spring

spring表达式

- #{123}、#{'jack'}：数字、字符串
- #{beanId}：另一个bean被引用
- #{beanId.propName}:操作数据
- #{beanId.toString()}:执行方法
- #{T(类).字段|方法}:静态方法或字段

案例

```
 <bean id="address" class="com.gyf.model.Address">
    <property name="name" value="天河"></property>
</bean>

<bean id="customer" class="com.gyf.model.Customer">
    <property name="name" value="#{'gyf'.toUpperCase()}"></property>
    <!-- Math.PI 调用静态方法-->
    <property name="pi" value="#{T(java.lang.Math).PI}"></property>

    <!--
    一个对象引用另外一个对象两写法
    1.ref: 引用<property name="address" ref="address"></property>
    2.SpEL:<property name="address" value="#{address}"></property>
    -->
    <property name="address" value="#{address}"></property>
</bean>
```

总结：spring表达式的作用就是在xml的属性注入和配置中，就把一些比如大写，比如引用等小的函数写入进去作用，而不是只能单纯的注入属性，还可以对属性进行修改，或者引用其他bean的属性等。

在xml中集合的注入

包括LIST<String>，Set<String>,Map<String,String>,properties,String[]等形式，具体的注入格式如下所示

```
<property name="cars">
            <!--1. list数据注入-->
            <list>
                <value>ofo</value>
                <value>mobai</value>
                <value>宝马</value>
            </list>
```

```
        </property>

        <!-- set数据注入-->
        <property name="pats">
            <set>
                <value>小黑</value>
                <value>小黄</value>
                <value>小白</value>
            </set>
        </property>


        <!-- map数据注入-->
        <property name="infos">
            <map>
                <entry key="name" value="gyf"></entry>
                <entry key="age" value="108"></entry>
                <entry key="ip" value="127.0.0.1"></entry>
            </map>
        </property>

        <!--Properties 数据注入-->
        <property name="mysqlInfos">
            <props>
                <prop key="url">mysql:jdbc://localhost:3306/dbname</prop>
                <prop key="user">root</prop>
                <prop key="password">123456</prop>
            </props>
        </property>

        <!-- 数组注入-->
        <property name="members">
            <array>
                <value>哥哥</value>
                <value>弟弟</value>
                <value>妹妹</value>
                <value>姐姐</value>
            </array>
        </property>
```

spring的注解配置

主要包括Component,Autowired，Repository，Service，Controller，Qualifier，Resource

- @Component 主要是实现对bean的注解，把普通的pojo实例化到spring容器当中，相当于配置文件中的<bean id="" class=""/>
- @Respository 主要实现dao层的访问
- @Service 实现service层的访问
- @Controller 实现web（control）层的访问
- @Qualifier 前者都是通过类来进行映射和访问，该注解可以对相应的类设置id，通过具体的id来进行访问

- @Resource 主要是引入类

AOP编程

AOP-面向切面编程

- 是函数式编程的一种衍生范型
- 采取横向的抽取机制，取代了纵向继承的重复性代码
- 纵向继承概念：缺点依赖关系，耦合度高，代码重复性强

```
UserService extends BaseService(){
    add(){
        //log() 日志记录
        //dao方法
    }
    delete(){
        //log() 日志记录
        //dao方法
    }
}
StudentService extends BaseService(){
    add(){
        //log() 日志记录
        //dao方法
    }
    delete(){
        //log() 日志记录
        //dao方法
    }
}
BaseService
    public void log(){
    //日志记录，谁操作了代码
    //具体的操作代码
    }
```

- 横向继承概念：就是利用代理的方式，在类运行指定函数时，拦截方法并插入我们需要的类方法（比如log类等），所以相当于从一个横面切过去的，叫切面，也叫横向继承了。
- 总结来说，spring在运行期间通过代理方法向目标类植入增强代码。
- 可以对业务逻辑各部分进行分离，使得其耦合性降低，提高程序的可重用性
- AspectJ，基于java的AOP框架

AOP实现原理

- aop底层使用代理机制进行
- 接口+实现类：spring采用jdk的动态代理proxy
- 实现类： spring采用cglibzi字节码增强

AOP术语

- target：目标类，也就是需要被代理的类
- JointPoint：所有方法（可能被拦截注入增强方法的方法
- PointCut：已经确认被增强的连接点
- advice：增强代码
- Weaving：指把增强代码advice应用到目标对象target来创建新的代理对象proxy的处理过程
- proxy：新的代理类
- Aspect：切入点pointicut和advice的结合

jdk自己实现AOP代理

需要四个主要的类来实现AOP代理的功能

- 目标类（需要被注入方法的目标，主要是service接口和实现两个

```
public interface UserService{
    public void addUser();
}
public class UserServiceImpl implements UserService{
    @Override
    public void addUser(){
        System.out.println("添加用户");
    }
}
```

- 切面类(需要注入的方法)

```
public class MyAspect{
    public void before(){
       System.out.println("开启事务");
    }
    public voidafter(){
       System.out.println("提交事务");
    }
}
```

- 工厂类(把目标类方法进行拦截，生成代理类，并在代理类中运行相关的切面方法，并返回代理类,生成proxy代理类

```
public class MyBeanFactory{
   public static UserService creatUserService(){
       //目标类
       final UserService userService=new UserServiceImpl();
       //切面类
       final MyAspect myAspect=new MyAspect();
       //代理类
       <!-- newProcyInstance(Classloader loader,Class<?>[],InvocationHandler h)
```

```
        参数1：当前类的类加载器
        参数2：代理类所需要实现的接口，假如是一个类，就是该类下所有方法
        参数3：处理类，一般写匿名类--!>
        UserService proxyService=(UserService)Proxy.newProxyInstance(
        MyBeanFactory.class.getClassLoader(),
        userService.getClass().getInterfaces(),
        new InvocationHandler(){
            @Override
            public Object invoke(Object proxy,Method method,Objext[] args)
                throws Throwable{
                    aspect.before();
                    Object obj=method.invoke(userService,args)
                    aspect.afger();
                    return obj;
                }
        }
        );
        return proxyService;
    }
}
```

- 测试类

```
    @Test
    public void test2(){
        UserService userService=MyBeanFactory.creatUserService();
        uerService.addUser();
    }
```

cglib实现AOP编程

- cglib是通过将我们的目标类创建子类，然后每次执行的时候都会进行拦截，然后将子类方法注入运行并返回。 原理也是通通过实现代理类来进行拦截，加入增强方法后将原方法释放。但它创建代理类的方法和jdk不同，jdk生成的代理类只有一个，因为其被代理的目标是动态传入。而cglib是为每个目标类生成相应的子类，编译较慢，但在运行中因为已经静态编译生成，所以执行效率快。
- 还有在运行方法时，jdk的方法直接就是代理类属性了，但cglib的话还是原来的方法类
- jdk是针对有接口和实现类的，cglib是都可以

Spring编写代理半自动

- 导入jar包
  在原有的spring5个核心包外，还要导入AOP联盟和AOP实现两个jar包
- 代码实现 目标类

```
public interface IUserService {
    //切面编程
    public void addUser();
    public void updateUser();
```

```
    public void deleteUser();
    public int deleteUser(int id);
}
public class UserServiceImpl implements IUserService {
    @Override
    public void addUser() {
        System.out.println("添加用户。。。。");
    }

    @Override
    public void updateUser() {
        System.out.println("更新用户。。。。");
    }

    @Override
    public void deleteUser() {
        System.out.println("删除用户。。。。");
    }

    @Override
    public int deleteUser(int id) {
        System.out.println("通过"+id+"删除用户");
        return 1;
    }
}
```

- 切面类

```
/**
* 切面类：增加代码 与 切入点 结合
*/
public class MyAspect implements MethodInterceptor {

   @Override
   public Object invoke(MethodInvocation methodInvocation) throws Throwable {
       //拦截方法
       System.out.println("开启事务");
       //放行
       Object obj =methodInvocation.proceed();
       System.out.println("拦截");
       return obj;
   }
}
```

- xml配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p ="http://www.springframework.org/schema/p"
       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">


    <!--  配置UserService-->
    <bean id="userService" class="com.gyf.service.UserServiceImpl"></bean>

    <!--  配置切面类对象-->
    <bean id="myAspect" class="com.gyf.aspect.MyAspect"></bean>

    <!-- 配置代理对象
        默认情况下Spring的AOP生成的代理是JDK的Proxy实现的，如果没有借口，采用cglib字节码增强进
行方法注入，如果生命optimize=true，无论是否有接口，都采用cglib
    -->
    <bean id="serviceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">

        <!-- 接口 :如果只是一个接口，就写Value,如果是多个接口就写List-->
        <property name="interfaces"  value="com.gyf.service.IUserService">
        </property>

        <!-- 目标对象 -->
        <property name="target" ref="userService"/>

        <!-- 切面类-->
         <property name="interceptorNames" value="myAspect"></property>

        <!-- 配置使用cglib生成-->
        <property name="optimize" value="true"></property>

    </bean>
</beans>
```

- 测试类

```
public class Lesson07 {
    @Test
    public void test1() throws Exception {

        //获取Spring容器中代理对象
        ApplicationContext context = new ClassPathXmlApplicationContext("beans07.xml");

        IUserService userService = (IUserService) context.getBean("serviceProxy");

        userService.deleteUser(5);

    }
}
```

*** SpringAOP全自动编程

- 利用spring的的jar包xml配置就少了代理类的设置，然后在xml中的配置更加简单。
- 目标类

```
public interface IUserService {
    //切面编程
    public void addUser();
    public void updateUser();
    public void deleteUser();
    public int deleteUser(int id);
}
public class UserServiceImpl implements IUserService {
    @Override
    public void addUser() {
        System.out.println("添加用户。。。。");
    }

    @Override
    public void updateUser() {
        System.out.println("更新用户。。。。");
    }

    @Override
    public void deleteUser() {
        System.out.println("删除用户。。。。");
    }

    @Override
    public int deleteUser(int id) {
        System.out.println("通过"+id+"删除用户");
        return 1;
    }
}
public class StudentService {
    public void delete(){
        System.out.println("删除学生");

    }
    public void add(){
        System.out.println("add学生");
    }
    public void update(){
        System.out.println("update学生");
    }
}
```

- 切面类

```
public class MyAspect implements MethodInterceptor {

    @Override
    public Object invoke(MethodInvocation methodInvocation) throws Throwable {
        //拦截方法
        System.out.println("开启事务");
        //放行
        Object obj =methodInvocation.proceed();
        System.out.println("拦截");
        return obj;
```

```
    }
}
```

- xml配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p ="http://www.springframework.org/schema/p"
       xmlns:context ="http://www.springframework.org/schema/context"
       xmlns:aop ="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-
context.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!--  配置UserService-->
    <bean id="userService" class="com.gyf.service.UserServiceImpl"></bean>
    <bean id="studentService" class="com.gyf.service.StudentService"></bean>

    <!--  配置切面类对象-->
    <bean id="myAspect" class="com.gyf.aspect.MyAspect"></bean>

    <!--全自动的AOP配置
        1.在bean中配置aop约束
        2.配置aop:config内容，把切入点和通知结合
        3.True意味用cglib实现代理 -->
    <aop:config proxy-target-class="true">
        <!--切入点
            expression:表达式
            每个service的方法前后都要开启和结束事务
            proxy-target-class:使用cglib实现代理
     expression 表达式：*任意
                execution(*         com.gyf.service.*.    *       (..))
                          返回值     包名             类名 方法名   参数

              AOP:用于事务配置&日志记录  -->
        <aop:pointcut id="myPointcut" expression="execution(* com.gyf.service.*.*(..))"/>
        <!--通知  关联切入点-->
        <aop:advisor advice-ref="myAspect" pointcut-ref="myPointcut"></aop:advisor>
    </aop:config>

</beans>
```

- 测试类

```
public class Lesson08 {

    @Test
    public void test1() throws Exception {
```

```
        //获取Spring容器中代理对象
        ApplicationContext context = new ClassPathXmlApplicationContext("beans08.xml");

        IUserService userService = (IUserService) context.getBean("userService");

        StudentService studentService = (StudentService)
context.getBean("studentService");

        userService.deleteUser(5);

        studentService.add();
    }
}
```

AspectJ

AspectJ是一个基于java的面向切面编程的框架，在spring2.0之后新增了对AspectJ切点表达式的支持，主要用于自定义开发
aspectj有before，after，around等多个周期的方法，具体可以百度了解。

AspectJ基于XML切面编程的实现

- 目标类

```
public interface IUserService {
    //切面编程
    public void addUser();
    public void updateUser();
    public void deleteUser();
    public int deleteUser(int id);
}
public class UserServiceImpl implements IUserService {
    @Override
    public void addUser() {
        System.out.println("添加用户。。。。");
    }

    @Override
    public void updateUser() {
        System.out.println("更新用户。。。。");
    }

    @Override
    public void deleteUser() {
        System.out.println("删除用户。。。。");
    }

    @Override
    public int deleteUser(int id) {
        System.out.println("通过"+id+"删除用户");
        return 1;
    }
}
```

- 切面类

```
public class MyAspect3 {
    //JoinPoint 没有放行的概念
    public  void myBefore(JoinPoint jp){
        System.out.println("前置通知"+jo.getSignature.getName());
    }

    public  void after_returning(){
        System.out.println("后置通知");
    }
    //ProcessJointPoint 才有锁住和放行的概念
    public  Object myAround(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("环绕通知");
        System.out.println(pjp.getSignature().getName());//打印的就是执行的方法名
        System.out.println("开启事务");
        //放行
        Object obj=pjp.proceed();
        System.out.println("提交事务");
        return  obj;
    }
}
```

- xml配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-
context.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd ">


    <!--  配置UserService-->
    <bean id="userService" class="com.gyf.service.UserServiceImpl"></bean>

    <!-- 配置切面对象-->
    <bean id="myAspect3" class="com.gyf.aspect.MyAspect3"></bean>

    <!-- 配置 aop -->
    <aop:config>
        <!-- aop:指定切面-->
        <aop:aspect ref="myAspect3">
            <!--定义一个切入点-->
            <aop:pointcut id="myPointcut" expression="execution(*
com.gyf.service.UserServiceImpl.*(..))"/>
```

```
            <!-- 配置前置通知...-->
            <!--<aop:before method="myBefore" pointcut-ref="myPointcut" />

            <aop:after-returning method="after_returning" pointcut-ref="myPointcut"/>-->

            <!--配置环绕通知-->
            <aop:around method="myAround" pointcut-ref="myPointcut"></aop:around>

            <!-- 配置异常通知
            throwing="e" 值，是方法的参数名
            -->
            <aop:after-throwing method="myAfterThrowing" pointcut-ref="myPointcut"
throwing="e"/>

            <!--配置最终通知:不管有没有异常，最终通知都会执行-->
            <aop:after method="myAfter" pointcut-ref="myPointcut"></aop:after>

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

- 测试

```
public class Lesson10 {

    @Test
    public void test1() throws Exception {

        //获取Spring容器中代理对象
        ApplicationContext context = new ClassPathXmlApplicationContext("beans10.xml");

        IUserService userService = (IUserService) context.getBean("userService");

        userService.deleteUser();



    }
}
```

AspectJ基于注解来进行AOP编程

- 基于注解的AspectJ主要是将在xml配置中的aop配置，通过注解来实现，主要表现在xml文件和切面层的编写有不同。
- 一些逻辑的顺序可能会有不同，需要注意
- xml编写（主要是配置注解位置，和使其生效

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p ="http://www.springframework.org/schema/p"
       xmlns:context ="http://www.springframework.org/schema/context"
```

```
        xmlns:aop ="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                            http://www.springframework.org/schema/beans/spring-beans.xsd
                            http://www.springframework.org/schema/context
                            http://www.springframework.org/schema/context/spring-
context.xsd
                            http://www.springframework.org/schema/aop
                            http://www.springframework.org/schema/aop/spring-aop.xsd">


    <!-- 配置扫描注解的位置 -->
    <context:component-scan base-package="com.gyf"/>

    <!-- 配置aop注解生效-->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>

    <!--aop配置-->
    <aop:config>
        <aop:aspect ref="myAspect"></aop:aspect>
    </aop:config>
</beans>
```

- 切面层程序的编写

```
@Component
@Aspect
public class MyAspect {
    //声明一个公共的切入点
    @Pointcut("execution(* com.gyf.service.UserServiceImpl.*(..))")
    public void myPointcut(){}

    @Before("myPointcut()")
    public void myBefore(JoinPoint jp){
        System.out.println("1.前置通知..." + jp.getSignature().getName());//连接点方法名
    }

    /**
     * 后置通知获取service方法执行后的返回值
     * Object retValue:service方法执行的返回值,如果写了返回值，需要在xml中配置returning
     * @param jp
     */
    // <aop:after-returning method="myAfterReturning" pointcut-ref="myPointcut"
returning="retValue"/>
    @AfterReturning(pointcut = "myPointcut()",returning = "retValue")
    public void myAfterReturning(JoinPoint jp,Object retValue){
        System.out.println("3.后置通知..." +  jp.getSignature().getName());
        System.out.println("返回值:" + retValue);
    }


    @Around("myPointcut()")
    public Object myAround(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("2.环绕通知...开启事务..." + pjp.getSignature().getName());
        //放行
        Object retObj = pjp.proceed();
```

```
        System.out.println("4.环绕通知....提交事务...");
        return retObj;
    }


    @AfterThrowing(pointcut = "myPointcut()",throwing = "e")
    //Joinpoint无法放行
    public void myAfterThrowing(JoinPoint jp,Throwable e){
        System.out.println("异常通知..." + jp.getSignature().getName() + "===" +
e.getMessage() );
    }

    @After("myPointcut()")
    public void myAfter(JoinPoint jp){
        System.out.println("最终通知..." + jp.getSignature().getName());
    }
}
```

使用jdbctemplate来进行连接

- jdbcTemplate类似于dbutils，是用来操作jdbc的工具类，他需要依赖连接池DataSource来进行管理，而连接池主要有dbcp和成c3p0两种。
- 导入对应的jar包
  jar包主要包括c3p0，dbcp连接池，然后是mysql驱动已经在spring中的jdbc和事务
- 创建对应的数据库和表格
- 编写实体类
- 使用API，进行数据库连接和操作

```
public class Lesson04 {

    @Test
    public void test1() throws Exception {

       //1.创建数据源dbcp连接池
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql:///spring_day04");
        dataSource.setUsername("root");
        dataSource.setPassword("123456");

        //2.创建jdbcTemplate
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        //jdbcTemplate.update("update t_user set username = ?,password =? where id =
?","gan","13456",1);
        RowMapper<User> rowMapper=new BeanPropertyRowMapper<User>(User.class);
        User user=jdbcTemplate.queryForObject("select * from t_user where id=?",rowMapper
,1);
        System.out.println(user.toSting());

    }
}
```

- java里调用sql语句的写法
  https://www.cnblogs.com/caoyc/p/5630622.html

使用dbpc和c3p0来配置数据源，调用dao层方法

- 导入jar包
- 编写dao层以及其实现层

```
public interface IUserDao {
    public void add(User user);
}

public class UserDaoImpl implements IUserDao{
    private  JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override
    public void add(User user) {

        System.out.println("dao 添加用户:" + user);

        jdbcTemplate.update("insert t_user (username,password) values
(?,?)",user.getUsername(),user.getPassword());
    }
}
```

- 配置xml，引用dao层及配置数据源,主要是在利用dbcp数据池和c3p0数据池的时候，bean的属性存在一些区别需要注意。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p ="http://www.springframework.org/schema/p"
       xmlns:context ="http://www.springframework.org/schema/context"
       xmlns:aop ="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-
context.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- 配置DBCP dataSource 对象-->
  <!--  BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql:///spring_day04");
    dataSource.setUsername("root");
    dataSource.setPassword("123456");-->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
```

```
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql:///spring_day04"/>
        <property name="username" value="root"/>
        <property name="password" value="123456"/>
    </bean>

    <!-- 配置c3p0数据源
    注：dbcp和c3po的 数据库连接的参数的属性名是不一样
    please attention。。。。。
    -->
    <!--<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="com.mysql.jdbc.Driver"/>
        <property name="jdbcUrl" value="jdbc:mysql:///spring_day04"/>
        <property name="user" value="root"/>
        <property name="password" value="123456"/>
    </bean>-->

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

    <!-- 配置dao-->
    <bean id="userDao" class="com.gyf.dao.UserDaoImpl">
       <property name="jdbcTemplate" ref="jdbcTemplate"></property>
    </bean>
</beans>
```

- 基于xml配置bean时的property的依赖注入，都需要在对应的类里有相应的set方法
  https://blog.csdn.net/qq_39411607/article/details/79631484

jdbcdaosupport的运用

在对dao层的实现撰写时，我们是利用依赖的资源获取jdbctemplate的set方法，然后在xml的bean配置中进行调用，然后再在xml中配置jdbctemplate和datasource，如果我们将daoimpl继承JDbcsupport类，因为父类中就用对应的jdbctemplate的set方法，所我们可以在daoimpl中直接调用，getjdbctemplate的方法，以及在xml中，关于dao的配置也就可以直接指向datasource。
同时，可以通过properties文件，将数据池的相关信息配置，写在单独的文件中，xml的配置直接用spring的写法进行引用。

- dao实现

```
@Repository
public class UserDaoImpl2 extends JdbcDaoSupport implements IUserDao{

    @Override
    public void add(User user) {
        System.out.println("dao 添加用户:" + user);

        getJdbcTemplate().update("insert .t_user (username,password) values
(?,?)",user.getUsername(),user.getPassword());
    }
}
```

- xml配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p ="http://www.springframework.org/schema/p"
       xmlns:context ="http://www.springframework.org/schema/context"
       xmlns:aop ="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-
context.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- 配置DBCP dataSource 对象-->
  <!--  BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql:///spring_day04");
    dataSource.setUsername("root");
    dataSource.setPassword("123456");-->
    <!--<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql:///spring_day04"/>
        <property name="username" value="root"/>
        <property name="password" value="123456"/>
    </bean>-->

    <!-- 读取db.properties数据-->
    <context:property-placeholder location="classpath:db.properties"></context:property-
placeholder>
    <!-- 配置c3p0数据源
    注：dbcp和c3po的 数据库连接的参数的属性名是不一样
    please attention。。。。。
    -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${driverClass}"/>
        <property name="jdbcUrl" value="${jdbcUrl}"/>
        <property name="user" value="${user}"/>
        <property name="password" value="${password}"/>
    </bean>

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

    <!-- 配置dao-->
    <bean id="userDao" class="com.gyf.dao.UserDaoImpl2">
        <!--<property name="jdbcTemplate" ref="jdbcTemplate"></property>-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>
</beans>
```

- jdbcDaoSupport的源码

```
public abstract class JdbcDaoSupport extends DaoSupport {
    private JdbcTemplate jdbcTemplate;

    public JdbcDaoSupport() {
    }

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

    }

    protected JdbcTemplate createJdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    public final DataSource getDataSource() {
        return this.jdbcTemplate != null ? this.jdbcTemplate.getDataSource() : null;
    }

    public final void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
        this.initTemplateConfig();
    }

    public final JdbcTemplate getJdbcTemplate() {
        return this.jdbcTemplate;
    }
    .......
```

在spring中配置事务，并用转账来进行测试

事务配置时重要的3个概念

- 事务管理器 事务管理器，是spring用来管理事务的接口
- 事务详情 事务详情主要是定义了事务管理的相关属性，如隔离级别，是否只读等等
- 事务状态 事务状态用于记录当前事务的运行状态，比如是否有保存点，事务是否完成等

这里主要介绍利用xml工厂bean自动生成代理的过程

- xml配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p ="http://www.springframework.org/schema/p"
       xmlns:context ="http://www.springframework.org/schema/context"
```

```
        xmlns:aop ="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                            http://www.springframework.org/schema/beans/spring-beans.xsd
                            http://www.springframework.org/schema/context
                            http://www.springframework.org/schema/context/spring-
context.xsd
                            http://www.springframework.org/schema/aop
                            http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- 读取db.properties数据-->
    <context:property-placeholder location="classpath:db.properties"></context:property-
placeholder>

    <!-- 配置c3p0数据源
    注：dbcp和c3po的 数据库连接的参数的属性名是不一样
    please attention。。。。。
    -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${driverClass}"/>
        <property name="jdbcUrl" value="${jdbcUrl}"/>
        <property name="user" value="${user}"/>
        <property name="password" value="${password}"/>
    </bean>


    <!-- 配置dao-->
    <bean id="accountDao" class="com.gyf.dao.impl.AccountDaoImpl">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

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



    <!--配置service-->
    <bean id="accountService" class="com.gyf.service.impl.AccountServiceImpl2">
        <property name="accountDao" ref="accountDao"></property>
    </bean>

    <!-- 配置工厂代理-->
    <bean id="proxyService"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <!--接口-->
        <property name="proxyInterfaces" value="com.gyf.service.IAccountService">
</property>
        <!--目标对象-->
        <property name="target" ref="accountService"></property>
        <!--切面对象:Spring做，就不用写-->

        <!-- 事务管理器-->
        <property name="transactionManager" ref="txManager"/>
```

```
        <!--transactionAttributes:事务属性/详情配置
            key:写方法名
            value写 事务配置
            格式：PROPAGATION,ISOLATION,readOnly,-Exception,+Exception,即在某种错误下也进行提交任务，不会发生回滚
              传播行为         隔离级别            是否只读    异常回滚 异常提交
        -->
        <property name="transactionAttributes">
            <props>
                <prop key="transfer">PROPAGATION_REQUIRED,ISOLATION_DEFAULT,+java.lang.ArithmeticException</prop>
                <prop key="add">PROPAGATION_REQUIRED,ISOLATION_DEFAULT</prop>
                <prop key="delete">PROPAGATION_REQUIRED,ISOLATION_DEFAULT</prop>
                <prop key="update">PROPAGATION_REQUIRED,ISOLATION_DEFAULT</prop>
                <prop key="find">PROPAGATION_REQUIRED,ISOLATION_DEFAULT,readOnly</prop>
            </props>
        </property>
    </bean>
</beans>
```

- dao层和service层的相关代码

```
public interface IAccountDao {
    //扣钱
    public void out(String outer,Integer money);
    //进帐
    public void in(String inner,Integer money);
}

public class AccountDaoImpl extends JdbcDaoSupport implements IAccountDao{
    @Override
    public void out(String outer, Integer money) {
        getJdbcTemplate().update("update account set money = money - ? where username = ?",money,outer);
    }
    @Override
    public void in(String inner, Integer money) {
        getJdbcTemplate().update("update account set money = money + ? where username = ?",money,inner);
    }
}

public interface IAccountService {

    /**
     * 转账
     * @param outer 转出帐号
     * @param inner 转入帐号
     * @param money 转入金额
     */
    public void transfer(String outer,String inner,Integer money);
}

public class AccountServiceImpl2 implements IAccountService {
```

```
    private IAccountDao accountDao;//由spring注入
    public void setAccountDao(IAccountDao accountDao) {
        this.accountDao = accountDao;
    }

    @Override
    public void transfer(String outer, String inner, Integer money) {
        //扣钱
        accountDao.out(outer,money);
        //int i = 10 / 0;
        //进帐
        accountDao.in(inner,money);
    }
}
```

- 测试类

```
public class Lesson10 {

    @Test
    public void test1(){

        //转帐测试
        //获取Service
        ApplicationContext context = new ClassPathXmlApplicationContext("beans10.xml");

        //获取service代理对象
        IAccountService accountService = (IAccountService)
context.getBean("proxyService");
        accountService.transfer("jack","rose",100);
    }
}
```

基于SpringAop来进行事务的管理

- 这里增强的方法就是我们的事务管理，然后我们用spring自己的aop管理来进行操作，主要区别是在xml中bean的配置上，我们就是直接配置aop而不是代理工厂了，以及在test中的引用，不是获取代理工厂的bean类，二是直接获取service方法的bean类。
- xml配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-
```

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

    <!-- 读取db.properties数据-->
    <context:property-placeholder location="classpath:db.properties"></context:property-
placeholder>

    <!-- 配置c3p0数据源
    注：dbcp和c3po的 数据库连接的参数的属性名是不一样
    please attention。。。。。
    -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${driverClass}"/>
        <property name="jdbcUrl" value="${jdbcUrl}"/>
        <property name="user" value="${user}"/>
        <property name="password" value="${password}"/>
    </bean>


    <!-- 配置dao-->
    <bean id="accountDao" class="com.gyf.dao.impl.AccountDaoImpl">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

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

    <!--配置service-->
    <bean id="accountService" class="com.gyf.service.impl.AccountServiceImpl2">
        <property name="accountDao" ref="accountDao"></property>
    </bean>

    <!-- 使用spring的aop去配置-->
    <!--配置通知事务的管理器，就是加强的方法是一个事务，我们配这个事务的管理器-->
    <tx:advice id="txAdvice" transaction-manager="txManager">
        <!--事务详情-->
        <!--传播行为，隔离级别 ,可以不配置默认，但对应方法一定要配置-->
        <tx:attributes>
            <tx:method name="transfer" propagation="REQUIRED" isolation="DEFAULT"/>
        </tx:attributes>
    </tx:advice>

    <aop:config>
        <aop:pointcut id="myPointCut" expression="execution(*com.gyf.service...*.*(..))">
</aop:pointcut>
        <!--主要配置的就是增强的事务和切入点 以及关联-->
        <aop:advisor advice-ref="txAdvice" pointcut-ref="myPointCut"></aop:advisor>
    </aop:config>

</beans>
```

- 测试类的调用

```
public class Lesson11 {

    @Test
    public void test1(){

        //转帐测试
        //获取Service
        ApplicationContext context = new ClassPathXmlApplicationContext("beans11.xml");

        //获取service代理对象
        IAccountService accountService = (IAccountService)
context.getBean("accountService");
        accountService.transfer("jack","rose",100);
    }
}
```

注解配置事务

- 主要是将xml中关于事务的配置编程注解配置的开启，然后在需要加强的方法上加上对应的注解
- xml配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-
context.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd">

    <!-- 读取db.properties数据-->
    <context:property-placeholder location="classpath:db.properties"></context:property-
placeholder>

    <!-- 配置c3p0数据源
    注：dbcp和c3po的 数据库连接的参数的属性名是不一样
    please attention。。。。。
    -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${driverClass}"/>
        <property name="jdbcUrl" value="${jdbcUrl}"/>
```

```
        <property name="user" value="${user}"/>
        <property name="password" value="${password}"/>
    </bean>


    <!-- 配置dao-->
    <bean id="accountDao" class="com.gyf.dao.impl.AccountDaoImpl">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!--配置service-->
    <bean id="accountService" class="com.gyf.service.impl.AccountServiceImpl2">
        <property name="accountDao" ref="accountDao"></property>
    </bean>

    <!-- 配置事务管理器-->
    <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <!--配置dataSource-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>
    <!--开启事务注解驱动-->
    <tx:annotation-driven transaction-manager="txManager"></tx:annotation-driven>

</beans>
```

- service方法加注解,通过@Transactional来进行注解，其中传播和隔离级别默认可以不写

```
//@Transactional(propagation = Propagation.REQUIRED,isolation = Isolation.DEFAULT)
@Transactional
public class AccountServiceImpl2 implements IAccountService {

    private IAccountDao accountDao;//由spring注入

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



    @Override
    public void transfer(String outer, String inner, Integer money) {
        //扣钱
        accountDao.out(outer,money);
        //int i = 10 / 0;
        //进帐
        accountDao.in(inner,money);
    }
}
```

ssh整合

将struts2，spring和hibernate三者进行整理，然后将strust和hibernate的任务都交由spring来统一管理