Spring

## 1.简介与概念

## 2.IOC(控制反转(依赖注入)):

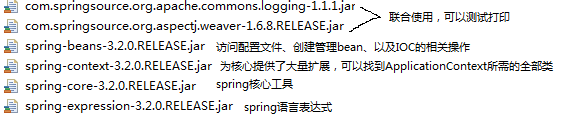
IOC : inverse of Control 控制反转:对组件对象控制权的转移,从程序代码本身转移到了容器。

依赖注入: 应用组建不应该负责查找资源或者其他依赖的协作对象。配置对象的工作应该由IOC负责，“查找资源”的逻辑应该从应用组件的代码中抽取出来，交给IOC容器负责。

组件 : 对数据与方法的简单封装。

1. **Spring的bean管理(xml配置)**

**导入jar**



一.使用无參构造创建对象

1. 编写实体类

**public** **interface** UserDao {

**public** String sayHello();

}

**public** **class** UserDaoImpl **implements** UserDao{

@Override

**public** String sayHello() {

**return** "Hello World";

}

}

1. 配置xml(把new的过程交给spring实现)

id:根据id属性得到配置对象，id属性不能相同，不能有特殊符号

class ： 全类名

<bean id=*"userDaoImpl"* class=*"com.zhuru.one.UserDaoImpl"*></bean>

1. 编写测试类

ApplicationContext ac;

@Before

**public** **void** setUp() **throws** Exception {

//获取xml的路径

ac = **new** ClassPathXmlApplicationContext("beans.xml");

}

@Test

**public** **void** test() {

UserDaoImpl udi = (UserDaoImpl) ac.getBean("userDaoImpl");

System.***out***.println(udi.sayHello());

}

1. 使用静态工厂创建对象

（1）创建工厂类，在工厂类提供静态的方法，这个方法返回类对象

（2）调用工厂类的方法时候，直接使用类名.方法名称可以调用

java代码

**public** **class** JFactory {

**public** **static** UserDaoImpl getUDI(){

**return** **new** UserDaoImpl();

}

}

xml配置

<bean id=*"jFactory"* class=*"com.wucan.one.JFactory"* factory-method=*"getUDI"*></bean>

测试

@Test

**public** **void** test2(){

ac.getBean("jFactory", UserDaoImpl.**class**).*Jtest*();

}

1. 使用实例工厂创建对象

java代码

**public** **class** SFactory {

**public** UserDaoImpl getUDI(){

**return** **new** UserDaoImpl();

}

}

xml配置

<!-- 使用实例工厂创建对象 -->

<bean id=*"sFactory"* class=*"com.wucan.one.SFactory"*></bean>

<bean id=*"userDaoImpl2"* factory-bean=*"sFactory"* factory-method=*"getUDI"*></bean>

测试

//使用实例工厂创建对象

@Test

**public** **void** test3(){

ac.getBean("userDaoImpl2", UserDaoImpl.**class**).*Jtest*();

}

1. **注入(给属性赋值)**
2. **set注入**

**java代码**

**有一个person类，里面有有属性 name 与 id ，私有化提供set与get方法，重写toString**

**private** Person person;

**private** String name;

**private** **int** id;

**private** List<String> list;

**private** Map<String, String> map;

**private** String[] string;

**private** Set<String> set;

**private** List<Person> lp;

**private** Set<Person> sp;

**private** Map<String, Person> mp;

私有化，提供set与get，重写toString

XML

<!--set注入-->

<bean id=*"person"* class=*"com.wucan.one.Person"*>

<property name=*"name"* value=*"小李"*></property>

<property name=*"id"* value=*"18"*></property>

</bean>

<bean id=*"person2"* class=*"com.wucan.one.Person"*>

<property name=*"name"* value=*"小华"*></property>

<property name=*"id"* value=*"15"*></property>

</bean>

<bean id=*"testPerson"* class=*"com.wucan.one.TestPerson"*>

<!--注入对象，注入这个对象之前必须给属性赋值 -->

<!--name:代表在此类中的对象名

ref：引入目标对象

-->

<property name=*"person"* ref=*"person"*></property>

<!--直接利用set注入给属性赋值 -->

<property name=*"name"* value=*"小茗"*></property>

<property name=*"id"* value=*"1"*></property>

<!--给list集合赋值 -->

<property name=*"list"*>

<list>

<value>1</value>

<value>1</value>

<value>1</value>

</list>

</property>

<!--给map集合赋值 -->

<property name=*"map"*>

<map>

<entry key=*"iphoneNumber"* value=*"13437278267"*/>

<entry key=*"address"* value=*"海口"* />

</map>

</property>

<!--给数组赋值 -->

<property name=*"string"*>

<list>

<value>7</value>

<value>8</value>

<value>9</value>

</list>

</property>

<!--给set集合赋值 -->

<property name=*"set"*>

<set>

<value>4</value>

<value>5</value>

<value>6</value>

</set>

</property>

<property name=*"lp"*>

<list>

<ref bean=*"person"*/>

<ref bean=*"person2"*/>

</list>

</property>

<property name=*"sp"*>

<set>

<ref bean=*"person"*/>

<ref bean=*"person2"*/>

</set>

</property>

<property name=*"mp"*>

<map>

<entry key=*"1"* value-ref=*"person"*/>

<entry key=*"2"* value-ref=*"person2"*/>

</map>

</property>

</bean>

测试

TestPerson p = ac.getBean("testPerson", TestPerson.**class**);

System.***out***.println(p.toString());

1. **构造注入**

**java代码**

String name;

**int** age;

String[] string;

List<String> list;

Set<String> set;

Map<String, String> map;

List<Students> ls;

Set<Students> ss;

Map<String,Students> ms;

**public** Stduent(Map<String, String> map,String name,**int** age,String[] string,List<String> list,Set<String> set,List<Students> ls,Set<Students> ss,Map<String,Students> ms){

**this**.name = name;

**this**.age = age;

**this**.string = string;

**this**.list = list;

**this**.set = set;

**this**.map = map;

**this**.ls = ls;

**this**.ss = ss;

**this**.ms = ms;

xml

<!-- 构造注入 -->

<bean id=*"students"* class=*"com.wucan.one.Students"*>

<constructor-arg name=*"name"* value=*"小华"*></constructor-arg>

<constructor-arg name=*"age"* value=*"15"*></constructor-arg>

</bean>

<bean id=*"students2"* class=*"com.wucan.one.Students"*>

<constructor-arg name=*"name"* value=*"疾风"*></constructor-arg>

<constructor-arg name=*"age"* value=*"6"*></constructor-arg>

</bean>

<bean id=*"student"* class=*"com.wucan.one.Stduent"*>

<!--直接给属性赋值 -->

<!--name的值为属性名，value的值为给属性赋值 -->

<constructor-arg name=*"name"* value=*"小茗"*></constructor-arg>

<constructor-arg name=*"age"* value=*"18"*></constructor-arg>

<constructor-arg name=*"string"*>

<list>

<value>1</value>

<value>2</value>

<value>3</value>

</list>

</constructor-arg>

<constructor-arg name=*"list"*>

<list>

<value>4</value>

<value>5</value>

<value>6</value>

</list>

</constructor-arg>

<constructor-arg name=*"set"*>

<set>

<value>s</value>

<value>e</value>

<value>t</value>

</set>

</constructor-arg>

<constructor-arg name=*"map"*>

<map>

<entry key=*"iphoneNumber"* value=*"13465479"*></entry>

<entry key=*"address"* value=*"海南"*></entry>

</map>

</constructor-arg>

<constructor-arg name=*"ls"*>

<list>

<ref bean=*"students"*/>

<ref bean=*"students2"*/>

</list>

</constructor-arg>

<constructor-arg name=*"ss"*>

<set>

<ref bean=*"students"* />

<ref bean=*"students2"* />

</set>

</constructor-arg>

<constructor-arg name=*"ms"*>

<map>

<entry key=*"1"* value-ref=*"students"*/>

<entry key=*"2"* value-ref=*"students2"*/>

</map>

</constructor-arg>

</bean>

测试

//构造注入

@Test

**public** **void** test5(){

Stduent stu = ac.getBean("student", Stduent.**class**);

System.***out***.println(stu.toString());

}

空间名注入:

在 xsi:schemaLocation头部信息里面 加入 xmlns:p=*"http://www.springframework.org/schema/p"*

<bean id=*"car"* class=*"com.wucan.one.Car"* p:weight=*"10"*></bean>

空间名取名为 : p

1. **IOC与DI**

IOC：控制反转

把对象的创建交给spring进行管理

DI ：依赖注入

创建对象的过程中向属性添加值

IOC与DI的关系

1. 依赖注入不能单独完成，是在控制反转的基础上完成的
2. 注入类里面的属性值不能直接注入，先创建类的对象再完成注入
3. **注解配置bean**
4. 导入spring-aopjar包
5. 在xml配置文件的头部 shema里面加入*http://www.springframework.org/schema/context/spring-context-3.1.xsd*
6. 在配置文件中开启扫描 包名

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

java代码

**import** org.springframework.stereotype.Component;

@Component(value="stus")

**public** **class** Student {

**public** **static** **void** Jtest(){

System.***out***.println("我是静态的方法");

}

**public** **void** Ptest(){

System.***out***.println("我是非静态方法");

}

}

测试

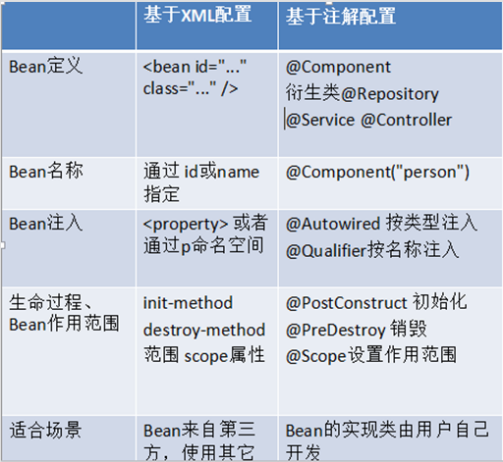
ApplicationContext ac = new ClassPathXmlApplicationContext(“beans.xml”);

student = ac.getBean("stus", Student.**class**);

student.Ptest();

student.*Jtest*();

1. **创建对象的注解**
2. **@Component(value=” ... ”)一般类**
3. **@Repository(value=”...”)对于Dao层类进行标注 持久层**
4. **@Service(value=”...”)对service层进行标注 业务层**
5. **Controller(value=”...”)堆controller层进行标注 web层**
6. **@AutoWired 自动装配，默认按类型进行装配，按名称注入**
7. **@Qualifier:强制使用名称注入**
8. **@Resource 相当于@Autowired和@Qualifier一起使用**
9. **@Scope: \* singleton:单例\* prototype:多例**



**java代码**

@Service(value="udao")

**public** **class** UserDao {

**public** **void** test(){

System.***out***.println("测试注解");

}

}

@Component(value="uService")

**public** **class** UserService {

// @Autowired

// private UserDao userdao;

@Resource(name="udao")

**private** UserDao userdao;

**public** **void** utest(){

userdao.test();

}

}

@Test

**public** **void** test2(){

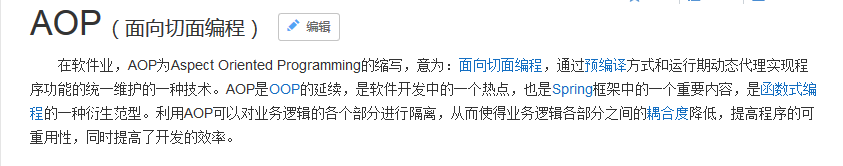
UserService us = ac.getBean("uService", UserService.**class**);

us.utest();

}

**AOP**

**什么是AOP?**



1. **aop面向切面编程.扩展功能不是修改源代码实现**
2. **aop采用横向抽取机制实现。**
3. **aop底层采用动态代理实现**

**为什么学习aop?**

**在不修改源代码的情况下对程序进行增强**

**AOP术语**

**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**

**pointCut 切入点:实际增强的方法**

**advice(通知/增强)：所谓通知是指拦截到joinpoint以后所有做的事情就是通知。**

**通知类型：**

**前置通知**

**后置通知**

**环绕通知**

**异常通知**

**最终通知**

**aspect 切面:是切入点和通知的结合。把具体的逻辑用到具体的方法上**

**Target 目标对象:代理的目标对象(要增强的类)**

**Weaving 织入:把增强应用到目标的过程(adivice到target的过程)**

**proxy 代理: 一个类被aop织入增强后，就产生一个结果代理类**

**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的添加一些方法**

**Spring的AOP操作**

1. **在Spring中使用aspectj实现aop操作**

aspectj不是spring的一部分，单独面向切面框架，单独和spring一起操作aop

要使用必须先要导包 sapectjweaver.jar

1. **aspectj实现的两种方式**
   1. **配置**

**Aspej表达式**

**1 使用aspectj的操作，基于 表达式 形式进行 切入点 配置**

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

**（1）对User类里面的add方法**

**- 第一个参数： \* 表示类里面方法public、private都增强**

**- 添加空格，第二个参数：增强的方法所在类全路径**

**--- 类的全路径后面增强的方法名称**

**execution(\* cn.itcast.User.add(..))**

**（2）对User类里面的所有方法**

**execution(\* cn.itcast.User.\*(..))**

**（3）对所有类里面的所有的方法**

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

**java代码**

**public** **class** OrderDaoImpl{

String name;

**public** **void** select() {

System.***out***.println("查询订单"+name.charAt(0));

}

}

通知类代码

**public** **class** TestOrder {

**public** **void** befores(){

System.***out***.println("我是前置通知");

}

**public** **void** afters(){

System.***out***.println("我是后置通知");

}

**public** **void** aroundTest(JoinPoint jp){

System.***out***.println("之前");

ProceedingJoinPoint pj = (ProceedingJoinPoint) jp;

**try** {

pj.proceed();

System.***out***.println("之后");

} **catch** (Throwable e) {

e.printStackTrace();

}

}

**public** **void** throwings(){

System.***out***.println("我是异常通知");

}

}

测试

ApplicationContext ac;

@Before

**public** **void** setUp() **throws** Exception {

ac = **new** ClassPathXmlApplicationContext("beans.xml");

}

@Test

**public** **void** test() {

OrderDaoImpl odi = ac.getBean("orderDaoImpl", OrderDaoImpl.**class**);

odi.select();

}

xml

<!-- 配置目标类 -->

<bean id=*"orderDaoImpl"* class=*"com.aopxml.one.OrderDaoImpl"*></bean>

<!-- 配置通知类 -->

<bean id=*"testOrder"* class=*"com.aopxml.one.TestOrder"*></bean>

<!--配置aop操作 -->

<aop:config>

<!--配置切入点,对目标增强 -->

<aop:pointcut expression=*"execution(\* com.aopxml.one.OrderDaoImpl.\*(..))"* id=*"pointcut"*/>

<!-- 配置切面 -->

<aop:aspect ref=*"testOrder"*>

<!--增强类型 -->

<!-- <aop:after method="afters" pointcut-ref="pointcut"/>

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

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

<aop:after-throwing method=*"throwings"* pointcut-ref=*"pointcut"*/>

</aop:aspect>

</aop:config>

* 1. **注解**

java代码

**import** org.springframework.stereotype.Component;

@Component(value="book")

**public** **class** Book {

String name;

**public** **void** addBook(){

System.***out***.println("买了一本书"+name.charAt(0));

}

}

通知类

@Component//声明这是一个组建

@Aspect//声明这是一个切面bean

**public** **class** MyBook {

//定义切点

@Pointcut("execution(\* com.aopzhujie.one.Book.\*(..))")

**public** **void** testBook(){

}

/\*

\* 配置 前置通知，使用在方法testBook()上注册切入点

\* 同时接受JoinPoint切入点对象，可以没有该参数

\*/

// @Before("testBook()")

// public void beforeBook(){

// System.out.println("我是前置通知");

// }

//

// @After("testBook()")

// public void afterBook(){

// System.out.println("我是后置通知");

// }

//

// @Around("testBook()")

// public void aroundBook(ProceedingJoinPoint pj){

// System.out.println("之前");

// try {

// pj.proceed();

// System.out.println("之后");

// } catch (Throwable e) {

// e.printStackTrace();

// }

// }

**final** **static** Log ***log*** = LogFactory.*getLog*(MyBook.**class**);

@AfterThrowing(pointcut="testBook()",throwing="ex")

**public** **void** throwingBook(JoinPoint jp,Exception ex){

**if**(***log***.isInfoEnabled()){

***log***.info("afterThrow"+jp+"\t"+ex.getMessage());

}

System.***out***.println("我是异常通知");

}

测试

ApplicationContext ac;

@Before

**public** **void** setUp() **throws** Exception {

ac = **new** ClassPathXmlApplicationContext("beans.xml");

}

@Test

**public** **void** test() {

Book bk = ac.getBean("book", Book.**class**);

bk.addBook();

}

**Log4j**

导入jar

导入配置文件 log4j.properties

##root logger , local level: debug, output: stdout ##

## root logger: better performance than others ##

/\*将等级为INFO的日志输出到stdout这个目的地 \*/

log4j.rootLogger=info,stdout

/\*每天成产一个日子文件\*/

log4j.appender.F=org.apache.log4j.DailyRollingFileAppender

/\*时间格式\*/

log4j.appender.F.DatePattern='.'yyyy-MM-dd

/\*输出端的文件名\*/

log4j.appender.F.File=logs/ssm-test.log

/\*定义名为stdout的输出端的layont类型

HTMLLayout HTML表格形式布局

PatternLayout 灵活的指定布局格式

SimpleLayout 包含日志信息级别和信息字符串

TTCCLayout 包含日志产生的时间、线程、类别等信息

\*/

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

/\*如果使用pattern布局就要指定打印信息的具体格式ConversionPattern,打印参数如下

%m 输出代码中指定的消息

%p 输出优先级，即DEBUG 、INFO、WARN、ERROR、FATAL

%r 输出自应用启动到输出该log信息耗费的毫秒数

%c 输出所属类目，通常就是所在类的全名

%t 输出产生该日志的线程名

%n 输出一个回车换行符，Windows平台为“rn” Unix为“n”

%d 输出日志时间点或日期或时间，默认格式ISO-8859-1，也可以在其后指定格式 %d{yyyy MMM dd HH:mm:ss,SSS}

输出 : 2017年10月8日 09：10：10，921

%| 输出日志事件的发生位置，包括类名、发生的线程、以及在代码中的行数

QC 是log信息的开头，可以为任意字符，一般为项目简称

\*/

log4j.appender.F.layout.ConversionPattern=[%d{ISO8601}|%t|%-5p|%C] - %m%n

log4j.appender.F.encoding=UTF-8

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

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

log4j.appender.stdout.layout.ConversionPattern=[%d{HH:mm:ss}|%-5p|%c] - %m%n

#show application DEBUG log

log4j.logger.com.asecs = DEBUG,F

#show ibatis DEBUG log

log4j.logger.com.ibatis = DEBUG

log4j.logger.com.ibatis.common.jdbc.SimpleDataSource = DEBUG

log4j.logger.com.ibatis.common.jdbc.ScriptRunner = DEBUG

log4j.logger.com.ibatis.sqlmap.engine.impl.SqlMapClientDelegate = DEBUG

log4j.logger.java.sql.Connection = DEBUG

log4j.logger.java.sql.Statement = DEBUG

log4j.logger.java.sql.PreparedStatement = DEBUG

log4j.logger.java.sql.ResultSet = DEBUG

**Spring的JDBC模版(JDBCTemplate)**

1. **导入jar**
2. **编写mysql数据库的连接类**

**private** **static** String *DB\_USERNAME*;

**private** **static** String *DB\_PASSWORD*;

**private** **static** String *DB\_DRIVER*;

**private** **static** String *DB\_URL*;

**private** **static** **final** String ***DB\_CONFIG\_FILE*** = "db.properties";

**static** {

Properties properties = **new** Properties();

**try** {

// 获得当前类的路径，加载指定属性文件,配置文件在src下(Java代码默认路径)

properties.load(DBUtil.**class**.getClassLoader().getResourceAsStream(***DB\_CONFIG\_FILE***));

*DB\_DRIVER* = properties.getProperty("jdbc.driver");

*DB\_URL* = properties.getProperty("jdbc.url");

*DB\_USERNAME* = properties.getProperty("jdbc.user");

*DB\_PASSWORD* = properties.getProperty("jdbc.password");

} **catch** (Exception e) {

e.printStackTrace();

}

}

**public** **static** DataSource getDataSource() {

DriverManagerDataSource dataSource = **new** DriverManagerDataSource(*DB\_URL*, *DB\_USERNAME*, *DB\_PASSWORD*);

dataSource.setUrl(*DB\_URL*);

dataSource.setUsername(*DB\_USERNAME*);

dataSource.setPassword(*DB\_PASSWORD*);

dataSource.setDriverClassName(*DB\_DRIVER*);

**return** dataSource;

}

**public** **static** **void** main(String[] args) **throws** IOException {

System.***out***.println(*getDataSource*());

}

1. **db.properties文件**

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf8

jdbc.user=root

jdbc.password=root

jdbc.initialPoolSize=20 //初始化最大连接数

jdbc.maxPoolSize=100

jdbc.minPoolSize=10

jdbc.maxIdleTime=600

jdbc.acquireIncrement=5

jdbc.maxStatements=5

jdbc.idleConnectionTestPeriod=60

1. 测试

**import** java.sql.PreparedStatement;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**import** org.junit.Before;

**import** org.junit.Test;

**import** org.springframework.jdbc.core.BatchPreparedStatementSetter;

**import** org.springframework.jdbc.core.BeanPropertyRowMapper;

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.jdbc.core.RowCallbackHandler;

**public** **class** TestJDBCTemplate {

//创建数据库连接模版

JdbcTemplate jt;

@Before

**public** **void** before(){

//首先获取数据库连接

jt = **new** JdbcTemplate(DBUtil.*getDataSource*());

}

//查询数据集合

@Test

**public** **void** listPerson(){

**final** List<Person> list = **new** ArrayList<>();

// String sql = "select \* from person where \_id > ?";

// Object[] params = new Object[]{5};

String sql = "select \* from person";

Object[] params = **new** Object[]{};

jt.query(sql, params,**new** RowCallbackHandler(){

@Override

**public** **void** processRow(ResultSet arg0) **throws** SQLException {

Person p = **new** Person();

//设置要查询的字段

p.set\_id(arg0.getInt("\_id"));

p.set\_username(arg0.getString("\_username"));

p.set\_password(arg0.getString("\_password"));

p.set\_email(arg0.getString("\_email"));

p.set\_address(arg0.getString("\_address"));

list.add(p);

}

});

Iterator<Person> it = list.iterator();

**while**(it.hasNext()){

System.***out***.println(it.next());

}

}

//查询一条数据

@Test

**public** **void** select(){

String sql = "select \* from person where \_id = ?";

Person person = jt.queryForObject(sql,**new** Object[]{5},**new** BeanPropertyRowMapper<Person>(Person.**class**));

System.***out***.println(person);

}

//修改数据

@Test

**public** **void** update(){

String sql = "update person set \_username=?,\_password=? where \_id=?";

Object[] param = **new** Object[]{"阿花","admins",5};

**int** code = jt.update(sql, param);

System.***out***.println(code);

}

//删除数据

@Test

**public** **void** delete(){

String sql = "delete from person where \_id = ?";

Object[] params = **new** Object[]{4};

**int** code = jt.update(sql, params);

System.***out***.println(code);

}

//给数据添加一条信息

@Test

**public** **void** inserts(){

String sql = "insert into person (\_username,\_password,\_email,\_address) values(?,?,?,?)";

Object[] params = **new** Object[]{"小锅","789456","78456@qq.com","海南"};

**int** code = jt.update(sql, params);

System.***out***.println(code);

}

//给数据库添加多条信息

@Test

**public** **void** insert(){

String sql = "insert into person (\_username,\_password,\_email,\_address) values(?,?,?,?)";

jt.batchUpdate(sql, **new** BatchPreparedStatementSetter() {

@Override

**public** **void** setValues(PreparedStatement ps, **int** i) **throws** SQLException {

ps.setString(1,"小华"+i);

ps.setString(2,"123456"+i);

ps.setString(3,"75286164"+i+"qq.com");

ps.setString(4,"海口"+i);

}

//循环的次数 (相当于把值添加几次)

@Override

**public** **int** getBatchSize() {

**return** 5;

}

});

}

}

**XML配置JBDCTemplate 与数据库连接池 与 事物 完成银行转账功能**

1. **获取数据库信息的db.properties文件**

<context:property-placeholder location=*"classpath:db.properties"* />

1. **配置C3P0连接池 导入c3p0连接词包**

<!-- c3p0连接池 -->

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

destroy-method=*"close"*>

<property name=*"driverClass"* value=*"${jdbc.driver}"* />

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

<property name=*"user"* value=*"${jdbc.user}"* />

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

<!-- 指定连接池中保留的最大连接数. Default:15-->

<property name=*"maxPoolSize"* value=*"${jdbc.maxPoolSize}"*/>

<!-- 指定连接池中保留的最小连接数-->

<property name=*"minPoolSize"* value=*"${jdbc.minPoolSize}"*/>

<!-- 指定连接池的初始化连接数 取值应在minPoolSize 与 maxPoolSize 之间.Default:3-->

<property name=*"initialPoolSize"* value=*"${jdbc.initialPoolSize}"*/>

<!-- 最大空闲时间,60秒内未使用则连接被丢弃。若为0则永不丢弃。 Default:0-->

<property name=*"maxIdleTime"* value=*"${jdbc.maxIdleTime}"*/>

<!-- 当连接池中的连接耗尽的时候c3p0一次同时获取的连接数. Default:3-->

<property name=*"acquireIncrement"* value=*"${jdbc.acquireIncrement}"*/>

<!-- JDBC的标准,用以控制数据源内加载的PreparedStatements数量。

但由于预缓存的statements属于单个connection而不是整个连接池所以设置这个参数需要考虑到多方面的因数.如果maxStatements与maxStatementsPerConnection均为0,则缓存被关闭。Default:0-->

<property name=*"maxStatements"* value=*"${jdbc.maxStatements}"*/>

<!-- 每60秒检查所有连接池中的空闲连接.Default:0 -->

<property name=*"idleConnectionTestPeriod"* value=*"${jdbc.idleConnectionTestPeriod}"*/>

</bean>

1. **配置JDBCTemplate模版**

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

<!--注入c3p0连接池 -->

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

</bean>

1. **配置JDBC事物管理器**

<bean id=*"transactionManager"*

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

<!-- 注入才c3p0数据库连接池 -->

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

</bean>

1. **配置JDBC事物模版**

<bean id=*"transactionTemplate"*

class=*"org.springframework.transaction.support.TransactionTemplate"*>

<!--引入事物管理器-->

<property name=*"transactionManager"* ref=*"transactionManager"*></property>

</bean>

1. **编写java类**
2. **一个增删改查的接口与类**

**public** **interface** AccountDao {

**public** **void** inMoney(String name,**double** money);

**public** **void** outMoney(String name,**double** money);

}

**import** org.springframework.jdbc.core.support.JdbcDaoSupport;

**public** **class** Accounts **extends** JdbcDaoSupport **implements** AccountDao{

//入账

@Override

**public** **void** inMoney(String name, **double** money) {

String sql = "update salarys set salaryes=salaryes+? where username=?";

getJdbcTemplate().update(sql, money, name);

}

//出账

@Override

**public** **void** outMoney(String name, **double** money) {

String sql = "update salarys set salaryes=salaryes-? where username=?";

getJdbcTemplate().update(sql, money, name);

}

}

**一个操作转账的接口与类**

**public** **interface** AccountService {

//转账

**public** **void** transAccount(String outName,String inName,**double** money);

}

**public** **class** AccountServiceImpl **implements** AccountService{

**private** AccountDao ad;

String name;

**public** **void** setAd(AccountDao ad) {

**this**.ad = ad;

}

@Override

**public** **void** transAccount(String outName, String inName, **double** money) {

//出账

ad.outMoney(outName, money);

// int i = 1/0;

System.***out***.println(name.charAt(0));

//入账

ad.inMoney(inName, money);

}

}

**测试**

**public** **class** TestAccount {

ApplicationContext ac;

AccountServiceImpl as;

@Before

**public** **void** before(){

ac = **new** ClassPathXmlApplicationContext("beans.xml");

}

@Test

**public** **void** test(){

as = ac.getBean("accountsim",AccountServiceImpl.**class**);

as.transAccount("小王","小李",500);

}

}

**xml继续配置**

<!-- 配置目标类 sql操作类 -->

<bean id=*"accounts"* class=*"com.jdbcc3p0.one.Accounts"*>

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

</bean>

<bean id=*"accountsim"* class=*"com.jdbcc3p0.one.AccountServiceImpl"*>

<property name=*"ad"* ref=*"accounts"* />

</bean>

<!--配置事物与隔离级别 -->

<!--事物代理工厂 -->

<bean id=*"accountSeroveProxy"* class=*"org.springframework.transaction.interceptor.TransactionProxyFactoryBean"*>

<!--被代理的类 -->

<property name=*"target"* ref=*"accountsim"*></property>

<!--添加事物管理器 -->

<property name=*"transactionManager"* ref=*"transactionManager"*></property>

<!--添加事物属性 -->

<property name=*"transactionAttributes"*>

<props>

<!-- PROPAGATION 事物传播行为

PROPAGATION\_REQUIRED 如果没事物就新建一个，如果有就加入进来(最常见的选择)

PROPAGATION\_SUPPORTS 支持当前事物，如果没有事物就已非事物方式执行

PROPAGATION\_MANDATORY 使用当前的事物，如果没有就抛出异常

PROPAGATION\_REQUIRES\_NEW 新建事物，如果当前存在事物就把事物挂起

PROPAGATION\_NOT\_SUPPORTED 以非事物的方式执行操作，如果存在事物就将事物挂起

PROPAGATION\_NEVER 以非事物的方式执行，如果存在事物就抛出异常

PROPAGATION\_NESTED 如果当前存在事物，则在嵌套事物内执行。如果没有事物，则执行 PROPAGATION\_REQUIRED 类似的操作

ISOLATION 事物隔离级别

readOnly 只读

-Exception 发生哪些异常回滚事物

+Exception 发生哪些异常不回滚事物

-->

<!--key 对应需要事物操作的方法 \* 代表通配符 -->

<prop key=*"tran\*"*>PROPAGATION\_REQUIRED</prop>

</props>、

</property>

</bean>

<!-- 配置事物的通知 -->

<tx:advice id=*"myAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<tx:method name=*"trans\*"* propagation=*"REQUIRED"* />

</tx:attributes>

</tx:advice>

<aop:config>

*execution(\* com.jdbcc3p0.one..\*.\*(..)) 此包下的所有类的所有方法所有参数类型*

<aop:pointcut id=*"pointcut1"* expression=*"execution(\* com.jdbcc3p0.one..\*.\*(..))"* />

<!-- 配置切面 -->

<aop:advisor advice-ref=*"myAdvice"* pointcut-ref=*"pointcut1"* />

</aop:config>

**XML JDBCTemplate**

ApplicationContext ac;

JdbcTemplate jt;

@Before

**public** **void** before(){

ac = **new** ClassPathXmlApplicationContext("beans.xml");

jt = (JdbcTemplate) ac.getBean("jdbcTemplate");

}

@Test

**public** **void** select(){

String sql = "select \* from person where \_id = ?";

Person sa = jt.queryForObject(sql, **new** Object[]{5},**new** BeanPropertyRowMapper<Person>(Person.**class**));

System.***out***.println(sa);

}

**以之前的操作一样**