链滴

Spring 框架学习笔记

作者：rawchen

原文链接：https://ld246.com/article/1597972445175

来源网站：链滴

GitHub：https://github.com/yoyling/SpringStudy

SpringStudy

Spring Framework 5 学习记录4阶段

（1）spring框架的概述以及spring中基于XML的IOC配置

1. spring的概述

spring是什么、两大核心、发展历程和优势、体系结构

2. 程序的耦合及解耦

曾经案例中问题、工厂模式解耦

3. IOC概念和spring中的IOC

spring中基于XML的IOC环境搭建

4. 依赖注入（Dependency Injection）

（2）spring中基于注解的IOC和ioc的案例

1. spring中ioc的常用注解

2. 案例使用xml方式和注解方式实现单表的CRUD操作

持久层技术选择：dbutils

3. 改造基于注解的ioc案例，使用纯注解的方式实现

spring的一些新注解使用

4. spring和Junit整合

（3）spring中的aop和基于XML以及注解的AOP配置

1. 完善我们的account案例

2. 分析案例中问题

3. 回顾之前讲过的一个技术：动态代理

4. 动态代理另一种实现方式

5. 解决案例中的问题

6. AOP的概念

7. spring中的AOP相关术语

8. spring中基于XML和注解的AOP配置

（4）spring中的JdbcTemlate以及Spring事务控制

1. spring中的JdbcTemplate

- JdbcTemplate的作用：

它就是用于和数据库交互的，实现对表的CRUD操作

- 如何创建该对象：
- 对象中的常用方法：

2. 作业：

- spring基于AOP的事务控制

3. spring中的事务控制

- 基于XML的
- 基于注解的

01-01jdbc

程序的耦合

- **耦合**：程序间的依赖关系

包括：

- 类之间的依赖
- 方法间的依赖

- **解耦**：
  - 降低程序间的依赖关系
- 实际开发中：
  - 应该做到：编译期不依赖，运行时才依赖。
- 解耦的思路：
  - 第一步：使用反射来创建对象，而避免使用new关键字。
  - 第二步：通过读取配置文件来获取要创建的对象全限定类名

接下来的Spring学习数据库都为springtest，表只有一个为account。

springtest库 -> account表

```
create table account(
    id int primary key auto_increment,
    name varchar(40),
    money float
)character set utf8 collate utf8_general_ci;
```

```
insert into account(name,money) values('aaa',1000);
insert into account(name,money) values('bbb',1000);
insert into account(name,money) values('ccc',1000);
```

下面演示了 **JDBC** 查询数据库数据的一般步骤。

```
public static void main(String[] args) throws  Exception{
    //1.注册驱动
    //DriverManager.registerDriver(new com.mysql.jdbc.Driver());
    Class.forName("com.mysql.jdbc.Driver");

    //2.获取连接
    Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/springtest",
root","root");
    //3.获取操作数据库的预处理对象
    PreparedStatement pstm = conn.prepareStatement("select * from account");
    //4.执行SQL，得到结果集
    ResultSet rs = pstm.executeQuery();
    //5.遍历结果集
    while(rs.next()){
        System.out.println(rs.getString("name"));
    }
    //6.释放资源
    rs.close();
    pstm.close();
    conn.close();
}
```

01-02factory

一个创建Bean对象的工厂

Bean：在计算机英语中，有可重用组件的含义。

JavaBean：用java语言编写的可重用组件。

javabean > 实体类

它就是创建我们的service和dao对象的。

- 第一个：需要一个配置文件来配置我们的 **service** 和**dao**
  - 配置的内容：唯一标识=全限定类名（key = value)
- 第二个：通过读取配置文件中配置的内容，反射创建对象
  - 我的配置文件可以是 **xml**也可以是**properties**

```
public class BeanFactory {
    //定义一个Properties对象
    private static Properties props;

    //定义一个Map,用于存放我们要创建的对象。我们把它称之为容器
    private static Map<String,Object> beans;

    //使用静态代码块为Properties对象赋值
```

```
    static {
        try {
            //实例化对象
            props = new Properties();
            //获取properties文件的流对象
            InputStream in = BeanFactory.class.getClassLoader().getResourceAsStream("bean.pro
perties");
            props.load(in);
            //实例化容器
            beans = new HashMap<String,Object>();
            //取出配置文件中所有的Key
            Enumeration keys = props.keys();
            //遍历枚举
            while (keys.hasMoreElements()){
                //取出每个Key
                String key = keys.nextElement().toString();
                //根据key获取value
                String beanPath = props.getProperty(key);
                //反射创建对象
                Object value = Class.forName(beanPath).newInstance();
                //把key和value存入容器中
                beans.put(key,value);
            }
        }catch(Exception e){
            throw new ExceptionInInitializerError("初始化properties失败！");
        }
    }

    /**
     * 根据bean的名称获取对象
     * @param beanName
     * @return
     */
    public static Object getBean(String beanName){
        return beans.get(beanName);
    }
}
```

bean.properties

```
accountService=com.yoyling.service.impl.AccountServiceImpl
accountDao=com.yoyling.dao.impl.AccountDaoImpl
```

01-03spring

把对象的创建交给spring来管理

1. 获取核心容器对象

```
ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");
```

2. 根据id获取Bean对象

```
IAccountService as = (IAccountService)ac.getBean("accountService");
IAccountDao adao = ac.getBean("accountDao",IAccountDao.class);
```

----------BeanFactory----------

Resource resource = new ClassPathResource("bean.xml");

BeanFactory beanFactory = new XmlBeanFactory(resource);

IAccountService as = (IAccountService)beanFactory.getBean("accountService");

* ApplicationContext的三个常用实现类

  - **ClassPathXmlApplicationContext** 可以加载类路径下的配置文件
  - **FileSystemXmlApplicationContext** 可以加载磁盘任意路径下的配置文件(必须有访问权限)
  - **AnnotationConfigApplicationContext** 它是用于读取注解创建容器的

* 核心容器的两个接口引发的出的问题

  * **ApplicationContext**   单例对象适用   采用此接口

    它在构建核心容器时，创建对象采取的策略是立即加载的方式，一读取文件马上就创建配置文件中置的对象。

  * **BeanFactory**   多例对象适用

    它在构建核心容器时，创建对象采取的策略是延迟加载的方式，什么时候根据id获取对象了，什么候才真正创建了对象。

01-04bean

spring对bean的管理细节

1. 创建bean的三种方式

   - 第一种方式：**适用默认构造函数创建。**

   - 在spring的配置文件中使用bean标签，配以id和class属性之后，且没有其他属性和标签时。
   - 采用的就是默认构造函数创建bean对象，此时如果类中没有默认构造函数，则对象无法创建。

   ```Java
   <bean id="accountService" class="com.yoyling.service.impl.AccountServiceImpl"></bean>
   ```

● 第二种方式： **使用普通工厂中的方法创建对象** (使用某个类中的方法创建对象，并存入spring容器)

```
<bean id="instanceFactory" class="com.yoyling.factory.InstanceFactory"></bean>
<bean id="accountService" factory-bean="instanceFactory" factory-method="getAccountSer
ice"></bean>
```

● 第三种方式： **使用工厂中的静态方法创建对象**（使用某个类中的静态方法创建方法并存入spring器）

```
<bean id="accountService" class="com.yoyling.factory.StaticFactory"></bean>
```

2. bean对象作用范围

- bean标签的 **scope**属性：
    - 作用：用于指定bean的作用范围
    - 取值：常用单例和多例
        - **singleton** 单例的（默认值）
        - **prototype** 多例的
        - **request** 作用于web应用的请求范围
        - **session** 作用于web应用的绘画范围
        - **global-session** 作用于集群环境的会话范围（全局会话范围），当不是集群环境时，就是session

```
<bean id="accountService" class="com.yoyling.service.impl.AccountServiceImpl" scope="prototype"></bean>
```

3. bean对象的生命周期

- 单例对象-和容器相同（立即）
    - 出生：当容器创建时对象出生
    - 活着：只要容器还在，对象一直活
    - 死亡：容器销毁，对象销毁
- 多例对象（延迟）
    - 出生：当我们使用对象时，spring框架为我们创建
    - 活着：对象只要在使用过程中就一直活着
    - 死亡：当对象长时间不用，且没有别的对象引用时，由Java垃圾回收器回收

01-05DI

依赖注入：

- Dependency Injection

IOC的作用：

- 降级程序间的耦合（依赖关系）

依赖关系的管理：

- 以后都交给了spring来维护

在当前类中需要用到其它类的对象，由spring为我们提供，我们只需要在配置文件中说明

依赖关系的维护：

- 就称之为依赖注入。

依赖注入：

- 能注入的数据有三类：
  - 基本类型和string
  - 其他bean类型（在配置文件中或者注解配置过的bean）
  - 复杂类型/集合类型
- 注入的方式有三种：
  - 第一种：使用构造函数提供
  - 第二种：使用set方法
  - 第三种：使用注解提供

构造函数注入：

- 使用 **constructor-arg**
- 便签出现的位置：bean标签的内部
- 标签中的属性：
  - **type**：用于指定要注入的数据的数据类型，也是构造函数中某些参数的类型
  - **index**：用指定要注入的数据给构造函数中的指定索引位置的参数赋值，索引从0开始
  - **name**：用于指定给构造函数中指定名称的参数赋值 （常用的）
  - ==========以上三个用于指定给构造函数中哪个参数赋值==========
  - **value**：用于提供基本类型和String类型的数据
  - **ref**：用于指定其它的bean类型数据，它指的就是在spring的IOC核心容器中出现过的bean对象
- 优势：
  - 在获取bean对象时，注入数据是必须的操作，否则对象无法创建成功。
- 弊端：
  - 改变了bean对象的实例化方式，使我们在创建对象时，如果用不到这些数据也必须提供。

```
<bean id="accountService" class="com.yoyling.service.impl.AccountServiceImpl">
   <constructor-arg name="name" value="泰雷瑟"></constructor-arg>
   <constructor-arg name="age" value="18"></constructor-arg>
   <constructor-arg name="birthday" ref="now"></constructor-arg>
</bean>

<!-- 配置一个日期对象 -->
<bean id="now" class="java.util.Date"></bean>
```

set方法注入（更常用）

- 涉及的标签： **property**
- 出现的位置：bean标签的内部
- 标签的属性：
  - **name**：用于指定注入时所调用的set名称
  - **value**：用于提供基本类型和String类型的数据
  - **ref**：用于指定其它的bean类型数据，它指的就是在spring的IOC核心容器中出现过的bean对象
- 优势：
  - 创建对象时，没有明确限制，可直接使用默认构造函数
- 弊端：
  - 如果有某个成员必须有值，则获取对象时有可能set方法没有执行。

```
<bean id="accountService2" class="com.yoyling.service.impl.AccountServiceImpl2">
   <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/集合类型的注入

- 用于给List结构集合注入的标签有：
  - **List、Array、Set**
- 用于给Map结构集合注入的标签有：
  - **Map、Props**
- 结构相同标签可以互换

```
<bean id="accountService3" class="com.yoyling.service.impl.AccountServiceImpl3">
   <property name="myStrs">
      <array>
         <value>AAA</value>
         <value>BBB</value>
         <value>CCC</value>
      </array>
   </property>

   <property name="myList">
      <list>
         <value>AAAList</value>
         <value>BBBList</value>
         <value>CCCList</value>
      </list>
   </property>

   <property name="mySet">
      <set>
```

```
            <value>AAASet</value>
            <value>BBBSet</value>
            <value>CCCSet</value>
        </set>
    </property>

    <property name="myMap">
        <map>
            <entry key="a" value="AAAMap"></entry>
            <entry key="b">
                <value>BBBMap</value>
            </entry>
        </map>
    </property>

    <property name="myProps">
        <props>
            <prop key="textC">CCCProps</prop>
            <prop key="textD">DDDProps</prop>
        </props>
    </property>
</bean>
```

02-01anno_IoC

- 曾经XML配置：

```
<bean id="accountService" class="com.yoyling.service.impl.AccountServiceImpl"
    scope="" init-method="" destroy-method="" >
    <property name="" value="" | ref=""></property>
</bean>
```

用于创建对象的注解

- 它们的作用就和在XML配置文件中编写一个 <bean>标签实现的功能是一样的
- **@Component**
- 作用：用于把当前类存入spring容器中
    - 属性：value 用于指定bean的id，不写时，默认值为当前类名，且首字母改小写。
- **@Controller**

------ 一般用在表现层

- **@Service**

------ 一般用在业务层

- **@Repository**

------ 一般用于持久层

- 以上三个注解他们的作用和属性与 **Component**是一模一样。
- 他们三个是spring框架为我们提供明确的三层使用的注解，使我们的三层对象更加清晰。

用于注入数据的注解

- 它们的作用就和在XML配置文件中的 <bean>标签中写一个<property>标签的作用是一样的
- **@Autowired**:
- 作用：自动按照类型注入，只要容器中有唯一的一个bean对象类型和要注入的变量类型匹配就可以入成功
  - 如果ioc容器中没有任何bean的类型和要注入的变量类型匹配，则报错。
  - 如果ioc容器中有多个类型匹配时：
- 出现位置：
  - 可以是变量上，也可以是方法上
- 细节：
  - 在使用注解注入时，set方法就不是必须的。
- **@Qualifier**:
- 作用：在按照类中注入的基础之上再按照名称注入。它在给类成员注入时不能单独使用。但是在给法参数注入时可以
  - 属性：
    - value：用于指定注入bean的id。
- **@Resource**:
  - 作用：直接按照bean的id注入，它可以独立使用
  - 属性：
    - name：用于指定bean的id
  - 以上三个注解都只能注入其他bean类型的数据，而基本类型和String类型无法使用上述注解实。
  - 另外集合类型的注入只能通过XML来实现。
- **@Value**:
  - 作用：用于注入基本类型和String类型的数据
  - 属性：
    - value：用于指定数据的值，它可以用spring中SpEL(Spring的el表达式)
      - **SpEL的写法：${表达式}**

用于改变作用范围的

- 它们的作用就和在 <bean>标签中使用scope属性实现的功能是一样的
- **@Scope**
  - 作用：用于指定bean的作用范围

- 属性：
    - value：指定范围的取值。常用取值　**singleton****prototype**

和生命周期相关(了解)

- 它们的作用就和在　<bean>标签中使用**init-method**、**destroy-method**是一样的
- **@PreDestroy**
    - 作用：用于指定销毁方法
- **@PostCoustruct**
    - 用于指定初始化方法

---

bean.xml

```
<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">

    <!-- 告知spring在创建容器时要扫描的包，配置所需的标签不是在beans这个约束中而是一个名称context名称空间和约束中-->

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

@Component("accountService")

IAccountService as = (IAccountService)ac.getBean("accountService");
```

02-02/03 xml_ioc/anno_ioc

基于xml的配置代码、基于set与构造方法注入相关代码注释均在：

day_02_02account_xml_ioc

day_02_03account_anno_ioc

由于无法对jar包中的类实现注解，因此采用 **SpringConfiguration** 配置类来实现无xml，下面将介怎么使用配置类

02_04annoioc_withoutxml

```
package com.yoyling.config;
```

```
import com.mchange.v2.c3p0.ComboPooledDataSource;
import org.apache.commons.dbutils.QueryRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;
import java.beans.PropertyVetoException;

/**
 * 该类为一个配置类，它的作用和bean.xml是一样的
 * spring中的新注解：
 *
 * @Configuaration
 *    作用：指定当前类为一个配置类
 * @ComponentScan
 *    作用：用于通过注解指定spring在创建容器时要扫描的包
 *    属性：
 *        value：它和basePackages的作用是一样的，都是用于指定创建容器时要扫描的包。
 *            我们使用此注解等同于在xml中配置了
 *            <context:component-scan base-package="com.yoyling"></context:component-sca
>
 * @Bean
 *    作用：用于把当前方法的返回值作为bean对象存入spring的ioc容器中
 *    属性：
 *        name:用于指定bean的id，不写时默认值是当前方法的名称
 *    细节：
 *        当我们使用注解配置方法时，如果方法有参数，spring框架会去容器中查找有没有可用的bea
对象。
 *        查找方式和Autowired注解的作用是一样的
 * @Import
 *    作用：用于导入其它配置类
 *    属性：
 *        value：用于指定其他配置类的字节码
 *        当我们使用Import注解后，有Import注解的类就是父配置类，而导入的都是子配置类
 * @PropertySource
 *    作用：用于指定properties文件的位置
 *    属性：
 *        value：指定文件的名称和路径
 *            关键字：classpath：表示类路径下
 * 注解和xml如何选择？没有选择权利下以公司为准。
 * 实际开发中哪种更方便用哪种配置，如果存在jar包中的类用xml更直接更省事。自己写的类注解更
便。
 */
@Configuration
@ComponentScan("com.yoyling")
public class SpringConfiguration {

    /**
     * 用于创建一个QueryRunner对象
     * @param dataSource
     * @return
     */
```

```
    @Bean(name = "queryRunner")
    public QueryRunner createQueryRunner(DataSource dataSource) {
        return new QueryRunner(dataSource);
    }

    /**
     * 创建数据源对象
     * @return
     */
    @Bean(name = "dataSource")
    public DataSource createDataSource() {
        try {
            ComboPooledDataSource ds = new ComboPooledDataSource();
            ds.setDriverClass("com.mysql.jdbc.Driver");
            ds.setJdbcUrl("jdbc:mysql://localhost:3306/springtest");
            ds.setUser("root");
            ds.setPassword("root");
            return ds;
        } catch (PropertyVetoException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

现在就可以在**Junit**单元测试中注释掉**ClassPathXmlApplicationContext**配置的**xml**文件的代码了

```
//ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");
  ApplicationContext ac = new AnnotationConfigApplicationContext(SpringConfiguration.class
;
```

现在的**QueryRunner**对象是单例的，给它一个**Scope**注解就是多例对象了

```
@Scope("prototype")
@Bean(name = "queryRunner")
public QueryRunner createQueryRunner(DataSource dataSource) {
    return new QueryRunner(dataSource);
}
```

如果想把**SpringConfiguration**里面的QueryRunner和DataSource配置拿出来拆分一个**JdbcConfig**则就需要确定**SpringConfiguration**为主配置类，加**@Configuration**注解，再加上**@Import(Jdbc onfig.class)**注解。

这样就不用对测试类中**AnnotationConfigApplicationContext()加入两个反射class，也不用对ComponentScan**数组多添加一个扫描路径。

把jdbc配置放到properties文件中，这样就脱离了class文件。

```
@Value("${jdbc.driver}")
private String driver;

@Value("${jdbc.url}")
private String url;

@Value("${jdbc.username}")
```

```
private String username;

@Value("${jdbc.password}")
private String password;

/**
 * 创建数据源对象
 * @return
 */
@Bean(name = "dataSource")
public DataSource createDataSource() {
    try {
        ComboPooledDataSource ds = new ComboPooledDataSource();
        ds.setDriverClass(driver);
        ds.setJdbcUrl(url);
        ds.setUser(username);
        ds.setPassword(password);
        return ds;
    } catch (PropertyVetoException e) {
        e.printStackTrace();
        return null;
    }

}
```

jdbcConfig.properties

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/springtest
jdbc.username=root
jdbc.password=root
```

注解和xml如何选择？没有选择权利下以公司为准，实际开发中哪种更方便用哪种配置，如果存在jar中的类用xml更直接更省事。自己写的类注解更方便。

Spring5整合Junit单元测试

使用Junit单元测试：测试我们的配置

1. 应用程序的入口

main方法

2. junit单元测试中没有main方法也能执行

junit集成了一个main方法

该方法就会判断当前测试类中那些方法有**@Test**注解

junit就会让有Test注解的方法执行

3. junit不会管我们是否采用spring框架

在执行测试方法时，junit根本不知道我们是不是使用了spring框架

所以也就不会为我们读取配置文件/配置类创建sping核心容器

4. 由以上三点可知

当测试方法执行时，没有ioc容器，就算写了Autowired注解，也无法实现注入

Spring整合了junit的配置

1. 导入spring整合junit的jar（坐标）
2. 使用Junit提供的一个注解把原来的main方法替换了，替换为spring提供的

@Runwith

3. 告知spring的运行器，spring和ioc创建是基于xml还是注解的，并说明位置

@ContextConfiguration

locations:指定xml文件的位置+classpath关键字，表示在类路径下

classes:指定注解类所在的位置

当我们使用spring5.x的b版本的时候，要求junite的jar必须是4.12及以上

当我们使用spring5.x的b版本的时候，要求junite的jar必须是4.12及以上
当我们使用spring5.x的b版本的时候，要求junite的jar必须是4.12及以上

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SpringConfiguration.class)
public class AccountServiceTest {

//    private ApplicationContext ac;
    @Autowired
    private AccountService as;

//    @Before
//    public void init() {
//        //1.获取容器
//        ApplicationContext ac = new AnnotationConfigApplicationContext(SpringConfiguratio
.class);
//        //2.得到业务层对象
//        AccountService as = ac.getBean("accountService",AccountService.class);
//    }

    @Test
    public void testFinAll() {
        //3.执行方法
        List<Account> accounts = as.findAllAccount();
        for (Account account : accounts) {
            System.out.println(account);
        }
    }
}
```

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
```

xml配置的spring整合Junit单元测试（参考 day_02_02account_xml_ioc）

```
/**
 * 使用Junit单元测试：测试我们的配置
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:bean.xml")
public class AccountServiceTest {

    @Autowired
    private AccountService as;

    @Test
    public void testFinAll() {
//        //1.获取容器
//        ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");
//        //2.得到业务层对象
//        AccountService as = ac.getBean("accountService",AccountService.class);
        //3.执行方法
        List<Account> accounts = as.findAllAccount();
        for (Account account : accounts) {
            System.out.println(account);
        }
    }
}
```

03_01account

对于之前例子**AccountServiceImpl**新增一个转账方法，如果中间出现算数异常则造成钱款数据异常因此spring需要解决的事就是对于事务一致性的问题。

```
public void transfer(String sourceName, String targetName, Float money) {
    //1.根据名称查询转出账户
    Account source = accountDao.findAccountByName(sourceName);
    //2.根据名称查询转入账户
    Account target = accountDao.findAccountByName(targetName);
    //3.转出账户减钱
    source.setMoney(source.getMoney()-money);
    //4.转入账户加钱
    target.setMoney(target.getMoney()+money);
    //5.更新转出账户
    accountDao.updateAccount(source);
    int i = 1/0;
    //6.更新转入账户
    accountDao.updateAccount(target);
}
```

代码看出与数据库交互4次或者获取了4次连接，每个**Connection**都有自己独立的事务。

ThreadLocal 对象解决事务一致性的问题：

把**Connection**和当前线程绑定，从而使一个线程中只有一个能控制事务的对象。

编写一个连接工具类，它用于从数据源中获取一个连接，并且实现和线程的绑定。

编写一个事务管理相关工具类，它包含了开启事务、提交事务、回滚事务和释放事务。

ConnectionUtils.java

```
package com.yoyling.utils;


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

/**
 * 连接的工具类，它用于从数据源中获取一个连接，并且实现和线程的绑定
 */
public class ConnectionUtils {

    private ThreadLocal<Connection> tl = new ThreadLocal<Connection>();

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    /**
     * 获取当前线程上的连接
     * @return
     */
    public Connection getThreadConnection() {
        try {
            //1.先从ThreadLocal上获取
            Connection conn = tl.get();
            //2.判断当前线程上是否有连接
            if (conn == null) {
                //3.从数据源中获取一个连接，并且存入ThreadLocal中,线程绑定
                conn = dataSource.getConnection();
                tl.set(conn);
            }
            //4.返回当前线程上的连接
            return conn;
        } catch (SQLException e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
```

```
     * 把连接和线程解绑
     */
    public void removeConnection() {
        tl.remove();
    }
}
```

TransactionManager.java

```
package com.yoyling.utils;

import java.sql.SQLException;

/**
 * 和事务管理相关的工具类，它包含了开启事务、提交事务、回滚事务和释放事务
 */
public class TransactionManager {

    private ConnectionUtils connectionUtils;

    public void setConnectionUtils(ConnectionUtils connectionUtils) {
        this.connectionUtils = connectionUtils;
    }

    /**
     * 开启事务
     */
    public void beginTransaction() {
        try {
            connectionUtils.getThreadConnection().setAutoCommit(false);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    /**
     * 提交事务
     */
    public void commit() {
        try {
            connectionUtils.getThreadConnection().commit();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    /**
     * 回滚事务
     */
    public void rollback() {
        try {
            connectionUtils.getThreadConnection().rollback();
        } catch (SQLException e) {
            e.printStackTrace();
```

```
        }
    }

    /**
     * 释放连接
     */
    public void release() {
        try {
            connectionUtils.getThreadConnection().close();//还回连接池中
            connectionUtils.removeConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

AccountServiceImpl.java

```
package com.yoyling.service.impl;

import com.yoyling.dao.AccountDao;
import com.yoyling.domain.Account;
import com.yoyling.service.AccountService;
import com.yoyling.utils.TransactionManager;

import java.util.List;

/**
 * 账户的业务实现类
 *
 * 事务控制应该都是在业务层
 */
public class AccountServiceImpl_OLD implements AccountService {

    private AccountDao accountDao;
    private TransactionManager transactionManager;

    public void setTransactionManager(TransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    public void transfer(String sourceName, String targetName, Float money) {
        try {
            //1.开启事务
            transactionManager.beginTransaction();
            //2.执行操作
            //2.1根据名称查询转出账户
            Account source = accountDao.findAccountByName(sourceName);
            //2.2根据名称查询转入账户
            Account target = accountDao.findAccountByName(targetName);
```

```
            //2.3转出账户减钱
            source.setMoney(source.getMoney()-money);
            //2.4转入账户加钱
            target.setMoney(target.getMoney()+money);
            //2.5更新转出账户
            accountDao.updateAccount(source);
            int i = 1/0;
            //2.6更新转入账户
            accountDao.updateAccount(target);
            //3.提交事务
            transactionManager.commit();
        } catch (Exception e) {
            e.printStackTrace();
            //4.回滚操作
            transactionManager.rollback();
        }finally {
            //5.释放连接
            transactionManager.release();
        }
    }
}
```

03_02proxy

最后在bean.xml中注入各个类带有set方法的对象到相应的bean中。我们可以发现，servlceImpl中码冗余了，如果出现其他任意FindAll()、deleteAccount()方法，则要多次写入以上5个事件，且如果务方法名改变则需要重构所有service层类文件相同的方法。

因此下面我们就会使用动态代理来解决对于方法的增强，主要有**JDK官方的Proxy**、以及**cglib库**。

参考之前文章：框架底层核心知识点分析总结

参考最近的文章：动态代理 | CGlib | AOP

后面Spring框架为我们提供AOP编程，使用注解简化了以上步骤，Spring框架在使用代理模式时，会据当前java文件是类或者是接口，然后采用不同的代理方式，在spring4.0以后的版本（自动整合了cglb代理）

JDK动态代理和CGLIB动态代理区别：

- JDK动态代理基于接口实现的。也就是说，如果使用JDK动态代理，必须提供接口

CGLIB动态代理基于类实现的。也就是说，只需提供一个类即可，CGLIB会给这个类生成代理对象

proxy\Client.java

```
public class Client {
    public static void main(String[] args) {
        final Producer producer = new Producer();

        /**
         * 动态代理：
         * 特点：字节码随用随创建，随用随加载
         * 作用：不修改源码基础上对方法增强
```

```
        * 分类：
        *   基于接口的动态代理
        *   基于子类的动态代理
        * 基于接口的动态代理：
        *   涉及的类：Proxy
        *   提供者：JDK官方
        * 如何创建代理对象：
        *   使用Proxy类中的newProxyInstance方法
        * 创建代理对象的要求：
        *   被代理类最少实现一个接口，如果没有则不能使用
        * newProxyInstance方法的参数：
        *   ClassLoader 类加载器
        *     用于加载代理对象字节码的，和被代理对象使用相同的类加载器，固定写法。
        *   Class[] 字节码数组
        *     让代理对象和被代理对象有相同方法，固定写法。
        *   InvocationHandler 用于提供增强的代码
        *     写如何代理，我们一般是写一个接口实现类，通常情况是匿名内部类，但不是必须的
        *     此接口的实现类都是谁用谁写。
        *
        */
       IProducer proxyProducer = (IProducer) Proxy.newProxyInstance(producer.getClass().getC
assLoader(), producer.getClass().getInterfaces(), new InvocationHandler() {
           /**
            * 作用：执行被代理对象的任何借口方法都会经过该方法
            * 方法参数的含义
            * @param proxy     代理对象的引用
            * @param method    当前执行的方法
            * @param args      当前执行方法所需的参数
            * @return          和被代理对象方法有相同的返回值
            * @throws Throwable
            */
           public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
               //提供增强的代码
               Object returnValue = null;
               //1.获取方法执行的参数
               Float money = (Float)args[0];
               //2.判断当前方法是不是销售
               if ("saleProduct".equals(method.getName())) {
                   returnValue = method.invoke(producer,money * 0.8f);
               }
               return returnValue;
           }
       });
       proxyProducer.saleProduct(10000f);

   }
}
```

cglib\Client.java

```
public class Client {
    public static void main(String[] args) {
        final Producer producer = new Producer();
```

```
        /**
         * 动态代理：
         * 特点：字节码随用随创建，随用随加载
         * 作用：不修改源码基础上对方法增强
         * 分类：
         *   基于接口的动态代理
         *   基于子类的动态代理
         * 基于子类的动态代理：
         *   涉及的类：Enhancer
         *   提供者：cglib库
         * 如何创建代理对象：
         *   使用Enhancer类中的create方法
         * 创建代理对象的要求：
         *   被代理类不能是最终类
         * create方法的参数：
         *   Class 字节码
         *      它是用于指定被代理对象的字节码。
         *   Callback 用于提供增强的代码
         *      写如何代理，我们一般是写一个接口实现类，通常情况是匿名内部类，但不是必须的
         *      此接口的实现类都是谁用谁写。
         *      我们一般写的都是该接口的子接口实现类：MethodInterceptor
         *
         */
        Producer cglibProducer = (Producer) Enhancer.create(producer.getClass(), new MethodIn
erceptor() {
            /**
             * 执行被代理对象的任何方法都会经过该方法
             * @param proxy
             * @param method
             * @param args
             *     以上三个参数和基于接口的动态代理中invoke方法的参数是一样的
             * @param methodProxy   当前执行方法的代理对象
             * @return
             * @throws Throwable
             */
            public Object intercept(Object proxy, Method method, Object[] args, MethodProxy me
hodProxy) throws Throwable {
                //提供增强的代码
                Object returnValue = null;
                //1.获取方法执行的参数
                Float money = (Float)args[0];
                //2.判断当前方法是不是销售
                if ("saleProduct".equals(method.getName())) {
                    returnValue = method.invoke(producer,money * 0.8f);
                }
                return returnValue;
            }
        });
        cglibProducer.saleProduct(12000f);
    }
}
```

在JDK8之前，如果我们在匿名内部类中需要访问局部变量，那么这个局部变量必须用final修饰符修。

cglib 动态代理需要引入坐标：

```
<dependencies>
  <dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>3.1</version>
  </dependency>
</dependencies>
```

AOP

Aspect Oriented Programming，面向切面编程。通过预编译方式和运行期动态代理实现程序功能统一维护的一种技术。是OOP的延续，软件开发的热点，也是Spring框架中的一个重要内容，是函数编程的一种衍生范例。利用AOP可以对业务逻辑各部分进行隔离。使业务逻辑各部分之间的耦合度降。提高程序重用性，提高开发效率。

简单说它就是把程序重复的代码抽取出来，需要执行时使用动态代理技术在不修改源码的基础上，对们已有方法进行增强。

作用：

- 程序运行期间，不修改源码对已有方法进行增强。

优势：

- 减少重复代码
- 提高开发效率
- 维护方便

我们学习Spring的aop，就是通过配置的方式，实现上一章内容。

AOP相关术语：

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只支持方法类型连接点。

Pointcut（切入点）：

- 所谓切入点是指我们要对哪些Joinpoint进行拦截的定义。

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。

学习Spring中的AOP要明确的事

a、开发阶段(我们做的)

● 编写核心业务代码(开发主线):大部分程序员来做，要求熟悉业务需求。
● 把公用代码抽取出来，制作成通知。(开发阶段最后再做) : AOP编程人员来做。
● 在配置文件中，声明切入点与通知间的关系，即切面。: AOP编程人员来华。

b、运行阶段(spring框架完成的)

● Spring框架监控切入点方法的执行。一旦监控到切入点方法被运行，使用代理机制，动态创建目标象的代理对象，根据通知类别，在代理对象的对应位置，将通知对应的功能织入，完成完整的代码逻运行。

03_03springAOP

记录日志：通过aop编写记录日志的工具类Logger，计划让其在切入点方法之前执行打印日志方法printLog()（切入点方法就是业务层方法）

Spring中基于XML的AOP配置步骤：

1. 把通知的Bean也交给Spring来管理
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** 标签内部使用对应的标签来配置通知的类型

我们现在的实例是让printLog方法在切入点方法执行之前执行，所以是前置通知**aop:before** 表示配前置通知**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

达式，该表达式含义指的是对业务层中哪些方法增强

切入点的表达式写法：

关键字：**execution(表达式)**

表达式：**访问修饰符 返回值 包名.包名...类名.方法名(参数列表)**

标准表达式写法：**public void com.yoyling.service.impl.AccountServiceImpl.saveAccount()**

bean.xml

```
<?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: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/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">
    <!-- 配置Spring的IOC，把service对象配置进来 -->
    <bean id="accountService" class="com.yoyling.service.impl.AccountServiceImpl"></bean>

    <!-- 配置Logger类 -->
    <bean id="logger" class="com.yoyling.utils.Logger"></bean>

    <!-- 配置AOP -->
    <aop:config>
        <!-- 配置切面 -->
        <aop:aspect id="logAdvice" ref="logger">
            <!-- 配置通知的类型，且建立通知方法和切入点方法的关联 -->
            <aop:before method="printLog" pointcut="execution(public void com.yoyling.service.
mpl.AccountServiceImpl.saveAccount())"></aop:before>
        </aop:aspect>
    </aop:config>
</beans>
```

pom.xml

```
<dependencies>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.0.2.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.8.7</version>
    </dependency>

</dependencies>
```

切入点表达式的通用写法：

访问修饰符 返回值 包名.包名...类名.方法名(参数列表)

标准表达式写法：

public void com.yoyling.service.impl.AccountServiceImpl.saveAccount()

访问修饰符可省略

void com.yoyling.service.impl.AccountServiceImpl.saveAccount()

返回值可以使用通配符，表示任意返回值

* com.yoyling.service.impl.AccountServiceImpl.saveAccount()

包名可以使用通配符，表示任意包，但是有几级包就写几个 *.

* *.*.*.*.AccountServiceImpl.saveAccount()

包名可使用..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：

- 可以写基本类型：

基本类型直接写名称 **int**
引用类型写包名.类名 **java.lang.String**

- 可以使用通配符表示任意类型但必须有参数
- 可以使用..有无参数均可，参数类型可任意

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.yoyling.service.impl.*.*(..)

03_04adviceType

4种常用的通知类型：

- **前置aop:before****
- **后置aop:after-returning**
- **异常aop:after-throwing**

- **最终aop:after**

配置切入点表达式 **id**属性用于指定表达式的唯一标志。**expression**属性用于指定表达式内容

- 此标签写在 **aop:aspect**标签内部只能当前切面使用。
- 它还可以写在 **aop:aspect**外面，此时就变成了所有切面可用。

spring约束**aop:pointcut**标签必须出现在切面之前。

```
<!-- 配置AOP -->
<aop:config>
    <aop:pointcut id="pt1" expression="execution(* com.yoyling.service.impl.*.*(..))"/>

    <!-- 配置切面 -->
    <aop:aspect id="logAdvice" ref="logger">
        <!-- 配置前置通知：在切入点方法执行之前执行 -->
        <aop:before method="beforePrintLog" pointcut-ref="pt1"></aop:before>

        <!-- 前置后置通知：在切入点方法正常执行之后执行。它和异常通知永远只能执行一个 -->
        <aop:after-returning method="afterReturningPrintLog" pointcut-ref="pt1"></aop:after-
eturning>

        <!-- 配置异常通知：在切入点方法执行产生异常之后执行。它和后置通知永远只能执行一个 -->
        <aop:after-throwing method="afterThrowingPrintLog" pointcut-ref="pt1"></aop:after-
hrowing>

        <!-- 配置最终通知：无论切入点方法是否正常执行它都会在其后面执行 -->
        <aop:after method="afterPrintLog" pointcut-ref="pt1"></aop:after>
    </aop:aspect>
</aop:config>
```

环绕通知：

问题：

当我们配置了环绕通知之后，切入点方法没有执行，而通知方法执行了。

分析：

通过对比动态代理中的环绕通知代码，发现**动态代理中的环绕通知**有**明确的切入点方法调用**，而我们代码中没有。

解决：

Spring框架为我们提供了一个接口**ProceedingJoinPoint**。该接口有一个方法**proceed()**，此方法相当于明确调用切入点的方法。

该接口可以作为环绕通知的方法参数，在程序执行时spring框架会为我们提供该接口实现类供我们使。

spring中的环绕通知：

它是spring框架为我们提供的一种可以在代码中手动控制增强方法何时执行的方式

```
<aop:config>
```

```
    <aop:pointcut id="pt1" expression="execution(* com.yoyling.service.impl.*.*(..))"/>

        <!-- 配置切面 -->
        <aop:aspect id="logAdvice" ref="logger">

        <!-- 配置环绕通知 详细的注释请看Logger类中 -->
        <aop:around method="aroundPrintLog" pointcut-ref="pt1"></aop:around>

    </aop:aspect>
</aop:config>
```

Logger.java

```
public Object aroundPrintLog(ProceedingJoinPoint pjp) {
    Object rtValue = null;
    try {
        Object[] args = pjp.getArgs();//得到方法执行所须的参数
        System.out.println("Logger类中的aroundPrintLog方法开始记录日志..前置");
        rtValue = pjp.proceed(args);//明确调用业务层方法（切入点方法）
        System.out.println("Logger类中的aroundPrintLog方法开始记录日志..后置");
        return rtValue;
    } catch (Throwable throwable) {
        System.out.println("Logger类中的aroundPrintLog方法开始记录日志..异常");
        throw new RuntimeException(throwable);
    } finally {
        System.out.println("Logger类中的aroundPrintLog方法开始记录日志..最终");
    }
}
```

03_05annotationAOP

基于注解的AOP：

@Service("accountService")　@Component("logger")

@Before("pt1()")　@AfterReturning("pt1()")　@AfterThrowing("pt1()")　@After("pt1()")　@Around("pt1()")

@Aspect　@Pointcut("execution()")

```
<?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:aop="http://www.springframework.org/schema/aop"
       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/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

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

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

Logger.java

```
package com.yoyling.utils;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

/**
 * 用于记录日志的工具类，它提供了公共的代码
 */
@Component("logger")
@Aspect//表示当前类是一个切面
public class Logger {

    @Pointcut("execution(* com.yoyling.service.impl.*.*(..))")
    private void pt1(){}

//    @Before("pt1()")
    public void beforePrintLog() {
        System.out.println("前置通知Logger类中的beforePrintLog方法开始记录日志");
    }

//    @AfterReturning("pt1()")
    public void afterReturningPrintLog() {
        System.out.println("后置通知Logger类中的afterReturningPrintLog方法开始记录日志");
    }

//    @AfterThrowing("pt1()")
    public void afterThrowingPrintLog() {
        System.out.println("异常通知Logger类中的afterThrowingPrintLog方法开始记录日志");
    }

//    @After("pt1()")
    public void afterPrintLog() {
        System.out.println("最终通知Logger类中的afterPrintLog方法开始记录日志");
    }

    @Around("pt1()")
    public Object aroundPrintLog(ProceedingJoinPoint pjp) {
        Object rtValue = null;
        try {
            Object[] args = pjp.getArgs();//得到方法执行所须的参数

            System.out.println("Logger类中的aroundPrintLog方法开始记录日志..前置");

            rtValue = pjp.proceed(args);//明确调用业务层方法（切入点方法）
```

```
            System.out.println("Logger类中的aroundPrintLog方法开始记录日志..后置");

            return rtValue;
        } catch (Throwable throwable) {

            System.out.println("Logger类中的aroundPrintLog方法开始记录日志..异常");
            throw new RuntimeException(throwable);
        } finally {
            System.out.println("Logger类中的aroundPrintLog方法开始记录日志..最终");
        }
    }
}
```

04_01jdbcTemplate

以前的学的JDBC原始写法简单插入数据库数据：要写加载数据库驱动、获取连接、执行语句对象、闭数据库连接。

现在使用spring内置的数据源和JdbcTemplate写法可简化为：

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

/**
 * JdbcTemplate的最基本用法
 */
public class JdbcTemplateDemo1 {
    public static void main(String[] args) {
        //准备数据源：spring的内置数据源
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/springtest");
        ds.setUsername("root");
        ds.setPassword("root");

        //1.创建对象JdbcTemplate
        JdbcTemplate jt = new JdbcTemplate();
        //给jt设置数据源
        jt.setDataSource(ds);
        //2.执行操作
        jt.execute("insert into account(name,money)values('ccc',1000)");
    }
}
```

以上耦合太多，且配置写死了。下面使用ioc容器配置：

```
<!-- 配置JdbcTemplate -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 配置数据源 -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSour
e">
```

```
    <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
    <property name="url" value="jdbc:mysql://localhost:3306/springtest"></property>
    <property name="username" value="root"></property>
    <property name="password" value="root"></property>
</bean>

//1.获取容器
ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");
//2.获取对象
JdbcTemplate jt = ac.getBean("jdbcTemplate", JdbcTemplate.class);
//3.执行操作
jt.execute("insert into account(name,money)values('ddd',222)");
```

JdbcTemplate的CRUD：

```
//保存
jt.update("insert into account(name,money)values(?,?)","eee",3333f);
//更新
jt.update("update account set name = ?,money = ? where id = ?","test",4567f,7);
//删除
jt.update("delete from account where id = ?",8);
//查询所有
List<Account> accounts = jt.query("select * from account where money > ?",new BeanProper
yRowMapper<Account>(Account.class),1000f);
//查询一个
List<Account> accounts = jt.query("select * from account where id = ?",new BeanPropertyRo
Mapper<Account>(Account.class),1);
System.out.println(accounts.isEmpty()?"没有内容":accounts.get(0));
//查询返回一行一列（使用聚合函数，但是不加group by）
Long count = jt.queryForObject("select count(*) from account where money > ?",Long.class,1
00f);
```

Spring之JdbcDaoSupport的使用：

JdbcDaoSupport是Spring内置的一个Dao层的基类，其内部定义了JdbcTemplate的set方法，这样们自己的dao类只需要继承JdbcDaoSupport类，就可以省略JdbcTemplate的set方法书写了，通过看源码你会发现，该方法是final修饰的。还提供了注入数据库连接池的set方法。

```
public final void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
    initTemplateConfig();
}
public final void setDataSource(DataSource dataSource) {
    if (this.jdbcTemplate == null || dataSource != this.jdbcTemplate.getDataSource()) {
        this.jdbcTemplate = createJdbcTemplate(dataSource);
        initTemplateConfig();
    }
}
```

说白了，JdbcDaoSupport就是为了简化我们dao类有关JdbcTemplate的注入的相关工作量。下是JdbcDaoSupport的使用：

```
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import blog.csdn.net.mchenys.domain.Account;
```

```
//dao层，继承JdbcDaoSupport
public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {
    @Override
    public void save(Account account) {
        // 从父类获取jdbcTemplate
        getJdbcTemplate().update("insert into t_account values(null,?,?)",
                account.getName(), account.getMoney());
    }
}
```

04_02accountAopTransactionXml

之前转账的问题基于XML配置文件的AOP实现事务控制

04_03accountAopTransactionAnnotation

之前转账的问题基于Annotation注解的AOP实现事务控制

不配置环绕通知的情况下报错：

com.mysql.jdbc.exceptions.jdbc4.MySQLNonTransientConnectionException: **Can't call rollback when autocommit=true**

是因为spring后置通知和最终通知的顺序与xml配置下实现AOP功能的顺序有些出入。正常返回的执顺序应该是：

环绕前置—>普通前置—>目标方法执行—>环绕后置—>环绕最终—>普通后置—>普通最终

环绕通知的顺序是正常的，而普通的后置通知和最终通知顺序不对。因此：

如果在省时省力且对顺序无要求的情况下使用注解，在要求顺序的情况下那就得使用xml配置或者使环绕通知，这是最正确的。

环绕通知：具体参考：03_04adviceType

```
package com.yoyling.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.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.sql.SQLException;

/**
 * 和事务管理相关的工具类，它包含了开启事务、提交事务、回滚事务和释放事务
 */
@Component("txManager")
@Aspect
public class TransactionManager {
```

```
@Autowired
private ConnectionUtils connectionUtils;

@Pointcut("execution(* com.yoyling.service.impl.*.*(..))")
private void pt1(){}

/**
 * 开启事务
 */
public void beginTransaction() {
    try {
        connectionUtils.getThreadConnection().setAutoCommit(false);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

/**
 * 提交事务
 */
public void commit() {
    try {
        connectionUtils.getThreadConnection().commit();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

/**
 * 回滚事务
 */
public void rollback() {
    try {
        connectionUtils.getThreadConnection().rollback();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

/**
 * 释放连接
 */
public void release() {
    try {
        connectionUtils.getThreadConnection().close();//还回连接池中
        connectionUtils.removeConnection();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

@Around("pt1()")
public Object aroundAdvice(ProceedingJoinPoint pjp) {
    Object rtValue =null;
```

```
        try {
            //1.获取参数
            Object[] args = pjp.getArgs();
            //2.开启事务
            this.beginTransaction();
            //3.执行方法
            rtValue =  pjp.proceed(args);
            //4.提交事务
            this.commit();
            //返回结果
            return rtValue;
        } catch (Throwable e) {
            //5.回滚事务
            this.rollback();
            throw new RuntimeException(e);
        } finally {
            //6.释放资源
            this.release();
        }
    }
}
```

04_04transaction

转账事务控制的代码准备。

现总结下三类事务管理的做法：

1. 传统的使用JDBC的事务管理，使用DataSource，从数据源中获取connection，通过con的api进行RUD，手动的进行commit或者rollback。上面案例04_0204_02accountAopTransactionXml、04_0accountAopTransactionAnnotation已经实现了。

2. 使用spring的声明式事务处理。下面04_05、04_06将要做的。

3. 应用spring提供的编程式的事务管理。

04_05transactionXml

基于XML的声明式事务控制

spring中基于XML的声明式事务控制配置步骤：

1. 配置事务管理器

2. 配置事务的通知

此时需要导入事务的约束 tx的名称空间和约束，同时也需要aop的

使用 **tx:advice** 标签配置事务的通知

属性：

id：给事务通知起一个唯一标识

transaction-manager：给事务通知提供一个事务管理器引用

3. 配置AOP中的通用切入点表达式

4. 建立事务通知和切入点表达式对应的关系

5. 配置事务的属性

是在事务的通知 **tx:advice** 标签的内部

配置事务的属性：

isolation：事务隔离级别。默认DEFAULT，表示使用数据库的默认隔离级别。

propagation：传播行为。默认REQUIRED，表示一定会有事务，增删改的选择。查询方法可以选择UPPORTS。

read-only：是否只读。只有查询方法才能设置为true。默认为false表示读写。

timeout：超时时间，默认-1表示永不超时。秒为单位。

rollback-for：用于指定一个异常，产生该异常事务回滚，产生其它异常不回滚。无默认，表示任何常都回滚。

no-rollback-for：用于指定一个异常，产生该异常事务不回滚，产生其它异常回滚。无默认，表示何异常都回滚。

```
<!-- 配置业务层 -->
<bean id="accountService" class="com.yoyling.service.impl.AccountServiceImpl">
  <property name="accountDao" ref="accountDao"></property>
</bean>

<!-- 配置账户的持久层 -->
<bean id="accountDao" class="com.yoyling.dao.impl.AccountDaoImpl">
  <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 配置数据源 -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSour
e">
  <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
  <property name="url" value="jdbc:mysql://localhost:3306/springtest"></property>
  <property name="username" value="root"></property>
  <property name="password" value="root"></property>
</bean>

<!-- 配置事务管理器 -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTra
sactionManager">
  <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 配置事务的通知 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <!-- 配置事务的属性 -->
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED" read-only="false"></tx:method>
    <tx:method name="find*" propagation="SUPPORTS" read-only="true"></tx:method>
```

```
    </tx:attributes>
</tx:advice>

<!-- 配置aop -->
<aop:config>
    <!-- 配置切入点表达式 -->
    <aop:pointcut id="pt1" expression="execution(* com.yoyling.service.impl.*.*(..))"></aop:po
ntcut>
    <!-- 建立切入点表达式和事务通知的关系 -->
    <aop:advisor advice-ref="txAdvice" pointcut-ref="pt1"></aop:advisor>
</aop:config>
```

声明式事务的好处在于可以一次配置，后面再没有什么事务的问题干扰。

04_06transactionAnnotation

基于注解的声明式事务控制

spring中基于注解的声明式事务控制配置步骤

1. 配置事务管理器
2. 开启spring对注解事务的支持
3. 在需要事务支持的地方使用@Transactional注解

```
<!-- 配置事务管理器 -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTra
sactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 开启spring对注解事务的支持 -->
<tx:annotation-driven transaction-manager="transactionManager"></tx:annotation-driven>

@Service("accountService")
@Transactional(propagation = Propagation.SUPPORTS,readOnly = true)//只读型事务的配置
public class AccountServiceImpl implements AccountService {
    @Transactional(propagation = Propagation.REQUIRED,readOnly = false)
    @Override
    //需要的是读写型事务配置
    public void transfer(String sourceName, String targetName, Float money) {}
}
```

day_04_07annotation_tx_withoutxml

基于纯注解的声明式事务控制

测试类：AccountServiceTest.java

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SpringConfiguration.class)
public class AccountServiceTest {
```

```
    @Autowired
    private AccountService as;

    @Test
    public void testTransfer() {
        as.transfer("aaa","bbb",100f);
    }
}
```

spring配置类：SpringConfiguration.java

```
@Configuration
@ComponentScan("com.yoyling")
@Import({JdbcConfig.class,TransactionConfig.class})
@PropertySource("jdbcConfig.properties")
@EnableTransactionManagement
public class SpringConfiguration {
}
```

和连接数据库相关的配置类：JdbcConfig.java

```
public class JdbcConfig {

    @Value("${jdbc.driver}")
    private String driver;

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    /**
     * 创建JdbcTemplate对象
     * @param dataSource
     * @return
     */
    @Bean(name = "jdbcTemplate")
    public JdbcTemplate createJdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    /**
     * 创建数据源对象
     * @return
     */
    @Bean(name = "dataSource")
    public DataSource createDateSource() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(driver);
        ds.setUrl(url);
```

```
        ds.setUsername(username);
        ds.setPassword(password);
        return ds;
    }
}
```

和事务相关的配置类：TransactionConfig.java

```
public class TransactionConfig {
    /**
     * 用于创建事务管理器对象
     * @param dataSource
     * @return
     */
    @Bean(name = "transactionManager")
    public PlatformTransactionManager createTransactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

jdbcConfig.properties

```
jdbc.driver = com.mysql.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/springtest
jdbc.username = root
jdbc.password = root
```

service实现类

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

04_08accountTransaction

编程式事务控制（少用）

通过前面的示例可以发现，事务管理方式很容易理解，代码中显式调用**beginTransaction()**、**commi()**、**rollback()**。或者通过 Spring 提供的事务管理 API，将事务操作委托给底层的持久化框架来执行但是事务管理的代码散落在业务逻辑代码中，破坏了原有代码的条理性，并且每一个业务方法都包含类似的启动事务、提交/回滚事务的样板代码。因此spring又提供了编程式事务管理。

TransactionTemplate

TransactionTemplate 的 execute() 方法有一个 **TransactionCallback** 类型的参数，该接口中定义一个 **doInTransaction()** 方法，通常我们以**匿名内部类**的方式实现 TransactionCallback 接口，并其 doInTransaction() 方法中书写业务逻辑代码。这里可以使用默认的事务提交和回滚规则，这样在务代码中就不需要显式调用任何事务管理的 API。

```
public Account findAccountById(Integer accountId) {
    return transactionTemplate.execute(new TransactionCallback<Account>() {
        public Account doInTransaction(TransactionStatus status) {
            return accountDao.findAccountById(accountId);
        }
    });
}
```

也能看出虽然spring提供的 TransactionTemplate 简化了事务控制，但还是使得业务层代码重复。此，一般开发事务控制使用声明式的比较多。