Spring框架学习

第一章:创建对象与依赖注入

1. Spring中基于XML的IOC环境搭建

```
1. 创建一个Spring的配置文件,名为bean.xml
2. 在配置文件中引入最基本的约束


        <?xml version="1.0" encoding="UTF-8"?>
        <beans xmlns="http://www.springframework.org/schema/beans"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">

            <bean id = "ServiceDaoImpl" class="com.dao.ServiceDaoImpl"></bean>

            <bean id="ServiceImpl" class="com.service.ServiceImpl"></bean>

        </beans>

3.  在配置文件中使用bean标签来管理对象
4. 代码使用Spring容器示例

          ApplicationContext applicationContext =  new ClassPathXmlApplicationContext("bean.xml");
          IServiceDao iServiceDao = (IServiceDao) applicationContext.getBean("ServiceDaoImpl");
          IService service = (IService)applicationContext.getBean("ServiceImpl") ;
          System.out.println(iServiceDao);
          System.out.println(service);
```

2. 在spring配置文件中创建对象的三种方式

1. 使用默认构造函数创建对象

   - 待创建的java类

     ```
     public class ServiceImpl implements IService {
         public void save() {
             System.out.println("方法被执行了");
         }
     }
     ```

   - 在spring配置文件中的内容,使用默认构造函数创建出ServiceImpl对象

     ```
     <bean id="ServiceImpl" class="service.ServiceImpl" scope="singleton"></bean>
     ```

2. 使用某个类中的方法创建对象

   - 创建对象的类

     public class methodFactory {

     ```
     public IService getService(){
         return new ServiceImpl();
     }
     }
     ```

   - 在配置文件中,使用getService方法创建出ServiceImpl对象

     ```
     - 先创建出methodFactory这个对象
     ```

```
<bean id = "methodFactory" class="Factory.methodFactory"></bean>
- 再使用methodFctory对象中的getServic方法创建出ServiceImpl对象
<bean id = "methodbean" factory-bean="methodFactory" factory-method="getServic"></bean>
```

3. 使用某个类的静态方法创建对象
    - 静态工厂类

```
public class staticFactory {

    public static IService getServic(){
        return new ServiceImpl();
    }
}
```

    - 在配置文件中使用该静态方法创建对象

```
<bean id = "staticFactory" class="Factory.staticFactory" factory-method="getServic"></bean>
```

3. bean的作用范围和生命周期

1. 对象的作用范围:

```
1. singleton:单例的
2. prototype:多例的
3. request:web应用下的请求对象
4. session:web应用下的session范围
5. global-session:作用于集群环境下的session
```

2. bean的生命周期:

```
1. 单例对象:
   出生:容器创建时对象自动创建
   活着：容器活着它就活着
   销毁：容器销毁时便销毁
   容器中的对象与容器同生共死

2. 多例对象
   出生:当使用到对象时对象就会被创建
   活着：使用过程中就活着
   销毁：由垃圾回收器对其进行统一回收
```

4. 依赖注入:在对象属性中注入对应的值

1. 使用构造函数
    - java类

```
public class ServiceImpl implements IService {
    private int age;
    private String name;
    private Date birthday;

    public ServiceImpl(int age, String name, Date birthday) {
        this.age = age;
        this.name = name;
        this.birthday = birthday;
    }
}
```

    - 配置文件中的内容

```
 <bean id = "ServiceImpl" class="com.service.ServiceImpl" >
```

```
            <constructor-arg name="name" value="李四"></constructor-arg>
            <constructor-arg name="age" value="20"></constructor-arg>
            <constructor-arg name="birthday" ref = "birthday"></constructor-arg>
 </bean>
 <bean id = "birthday" class="java.util.Date"></bean>
name:指向对象中待注入数据的属性名
value：普通类型和String类型
ref:指向容器中其他的对象
```

2. 使用setter方法

   - 对应的java类

```
public class ServiceImpl implements IService {
        private int age;
        private String name;
        private Date birthday;

        public int getAge() {
            return age;
        }

        public void setAge(int age) {
            this.age = age;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public Date getBirthday() {
            return birthday;
        }

        public void setBirthday(Date birthday) {
            this.birthday = birthday;
        }

    }
```

   - 配置文件中的配置

```
<bean id = "ServiceImpl2" class="com.service.ServiceImpl2" >
            <property name="age" value="22"></property>
            <property name="birthday" ref="birthday"></property>
            <property name="name" value="李四"></property>
</bean>
<bean id = "birthday" class="java.util.Date"></bean>
```

   - 注意

```
1.底层使用反射,所以直接通过配置文件中属性的名字找到对应的setter方法,所以name属性的值与
  对象的属性名并没有必然的关系,而与对象中的setter方法相关
2.举个例子来说,设置year属性的setter方法名为setYear,那么此时标签property属性name的值为
  year,但是假如setter方法名为setMyYear,那么此时property属性的name的值就变为myYear
```

2. Spring常用注解的使用

1. Spring启动注解扫描的环境搭建

1)具体说明

```
- 加入新的约束
- 使用标签<context:component-scan base-package="" />开启注解扫描
```

2)配置文件的配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans.xsd
                http://www.springframework.org/schema/context
                http://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

  <context:component-scan base-package="com.spring.annoation.service"></context:component-scan>
</beans>
```

2. 创建对象的注解

```
    Component注解：默认创建的bean对象的名字为将类名首字母小写
    下面三个注解与上面注解的功能相同,不同是用于三层架构中,使属于三层中的类能够更加清晰得辨识出来

    @Controller：表现层
    @Repository:持久层
    @Service:业务层
```

3. 属性注入的注解

1. @Autowired和@Qualifer注解的使用
    - Autowired
        - 具体说明
            - 如果容器中中存在与要注入属性的变量具有相同类型的对象,那就使用容器中的对象自动注入到变量中 如果容器中不存在与将要注入变量具有相同类型的对象,那么将会报错
            - 这里有一个问题,如果存在多个同种类型但是id却不相同的bean,此时要求被注入的类变量的名称与bean的id 值相等才能成功注入,不然就失败了
            - 在使用注解时,此时set方法就不是必须的了,因为此时并非是使用反射来调用方法对变量进行赋值
        - 具体使用

          ```
          @Autowired
          private IServiceDao serviceDao = null;
          ```

    - @Qualifier注解
        - 具体说明
            - 可以使用@Qualifier(value="serviceDao")与 @Autowired注解配套使用,可以使用该注解指定类变量注入指定id的bean对象，@Qualifier注解单独使用将会报错,只有与@Autowired 配套使用才能生效
            - 当@Autowired注解与@Qualifer注解配合使用时,将先到容器中寻找符合类型要求的对象,然后将所筛选出的bean的id与@Qualifer注解中声明的值进行比较,相等就将该对象注入到变量中,如果找不到就失败
        - 具体代码

          ```
          @Autowired
          ```

```
@Qualifier(value="serviceDao")
private IServiceDao serviceDao = null;
```

2. @Resource注解
   - 具体说明
     - Resource注解注入指定id的bean对象
     - 因为@Resource(name = "serviceDao")可以直接根据id来指定bean对象,所以如果指定的类型与实际要注入的类型不一致的话,就会直接报错
     - 而对于
       @Autowired
       @Qualifier(value="serviceDao")
       则会找到容器中要注入的类的对象,所以此处不会由于注入错误类型而导致错误的情况
   - 代码示例:

```
@Resource(name = "serviceDao")
private IServiceDao serviceDao = null;
```

3. 注意:上面的注解只能注入bean类型,不能注入普通类型或者是String类型而对于复杂类型,比如array,set等的属性需要xml配置才能注入,不能基于注解的注入
4. 普通类型及String类型属性的注入,可以使用@Value注解

```
@Value(value = "22")
private int age;
@Value(value = "nihao")
private String name;
```

4. 作用范围的注解

- 具体说明

```
Scope注解：用于指定作用范围,常用属性为singleton,prototype,一个是单例一个是多例,不写默认是单例的
```

- 代码示例

```
@Scope(value = "singleton")
public class ServiceImpl implements IService {
       。。。。代码的具体实现
}
```

5. 生命周期的注解

- 具体说明

```
@PostConstruct对象创建时将会执行的方法
@PreDestroy对象销毁时将会执行的方法
```

- 代码示例

在类中对应的方法加上@PostConstruct注解,在创建这个对象时,便会执行这个方法
加上@PreDestroy注解,销毁这个对象时,便会执行销毁方法

```
public class ServiceImpl implements IService {

   @PostConstruct
   public void init() {
       System.out.println("对象开始创建");
```

```
    }
    @PreDestroy
    public void destroy() {
         System.out.println("对象的销毁开始执行");
    }

}
```

6. Spring和JUnit的整合

1. 分析前的说明

```
1. 应用程序的入口
    main方法
2. junit没有main方法也可以成功运行
    junit集成了一个main方法,这个方法会当前测试类中哪些方法存在@Test注解
    便让有@Test注解的方法执行
    使用invoke方法来调用方法
3. junit不会察觉到是否采用了Spring框架,所以也不会创建IOC核心容器
4. 所以即使是使用@Autowired注解也不会成功注入对应的对象
```

2. 解决方法

```
    使用新注解,将junit中的main方法创建出IOC核心容器,替换原先的main方法,使得main方法在执行方法前先创建出Spring容器
    @RunWith(SpringJUnit.class),这个SpringJUnit创建出来的main方法会先根据配置文件或配置类创建出Spring容器
    @ContextConfiguration(locations/classes),其中参数用于告知配置类或者是xml的位置
    location:用于告知配置文件的位置,classpath表明是位于类路径下的
    classes:用于告知配置类的具体类
```

3. 具体示例

(1)Spring使用xml文件和注解结合开发的方式

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:bean.xml")
public class Test {
    @Autowired
  private  IAccountService iAccountService;
    @org.junit.Test
    public void test01() throws SQLException {
        List<Account> list = iAccountService.findAll();
        for(Account account : list){
            System.out.println(account.toString());
        }
   }
}
```

(2)使用纯注解的配置方式

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SpringConfig.class)
public class Test {

    @Autowired
    private IAccountService accountService ;
    @org.junit.Test
    public void test01() throws SQLException {

        List<Account> list = accountService.findAll();
        for(Account account : list){
            System.out.println(account.toString());
        }
    }
}
```

7. 纯注解式的开发

1. 纯注解式开发步骤
    - 创建一个配置类,创建容器时直接传入该配置类的字节码
    - 配置类上方使用注解@ComponentScans来开启注解扫描
    - 可以使用@Import引入其他配置类
    - 一些配置信息存储在外部文件中,可以使用注解@PropertySource来引入外部文件,并且使用EL表达式来获取文件中的值
    - 第三方类库对象的创建可以使用配置类中方法加上@bean注解
2. 具体实例:创建了两个配置类
    - 具体代码

```
@ComponentScans(value = {@ComponentScan(value = "dao"),@ComponentScan(value = "Service")})
@Import(JdbcConfig.class)
public class SpringConfig {

}


@PropertySource("classpath:jdbc.properties")
public class JdbcConfig {

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

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

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

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


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

    @Bean(name="datasource")
    public DataSource createDataSource() throws PropertyVetoException {
        ComboPooledDataSource comboPooledDataSource = new ComboPooledDataSource();
        comboPooledDataSource.setUser(name);
        comboPooledDataSource.setDriverClass(driver);
        comboPooledDataSource.setPassword(password);
        comboPooledDataSource.setJdbcUrl(url);
        return comboPooledDataSource;
    }
}
```

    - 配置类的说明
        1. 在创建容器时传入主配置类SpringConfig的字节码,在主配置类上方加上@ComponentScans注解来开启注解扫描
        2. @Import注解来引入从配置文件
        3. 在从配置文件中使用注解@PropertySource来引入外部属性文件,并且使用@Value注解加上EL表达式来导入属性的值
        4. 在从配置文件中使用方法加上@Bean注解来创建第三方类库中类的对象

动态代理和AOP的使用

动态代理

1. JDK提供的动态代理的实现

- JDK提供的动态代理的说明

  1. 动态代理使用JDK提供的Proxy中的newProxyInstance方法来生成代理类
  2. 这个代理类是随时使用,字节码随时生成,并有类加载器进行加载
  3. 使用newProxyInstance方法需要传入3个参数

     - 第一个是被代理类的类加载器,一般是系统类加载器
     - 第二个参数是字节码数组,这个是接口的字节码数组,能够识别出该接口所具有的所有方法
     - 第三个方法是实现InvocationHandler接口,一般使用匿名内部类来实现,接口中的invoke方法是用于动态代理中编写增强代码的部分
- 代码说明

```
IProducer iProducer = (IProducer) Proxy.newProxyInstance
(Producer.class.getClassLoader(), producer.getClass().getInterfaces(),
 new InvocationHandler(){

                public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                    if (method.getName() == "sale") {
                        System.out.println("这是sale方法");
                    }
                    System.out.println("对所有的方法进行增强");
                    return method.invoke(producer,args);
                }
    });
```

- JDK生成的IProducer接口实现的代理类

```
public final class pro extends Proxy implements IProducer {
       private static Method m1;
       private static Method m3;
       private static Method m2;
       private static Method m4;
       private static Method m0;

       public pro(InvocationHandler var1) throws  {
           super(var1);
       }

       public final boolean equals(Object var1) throws  {
           try {
               return (Boolean)super.h.invoke(this, m1, new Object[]{var1});
           } catch (RuntimeException | Error var3) {
               throw var3;
           } catch (Throwable var4) {
               throw new UndeclaredThrowableException(var4);
           }
       }

       public final void sale(Float var1) throws  {
           try {
               super.h.invoke(this, m3, new Object[]{var1});
           } catch (RuntimeException | Error var3) {
               throw var3;
           } catch (Throwable var4) {
               throw new UndeclaredThrowableException(var4);
           }
       }

       public final String toString() throws  {
           try {
               return (String)super.h.invoke(this, m2, (Object[])null);
           } catch (RuntimeException | Error var2) {
               throw var2;
```

```
            } catch (Throwable var3) {
                throw new UndeclaredThrowableException(var3);
            }
        }

        public final void afterSale(Float var1) throws  {
            try {
                super.h.invoke(this, m4, new Object[]{var1});
            } catch (RuntimeException | Error var3) {
                throw var3;
            } catch (Throwable var4) {
                throw new UndeclaredThrowableException(var4);
            }
        }

        public final int hashCode() throws  {
            try {
                return (Integer)super.h.invoke(this, m0, (Object[])null);
            } catch (RuntimeException | Error var2) {
                throw var2;
            } catch (Throwable var3) {
                throw new UndeclaredThrowableException(var3);
            }
        }

        static {
            try {
                m1 = Class.forName("java.lang.Object").getMethod("equals", Class.forName("java.lang.Object"));
                m3 = Class.forName("proxy.interfaceProxy.IProducer").getMethod("sale", Class.forName("java.lang.Float"));
                m2 = Class.forName("java.lang.Object").getMethod("toString");
                m4 = Class.forName("proxy.interfaceProxy.IProducer").getMethod
                ("afterSale", Class.forName("java.lang.Float"));
                m0 = Class.forName("java.lang.Object").getMethod("hashCode");
            } catch (NoSuchMethodException var2) {
                throw new NoSuchMethodError(var2.getMessage());
            } catch (ClassNotFoundException var3) {
                throw new NoClassDefFoundError(var3.getMessage());
            }
        }
}
```

那么这是如何实现的呢?JDK提供的Proxy.newProxyInstance方法传入的第二个参数为被代理类实现接口的Class对象,查看上面中代理类的实现,我们可以猜测,代理类根据传入的接口的字节码,可以创建出代理类中对应的方法以及接口中存在的方法对象 而对于第三个参数InvocationHandler接口的实现类,在创建代理类时,代理类创建一个带参数的构造方法,直接将这个实现类对象作为参数传入,就像下面这样

```
  public pro(InvocationHandler var1) throws  {
    super(var1);
  }
```

在代理类中方法的实现,就直接调用InvoactionHandler接口实现类中的invoke方法,然后传入的参数为该方法对应的Method对象, 所以每执行一个接口方法,其实就是执行InvoactionHandler接口实现类中的invoke方法,只是每个方法传入的Method对象不同,在InvoactionHandler接口中的invoke中直接调用method的invoke方法

2. 继承实现子类的动态代理(cglib)：可以生成没有实现接口的类的代理对象

```
 这里是使用第三方类库来生成子类的代理类
 其中使用Enhancer.create方法来生成代理对象
      参数：
      Class:类的字节码,代理类与被代理类存在相同的方法
      callBack:使用实现该接口的类,增强被代理类

 callback接口使用MethodInterceptor的匿名实现类，参数说明
    Object o:
    Method  method:方法对象
    Object[] objects:方法的参数
```

```
    MethodProxy:代理对象

    使用这个方法创建出一个被代理类的增强类,cglib可以实现对子类的增强,通过传入的Class对象
    来生成与被代理对象具有同样方法的代理对象

    然后在intercept中编写出对方法增强的代码


final Producer producer = new Producer();
Producer producer1 = (Producer) Enhancer.create(Producer.class, new MethodInterceptor() {
    public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {
        System.out.println("所有的方法都被增强");
        return method.invoke(producer,objects);
    }
});
```

3. Spring中XML文件配置的AOP

1. AOP的基本概念
    - Spring将根据是否实现接口来决定使用JDK动态代理还是Cglib来实现AOP
    - AOP:面向切面编程,使用切面类中的方法来对要加强的类进行代码的增强,一次性可以多个类中的多个方法同时进行加强,这个就称之为面向切面编程
2. XML文件中配置AOP
    - 将通知类交给Spring进行管理
    - 使用aop:config标签表明开始AOP配置
    - 使用aop:aspect标签表明开始切面配置
        - id:切面的唯一标识
        - ref:通知类(增强类的标识)
    - 在aop:aspect内部使用对应标签来配置通知的类型
        - 前置通知:<aop:before method="beforeLog" pointcut-ref="pc1"></aop:before>
        - 后置通知:<aop:after-returning method="afterReturningLog" pointcut-ref="pc1"></aop:after-returning>
        - 异常通知:<aop:after-throwing method="catchLog" pointcut-ref="pc1"></aop:after-throwing>
        - 最终通知:<aop:after method="afterLog" pointcut-ref="pc1"></aop:after>
    - 各类通知的内部属性

      ```
      - method:指明增强的方法
      - pointcut:编写切入点表达式
      - pointcut-ref:使用之前编写好的切入点表达式
      - pointcut-ref编写的位置,在<aop:aspect></aop:aspect>处编写可以让所有的切面都使用
      ```

3. 切入点表达式的省略编写的方式

```
(一般写到业务层中所有的方法进行增强)
表达式的省略表达:
    1. 访问修饰符可以省略
    2. 返回值用 * 表示任意类型的返回值
    3. 包名可以使用通配符*来表示,有多少个包需要多少个通配符
    可以使用*..表示当前包及其子包
    类名和方法名使用通配符*来表示
    4. 参数可以使用*来表示,表示任意类型,但是必须有参数
    使用..来表示有无参数即可,有参数的话可以是任意类型参数

    实际开发中:一般切到业务层实现类下的所有方法
```

4. 环绕通知的编写
    - 编写环绕通知对应的代码,在切面类中编写该环绕方法

      ```
      public Object around(ProceedingJoinPoint proceedingJoinPoint){
          Object rtValue = null;
      ```

```
    try {
        System.out.println("进行前置通知");
        Object[] par = proceedingJoinPoint.getArgs();
        rtValue = proceedingJoinPoint.proceed(par);//明确执行方法,执行被切入的方法
        System.out.println("进行后置通知");
        return rtValue;
    } catch (Throwable throwable) {
        System.out.println("执行异常通知");
        throwable.printStackTrace();
    }finally {
        System.out.println("执行最终通知");
    }
    return null;
}
```

- 说明
  - AOP中环绕通知方法的增强,并不会调用method.invoke方法来调用原先待增强的方法,所以在编写环绕方法时需要使用Spring提供的ProceedingJoinPoint来调用原先的方法
  - 编写环绕通知,可以使用Spring提供的ProceedingJoinPoint类来直接调用类中的方法,环绕方法其实是提供了另一种实现增强的方式,不是通过配置的方式来对代码进行增强而是使用编码的方式来实现增强,可以自行决定到底要在何处进行增强
  - 配置AOP时,配置了环绕方法的增强就不使用其他类型的增强

5. AOP配置的具体代码示例

```
- 切面类
<bean id = "logger" class="util.Logger"></bean>
- AOP配置
<aop:config>
    <aop:pointcut id="pc1" expression="execution( *  service.AccountService.*(..))"/>
    <aop:aspect id = "loggerAop" ref="logger">
        <aop:before method="beforeLog" pointcut-ref="pc1"></aop:before>
        <aop:after-returning method="afterReturningLog"  pointcut-ref="pc1"></aop:after-returning>
        <aop:after method="afterLog" pointcut-ref="pc1"></aop:after>
        <aop:after-throwing method="catchLog" pointcut-ref="pc1"></aop:after-throwing>
        <aop:before method="beforeLog" pointcut="" />
        环绕通知
        <aop:around method="around" pointcut-ref="pc1"></aop:around>
    </aop:aspect>
</aop:config>
```

4. AOP基于注解的开发

1. 通用步骤

```
    1. 在容器中开启注解的扫描,扫描对应的包
    2. 在配置文件中开启aop配置的功能
       <aop:aspectj-autoproxy />
    3. 在类上加入对应的注解
        - 在切面类处加入@Aspect注解,用于声明这是一个切面类
        - 在切面类中对应的方法处加上下面的注解
         @Before("pc1()")
         @AfterReturning("pc1()")
         @AfterThrowing("pc1()")
         @After("pc1()")
        用于声明在方法中加入不同的增强
     4. 切入点的配置,在切面类内部编写这个方法
      @Pointcut("execution( *  service.AccountService.*(..))")
       private void pc1(){};

    5. 问题:使用注解的AOP,方法的执行顺序:前置通知,最终通知,后置通知,这样的执行顺序将会导致一些问题
```

2. 举例说明:声明为切面类,并且同时在此处声明切入点并配置方法类型

```
@Component("logger")
@Aspect//声明这是一个切面类
public class Logger {


    @Pointcut("execution( *  service.AccountService.*(..))")
    private void pc1(){};
//
    @Before("pc1()")
    public void beforeLog(){
        System.out.println("beforeLog开启记录日志_前置通知");
    }
    @AfterReturning("pc1()")
    public void afterReturningLog(){
        System.out.println("afterReturningLog开启记录日志_后置通知");
    }
    @AfterThrowing("pc1()")
    public void catchLog(){
        System.out.println("catchLog开启记录日志_异常通知");
    }
    @After("pc1()")
    public void afterLog(){
        System.out.println("afterLog开启记录日志_最终通知");
    }

    //环绕方法
    @Around("pc1()")
    public Object around(ProceedingJoinPoint proceedingJoinPoint){
        Object rtValue = null;
        try {
            System.out.println("进行前置通知");
            Object[] par = proceedingJoinPoint.getArgs();
            rtValue = proceedingJoinPoint.proceed(par);//明确执行方法,执行被切入的方法
            System.out.println("进行后置通知");
            return rtValue;
        } catch (Throwable throwable) {
            System.out.println("执行异常通知");
            throwable.printStackTrace();
        }finally {
            System.out.println("执行最终通知");
        }
        return null;
    }
}
```

JdbcTemplate及事务的使用

1. jdbcTemplate类的使用

- 将JdbcTemplate注入到dao层,在Dao层设置JdbcTemplate类型的变量

```
@Repository
public class AccountDaoImpl implements IAccountDao {

    private JdbcTemplate jdbcTemplate;

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

    public void save() {
       jdbcTemplate.update("insert into account values(33,'ccc','32434433','6565')");
    }

    public void delete(int i) {
```

```
        System.out.println("实现了删除");
    }
}
```

- 问题:

```
如果存在多个dao层,那么此时就存在重复代码
    private JdbcTemplate jdbcTemplate;

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

如何消除这些重复代码,可以将让dao层的实现类继承JdbcDaoSupport,这个类中就存在一个JdbcTemplate变量,可以使用getJdbcTemplate去获取该变量
然后使用这个变量去执行增删改查的代码
```

- 区别:

```
自行设置JdbcTemplate变量和继承JdbcDaoSupport所获取的JdbcTemplate变量的区别:
自行设置的变量可以通过注解方式进行注入,也可以使用XML文件格式进行注入
而通过继承所获取的变量,只能通过XML文件对变量进行注入
```

2. Spring基于XML文件的事务配置

- 配置好事务管理器,这个事务管理器需要一个数据源

```
    <bean id = "transManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
       <property name="dataSource" ref="datasource"></property>
    </bean>
```

- 配置一个事务的通知

```
    <tx:advice id="txManager" transaction-manager="transManager" >
        <tx:attributes>
            <tx:method name="findAll"/>
        </tx:attributes>
    </tx:advice>

    事务属性的设置
    tx:method:对某个方法的属性进行设置,并且必须进行设置,如果没有设置的话,那么肯定会发生错误


    isolation:设置事务的隔离级别,DEFAULT表示是默认的隔离级别
    propagation:设置事务的传播行为,默认是REQUIRED,表示一定会存在事务,当查询是可以使用SUPPORTS,表示是不使用事务
    read-only:设置事务是否为只读事务
    rollback-for:当某个方法发生异常,才发生回滚,默认事务都会回滚
    no-rollback-for:除了某个方法外,其他方法发生异常都会进行回滚,默认事务都会回滚
```

- 为切入点配置事务管理器,切入点就会增加对应的事务方法

```
    <aop:config>
       <aop:pointcut id="pc1" expression="execution(* Service.*.*(..))"/>
       <aop:advisor advice-ref="txManager" pointcut-ref="pc1"></aop:advisor>
    </aop:config>
```

- 在tx:advice标签内部需要为事务管理器设置属性,可以为加上事务的方法加上一些事务的属性,如果不加上,那么这个设置将会失败,并且所有需要加上事务的方法都需要进行设置， 比如在<tx:advice></tx:advice>标签内部设置事务方法的属性,其中 <tx:method>方法内部的name属性可以使用*来表示某一个方法,或者是find*来表示以find开头的所有方法

```
  <tx:attributes>
            <tx:method name="findAll"/>
  </tx:attributes>
```

3. 基于注解的事务配置

- 具体的配置步骤

  1. 首先配置一个spring提供的事务管理器

     ```
     <bean id = "transManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
          <property name="dataSource" ref="datasource"></property>
     </bean>
     ```

  2. 配置spring开启注解驱动的方式,此时需要设置好一个事务管理器

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

  3. 在需要配置事务的位置加上@Transactional注解

     ```
     @Transactional //为该类中所有的方法都加上事务,使用事务默认的配置
     //如果需要为不同的方法中事务设置不同的类型,此时可以在不同方法上加上@Transactional中设置不同的属性即可
     public class AccountServiceImpl implements IAccountService {
         此时如果在不同的方法配置上不同配置的事务,此时在不同的方法上都需要写上注解

     }
     ```