");

}

}

第三步：xml中进行配置aop标签

<bean id="a" class="com.wwl.A"></bean>

<bean id="b" class="com.wwl.B"></bean>

<aop:config>

<!-- 配置切入点 -->

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

<!-- 配置切面 :把增强应用到具体方法上面：这过程称为切面 -->

<aop:aspect ref="b">

<!--配置前置通知：b是myBook类中的方法名 -->

<aop:before method="b" pointcut-ref="c"/>

<!--配置后置通知：bb是myBook类中的方法名 -->

<aop:after method="bb" pointcut-ref="c"/>

<!--配置前置通知：huanRao是myBook类中的方法名 -->

<aop:around method="huanRao" pointcut-ref="c"/>

</aop:aspect>

</aop:config>

第四步：调用要被改造的方法即可

public static void main(String[] args) {

ApplicationContext ac=new ClassPathXmlApplicationContext("applicationContext.xml");

A b=(A) ac.getBean("a");

b.add();

}

总结：注意表达式不要写错，不然问题很难找出

注解操作过程

第一步 在核心配置文件中配置：<aop:aspectj-autoproxy/>自动代理（让AOP中的注解生效），

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

<aop:aspectj-autoproxy/>

第二步 创建被代理对象的类 A

@Component

public class A {

public void add() {

System.out.println("bool的add的方法");

}

public String select() { //有返回值

System.out.println("bool的select的方法");

return “曾曾曾”;

}

}

第三步 创建代理对象：定义增强类（执行A类add方法前的逻辑代码方法）

@Component

@Aspect //识别为一个切面类

public class B {

@Before(value="execution(\* com.wwl.A.\*(..))")

public void b() {

System.out.println("前置通知");

}

@AfterReturning(value="execution(\* com.wwl.A.\*(..))")

public void bb() {

System.out.println("后置通知");

}

//环绕通知

@Around(value="execution(\* com.wwl.A.\*(..))")

public void huanRao(ProceedingJoinPoint pjp) throws Throwable {

//方法之前

System.out.println("方法之前");

//被增强的方法

pjp.proceed();

//方法之后

System.out.println("方法之后");

}

//异常抛出通知

@AfterThroeing(value="execution(\* com.wwl.A.\*(..))")

public void dd(){

System.out.println(“异常抛出增强=======”);

}

//最终通知

@After (value="execution(\* com.wwl.A.\*(..))")

public void cc () {

System.out.println("最终通知");

}

-----------------------------------------------------------------------下面有返回值操作--------------------------------------------------------------------

@After (value="execution(\* com.wwl.A.\*(..))" ,returning=”str”)

public void b(Object str) { //注意 这里的 变量名要和returning=”str”一致

System.out.println("后置置通知"+str);

}

注意;此类还要交给spring管理，不然不起作用也就是要在类上面还要加入@Component

首先启动对@Aspect注解的支持：(注解本身是不能干活的，因为后面有处理器进行处理)

第四步 调用要被改造的方法即可

第一种原始测试

public static void main(String[] args) {

ApplicationContext ac=new ClassPathXmlApplicationContext("applicationContext.xml");

A b=(A) ac.getBean("a");

b.add();

}

第二种 配合junit使用

@Runwith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(“classpath: applicationContext.xml”)

public class Demo{

@Resource

private A a;

public void demo1(){

b.add();

}

}

传智另一个老师讲的增强类通用配置注解

1、声明为切面：在普通的java类（不是接口）加入@Aspect，声明它是切面

@Component

@Aspect

public class A(){}

2、定义切入点：创建一个方法，方法上面加入：@pointcut，它就是一个切入点

@pointcut（"execution(\* cn.wwl.service..\*.\*(..))"）

private void anyMethod（）{}

3、定义前置通知：方法上面加入：Before（"anyMethod（）"这里面是切入点的方法名称），

Before（"anyMethod（）&& args(name)"）意思是既要满足切入点条件，也要满足参数中只有一个名字要和name相同

@ Before（"anyMethod（））

public void doAccessCheck（String name）{

System.out.println("前置通知");

4、定义后置通知 ：方法上面加入：@afterReturning（"anyMethod（）"这里面是切入点的方法名称），

@afterReturning（pointcut="anyMethod（）",treturning="result"）意思是返回值是result，相当于execution(Sting cn.wwl.service..\*.\*(..))，说白点就是终极目标方法要带有返回值是String的，才会执行，同时后置通知还会收到目标方法返回来的参数

@afterReturning（pointcut="anyMethod（）",treturning="result"）

public void doAccessCheck（String result）{

System.out.println("后置通知");

5、定义最终通知 ：方法上面加入：@After（"anyMethod（）"这里面是切入点的方法名称），

@After（"anyMethod（）"）

public void doAccessCheck（）{

System.out.println("最终通知");

6、定义例外通知 ：方法上面加入：@AfterThrowing（"anyMethod（）"这里面是切入点的方法名称），

@AfterThrowing（"pointcut="anyMethod（）",throwing="e"）

public void doAccessCheck（Exception e）{

System.out.println("例外通知、异常通知");

7、定义环绕通知 ：方法上面加入：@Around（"anyMethod（）"这里面是切入点的方法名称），

@Around（"anyMethod（）"）

public Object doAccessCheck（ProceedingJoinPoint pjp）throws Throwable{

if(){//用此方法非常适合判断是否有权限

Object result =pjp.proceed();执行这个方法如果后面还有切面的话，执行顺序会变的，先执行后面的切面，如果没有切面了，再执行最终目标对象的业务方法，如果你不调用此方法，业务bean的方法都不会执行

}

System.out.println("进入方法");

Object result =pjp.proceed();

System.out.println("退出方法");

return result;

aop概念

横切性关注点：核心关注点和横切关注点。对于核心关注点而言，通常来说，实现这些关注点的模块是相互独立的，他们分别完成了系统需要的商业逻辑，这些逻辑与具体的业务需求有关。而对于日志、安全、持久化等关注点而言，他们却是商业逻辑模块所共同需要的，这些逻辑分布于核心关注点的各处。在AOP 中，诸如这些模块，都称为横切关注点。应用AOP 的横切技术，关键就是要实现对关注点的识别。

这四点要记住：连接点、切入点、通知、切面

joinpoint（连接点）：类里面的方法可以被增强的方法，这些方法称为连接点

pointcut（切入点）：类中方法看你选择哪个想被增强（表达式配置）比如实际操作中，只是增强了类里面add方法和update方法，增强的方法称为切入点

adcice（通知，增强）：增强的逻辑，称为增强，比如扩展日志功能，这个日志功能就成为增强

aspect（切面）：把增强应用到具体方法上面：这过程称为切面

target（目标对象）：代理的目标对象（）

weace（织入）：指‘增强’应用到‘要增强的类’对象并导致proxy（改造后的）对象创建的过程称为织入

Introduction（引入）：在不修改类代码的前提下，可以在运行期为类动态地添加一些方法或Field。（就是在代理对象中进行赋值）

Aspectj框架

Aspectj是一个面向切面的框架，它扩展了Java语言，Aspectj定义了AOP语法所以它有一个专门的编译器用来生成遵守Java字节编码规范的Class文件

Aspectj是一个基于Java语言的Aop框架

Spring2.0以后新增了对Aspectj切点表达式支持

@Aspectj是Aspectj1.5新增功能，通过JDK5注解技术，允许直接在Bean类中定义切面

新版本spring框架，建议使用Aspectj方式开发AOP

使用Aspectj需要导入Spring AOP和Aspectj相关JAR包

# 事务

spring事务管理两种方式

第一种 编程式事务管理（了解）就是纯代码写

第一步：配置平台事务管理器

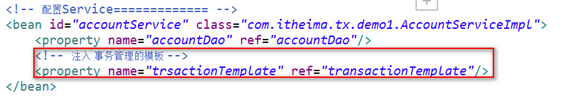


第二步：Spring提供了事务管理的模板类

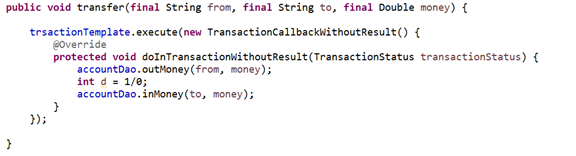
 配置事务的管理的模板类



第三步：在业务层注入事务管理的模板



编写事务管理的代码



第二种 声明式事务管理

基于xml配置文件实现

基于注解实现

事务配置操作

基于xml配置文件实现：转账操作

第一步 导入jar包

spring-tx-4.3.17.RELEASE.jar

第二步 写类service层AService类和dao层ADao类。方法：xiaoMing()是增加2元，xiaoLiang()是减少2元

public class AService {

private ADao aDao;

public void setaDao(ADao aDao) {

this.aDao = aDao;

}

public void getService() {

aDao.xiaoMing();

//制造异常

int i=9/0;

aDao.xiaoLiang();

}

}

Dao层：配合Spring的 jdbc模板：JdbcTemplate操作数据库

--------------------------------------------------------------新的spring的jdbc模板类开始------------------------------------------------------------------

//2018年10月16日看了新视频：可以继承JdbcTemplate模板此类，此类在xml中注入：DataSource即可

public class Adao extends JdbcTemplate{

public void xiaoMing(){

this.update("UPDATE ceshi SET money= money-? where name=?",2,"小明");

}

}

--------------------------------------------------------------新的spring的jdbc模板注入方式结束------------------------------------------------------------------

public class ADao {

private JdbcTemplate jt;

public void setJt(JdbcTemplate jt) {

this.jt = jt;

}

public void xiaoMing(){

jt.update("UPDATE ceshi SET money= money-? where name=?",2,"小明");

}

public void xiaoLiang(){

jt.update("UPDATE ceshi SET money= money+? where name=?",2,"小亮");

}

第三步 application.xml中配置

--------------------------------------------------------------新的spring的jdbc模板注入方式开始------------------------------------------------------------------

2018年10月16日新视频jdbc模板注入方式：在ADao类中继承JdbcTemplate，xml注入datasource即可

<!--注册自己实体类：只有这一步即搞定spring的jdbc模板的注入-->

<bean id="aDao" class="com.wwl.ADao">

<property name="dataSource" ref=" dataSource "/>

</bean>

--------------------------------------------------------------新的spring的jdbc模板注入方式结束------------------------------------------------------------------

<!-- 创建Jdbc模板对象(注册JdbcTemplate对象)-->

<bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">

<property name="dataSource" ref="dataSource"></property>

</bean>

<!-- 创建自己写的类对象（注册自己实体类）-->

<bean id="aDao" class="com.wwl.ADao">

<!—aDao类中需要： 注入属性JdbcTemplate -->

<property name="jt" ref="jt"/>

</bean>

<bean id="aService" class="com.wwl.AService">

<property name="aDao" ref="aDao"></property>

</bean>

<!-- 创建c3p0连接池对象-->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql:///wwl"></property>

<property name="user" value="root"></property>

<property name="password" value="123456"></property>

</bean>

事务配置操作：

<!-- 第一步配置（创建）事务管理器：DataSourceTransactionManager -->

注意：这里的是JdbcTemplate的事务对象，其它如hibernate有不同的事务对象

<bean id="transaction"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager" >

此对象里面有属性数据源，需配置上面的连接池（数据源）

<property name="dataSource" ref="dataSource"></property>

</bean>

<!-- 第二步 配置事务增强：类中刷选那个方法需要进行事务-->

<tx:advice id="txadvice" transaction-manager="transaction">

<tx:attributes>

<!—name：可以方法名，或通配符：add\* -->

<tx:method name="getService"/>

<!-- <tx:method name=""/> -->

</tx:attributes>

</tx:advice>

<!--第三步：整个工程进行刷选那个包，包中的那个类需要此事务支持-->

<aop:config>

<!-- 配置切入点：配置要增强的方法，这个方法叫切入点-->

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

<!-- 配置切面 :把增强应用到具体方法上面：这过程称为切面 -->

<aop:advisor advice-ref="txadvice" pointcut-ref="p"/>

</aop:config>

第四步 测试调用：没有配置事务出现异常：减少，不会增加，配置事务回滚，没有变化

public static void main(String[] args) {

ApplicationContext ac=new ClassPathXmlApplicationContext("applicationContext.xml");

AService as=(AService) ac.getBean("aService");

as.getService();

}

基于注解实现：转账操作

第一步 application.xml文件配置：

<context:component-scan base-package="com.wwl"/> 开启扫描注册bean

<!-- 创建c3p0连接池对象-->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql:///wwl"></property>

<property name="user" value="root"></property>

<property name="password" value="123456"></property>

</bean>

<!--（创建JdbcTemplate对象） 配置JDBC模板：连接数据库 -->

<bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">

<property name="dataSource" ref="dataSource"></property>

</bean>

配置事务管理器 不同操作数据库的对象都有不同的事务管理器，下面是JdbcTemplate的事务管理器

<bean id="transaction"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager" >

<property name="dataSource" ref="dataSource"></property>

</bean>

<!-- 开启事务注解生效 需注入事务管理器-->

<tx:annotation-driven transaction-manager="transaction"/>

</beans>

第二步 类中的操作

@Transactional

public class TestServiceBean implements TestService {

private TestDao dao;

public void setDao(TestDao dao) {

this.dao = dao;

}

@Transactional(propagation =Propagation.NOT\_SUPPORTED) 第一种情况 配置类中不需要事务的方法

public List getAll() {

return null;

}

//方法上注解属性会覆盖类注解上的相同属性

@Transactional(readOnly = false, propagation = Propagation.REQUIRES\_NEW)//第二种 配置新的事务

public void updateFoo(Foo foo) {

// do something

}

}

**@Transactional注解中常用参数说明**

属性 类型 描述

value String 可选的限定描述符，指定使用的事务管理器

propagation enum: Propagation 可选的事务传播行为设置

isolation enum: Isolation 可选的事务隔离级别设置

readOnly boolean 读写或只读事务，默认读写

timeout int (in seconds granularity) 事务超时时间设置

rollbackFor Class对象数组，必须继承自Throwable 导致事务回滚的异常类数组

rollbackForClassName 类名数组，必须继承自Throwable 导致事务回滚的异常类名字数组

noRollbackFor Class对象数组，必须继承自Throwable 不会导致事务回滚的异常类数组

noRollbackForClassName 类名数组，必须继承自Throwable 不会导致事务回滚的异常类名字数组

事物传播行为介绍: 事物超时设置:@Transactional(timeout=30) //默认是30秒

　　@Transactional(propagation=Propagation.REQUIRED) ：如果有事务, 那么加入事务, 没有的话新建一个(默认情况下)

　　@Transactional(propagation=Propagation.NOT\_SUPPORTED) ：容器不为这个方法开启事务

　　@Transactional(propagation=Propagation.REQUIRES\_NEW) ：不管是否存在事务,都创建一个新的事务,原来的挂起,新的执行完毕,继续执行老的事务

　　@Transactional(propagation=Propagation.MANDATORY) ：必须在一个已有的事务中执行,否则抛出异常

　　@Transactional(propagation=Propagation.NEVER) ：必须在一个没有的事务中执行,否则抛出异常(与Propagation.MANDATORY相反)

　　@Transactional(propagation=Propagation.SUPPORTS) ：如果其他bean调用这个方法,在其他bean中声明事务,那就用事务.如果其他bean没有声明事务,那就不用事务.

数据库读的三种问题

MYSQL: 默认为REPEATABLE\_READ级别

SQLSERVER: 默认为READ\_COMMITTED

脏读 : 一个事务读取到另一事务未提交的更新数据

不可重复读 : 一个事务读到另一个事务已经提交的update的数据，导致一个事务中多次查询结果不一致

虚读、幻读 : 一个事务读到另一个事务已经提交的insert的数据，导致一个事务中多次查询结果不一致。

事务隔离级别:解决三种读的问题

@Transactional(isolation = Isolation.READ\_UNCOMMITTED)：读取未提交数据(会出现脏读, 不可重复读) 基本不使用

@Transactional(isolation = Isolation.READ\_COMMITTED)：读取已提交数据(会出现不可重复读和幻读)

@Transactional(isolation = Isolation.REPEATABLE\_READ)：可重复读(解决脏读和不可重复读，虚读、幻读有可能发生)

@Transactional(isolation = Isolation.SERIALIZABLE)：单线程 解决所有读问题

spring的事务管理API

Spring 主要3个事务对象

1. PlatformTransactionManager(接口)：平台事务管理器

Spring针对不同的dao层框架，提供接口不同的实现类

DataSourceTransactionManager ：底层使用JDBC管理事务，mybatis使用此类

HibernateTransactionManager ：底层使用Hibernate管理事务

2、TransactionDefinition：事务定义信息（隔离、传播、超时、制度）

事务定义：用于定义事务的相关的信息，隔离级别、超时信息、传播行为、是否只读

传播行为介绍：

Spring中提供了七种事务的传播行为：

第一种情况：保证多个操作在同一个事务中

PROPAGATION\_REQUIRED ：默认值：如果A中有事务，使用A事务，如果A没有，创建新事务，将操作包含进来

通俗讲：B方法要调用的另一个类中A方法：A有事务使用A事务执行后续操作，没有则创建新事务执行后续操作

PROPAGATION\_SUPPORTS ：支持事务，如果A中有事务，使用A中的事务。如果A没有事务，不使用事务。

PROPAGATION\_MANDATORY：如果A中有事务，使用A中的事务。如果A没有事务，抛出异常。

第二种情况：保证多个操作不在同一个事务中

PROPAGATION\_REQUIRES\_NEW ：如果A有事务，将A的事务挂起（暂停），创建新事务，只包含自身操作。如果A中没有事务，创建一个新事务，包含自身操作。

通俗讲：B方法要调用的另一个类中A方法：不管A有没有事务，创建新事务执行后续操作(不包含调用A方法)，

PROPAGATION\_NOT\_SUPPORTED：如果A中有事务，将A的事务挂起。不使用事务管理。

PROPAGATION\_NEVER：如果A中有事务，报异常。

第三种情况：嵌套式事务

PROPAGATION\_NESTED ：嵌套事务，如果A中有事务，按照A的事务执行，执行完成后，设置一个保存点，执行B中的操作，如果没有异常，执行通过，如果有异常，可以选择回滚到最初始位置，也可以回滚到保存点。

3、TransactionStatus:事务具体运行状态

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。

commit()：用于提交TransactionStatus参数代表的事务，具体语义请参考Spring Javadoc；

rollback()：用于回滚TransactionStatus参数代表的事务

getPropagationBehavior()：返回定义的事务传播行为；

getIsolationLevel()：返回定义的事务隔离级别；

getTimeout()：返回定义的事务超时时间；

isReadOnly()：返回定义的事务是否是只读的；

getName()：返回定义的事务名字。

isNewTransaction()：返回当前事务状态是否是新事务；

hasSavepoint()：返回当前事务是否有保存点；

setRollbackOnly()：设置当前事务应该回滚；

isRollbackOnly(()：返回当前事务是否应该回滚；

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；

isCompleted():当前事务否已经完成。

3个事务管理API的关系

Spring进行事务管理的时候，首先平台事务管理器根据事务定义信息进行事务的管理，在事务管理过程中，产生各种状态，将这些状态的信息记录到事务状态的对象中。

# 核心XML配置

## Bean标签

属性

1、ID：为bean取名，id本身就是xml的一个属性，它会受到xml解析器验证，id是不能包含特殊字符的

2、Class：要被spring进行管理的类，全路径名

3、Scope:作用域，值：

① singleton默认值单例，

②prototype：多例模式(这种情况是在调用getBean进行实例化)，

③Request：创建对象把对象放到request域中

④Session：对象放到Session域中

⑤global Session：web项目中，应用在Porlet环境，如果没有此环境那么相当于session(好比单点登录)

应用场景：struts2的action类就要用到

ApplicationContext c=new ClassPathXmlApplicationContext("application.xml");

T test=(T) c.getBean("cc");// prototype调用它才实例化

4、Abstract：变成抽象类

5、lazy-init：在每一个spring ioc容器中一个bean定义只有一个对象实例，默认情况下会在容器启动时初始化bean

我们可以指定Bean lazy-init="true"来延迟初始化bean，这时候只有第一次获取bean才会初始化bean，如想对所有bean都应用延迟初始化，可在根节点beans进行设置：default-lazy-init=”true”

还有web中的：

6、init-method：用来初始化方法，随着对象实例化而立即执行：init-method=”方法名”

7、destroy-method：没有人为的话，默认一直在spring中，随着spring正常关闭才会销毁

8、Name：和id一样的，它是spring提供的属性，可以使用特殊字符,现在不用了，遗留问题：提供struts1使用的

注意：接收参数要AbstractApplicationContext才有调用close的方法

AbstractApplicationContext c=new ClassPathXmlApplicationContext("application.xml");

c.close();//正常关闭spring容器

## 注册扫描器

<contextcomponent-scan> 扫描类中带有注解的类进行注册

注意：配置了这个标签后，就不用配置contextannotation-config标签：因为已经包含了此标签

spring可以自动去扫描base-pack下面或者子包下面的java文件，如果扫描到有@Component @Controller@Service等这些注解的类，则把这些类注册为bean

两个子标签

1. contextinclude-filter

2. contextexclude-filter

## 获取propertiest配置文件

<context:property-placeholder> 用来获取propertiest文件的内容

<context:property-placeholder

location=" classpath:db.properties " 表示属性文件位置，多个之间通过如逗号/分号等分隔；

file-encoding="" 文件编码；

ignore-resource-not-found="" 如果属性文件找不到，是否忽略，默认false，即不忽略，找不到将抛出异常

ignore-unresolvable="" 是否忽略解析不到的属性，如果不忽略，找不到将抛出异常

properties-ref="" 本地java.util.Properties配置

local-override="" 是否本地覆盖模式，如果true，那么properties-ref的属性将覆盖location加载的属性

order="" 当配置多个<context:property-placeholder/>时的查找顺序

system-properties-mode="" 系统属性模式，ENVIRONMENT（默认），NEVER，OVERRIDE

ENVIRONMENT：将使用Spring 3.1提供的PropertySourcesPlaceholderConfigurer

NEVER：只查找properties-ref、location；

OVERRIDE： PropertyPlaceholderConfigurer使用，因为在spring 3.1之前版本是没有

例子：

properties文件写法：注意 key 前面最好加个前缀，不然spring有时候不识别(可能是关键字)

t.driverClass=com.mysql.jdbc.Driver 注意 key 最后加个前缀：t.driverClass、

t.url=jdbc:mysql:///wwl

t.username=root

t.password=123456

spring中引用：

导入properties：

<context:property-placeholder location="classpath:mysql-config.properties"/>

引用：

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="${t.driverClass}"></property>

<property name="jdbcUrl" value="${t.url}"></property>

<property name="user" value="${t.username}"></property>

<property name="password" value="${t.password}"></property>

</bean>

# SSH整合

前期准备

SSH：struts2，Hibernate，spring

版本： struts-2.3.24，spring-core-4.3.17.RELEASE.jar，hibernate-core-5.0.8.Final.jar

第一步：导入需要的jar包

Struts：

1、导入struts-2.3.24中的struts-2.3.24\apps示例程序struts2-blank.war里面的lib全部

2、整合包：解压根目录lib找到struts2-spring-plugin-2.3.24.jar

Hibernate：

1、导入hibernate-release-5.0.8.Final.zip中的lib\required文件夹下所有

2、导入Hibernate-release-5.0.7.Final.zip中的lib\Jpa文件下只有一个的jar

3、其它 jar：因为使用Hibernate时候，有日志信息输出，hibernate本身么有日志输出的jar包。导入其它日志的jar包：log4J-1.2.17,slf4j-api-1.6.1,slf4j-logj12-.17.2

4、整合包：在spring包中：spring-orm-4.3.17.RELEASE.jar

Spring：

1、Aop、beans、context、core、expression、web、tx、orm jdbc

2、log4j包：commons-logging-1.2.jar，log4j-1.2.17.jar（hibernate有重复）

数据源（连接池）：c3p0-0.9.5.2.jar、mchange-commons-java-0.2.11、c3p0-oracle-thin-extras-0.9.5.2

数据库：mysql-connector-java-5.1.46.jar

第二步：整合struts与spring：

创建action类：没有变化

package com.wwl.action;

import com.opensymphony.xwork2.ActionSupport;

public class UserAction extends ActionSupport{

//struts.xml中的action标签没有配置method的话默认进入此方法，

public String execute() throws Exception {

System.out.println("进来了吗");

return NONE;

}

public String add() throws Exception {

System.out.println("进来了吗dded");

return NONE;

}

}

Spring的application.xml文件配置：要变成多例模式

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

<bean id="userAction" class="com.wwl.action.UserAction" scope="prototype">

</bean>

Struts的struts.xml文件配置：这里有不同之处

小知识：此DTD约束可以在导入包的核心文件中找到，package中extends值忘记了也有

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE struts PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"

"http://struts.apache.org/dtds/struts-2.3.dtd">

<struts>

<package name="test" extends="struts-default" namespace="/">

<!-- 整合后的不同之处：class这个属性不是写类全路径了变成application.xml中bean标签的id名称 -->

<action name="\*" class="userAction" method="{1}">

<!-- <result name=""></result> -->

</action>

</package>

</struts>

web.xml配置：没有变化

寻找applicationContext.xml文件

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

把解析的资源文件加载到servletContext域中

<listener><listener-class>org.springframework.web.context.ContextLoaderListener</listener-class></listener>

配置struts过滤器加载类根路径的struts.xml配置文件

<filter>

<filter-name>struts</filter-name>

<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>struts</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

第三步：spring和Hibernate整合两种方式：

application.xml配置： 变化大

不同点：数据源c3p0的配置：不在hibernate中配置了

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"></property>

<property name="user" value="root"></property>

<property name="jdbcUrl" value="jdbc:mysql:///wwl"></property>

<property name="password" value="123456"></property>

</bean>

SessionFactory：之前hibernate使用代码两步进行获取的，现把方法封装在LocalSessionFactoryBean类中

<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

<property name="dataSource" ref="dataSource"></property>

第一种 引入hibernate-cfg.xml

<property name="configLocations" value="classpath:hibernate-cfg.xml"></property>

第二种：hibernate-cfg.xml都不用写了：把它的配置信息都这里写

<!-- 配置hibernate信息，可选的 -->

<property name="hibernateProperties">

<props>

<prop key="hibernate.show\_sql">true</prop>

<prop key="hibernate.format\_sql">true</prop>

<prop key="hibernate.hbm2ddl.auto">update</prop>

<prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>

</props>

</property>

<!-- 配置映射文件引入 -->

<property name="mappingResources">

<list>

<value>com/wwl/entity/User-hbm.xml</value>

</list>

</property>

</bean>

HibernateTemplate模板：对hibernate方法进行了封装

<bean id="hibernateTemplate" class="org.springframework.orm.hibernate5.HibernateTemplate">

<property name="sessionFactory" ref="sessionFactory"></property>

</bean>

Hibernate事务管理器

<bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">

<property name="sessionFactory" ref="sessionFactory"></property>

</bean>

事务注解生效

<tx:annotation-driven transaction-manager="transactionManager"/>

以下都是service、dao、aciton、三层和entity实体类的注册

<bean id="userAction" class="com.wwl.action.UserAction">

<property name="userService" ref="userService"></property>

<property name="user" ref="user"></property>

</bean>

<bean id="userService" class="com.wwl.service.UserServiceImpl">

<property name="userDao" ref="userDao"></property>

</bean>

<bean id="userDao" class="com.wwl.dao.UserDaoImpl">

<property name="hibernateTemplate" ref="hibernateTemplate"></property>

</bean>

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

Hibernate-cfg.xml 配置文件：有变化看红色的

<session-factory>

<!-- 控制台显示底层sql语句 -->

<property name="hibernate.show\_sql">true</property>

<!-- 对底层的sql语句进行格式化：也就是显示的sql语句排列好给你看 -->

<property name="hibernate.format\_sql">true</property>

<!-- hibernate帮创建表，需要配置下面:hibernate.hbm2ddl.auto

update:如果已经有表，更新，如果没有，创建 -->

<property name="hibernate.hbm2ddl.auto">update</property>

<!-- 配置数据库方言:如：

在mysql里面实现分页关键字limit，只能msyql使用，在oracle数据库，是rownum

让hibernate框架识别不同数据库的自己特有的语句 -->

<property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>

<!-- 绑定本地线程：这里出现问题：配置好了事务无效，两种解决方式：

第一种方式：值从thread改为：org.springframework.orm.hibernate5.SpringSessionContext，这样不会报错，

第二种方式：不进行配置此绑定线程

打印：Illegal access: this web application instance has been stopped already...

解决方案：修改tomcat目录下的server.xml,找到<Context>标签，把reloadble的属性值设为：reloadable="false"，即<Context reloadable="false">。

tomcat重新装载web应用程序失败导致的。当应用程序卸载时，并不会关闭所有的线程。当tomcat已经关闭了其类加载器后，一些线程依然会继续运行，

这样就导致出错。不过这个不影响正常使用，不管影响不影响，看到异常信息就不顺眼。

<property name="hibernate.current\_session\_context\_class">org.springframework.orm.hibernate5.SpringSessionContext</property>

实体类映射文件配置

<mapping resource="com/wwl/entity/User-hbm.xml"/>

</session-factory>

实体类的映射文件的配置：无变化

<hibernate-mapping>

<class name="com.wwl.entity.User" table="user">

<id name="id" column="id">

<generator class="native"></generator>

</id>

<property name="name" column="name"></property>

<property name="gender" column="gender"></property>

<property name="age" column="age"></property>

</class>

</hibernate-mapping>

Dao类层和setvice层类：

Service

private UserDao userDao;

public void setUserDao(UserDao userDao) {

this.userDao = userDao;

}

@Transactional

public void addService(User user) {

userDao.addDao(user);

}

Dao层

public class UserDaoImpl implements UserDao {

private HibernateTemplate hibernateTemplate;

public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {

this.hibernateTemplate = hibernateTemplate;

}

Hibernate再次封装模板的调用方法差不多

public void addDao(User user) {

hibernateTemplate.save(user);

}

# Log4j(日志使用)

spring框架整合了log4j使用非常简单了

log4j的使用需要jar包2个：

commons-logging-1.2.jar 这就是为什么一定要导入此包，spring整合了日志功能

log4j-1.2.17.jar 导入日志核心包

使用步骤：

第一步 导入log4j的jar包：log4j-1.2.17.jar

第二步 复制log4j的配置文件：log4j.properties，复制到src下面

这两步：控制台就有详细的日志输出，要把日志内容写入文件需要在配置文件写

注意：网上看到也可以配置监听类Log4jConfigListener此类在spring-web的jar包util中

log4j.properties内容

### 设置###

log4j.rootLogger = debug,stdout,D,E 可以设置info和debug

### 输出信息到控制抬 ###

log4j.appender.stdout = org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target = System.out

log4j.appender.stdout.layout = org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern = [%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} method:%l%n%m%n

### 输出DEBUG 级别以上的日志到=E://logs/error.log ###

log4j.appender.D = org.apache.log4j.DailyRollingFileAppender

log4j.appender.D.File = E://logs/log.log //log文件的路径设置

log4j.appender.D.Append = true

log4j.appender.D.Threshold = DEBUG

log4j.appender.D.layout = org.apache.log4j.PatternLayout

log4j.appender.D.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n

### 输出ERROR 级别以上的日志到=E://logs/error.log ###

log4j.appender.E = org.apache.log4j.DailyRollingFileAppender

log4j.appender.E.File =E://logs/error.log

log4j.appender.E.Append = true

log4j.appender.E.Threshold = ERROR

log4j.appender.E.layout = org.apache.log4j.PatternLayout

log4j.appender.E.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n

# 其它

Spring包的含义

1.spring-aop-3.2.2.jar

包含在应用中使用spring的aop特性时所需的类。

2.spring-aspects-3.2.2.jar

提供对aspectj的支持，以便可以方便的将面向方面的功能集成进ide中

3.spring-beans-3.2.2.jar

ioc（依赖注入）的基础实现，所有应用都要用到的，它包含访问配置文件、创建和管理bean以及进行inversion of control / dependency injection（ioc/di）操作相关的所有类。但是这个是个基础实现，一般我们在实际的开发过程中很少直接用到，它是对起到支撑作用的。

4.spring-context-3.2.2.jar

为spring核心提供了大量扩展。可以找到使用spring applicationcontext特性时所需的全部类，jdni所需的全部类，ui方面的用来与模板(templating)引擎如 velocity、freemarker、jasperreports集成的类，以及校验validation方面的相关类，还有ejb,cache,format,jms等等。

5.spring-context-support-3.2.2.jar

spring-context 的扩展支持，用于 mvc 方面

6.spring-core-3.2.2.jar

spring的核心包，包含spring框架基本的核心工具类，spring其它组件要都要使用到这个包里的类，是其它

组件的基本核心。包括asm，cglib以及相关的工具类

7.spring-expression-3.2.2.jar

spring表达式语言。

8.spring-instrument-3.2.2.jar

spring对服务器的代理接口。

9.spring-instrument-tomcat-3.2.2.jar

spring对tomcat连接池的集成。

10.spring-jdbc-3.2.2.jar

spring对jdbc的简单封装

11.spring-jms-3.2.2.jar

spring对jms(java message service)的封装，为了简化对jms api的使用

12.spring-orm-3.2.2.jar

包含spring对dao特性集进行了扩展，使其支持 ibatis、jdo、ojb、toplink，因为hibernate已经独立成包了，现在不包含在这个包里了。这个jar文件里大部分的类都要依赖spring-dao.jar里的类，用这个包时你需要同时包含spring-dao.jar包。spring 整合第三方的 orm 映射支持，如 hibernate 、ibatis、jdo 以及spring的jpa的支持。

13.spring-oxm-3.2.2.jar

spring 对object/xmi 的映射的支持，可以让java与xml之间来回切换。

14.spring-struts-3.2.2.jar

struts框架支持，可以更方便更容易的集成struts框架。

15.spring-test-3.2.2.jar

spring对junit框架的简单封装。

16.spring-tx.3.2.2.jar

spring提供对事务的支持，事务的相关处理以及实现类就在这个jar包中

17.spring-web-3.2.2.jar

包含web应用开发时，用到spring框架时所需的核心类，包括自动载入webapplicationcontext特性的类、struts与jsf集成类、文件上传的支持类、filter类和大量工具辅助类。

18.spring-webmvc-3.2.2.jar

spring mvc相关，实现springmvc的操作。

19.spring-webmvc-portlet-3.2.2.jar

spring mvc的增强扩展。

20: commons-logging-1.1.1

它不在整合包中：用来记录程序运行时的活动的日志记录。

在java平台中，有几个可以选择的日志记录的实现（log4j，JDK Logging API）