Spring框架学习

Spring的核心思想-IOC和AOP

- **IOC**
- **AOP**

IOC的基本介绍

什么是IOC?

IOC = Inversion of Control(控制反转),IOC只是一种技术思想

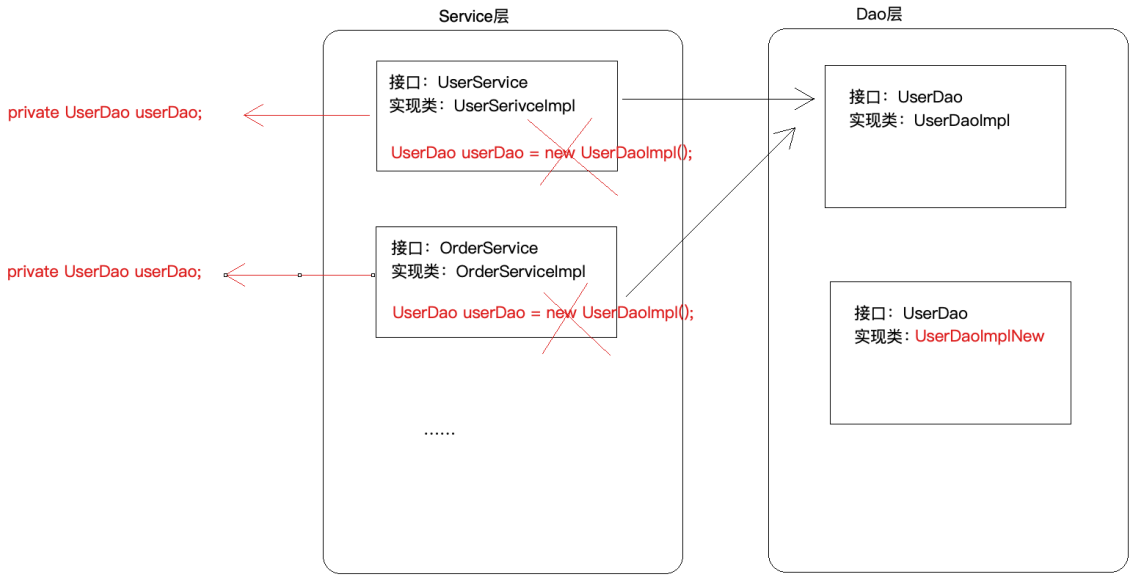
在java中使用IOC开发,无需自己new对象,使用IOC容器帮助我们实例化对象并管理它,我们需要使用哪个对象就从IOC容器中获取,对象的生命周期都由IOC容器进行管理

为什么称为控制反转?

- **控制**:指的是对象创建(实例化,管理)的权利
- **反转**:控制权交由外部管理

IOC解决了什么问题?

解决了对象之间耦合的问题



IOC与DI的区别?

IOC与DI描述的是一件事件,**IOC是目的,DI是手段**,DI就是依赖注入,也是Spring中实现IOC最常用的方式,站在容器的角度上看,DI将对象所需要依赖的对象都从容器中主动注入,达到IOC的目的

AOP的基本介绍

什么是AOP?

AOP = Aspect Oriented Programming 面向切面编程

在多个流程中出现了相同子流程的代码,我们称之为横切逻辑代码,例如在获取方法调用的时间消耗,如果手工在每个方法上面增加,导致编码重复并且不易修改,与业务逻辑代码混杂在一起,不方便管理

AOP提出来切面的概念,将横切逻辑代码与业务代码抽离出来,单独管理,达到非侵入式的效果

AOP解决了什么问题?

在不改变原有业务逻辑代码的同时,增强横切逻辑代码,从根本上解耦合,避免了横切代码重复的问题

为什么叫面向切面编程?

切:指的是横切的逻辑代码,在不修改原有业务逻辑代码的同时,只操作横切逻辑代码,面向横切逻辑编程

面:横切逻辑代码往往影响多个方法,每一个方法都像一个点,多个点构成一个面,称为切面

为什么要使用IOC和AOP?

常规版本代码,不使用IOC和AOP

一个简单的service层调用DAO层示例代码

pojo类:

```
public class Account {

    private String cardNo;
    private String name;
    private int money;
}
```

DAO类:

```
public interface AccountDao {
    Account queryAccountByCardNo(String cardNo) throws Exception;
    int updateAccountByCardNo(Account account) throws Exception;
}
public class JdbcAccountDaoImpl implements AccountDao {
    @Override
    public Account queryAccountByCardNo(String cardNo) throws Exception {
        //从连接池获取连接
         Connection con = DruidUtils.getInstance().getConnection();
        String sql = "select * from account where cardNo = ?";
        PreparedStatement preparedStatement = con.prepareStatement(sql);
        preparedStatement.setString(1,cardNo);
        ResultSet resultSet = preparedStatement.executeQuery();

        Account account = new Account();
        while(resultSet.next()) {
            account.setCardNo(resultSet.getString("cardNo"));
            account.setName(resultSet.getString("name"));
            account.setMoney(resultSet.getInt("money"));
        }
        resultSet.close();
        preparedStatement.close();
        con.close();
        return account;
    }
```

```
    @Override
    public int updateAccountByCardNo(Account account) throws Exception {

        // 从连接池获取连接
        // 改造为：从当前线程当中获取绑定的connection连接
        Connection con = DruidUtils.getInstance().getConnection();
        String sql = "update account set money=? where cardNo=?";
        PreparedStatement preparedStatement = con.prepareStatement(sql);
        preparedStatement.setInt(1,account.getMoney());
        preparedStatement.setString(2,account.getCardNo());
        int i = preparedStatement.executeUpdate();

        preparedStatement.close();
        con.close();
        return i;
    }
}
```

service类:

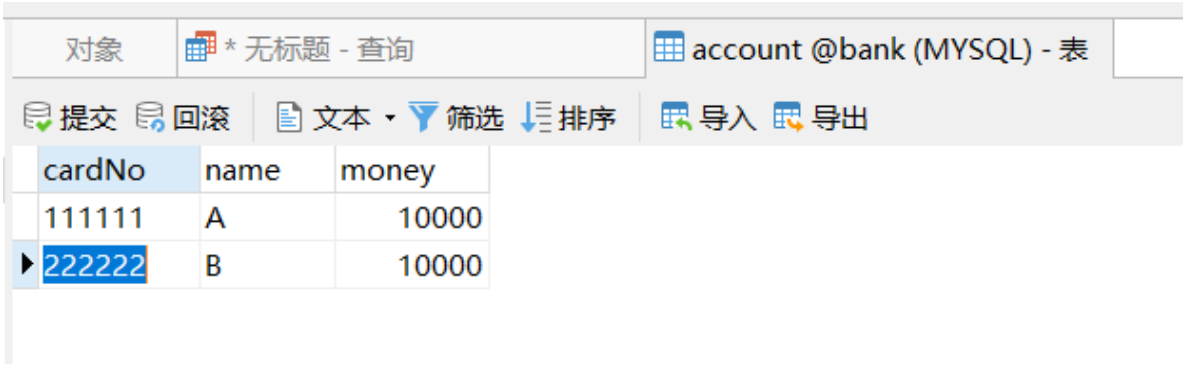
```
public interface TransferService {
    void transfer(String fromCardNo,String toCardNo,int money) throws Exception;
}

public class TransferServiceImpl implements TransferService {

    private AccountDao accountDao = new JdbcAccountDaoImpl();

    @Override
    public void transfer(String fromCardNo, String toCardNo, int money) throws
Exception {
            Account from = accountDao.queryAccountByCardNo(fromCardNo);
            Account to = accountDao.queryAccountByCardNo(toCardNo);
            from.setMoney(from.getMoney()-money);
            to.setMoney(to.getMoney()+money);
            accountDao.updateAccountByCardNo(to);
            accountDao.updateAccountByCardNo(from);
    }
}
```

Utils类:

```
public class DruidUtils {

    private DruidUtils(){
    }

    private static DruidDataSource druidDataSource = new DruidDataSource();


    static {
        druidDataSource.setDriverClassName("com.mysql.jdbc.Driver");
        druidDataSource.setUrl("jdbc:mysql://localhost:3306/bank");
        druidDataSource.setUsername("root");
```

```
        druidDataSource.setPassword("mysql");

    }

    public static DruidDataSource getInstance() {
        return druidDataSource;
    }

}
```

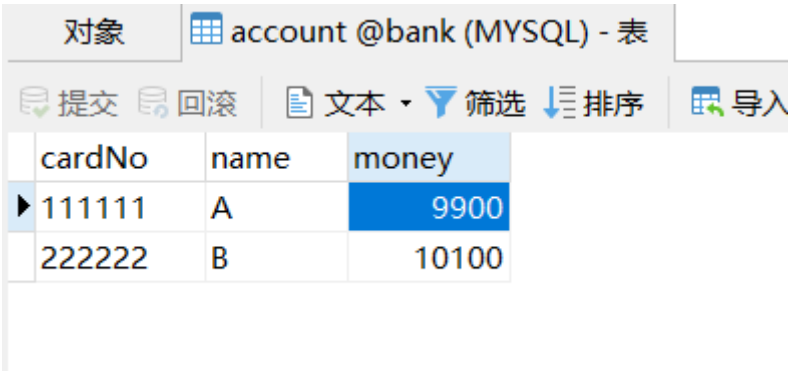
初始数据:



测试类:

```
public class Test {
    public static void main(String[] args) throws Exception {
        TransferService transferService = new TransferServiceImpl();
        transferService.transfer("111111","222222",100);
    }
}
```

执行一次结果为:



常规代码问题分析

- TransferService与AccountDao 两个接口都是用new 出来的对象,如果新增了新的接口实现类,需要切换实现类的时候,要修改相关的代码,耦合性太严重
- `JdbcAccountDaoImpl#updateAccountByCardNo` 两个方法之间没有用同一个事务进行控制,如果出现异常会导致数据异常

问题解决方案

1. 使用配置文件将对应接口的实现类注册在XML配置文件中,并使用反射进行初始化实例类,可以使用简单工厂模式获取对应实例

2. 同一个线程中,使用同一个Connection进行请求数据库,并增加事务控制

修正后代码实现

解析配置文件并使用工厂模式获取对应类实例

新增配置文件:

```
<?xml version="1.0" encoding="utf-8" ?>
<beans>
    <bean id = "accountDao" class ="com.zhengyao.dao.impl.JdbcAccountDaoImpl"/>
    <bean id = "transferService" class
="com.zhengyao.service.impl.TransferServiceImpl">
        <property id = "AccountDao" ref = "accountDao"/>
    </bean>
</beans>
```

新建BeansFactory

```
public class BeansFactory {

    private static Map<String, Object> classMap = new HashMap<>();

    static {
        InputStream resourceAsStream =
BeansFactory.class.getClassLoader().getResourceAsStream("config.xml");
        SAXReader saxReader = new SAXReader();
        try {
            //使用dom4j解析XML配置文件
            Document read = saxReader.read(resourceAsStream);
            Element rootElement = read.getRootElement();
            List<Element> list = rootElement.selectNodes("//bean");
            //先将所有的bean 给解析完成
            for (Element element : list) {
                String id = element.attributeValue("id");
                String className = element.attributeValue("class");
                Class<?> clazz = Class.forName(className);
                Object obj = clazz.newInstance();
                classMap.put(id,obj);
            }
            //解析所有的property,这里只配置了ref类型
            List<Element> propertyList = rootElement.selectNodes("//property");
            for (Element element : propertyList) {
                Element parent = element.getParent();
                String parentId = parent.attributeValue("id");
                String elementId = element.attributeValue("id");
                String ref = element.attributeValue("ref");
                Object object = classMap.get(parentId);
                Object refObj = classMap.get(ref);
                Method[] declaredMethods =
object.getClass().getDeclaredMethods();
                for (Method declaredMethod : declaredMethods) {
                    //使用set 方法注入
                    if (declaredMethod.getName().equals("set" + elementId)) {
                        declaredMethod.invoke(object,refObj);
                        break;
                    }
```

```
                }
                classMap.put(parentId, object);
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static Object getClass(String id){
        return classMap.get(id);
    }
}
```

原有代码中使用set方法注入:

```
public class TransferServiceImpl implements TransferService {

    private AccountDao accountDao;
    //使用set方法注入
    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    @Override
    public void transfer(String fromCardNo, String toCardNo, int money) throws
Exception {
    //省略相同代码
    }
```

测试类:

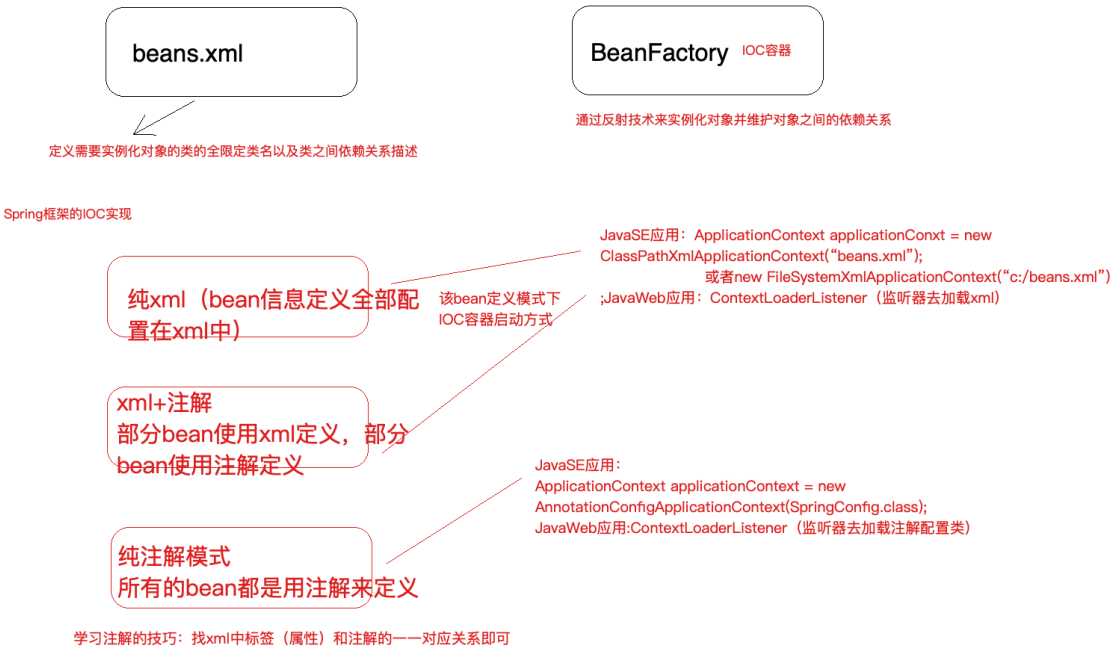
```
public class Test2 {

    public static void main(String[] args) throws Exception {
        TransferService transferService = (TransferService)
BeansFactory.getClass("transferService");
        transferService.transfer("111111","222222",100);
    }
}
```

扩展点-新增同一线程内事务控制

提示：使用动态代理新增事务控制代码

Spring中IOC应用

Spring配置的三种方式



纯xml模式

- xml 文件头

```
<?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
https://www.springframework.org/schema/beans/spring-beans.xsd">
```

- 实例化Bean的三种方式
  - 方式一：使用无参构造函数 在默认情况下，它会通过反射调用无参构造函数来创建对象。如果类中没有无参构造函数，将创建 失败。

```
<!--方式一：使用无参构造器（推荐）-->
    <bean id="connectionUtils"
class="com.zhengyao.edu.utils.ConnectionUtils"></bean>
```

  - 方式二：使用静态方法创建

    在实际开发中，我们使用的对象有些时候并不是直接通过构造函数就可以创建出来的，它可能在创 建的过程 中会做很多额外的操作。此时会提供一个创建对象的方法，恰好这个方法是static修饰的 方法，即是此种情 况。

```
<!--方式二：静态方法-->
   <bean id="connectionUtils" class="CreateBeanFactory" factory-
method="getInstanceStatic"/>
```

  - 方式三：使用实例化方法创建

    此种方式和上面静态方法创建其实类似，区别是用于获取对象的方法不再是static修饰的了，而是 类中的一 个普通方法。此种方式比静态方法创建的使用几率要高一些。 在早期开发的项目中，工厂类中的方法有可能是静态的，也有可能是非静态方法，当是非静态方法 时，即可 采用下面的配置方式：

```
<bean id="createBeanFactory" class="CreateBeanFactory"></bean>
    <bean id="connectionUtils" factory-bean="createBeanFactory"
factory-method="getInstance"/>
```

Bean的作用范围及生命周期

bean的作用范围

在spring框架管理Bean对象的创建时，Bean对象默认都是单例的，但是它支持配置的方式改 变作用范围。作用范围官方提供的说明如下图：

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of an HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.
application	Scopes a single bean definition to the lifecycle of a `ServletContext`. Only valid in the context of a web-aware Spring `ApplicationContext`.
websocket	Scopes a single bean definition to the lifecycle of a `WebSocket`. Only valid in the context of a web-aware Spring `ApplicationContext`.

在上图中提供的这些选项中，我们实际开发中用到最多的作用范围就是singleton（单例模式）和prototype（原型模式，也叫多例模式）

```
<bean id="accountDao" class="com.zhengyao.edu.dao.impl.JdbcAccountDaoImpl"
scope="singleton"></bean>
```

单例模式：singleton

- 对象出生：当创建容器时，对象就被创建了。
- 对象活着：只要容器在，对象一直活着。
- 对象死亡：当销毁容器时，对象就被销毁了。
- 一句话总结：单例模式的bean对象生命周期与容器相同。

多例模式：prototype

- 对象出生：当使用对象时，创建新的对象实例。
- 对象活着：只要对象在使用中，就一直活着。
- 对象死亡：当对象长时间不用时，被java的垃圾回收器回收了。
- 一句话总结：多例模式的bean对象，spring框架只负责创建，不负责销毁。

Bean的标签属性

在基于xml的IoC配置中，bean标签是最基础的标签。它表示了IoC容器中的一个对象。换句话 说，如果一个对象想让spring管理，在XML的配置中都需要使用此标签配置，Bean标签的属性如 下：

- id属性： 用于给bean提供一个唯一标识。在一个标签内部，标识必须唯一。
- class属性：用于指定创建Bean对象的全限定类名。
- name属性：用于给bean提供一个或多个名称。多个名称用空格分隔。
- factory-bean属性：用于指定创建当前bean对象的工厂bean的唯一标识。当指定了此属性之后，class属性失效。

- factory-method属性：用于指定创建当前bean对象的工厂方法，如配合factory-bean属性使用，则class属性失效。如配合class属性使用，则方法必须是static的。
- scope属性：用于指定bean对象的作用范围。通常情况下就是singleton。当要用到多例模式时，可以配置为prototype。
- init-method属性：用于指定bean对象的初始化方法，此方法会在bean对象装配后调用。必须是一个无参方法。
- destory-method属性：用于指定bean对象的销毁方法，此方法会在bean对象销毁前执行。它只能为scope是singleton时起作用。

DI 依赖注入的xml配置

依赖注入分类

按照注入的方式分类

- 构造函数注入：顾名思义，就是利用带参构造函数实现对类成员的数据赋值。
- set方法注入：它是通过类成员的set方法实现数据的注入。（使用最多的）

按照注入的数据类型分类

- 基本类型和String
  - 注入的数据类型是基本类型或者是字符串类型的数据。
- 其他Bean类型
  - 注入的数据类型是对象类型，称为其他Bean的原因是，这个对象是要求出现在IoC容器 中的。那么针对当前Bean来说，就是其他Bean了。

复杂类型（集合类型）

- 注入的数据类型是Aarry，List，Set，Map，Properties中的一种类型。
  - 依赖注入的配置实现之构造函数注入 顾名思义，就是利用构造函数实现对类成员的赋值。它的使用要求是，类中提供的构造函数参数个数必须和配置的参数个数一致，且数据类型匹配。同时需要注意的是，当没有无参构造时，则必须提供构造函数参数的注入，否则Spring框架会报错。

在使用构造函数注入时，涉及的标签是 constructor-arg ，该标签有如下属性：

- name：用于给构造函数中指定名称的参数赋值。
- index：用于给构造函数中指定索引位置的参数赋值。
- value：用于指定基本类型或者String类型的数据。
- ref：用于指定其他Bean类型的数据。写的是其他bean的唯一标识。

```
<!--<constructor-arg index="0" ref="connectionUtils"/>
<constructor-arg index="1" value="zhangsan"/>
<constructor-arg index="2" value="1"/>
<constructor-arg index="3" value="100.5"/>-->
```

依赖注入的配置实现之set方法注入 顾名思义，就是利用字段的set方法实现赋值的注入方式。此种方式在实际开发中是使用最多的注 入方式。

在使用set方法注入时，需要使用 property 标签，该标签属性如下：

- name：指定注入时调用的set方法名称。 （注：不包含set这三个字母,druid连接池指定属性名称）
- value：指定注入的数据。它支持基本类型和String类型。

- ref：指定注入的数据。它支持其他bean类型。写的是其他bean的唯一标识。
- 复杂数据类型注入 首先，解释一下复杂类型数据，它指的是集合类型数据。集合分为两类，一类是List结构（数组结构），一类是Map接口（键值对）。

```
<property name="myArray">
            <array>
                <value>array1</value>
                <value>array2</value>
                <value>array3</value>
            </array>
        </property>

        <property name="myMap">
            <map>
                <entry key="key1" value="value1"/>
                <entry key="key2" value="value2"/>
            </map>
        </property>

        <property name="mySet">
            <set>
                <value>set1</value>
                <value>set2</value>
            </set>
        </property>

        <property name="myProperties">
            <props>
                <prop key="prop1">value1</prop>
                <prop key="prop2">value2</prop>
            </props>
        </property>
```

```
private String[] myArray;
private Map<String,String> myMap;
private Set<String> mySet;
private Properties myProperties;

public void setMyArray(String[] myArray) { this.myArray = myArray; }

public void setMyMap(Map<String, String> myMap) { this.myMap = myMap; }

public void setMySet(Set<String> mySet) { this.mySet = mySet; }

public void setMyProperties(Properties myProperties) { this.myProperties = myProperties; }
```

xml与注解相结合模式

1. 实际企业开发中，纯xml模式使用已经很少了
2. 引入注解功能，不需要引入额外的jar
3. xml+注解结合模式，xml文件依然存在，所以，spring IOC容器的启动仍然从加载xml开始
4. 哪些bean的定义写在xml中，哪些bean的定义使用注解

xml中标签与注解的对应（IoC）

xml形式	对应的注解形式
标签	@Component("accountDao")，注解加在类上bean的id属性内容直接配置在注解后面如果不配置，默认定义个这个bean的id为类的类名首字母小写；另外，针对分层代码开发提供了@Componenet的三种别名@Controller、@Service、@Repository分别用于控制层类、服务层类、dao层类的bean定义，这四个注解的用法完全一样，只是为了更清晰的区分而已
标签的scope属性	@Scope("prototype")，默认单例，注解加在类上
标签的init-method属性	@PostConstruct，注解加在方法上，该方法就是初始化后调用的方法
标签的destory-method属性	@PreDestory，注解加在方法上，该方法就是销毁前调用的方法

DI 依赖注入的注解实现方式：

@Autowired（推荐使用）

@Autowired为Spring提供的注解，需要导入包 org.springframework.beans.factory.annotation.Autowired。@Autowired采取的策略为按照类型注入。

```
public class TransferServiceImpl {
    @Autowired
    private AccountDao accountDao;
}
```

如上代码所示，这样装配回去spring容器中找到类型为AccountDao的类，然后将其注入进来。这 样会产生一个问题，当一个类型有多个bean值的时候，会造成无法选择具体注入哪一个的情况， 这个时候我们需要配合着@Qualifier使用。@Qualifier告诉Spring具体去装配哪个对象。

```
public class TransferServiceImpl {
    @Autowired
    @Qualifier(name="jdbcAccountDaoImpl")
    private AccountDao accountDao;
}
```

纯注解模式

改造xm+注解模式，将xml中遗留的内容全部以注解的形式迁移出去，最终删除xml，从Java配置类启动 对应注解 @Configuration 注解，表名当前类是一个配置类

@ComponentScan 注解，替代 context:component-scan

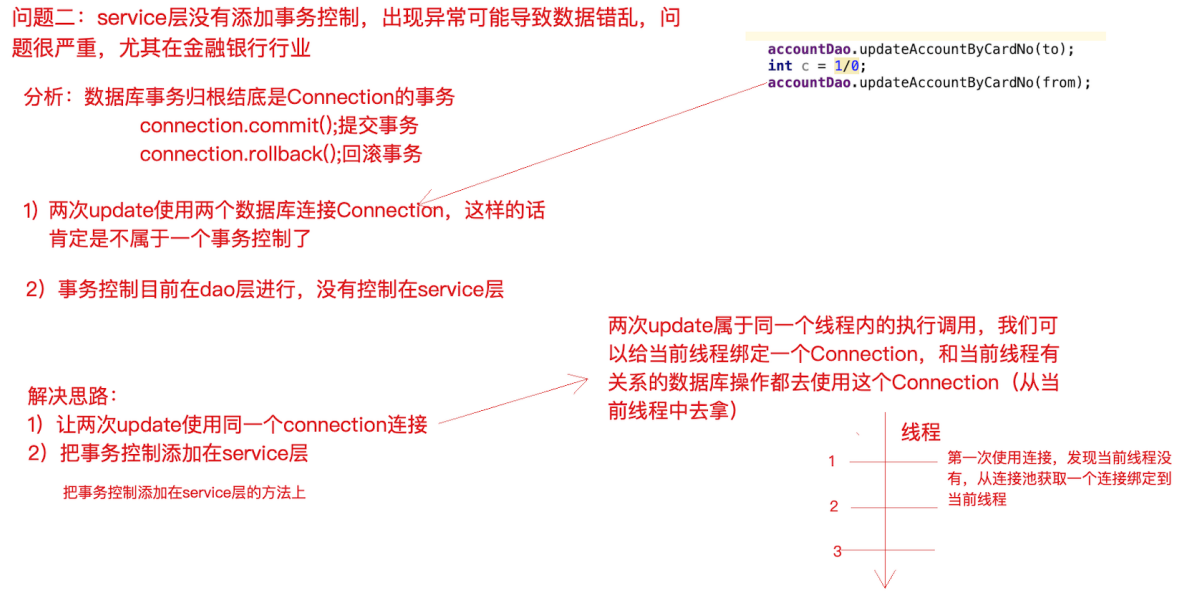
@PropertySource，引入外部属性配置文件

@Import 引入其他配置类

@Value 对变量赋值，可以直接赋值，也可以使用 ${} 读取资源配置文件中的信息

@Bean 将方法返回对象加入 SpringIOC 容器

自定义事务控制



新建一个手工事务控制器类

```
package com.zhengyao.edu.utils;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.sql.SQLException;

/**
 * @author zhengyao
 *
 * 事务管理器类：负责手动事务的开启、提交、回滚
 */
@Component("transactionManager")
public class TransactionManager {


    @Autowired
    private ConnectionUtils connectionUtils;




    // 开启手动事务控制
    public void beginTransaction() throws SQLException {
        connectionUtils.getCurrentThreadConn().setAutoCommit(false);
```

```
    }


    // 提交事务
    public void commit() throws SQLException {
        connectionUtils.getCurrentThreadConn().commit();
    }


    // 回滚事务
    public void rollback() throws SQLException {
        connectionUtils.getCurrentThreadConn().rollback();
    }
}
```

新建动态代理对象

新建一个JDK、CGLIB动态代理对象工厂，用于生成动态代理对象

```
package com.zhengyao.edu.factory;

import com.zhengyao.edu.utils.TransactionManager;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

/**
 * @author zhengyao
 *
 *
 * 代理对象工厂：生成代理对象的
 */


@Component("proxyFactory")
public class ProxyFactory {


    @Autowired
    private TransactionManager transactionManager;




    /**
     * Jdk动态代理
     * @param obj  委托对象
     * @return   代理对象
     */
    public Object getJdkProxy(Object obj) {
```

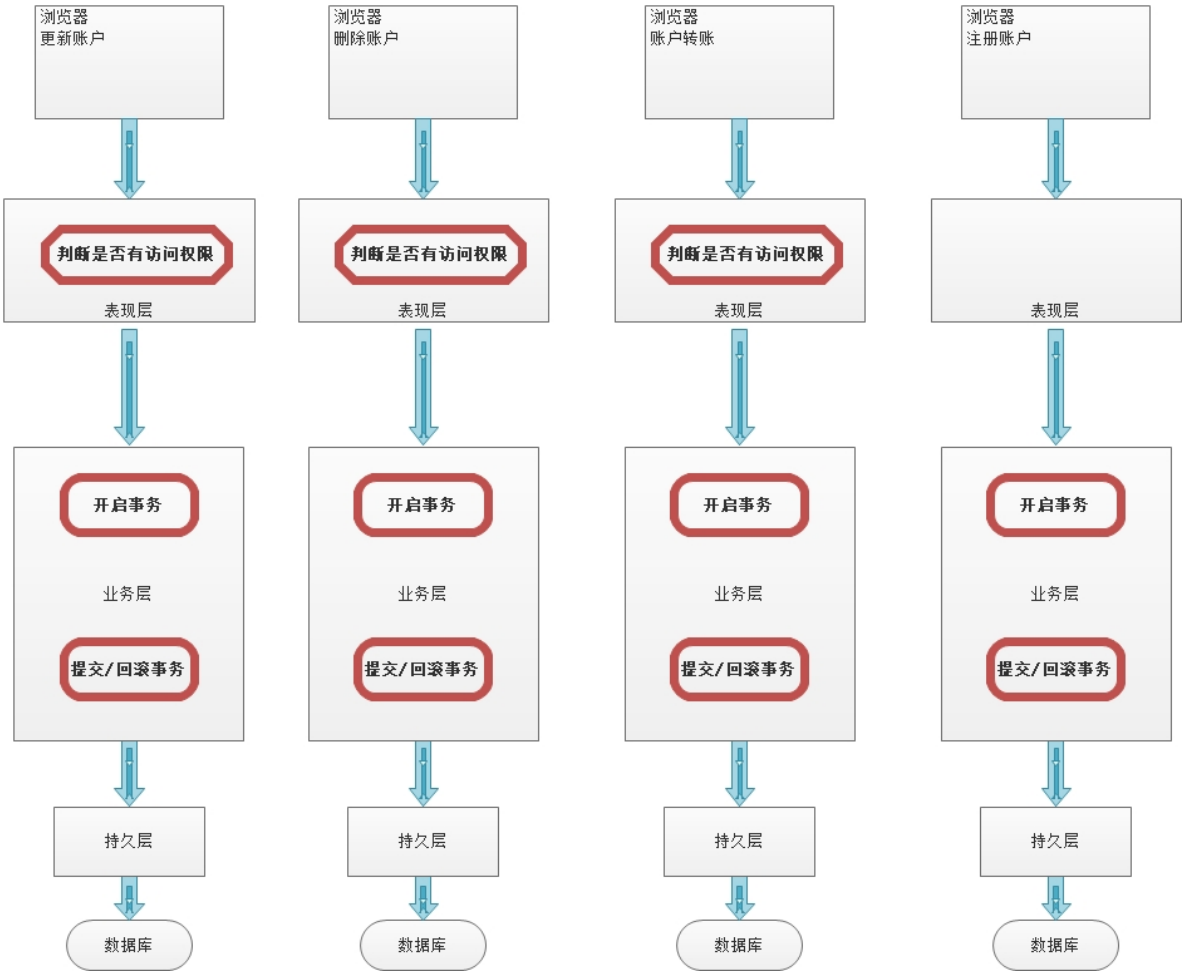
```
        // 获取代理对象
        return  Proxy.newProxyInstance(obj.getClass().getClassLoader(),
obj.getClass().getInterfaces(),
                new InvocationHandler() {
                    @Override
                    public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                        Object result = null;

                        try{
                            // 开启事务(关闭事务的自动提交)
                            transactionManager.beginTransaction();

                            result = method.invoke(obj,args);

                            // 提交事务

                            transactionManager.commit();
                        }catch (Exception e) {
                            e.printStackTrace();
                            // 回滚事务
                            transactionManager.rollback();

                            // 抛出异常便于上层servlet捕获
                            throw e;

                        }

                        return result;
                    }
                });

    }


    /**
     * 使用cglib动态代理生成代理对象
     * @param obj 委托对象
     * @return
     */
    public Object getCglibProxy(Object obj) {
        return  Enhancer.create(obj.getClass(), new MethodInterceptor() {
            @Override
            public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
                Object result = null;
                try{
                    // 开启事务(关闭事务的自动提交)
                    transactionManager.beginTransaction();

                    result = method.invoke(obj,objects);

                    // 提交事务

                    transactionManager.commit();
                }catch (Exception e) {
                    e.printStackTrace();
```

```
                    // 回滚事务
                    transactionManager.rollback();

                    // 抛出异常便于上层servlet捕获
                    throw e;

                }
                return result;
            }
        });
    }
}
```

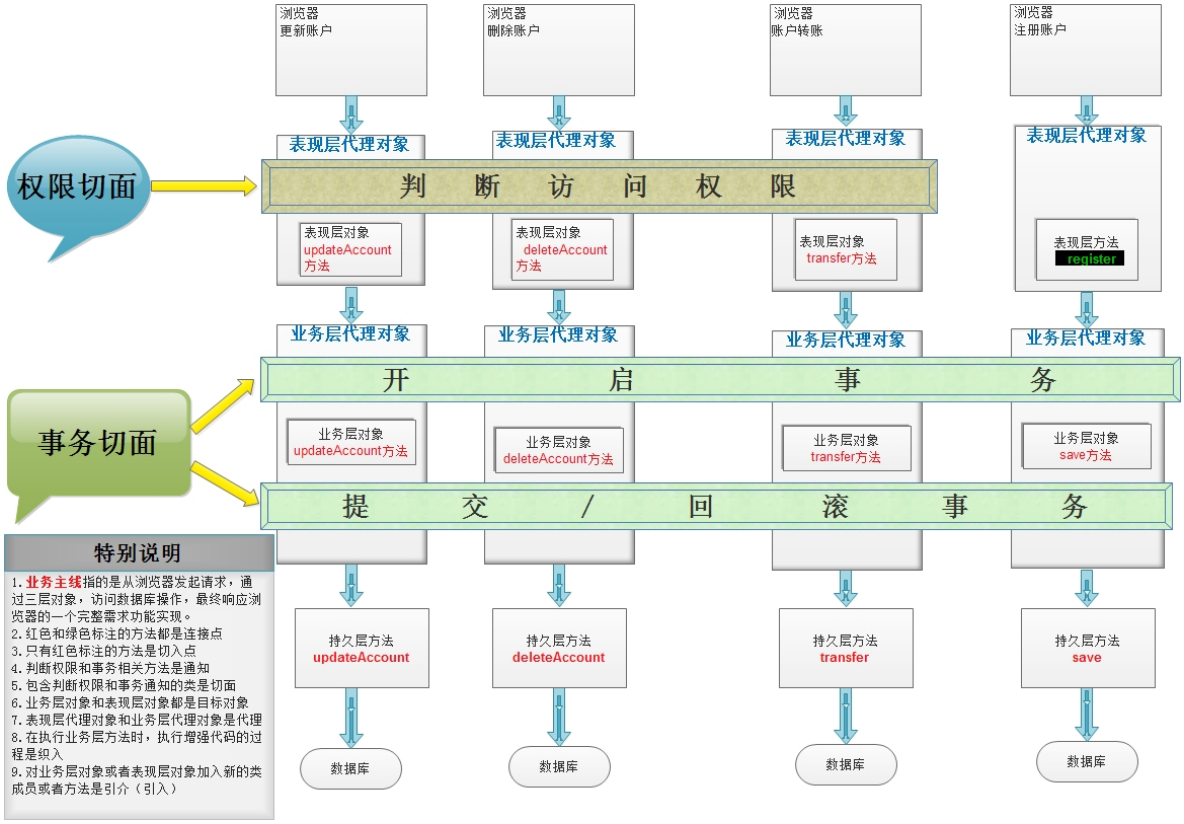
Spring中AOP的使用

AOP本质：在不改变原有业务逻辑的情况下增强横切逻辑，横切逻辑代码往往是权限校验代码、日志代码、事务控制代码、性能监控代码。

AOP 相关术语



上图描述的就是未采用AOP思想设计的程序，当我们红色框中圈定的方法时，会带来大量的重复劳动。程序中充斥着大量的重复代码，使我们程序的独立性很差。而下图中是采用了AOP思想设计的程序，它把红框部分的代码抽取出来的同时，运用动态代理技术，在运行期对需要使用的业务逻辑方法进行增 强



AOP 术语

名词	解释
Joinpoint(连接点)	它指的是那些可以用于把增强代码加入到业务主线中的点，那么由上图中我们可以看出，这些点指的就是方法。在方法执行的前后通过动态代理技术加入增强的代码。在Spring框架AOP思想的技术实现中，也只支持方法类型的连接点。
Pointcut(切入点)	它指的是那些已经把增强代码加入到业务主线进来之后的连接点。由上图中，我们看出表现层 transfer 方法就只是连接点，因为判断访问权限的功能并没有对其增强。
Advice(通知/增强)	它指的是切面类中用于提供增强功能的方法。并且不同的方法增强的时机是不一样的。比如，开启事务肯定要在业务方法执行之前执行；提交事务要在业务方法正常执行之后执行，而回滚事务要在业务方法执行产生异常之后执行等等。那么这些就是通知的类型。其分类有：前置通知 后置通知 异常通知 最终通知 环绕通知。
Target(目标对象)	它指的是代理的目标对象。即被代理对象。
Proxy(代理)	它指的是一个类被AOP织入增强后，产生的代理类。即代理对象。
Weaving(织入)	它指的是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。
Aspect(切面)	它指定是增强的代码所关注的方面，把这些相关的增强代码定义到一个类中，这个类就是切面类。例如，事务切面，它里面定义的方法就是和事务相关的，像开启事务，提交事务，回滚事务等等，不会定义其他与事务无关的方法。我们前面的案例中 TrasnactionManager 就是一个切面。

连接点：方法开始时、结束时、正常运行完毕时、方法异常时等这些特殊的时机点，我们称之为连接点，项目中每个方法都有连接点，连接点是一种候选点

切入点：指定AOP思想想要影响的具体方法是哪些，描述感兴趣的方法

Advice增强： 第一个层次：指的是横切逻辑 第二个层次：方位点（在某一些连接点上加入横切逻辑，那么这些连接点就叫做方位点，描述的是具体 的特殊时机）

Aspect切面：切面概念是对上述概念的一个综合 Aspect切面= 切入点+增强 = 切入点（锁定方法） + 方位点（锁定方法中的特殊时机）+ 横切逻辑

众多的概念，目的就是为了锁定要在哪个地方插入什么横切逻辑代码

Spring中AOP的代理选择

Spring 实现AOP思想使用的是动态代理技术默认情况下，Spring会根据被代理对象是否实现接口来选择使用JDK还是CGLIB。当被代理对象没有实现任何接口时，Spring会选择CGLIB。当被代理对象实现了接口，Spring会选择JDK官方的代理技术，不过我们可以通过配置的方式，让Spring强制使用CGLIB。

Spring中AOP的配置方式

在Spring的AOP配置中，也和IoC配置一样，支持3类配置方式。

第一类：使用XML配置

第二类：使用XML+注解组合配置

第三类：使用纯注解配置

XML 模式

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-aop</artifactId>
<version>5.1.12.RELEASE</version>
</dependency>
<dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjweaver</artifactId>
<version>1.9.4</version>
</dependency>
```

AOP核心配置

```
<!--
Spring基于XML的AOP配置前期准备：
在spring的配置文件中加入aop的约束
xmlns:aop="http://www.springframework.org/schema/aop"
http://www.springframework.org/schema/aop
https://www.springframework.org/schema/aop/spring-aop.xsd
Spring基于XML的AOP配置步骤：
第一步：把通知Bean交给Spring管理
第二步：使用aop:config开始aop的配置
第三步：使用aop:aspect配置切面
第四步：使用对应的标签配置通知的类型
入门案例采用前置通知，标签为aop:before
-->
<!--把通知bean交给spring来管理-->
<bean id="logUtil" class="com.zhengyao.utils.LogUtil"></bean>
<!--开始aop的配置-->
<aop:config>
```

```
<!--配置切面-->
<aop:aspect id="logAdvice" ref="logUtil">
<!--配置前置通知-->
<aop:before method="printLog" pointcut="execution(public *
com.zhengyao.service.impl.TransferServiceImpl.updateAccountByCardNo(com.zhengyao
.pojo.Account))"></aop:before>
</aop:aspect>
</aop:config>
```

切入点表达式

关于切入点表达式上述配置实现了对 TransferServiceImpl 的 updateAccountByCardNo 方法进行增强，在其 执行之前，输出了记录日志的语句。这里面，我们接触了一个比较陌生的名称：切入点表达式，它是做什么的呢？我们往下看。

概念及作用 切入点表达式，也称之为AspectJ切入点表达式，指的是遵循特定语法结构的字符串，其 作用是用于对符合语法格式的连接点进行增强。它是AspectJ表达式的一部分。

关于AspectJ

AspectJ是一个基于Java语言的AOP框架，Spring框架从2.0版本之后集成了AspectJ框架 中切入点表达式的部分，开始支持AspectJ切入点表达式。

切入点表达式使用示例

```
全限定方法名 访问修饰符 返回值 包名.包名.包名.类名.方法名(参数列表)
全匹配方式：
public void
com.zhengyao.service.impl.TransferServiceImpl.updateAccountByCardNo(c
om.zhengyao.pojo.Account)
访问修饰符可以省略
void com.zhengyao.service.impl.TransferServiceImpl.updateAccountByCardNo(c
om.zhengyao.pojo.Account)
返回值可以使用*，表示任意返回值
* com.zhengyao.service.impl.TransferServiceImpl.updateAccountByCardNo(c
om.zhengyao.pojo.Account)
    包名可以使用.表示任意包，但是有几级包，必须写几个
* ....TransferServiceImpl.updateAccountByCardNo(com.zhengyao.pojo.Accou
nt)
包名可以使用..表示当前包及其子包
* ..TransferServiceImpl.updateAccountByCardNo(com.zhengyao.pojo.Account
)
类名和方法名，都可以使用.表示任意类，任意方法
* ...(com.zhengyao.pojo.Account)
参数列表，可以使用具体类型
基本类型直接写类型名称 ： int
引用类型必须写全限定类名：java.lang.String
参数列表可以使用*，表示任意参数类型，但是必须有参数
* *..*.*(*)
参数列表可以使用..，表示有无参数均可。有参数可以是任意类型
* *..*.*(..)
全通配方式：
* *..*.*(..)
```

改变代理方式的配置

在前面我们已经说了，Spring在选择创建代理对象时，会根据被代理对象的实际情况来选择 的。被代理对象实现了接口，则采用基于接口的动态代理。当被代理对象没有实现任何接口 的时候，Spring会自动切换到基于子类的动态代理方式。 但是我们都知道，无论被代理对象是否实现接口，只要不是final修饰的类都可以采用cglib提 供的方式创建代理对象。所以Spring也考虑到了这个情况，提供了配置的方式实现强制使用 基于子类的动态代理（即cglib的方式），配置的方式有两种

- 使用aop:config标签配置

```
<aop:config proxy-target-class="true">
```

- 使用aop:aspectj-autoproxy标签配置

```
<!--此标签是基于XML和注解组合配置AOP时的必备标签，表示Spring开启注解配置AOP
的支持-->
<aop:aspectj-autoproxy proxy-target-class="true"></aop:aspectj-autoproxy>
```

五种通知类型

- 前置通知

```
<!--
作用:
用于配置前置通知。
出现位置:
它只能出现在aop:aspect标签内部
属性:
method:用于指定前置通知的方法名称
pointcut:用于指定切入点表达式
pointcut-ref:用于指定切入点表达式的引用
-->
<aop:before method="printLog" pointcut-ref="pointcut1">
</aop:before>
```

执行时机 前置通知永远都会在切入点方法（业务核心方法）执行之前执行。

细节 前置通知可以获取切入点方法的参数，并对其进行增强。

- 正常执行时通知

```
<!--
作用:
用于配置正常执行时通知
出现位置:
它只能出现在aop:aspect标签内部
属性:
method:用于指定后置通知的方法名称
pointcut:用于指定切入点表达式
pointcut-ref:用于指定切入点表达式的引用
-->
<aop:after-returning method="afterReturningPrintLog" pointcut-ref="pt1">
</aop:after-returning>
```

- 异常通知

```
<!--
作用:
用于配置异常通知。
出现位置:
它只能出现在aop:aspect标签内部
属性:
method:用于指定异常通知的方法名称
pointcut:用于指定切入点表达式
pointcut-ref:用于指定切入点表达式的引用
-->
<aop:after-throwing method="afterThrowingPrintLog" pointcut-ref="pt1">
</aop:after-throwing>
```

异常通知的执行时机是在切入点方法（业务核心方法）执行产生异常之后，异常通知执行。如果切入点方法执行没有产生异常，则异常通知不会执行。

细节

异常通知不仅可以获取切入点方法执行的参数，也可以获取切入点方法执行产生的异常信息。

- 最终通知

```
<!--
作用:
用于指定最终通知。
出现位置:
它只能出现在aop:aspect标签内部
属性:
method:用于指定最终通知的方法名称
pointcut:用于指定切入点表达式
pointcut-ref:用于指定切入点表达式的引用
-->
<aop:after method="afterPrintLog" pointcut-ref="pt1"></aop:after>
```

执行时机 最终通知的执行时机是在切入点方法（业务核心方法）执行完成之后，切入点方法返回之前执行。 换句话说，无论切入点方法执行是否产生异常，它都会在返回之前执行。

细节 最终通知执行时，可以获取到通知方法的参数。同时它可以做一些清理操作。

- 环绕通知

```
<!--
作用:
用于配置环绕通知。
出现位置:
它只能出现在aop:aspect标签的内部
属性:
method:用于指定环绕通知的方法名称
pointcut:用于指定切入点表达式
pointcut-ref:用于指定切入点表达式的引用
-->
<aop:around method="aroundPrintLog" pointcut-ref="pt1"></aop:around>
```

特别说明 环绕通知，它是有别于前面四种通知类型外的特殊通知。前面四种通知（前置，后置，异常和最终） 它们都是指定何时增强的通知类型。而环绕通知，它是Spring框架为我们提供的一种可以通过编码的 方式，控制增强代码何时执行的通知类型。它里面借助的ProceedingJoinPoint接口及其实现类， 实现手动触发切入点方法的调用。

```
/*@Around("pt1()")*/
public Object arroundMethod(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
    System.out.println("环绕通知中的beforemethod....");

    Object result = null;
    try{
        // 控制原有业务逻辑是否执行
        // result = proceedingJoinPoint.proceed(proceedingJoinPoint.getArgs());
    }catch(Exception e) {
        System.out.println("环绕通知中的exceptionmethod....");
    }finally {
        System.out.println("环绕通知中的after method....");
    }

    return result;
}
```

XML+注解模式

XML 中开启 Spring 对注解 AOP 的支持

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

```
@Component
@Aspect
public class LogUtils {


    @Pointcut("execution(* com.zhengyao.edu.service.impl.TransferServiceImpl.*
(..))")
    public void pt1(){

    }


    /**
     * 业务逻辑开始之前执行
     */
    @Before("pt1()")
    public void beforeMethod(JoinPoint joinPoint) {
        Object[] args = joinPoint.getArgs();
        for (int i = 0; i < args.length; i++) {
            Object arg = args[i];
            System.out.println(arg);
        }
        System.out.println("业务逻辑开始执行之前执行.......");
    }


    /**
     * 业务逻辑结束时执行（无论异常与否）
     */
    @After(value = "pt1()")
    public void afterMethod() {
        System.out.println("业务逻辑结束时执行，无论异常与否都执行.......");
    }
```

```
    /**
     * 异常时时执行
     */
    @AfterThrowing("pt1()")
    public void exceptionMethod() {
        System.out.println("异常时执行.......");
    }


    /**
     * 业务逻辑正常时执行
     */
    @AfterReturning(value = "pt1()",returning = "retVal")
    public void successMethod(Object retVal) {
        System.out.println("业务逻辑正常时执行.......");
    }


    /**
     * 环绕通知
     *
     */
    @Around("pt1()")
    public Object arroundMethod(ProceedingJoinPoint proceedingJoinPoint) throws
Throwable {
        System.out.println("环绕通知中的beforemethod....");

        Object result = null;
        try{
            // 控制原有业务逻辑是否执行
            // result =
proceedingJoinPoint.proceed(proceedingJoinPoint.getArgs());
        }catch(Exception e) {
            System.out.println("环绕通知中的exceptionmethod....");
        }finally {
            System.out.println("环绕通知中的after method....");
        }

        return result;
    }

}
```

注解模式

在使用注解驱动开发aop时，我们要明确的就是，是注解替换掉配置文件中的下面这行配置：

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

在配置类中使用如下注解进行替换上述配置

```
@Configuration
@ComponentScan("com.zhengyao")
@EnableAspectJAutoProxy //开启spring对注解AOP的支持
public class SpringConfiguration {
}
```

Spring对事务的支持

编程式事务：在业务代码中添加事务控制代码，这样的事务控制机制就叫做编程式事务

声明式事务：通过xml或者注解配置的方式达到事务控制的目的，叫做声明式事务

事务回顾

事务的概念

事务指逻辑上的一组操作，组成这组操作的各个单元，要么全部成功，要么全部不成功。从而确保了数据的准确与安全。

例如：A——B转帐，对应于如下两条sql语句:

```
/*转出账户减钱*/
update account set money=money-100 where name='a';
/**转入账户加钱*/
update account set money=money+100 where name='b';
```

这两条语句的执行，要么全部成功，要么全部不成功

事务的四大特性

原子性（Atomicity）

原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都 不发生。从操作的角度来描述，事务中的各个操作要么都成功要么都失败

一致性（Consistency）

事务必须使数据库从一个一致性状态变换到另外一个一致性状态。例如转账前A有1000，B有1000。转账后A+B也得是2000。一致性是从数据的角度来说的，（1000，1000） （900，1100），不应该出现（900，1000）

隔离性（Isolation）

事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，每个事务不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。比如：事务1给员工涨工资2000，但是事务1尚未被提交，员工发起事务2查询工资，发现工资涨了2000块钱，读到了事务1尚未提交的数据（脏读）

持久性（Durability） 持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障 也不应该对其有任何影响

事务的隔离级别

不考虑隔离级别，会出现以下情况：（以下情况全是错误的），也即为隔离级别在解决事务并发问题

脏读： 一个线程中的事务读到了另外一个线程中未提交的数据。

不可重复读：一个线程中的事务读到了另外一个线程中已经提交的update的数据（前后内容不一样,针对修改） 场景：

员工A发起事务1，查询工资，工资为1w，此时事务1尚未关闭

财务人员发起了事务2，给员工A张了2000块钱，并且提交了事务

员工A通过事务1再次发起查询请求，发现工资为1.2w，原来读出来1w读不到了，叫做不可重复读

虚读（幻读）：一个线程中的事务读到了另外一个线程中已经提交的insert或者delete的数据（前后条数不一样，针对插入数据） 场景：

事务1查询所有工资为1w的员工的总数，查询出来了10个人，此时事务尚未关闭

事务2财务人员发起，新来员工，工资1w，向表中插入了2条数据，并且提交了事务

事务1再次查询工资为1w的员工个数，发现有12个人

数据库共定义了四种隔离级别： **Serializable（串行化）**：可避免脏读、不可重复读、虚读情况的发生。（串行化） 最高

Repeatable read（可重复读）：可避免脏读、不可重复读情况的发生。(幻读有可能发生) 第二该机制下会对要update的行进行加锁

Read committed（读已提交）：可避免脏读情况发生。不可重复读和幻读一定会发生。 第三

Read uncommitted（读未提交）：最低级别，以上情况均无法保证。(读未提交) 最低

注意：级别依次升高，效率依次降低

MySQL的默认隔离级别是：REPEATABLE READ

查询当前使用的隔离级别： select @@tx_isolation;

设置MySQL事务的隔离级别： set session transaction isolation level xxx; （设置的是当前mysql连接会话的，并不是永久改变的）

Spring中的事务的传播行为

PROPAGATION_REQUIRED	**如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。**
PROPAGATION_SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行。
PROPAGATION_MANDATORY	使用当前的事务，如果当前没有事务，就抛出异常。
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，把当前事务挂起。
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，则抛出异常。
PROPAGATION_NESTED	如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION_REQUIRED类似的操作。

Spring 声明式事务配置

纯xml模式

```
<dependency>
```

```
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.12.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.4</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.1.12.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.1.12.RELEASE</version>
</dependency>
```

XML的配置

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!--定制事务细节，传播行为、隔离级别等-->
    <tx:attributes>
    <!--一般性配置-->
    <tx:method name="*" read-only="false"
    propagation="REQUIRED" isolation="DEFAULT" timeout="-1"/>
    <!--针对查询的覆盖性配置-->
    <tx:method name="query*" read-only="true"
    propagation="SUPPORTS"/>
    </tx:attributes>
</tx:advice>
<aop:config>
    <!--advice-ref指向增强=横切逻辑+方位-->
    <aop:advisor advice-ref="txAdvice"
pointcut="execution(*com.lagou.edu.service.impl.TransferServiceImpl.*(..))"/>
</aop:config>
```

基于XML+注解

```
<!--配置事务管理器-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
<property name="dataSource" ref="dataSource"></property>
</bean>
<!--开启spring对注解事务的支持-->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

在接口、类或者方法上添加@Transactional注解

```
@Transactional(readOnly = true,propagation = Propagation.SUPPORTS)
```

基于纯注解

Spring基于注解驱动开发的事务控制配置，只需要把 xml 配置部分改为注解实现。只是需要一个 注解替换掉xml配置文件中的 `<tx:annotation-driven transaction-manager="transactionManager"/>` 配置。在 Spring 的配置类上添加 @EnableTransactionManagement 注解即可

```
@EnableTransactionManagement//开启spring注解事务的支持
public class SpringConfiguration {
}
```