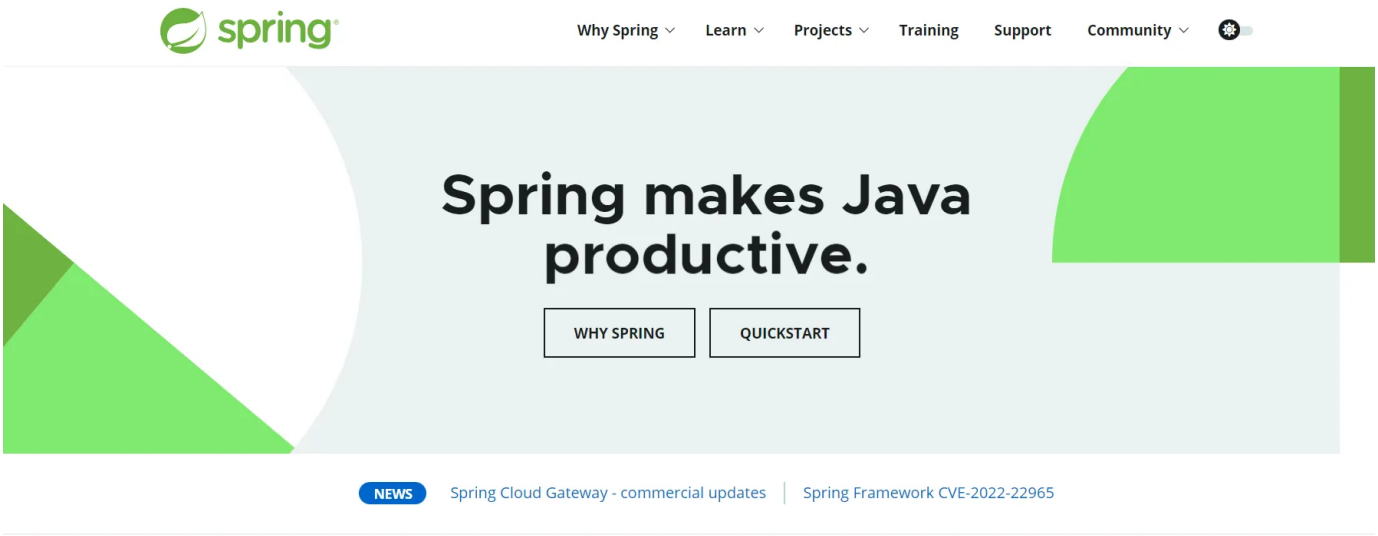
Spring

1.Spring入门

1.1 Spring是什么

Spring是分层的JavaSE/EE full-stack(一站式) 轻量级开源框架 以IoC（Inverse of Control 反转控制）和AOP（Aspect Oriented Programming 面向切面编程为 内核）

官网：http://www.springsource.org/

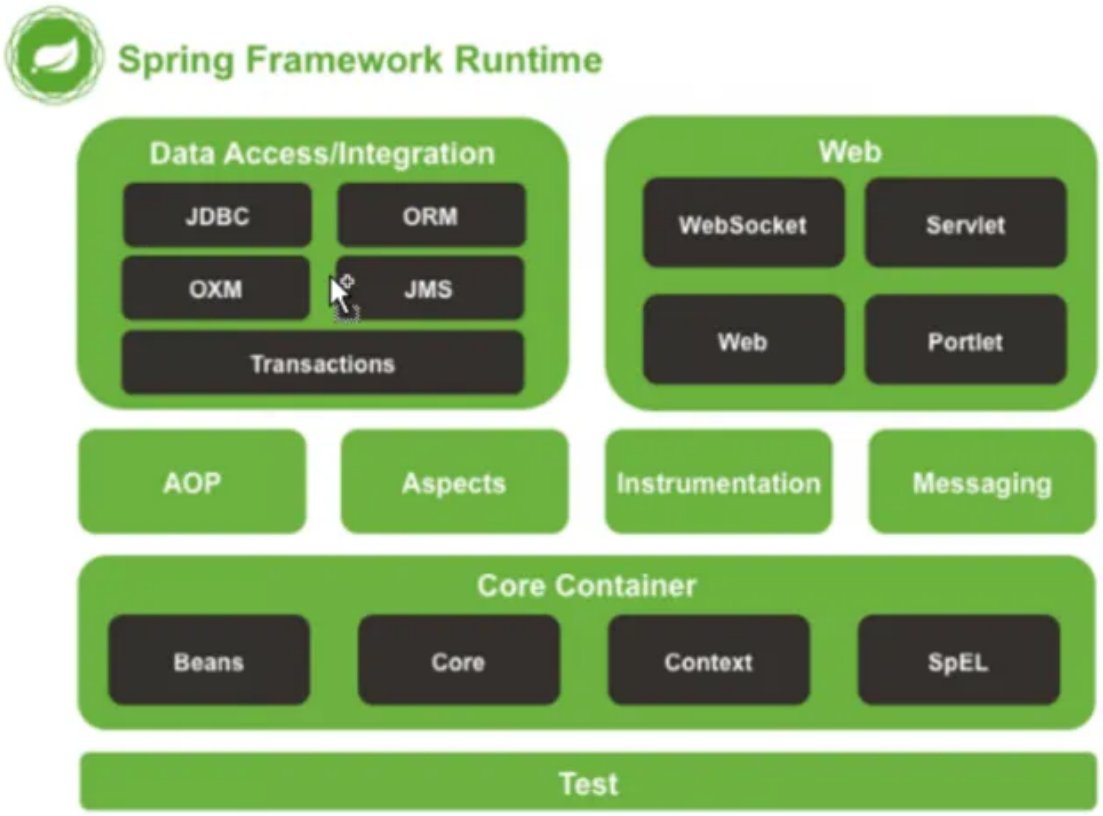


Spring旗下的众多项目

项目列表：https://spring.io/projects

1.2 Spring Framework

Spring 基础框架，可以视为 Spring 基础设施，基本上任何其他 Spring 项目都是以 Spring Framework 为基础的。



ssh spring+struts2+hibernate

ssm spring+springmvc+mybatis

springdatajpa

①Spring Framework优良特性

- 非侵入式：使用 Spring Framework 开发应用程序时，Spring 对应用程序本身的结构影响非常小。对领域模型可以做到零污染；对功能性组件也只需要使用几个简单的注解进行标记，完全不会破坏原有结构，反而能将组件结构进一步简化。这就使得基于 Spring Framework 开发应用程序时结构清晰、简洁优雅。
- 控制反转：IOC——Inversion of Control，翻转资源获取方向。把自己创建资源、向环境索取资源变成环境将资源准备好，我们享受资源注入。
- 面向切面编程：AOP——Aspect Oriented Programming，在不修改源代码的基础上增强代码功能。
- 容器：Spring IOC 是一个容器，因为它包含并且管理组件对象的生命周期。组件享受到了容器化的管理，替程序员屏蔽了组件创建过程中的大量细节，极大的降低了使用门槛，大幅度提高了开发效率。

- 组件化：Spring 实现了使用简单的组件配置组合成一个复杂的应用。在 Spring 中可以使用 XML 和 Java 注解组合这些对象。这使得我们可以基于一个个功能明确、边界清晰的组件有条不紊的搭建超大型复杂应用系统。
- 声明式：很多以前需要编写代码才能实现的功能，现在只需要声明需求即可由框架代为实现。
- 一站式：在 IOC 和 AOP 的基础上可以整合各种企业应用的开源框架和优秀的第三方类库。而且 Spring 旗下的项目已经覆盖了广泛领域，很多方面的功能性需求可以在 Spring Framework 的基础上全部使用 Spring 来实现。

②Spring Framework五大功能模块

功能模块	功能介绍
Core Container	核心容器，在 Spring 环境下使用任何功能都必须基于 IOC 容器。
AOP&Aspects	面向切面编程
Testing	提供了对 junit 或 TestNG 测试框架的整合。
Data Access/Integration	提供了对数据访问/集成的功能。
Spring MVC	提供了面向Web应用程序的集成功能。

2.IOC&DI

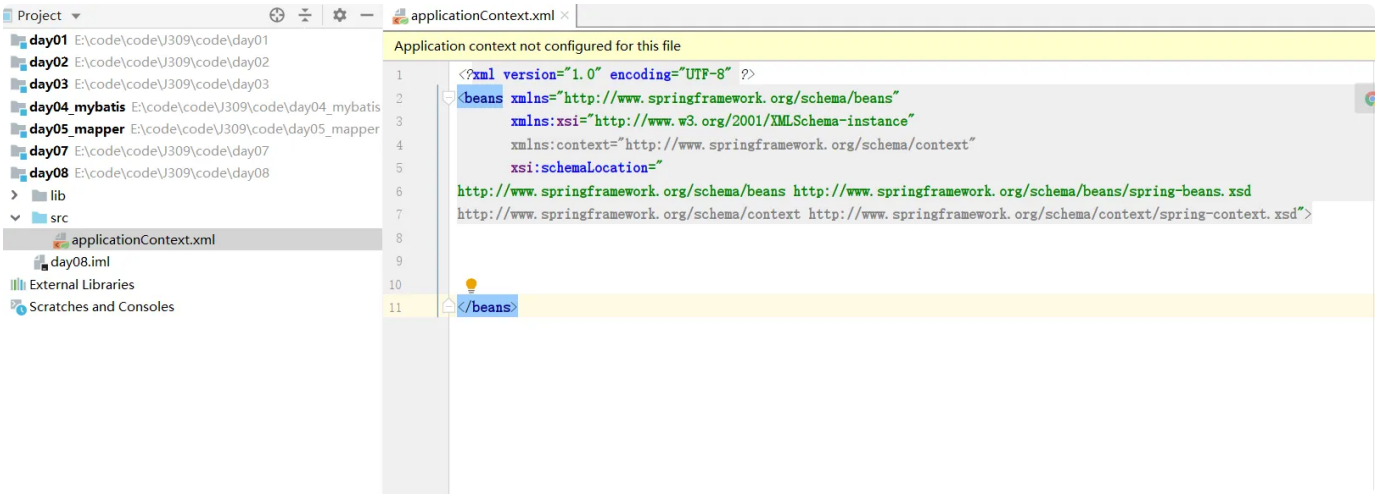
IOC：Inversion of Control，翻译过来是**反转控制**。

2.1 入门体验

2.1.1 导包 IOC和DI的jar包

spring-beans-4.1.3.RELEASE.jar	2016/5/24 16:56
spring-context-4.1.3.RELEASE.jar	2016/5/24 16:56
spring-context-support-4.1.3.RELEA...	2016/5/24 16:56
spring-core-4.1.3.RELEASE.jar	2016/5/24 16:56
spring-expression-4.1.3.RELEASE.jar	2016/5/24 16:56

2.1.2 准备配置文件



Java

```java
/**
 * Created by zxd on 2022/10/21 10:51
 */
public class CVN {

    //注入普通属性
    private String name;

    private Long length;

    private Long width;

    //注入的是一个已经被spring管理器的对象
    private Fighter fighter;

    public Fighter getFighter() {
        return fighter;
    }

    public void setFighter(Fighter fighter) {
        this.fighter = fighter;
    }

    public String getName() {
        return name;
    }

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

    public Long getLength() {
        return length;
    }

    public void setLength(Long length) {
        this.length = length;
    }

    public Long getWidth() {
        return width;
    }

    public void setWidth(Long width) {
        this.width = width;
```

```
    }


    @Override
    public String toString() {
        return "CVN{" +
                "name='" + name + '\'' +
                ", length=" + length +
                ", width=" + width +
                ", fighter=" + fighter +
                '}';
    }
}
```

Java

```
/**
 * Created by zxd on 2022/10/21 15:26
 */
public class Fighter {

    private String name;

    public String getName() {
        return name;
    }

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

    @Override
    public String toString() {
        return "Fighter{" +
                "name='" + name + '\'' +
                '}';
    }
}

```

2.1.3 配置

```Java
<?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">



    <bean id="cvn" class="com.gxa.bean.CVN"/>



</beans>
```

2.1.4 测试

```java
/**
 * Created by zxd on 2022/10/24 10:47
 */
public class TestSpring {


    @Test
    public void testSpring(){

        //1.创建spring容器
        ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");


        //2.从spring容器中获取指定的bean
        CVN cvn = (CVN) applicationContext.getBean("cvn2");

        System.out.println(cvn);


    }
}

```

2.2 Spring中的IOC实现

Spring 的 IOC 容器就是 IOC 思想的一个落地的产品实现。IOC 容器中管理的组件也叫做 bean。在创建 bean 之前，首先需要创建 IOC 容器。

2.2.1 BeanFctory

这是 IOC 容器的基本实现，是 Spring 内部使用的接口。面向 Spring 本身，不提供给开发人员使用。

Spring容器(工程)顶层接口，封装了spring工厂最基本的方法的规范

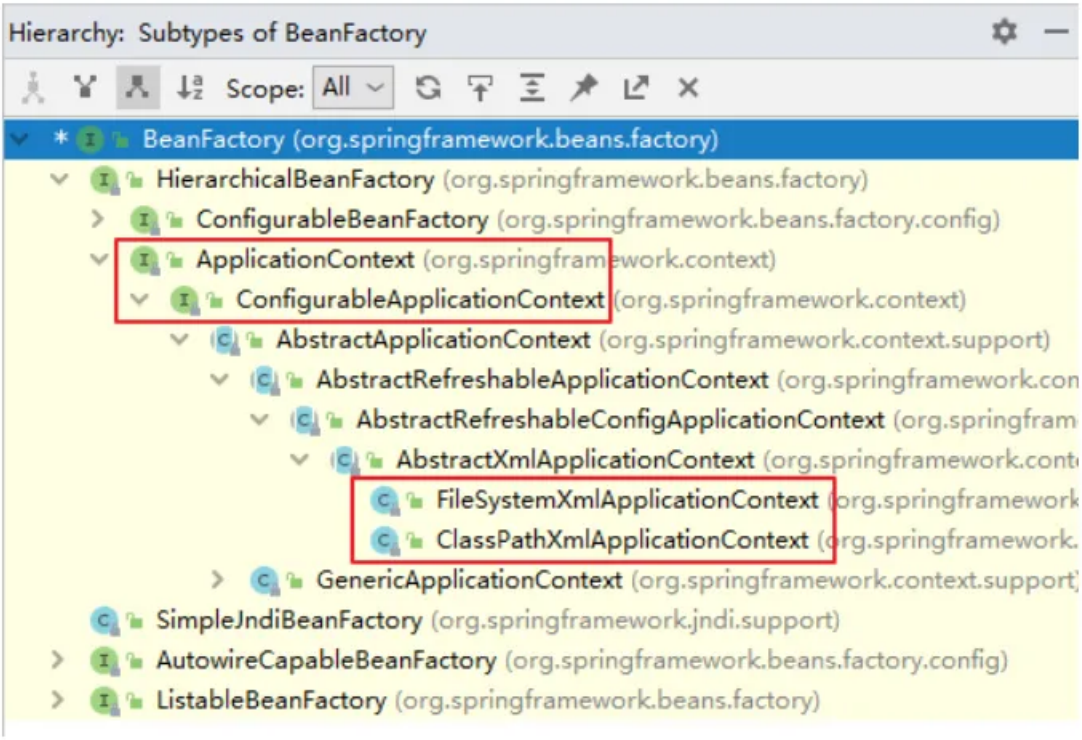
Java

```java
public interface BeanFactory {
    String FACTORY_BEAN_PREFIX = "&";

    //获取spring容器中指定的bean对象
    Object getBean(String var1) throws BeansException;

    <T> T getBean(String var1, Class<T> var2) throws BeansException;

    <T> T getBean(Class<T> var1) throws BeansException;

    Object getBean(String var1, Object... var2) throws BeansException;

    <T> T getBean(Class<T> var1, Object... var2) throws BeansException;

    boolean containsBean(String var1);

    boolean isSingleton(String var1) throws NoSuchBeanDefinitionException;

    boolean isPrototype(String var1) throws NoSuchBeanDefinitionException;

    boolean isTypeMatch(String var1, Class<?> var2) throws NoSuchBeanDefinitionException;

    Class<?> getType(String var1) throws NoSuchBeanDefinitionException;

    String[] getAliases(String var1);
}

```

2.2.2 ApplicationContext

BeanFactory 的子接口，提供了更多高级特性。面向 Spring 的使用者，几乎所有场合都使用 ApplicationContext 而不是底层的 BeanFactory。

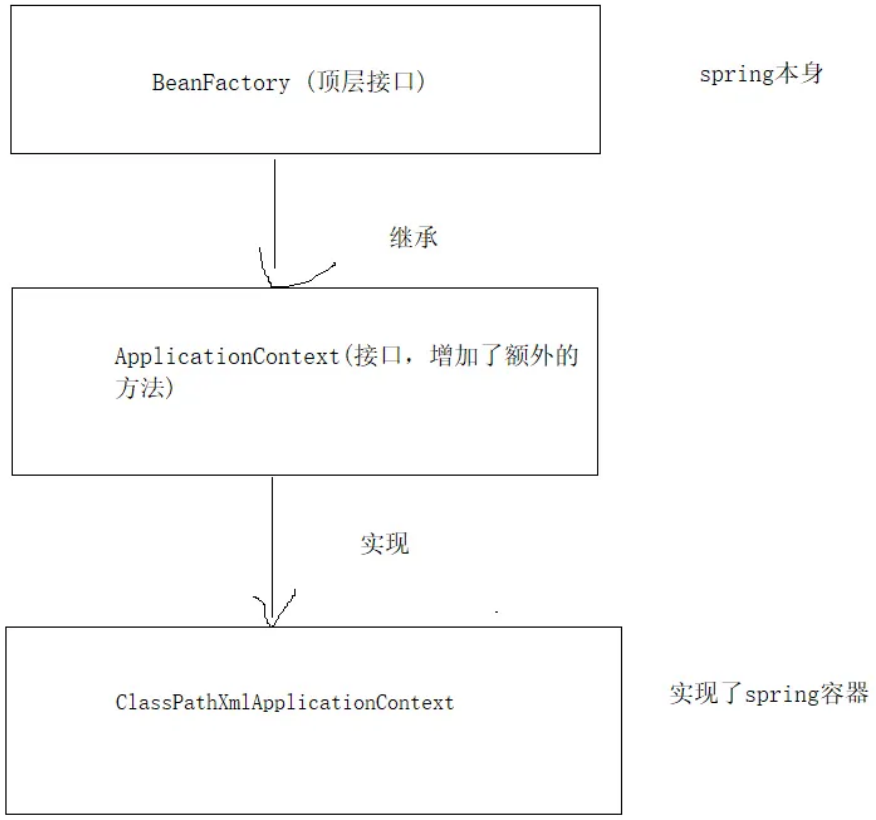
以后在 Spring 环境下看到一个类或接口的名称中包含 ApplicationContext，那基本就可以断定，这个类或接口与 IOC 容器有关。



2.2.3 ApplicationContext的主要实现类

类型名	简介
ClassPathXmlApplicationContext	通过读取类路径下的 XML 格式的配置文件创建 IOC 容器对象

FileSystemXmlApplicationContext	通过文件系统路径读取 XML 格式的配置文件创建 IOC 容器对象
ConfigurableApplicationContext(接口)	ApplicationContext 的子接口，包含一些扩展方法 refresh() 和 close() ，让 ApplicationContext 具有启动、关闭和刷新上下文的能力。
WebApplicationContext(接口)	专门为 Web 应用准备，基于 Web 环境创建 IOC 容器对象，并将对象引入存入 ServletContext 域中。



spring控制反转 把对象的创建权交给spring容器进行管理

Spring提供了几种实例化对象的方式

1.构造方法实例化(默认会使用无参构造器)　　99%

2.静态工厂实例化 了解

3.实例化工厂实例化 了解

2.2.4 无参构造器实例化(重点)

Java

```
1    <bean id="cvn" class="com.gxa.bean.CVN"/>
```

2.2.5 静态工厂实例化

编写静态工厂

```java
/**
 * Created by zxd on 2022/10/24 11:22
 */
public class StaticFactory {


    /**
     * 创建航母
     * @return
     */
    public static CVN  createCvn(){
        CVN cvn=new CVN();
        cvn.setName("福建号");
        return cvn;
    }

}


```

```Java
<?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">



    <!--1.无参构造器实例化 -->
    <bean id="cvn" class="com.gxa.bean.CVN"/>


    <!--2.静态工厂实例化-->
    <bean id="cvn2" class="com.gxa.factory.StaticFactory" factory-method="createCvn"/>


</beans>
```

2.2.6 实例化工厂实例化

工厂创建创建对象的方法是非静态方法

因为工程的方法是非静态的，先把让spring创建工厂的对象，把工厂交给spring进行管理，然后有spring来调用工厂创建对象的非静态方法来创建对应的对象，把这个对象交给spring进行管理

```java
/**
 * Created by zxd on 2022/10/24 11:22
 */
public class NoStaticFactory {


    /**
     * 创建航母
     * @return
     */
    public  CVN  createCvn(){
        CVN cvn=new CVN();
        cvn.setName("福建号");
        return cvn;
    }

}
```

```java
   <!--3.实例化工厂实例化-->
    <!--先创建工厂对象-->
    <bean id="noStaticFactory" class="com.gxa.factory.NoStaticFactory"></bean>

    <bean id="cvn3" factory-bean="noStaticFactory" factory-method="createCvn"/>

```

2.2.7 FactoryBean接口

Java

```java
public interface FactoryBean<T> {

    //工厂创建对象的方法  spring会调用getObject来创建对象
    T getObject() throws Exception;

    //获取对象的类型
    Class<?> getObjectType();

    //是否单例
    boolean isSingleton();
}

```

Java

```java
/**
 * Created by zxd on 2022/10/24 11:40
 */
public class CvnFactoryBean implements FactoryBean<CVN> {

    //工厂创建对象的方法  spring会调用getObject来创建对象
    @Override
    public CVN getObject() throws Exception {
        CVN cvn=new CVN();
        cvn.setName("福建号");
        return cvn;
    }

    @Override
    public Class<?> getObjectType() {
        return CVN.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}

```

配置

```Java
<bean id="cvn5" class="com.gxa.factory.CvnFactoryBean"></bean>
```

spring会检测这个类是否实现了FactoryBean这个接口，如果实现了接口代表当前类是一个工厂，spring会自动创建工厂对象，然后自动的去调用getObject方法来创建对象交给spring进行管理

BeanFactory和FactoryBean的区别?

1.BeanFactory代表spring容器本身，是spring容器的顶层接口

2.FactoryBean是一个工厂Bean的接口，创建某个对象的

(spring会检测这个类是否实现了FactoryBean这个接口，如果实现了接口代表当前类是一个工厂，spring会自动创建工厂对象，然后自动的去调用getObject方法来创建对象交给spring进行管理)

2.2.7 bean的作用范围

单例　一个bean标签创建出来的对象在整个spring容器只有一个

在Spring中可以通过配置bean标签的scope属性来指定bean的作用域范围，各取值含义参加下表：

取值	含义	创建对象的时机
singleton	在IOC容器中，这个bean的对象始终为单实例	IOC容器初始化时

prototype	这个bean在IOC容器中有多个实例	获取bean时

如果是在WebApplicationContext环境下还会有另外两个作用域（但不常用）：

取值	含义
request	在一个请求范围内有效
session	在一个会话范围内有效

2.2.8 自定义初始化和销毁的方法

init-method 指定初始化的方法 spring创建对象后调用初始化的方法

destroy-method 指定销毁的方法 spring在销毁对象前调用该销毁的方法

Java

```java
/**
 * Created by zxd on 2022/10/21 10:51
 */
public class CVN {

    //注入普通属性
    private String name;

    private Long length;

    private Long width;

    //注入的是一个已经被spring管理器的对象
    private Fighter fighter;


    public void initMethod(){
        System.out.println("对象被初始化了...");
    }

    public void destroyMethod(){
        System.out.println("对象被销毁了...");
    }



    public Fighter getFighter() {
        return fighter;
    }

    public void setFighter(Fighter fighter) {
        this.fighter = fighter;
    }

    public String getName() {
        return name;
    }

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

    public Long getLength() {
        return length;
    }
```

```
46
47      public void setLength(Long length) {
48          this.length = length;
49      }
50
51      public Long getWidth() {
52          return width;
53      }
54
55      public void setWidth(Long width) {
56          this.width = width;
57      }
58
59
60      @Override
61      public String toString() {
62          return "CVN{" +
63                  "name='" + name + '\'' +
64                  ", length=" + length +
65                  ", width=" + width +
66                  ", fighter=" + fighter +
67                  '}';
68      }
69  }
70
```

Java

```
1     <!--1.无参构造器实例化 -->
2       <bean id="cvn" class="com.gxa.bean.CVN" scope="singleton" init-method=
    "initMethod" destroy-method="destroyMethod"/>
3
```

测试

```java
    @Test
    public void testSpring(){

        //1.创建spring容器
        ApplicationContext applicationContext = new ClassPathXmlApplicatio
nContext("applicationContext.xml");


        //2.从spring容器中获取指定的bean
        CVN cvn = (CVN) applicationContext.getBean("cvn");

        System.out.println(cvn);

        CVN cvn2 = (CVN) applicationContext.getBean("cvn");

        System.out.println(cvn2);


    }
```

如果要测试销毁的方法，必须要使用ClassPathXmlApplicationContext来调用

```java
    @Test
    public void testSpring(){

        //1.创建spring容器
        ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");


        //2.从spring容器中获取指定的bean
        CVN cvn = (CVN) applicationContext.getBean("cvn");

        System.out.println(cvn);

        CVN cvn2 = (CVN) applicationContext.getBean("cvn");

        System.out.println(cvn2);

        applicationContext.close();

    }
```

如果sope属性配置为prototype，不会调用销毁的方法，因为spring容器不会保存对象，只会每次获取的时候创建

```java
    <!--1.无参构造器实例化 -->
    <bean id="cvn" class="com.gxa.bean.CVN" scope="prototype" init-method="initMethod" destroy-method="destroyMethod"/>
```

ssh　spring+struts2+hibernate

struts2封装Filter　UserAction DepartmentAction　..　多例的　spring整合struts2　scope="prototype"　复杂3倍，性能低下

springmvc 封装servlet UserConteoller DepartmentController 单例的

简单，性能高

2.3 DI(依赖注入)

在创建对象的时候给属性赋值

2.3.1 构造器器注入

Java

```java
/**
 * Created by zxd on 2022/10/21 15:26
 */
public class Fighter {

    private String name;

    private String color;

    public Fighter(){

    }

    public Fighter(String name, String color) {
        this.name = name;
        this.color = color;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public String getName() {
        return name;
    }

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

    @Override
    public String toString() {
        return "Fighter{" +
                "name='" + name + '\'' +
                '}';
    }
}

```

```Java
    <bean id="fighter" class="com.gxa.bean.Fighter">
        <constructor-arg name="name" value="飞鲨"/>
        <constructor-arg name="color" value="银色"/>
    </bean>
```

```Java
    @Test
     public void testSpring2(){

         //1.创建spring容器
         ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");


         Fighter fighter = applicationContext.getBean(Fighter.class);

         System.out.println(fighter);

     }
```

也可以通过序号来指定构造器参数的位置,不推荐

```Java
    <bean id="fighter" class="com.gxa.bean.Fighter">
        <constructor-arg name="name" value="飞鲨"/>
        <constructor-arg index="1" value="银色"/>
    </bean>
```

2.3.2 set方法注入(重要)

Java

```java

/**
 * Created by zxd on 2022/10/21 10:51
 */
public class CVN {

    //注入普通属性
    private String name;

    private Long length;

    private Long width;

    //注入的是一个已经被spring管理器的对象
    private Fighter fighter;


    public void initMethod(){
        System.out.println("对象被初始化了...");
    }

    public void destroyMethod(){
        System.out.println("对象被销毁了...");
    }



    public Fighter getFighter() {
        return fighter;
    }

    public void setFighter(Fighter fighter) {
        this.fighter = fighter;
    }

    public String getName() {
        return name;
    }

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

    public Long getLength() {
        return length;
```

```java
    }

    public void setLength(Long length) {
        this.length = length;
    }

    public Long getWidth() {
        return width;
    }

    public void setWidth(Long width) {
        this.width = width;
    }


    @Override
    public String toString() {
        return "CVN{" +
                "name='" + name + '\'' +
                ", length=" + length +
                ", width=" + width +
                ", fighter=" + fighter +
                '}';
    }
}
```

Java

```xml
  <bean id="cvn" class="com.gxa.bean.CVN">
        <property name="name" value="福建号"/>
        <property name="width" value="90"/>
        <property name="length" value="300"/>
        <property name="fighter" ref="fighter"/>
    </bean>

```

2.3.3 集合注入(了解)

Java

```java
/**
 * Created by zxd on 2022/10/24 15:31
 */
public class CollectionBean {

    private List list;

    private Set set;

    private Map map;

    private Properties properties;


    public List getList() {
        return list;
    }

    public void setList(List list) {
        this.list = list;
    }

    public Set getSet() {
        return set;
    }

    public void setSet(Set set) {
        this.set = set;
    }

    public Map getMap() {
        return map;
    }

    public void setMap(Map map) {
        this.map = map;
    }

    public Properties getProperties() {
        return properties;
    }

    public void setProperties(Properties properties) {
        this.properties = properties;
    }
```

```
46
47      @Override
48      public String toString() {
49          return "CollectionBean{" +
50                  "list=" + list +
51                  ", set=" + set +
52                  ", map=" + map +
53                  ", properties=" + properties +
54                  '}';
55      }
56  }
57
```

配置

Java

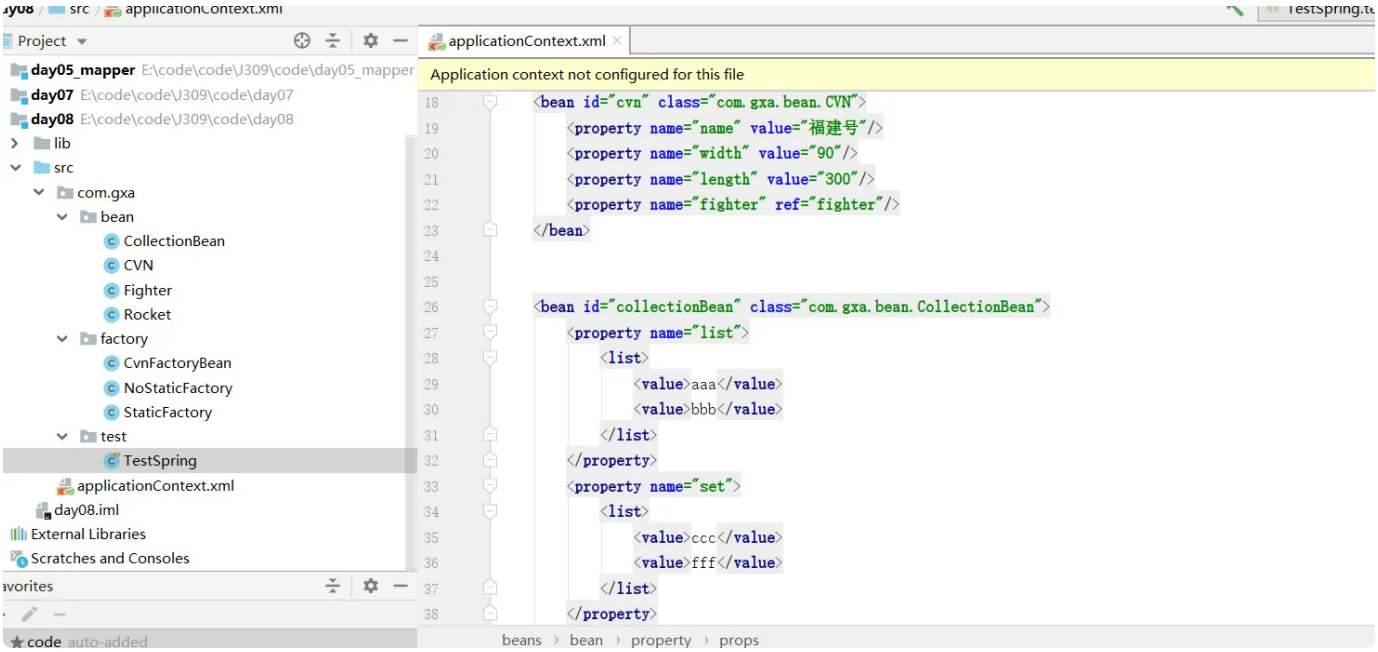
```
1     <bean id="collectionBean" class="com.gxa.bean.CollectionBean">
2           <property name="list">
3               <list>
4                   <value>aaa</value>
5                   <value>bbb</value>
6               </list>
7           </property>
8           <property name="set">
9               <list>
10                  <value>ccc</value>
11                  <value>fff</value>
12              </list>
13          </property>
14          <property name="map">
15              <map>
16                  <entry key="001" value="sss"/>
17              </map>
18          </property>
19          <property name="properties">
20              <props>
21                  <prop key="001">csc</prop>
22              </props>
23          </property>
24      </bean>
```

测试

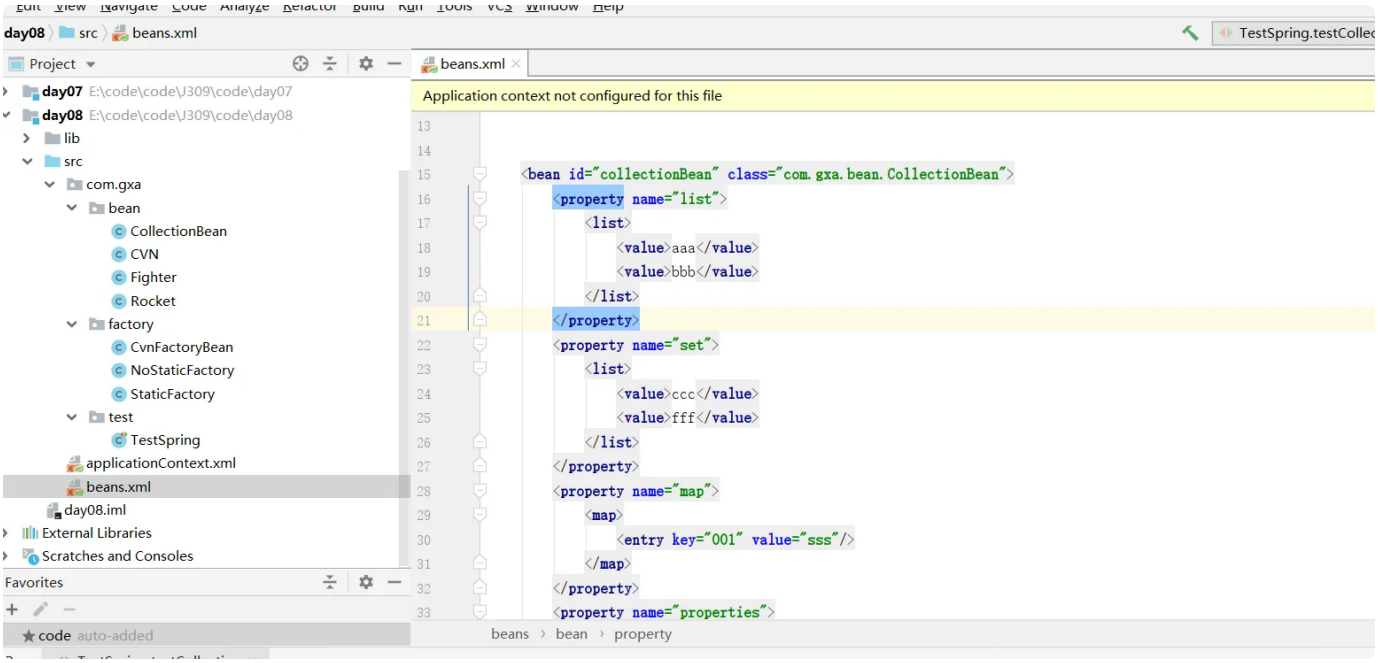
```java
    @Test
    public void testCollection(){

        //1.创建spring容器
        ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");


        //2.从spring容器中获取指定的bean
        CollectionBean collectionBean = (CollectionBean) applicationContext.getBean("collectionBean");

        System.out.println(collectionBean);


    }
```

2.4 配置文件的加载方式

spring配置文件的内容越来越多，分成几个配置文件



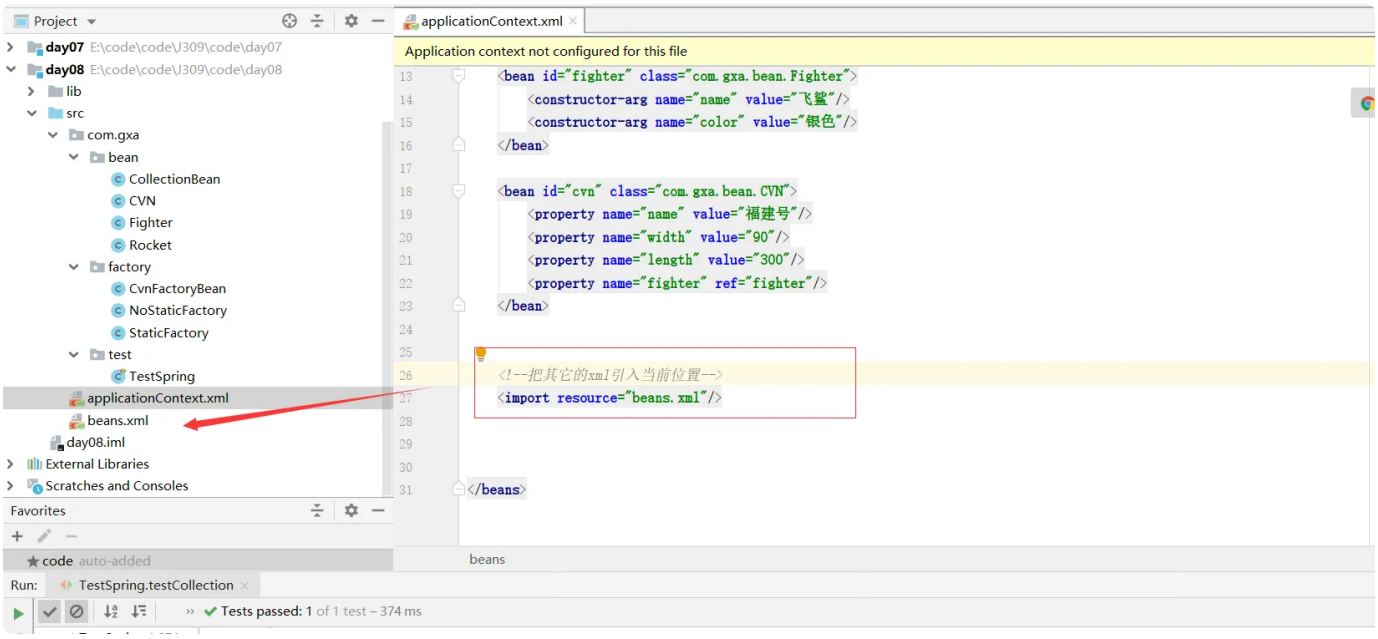
2.4.1 方式一



Java

```java
    @Test
    public void testCollection(){

        //1.创建spring容器
        ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml","beans.xml");


        //2.从spring容器中获取指定的bean
        CollectionBean collectionBean = (CollectionBean) applicationContext.getBean("collectionBean");

        System.out.println(collectionBean);


    }
```

2.4.2 方式二



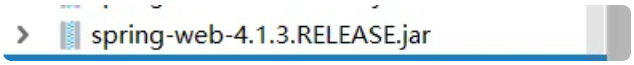
Java

```java
    @Test
    public void testCollection(){

        //1.创建spring容器
        ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");


        //2.从spring容器中获取指定的bean
        CollectionBean collectionBean = (CollectionBean) applicationContext.getBean("collectionBean");

        System.out.println(collectionBean);


    }
```

2.5 spring整合servlet

spring容器交给servletContext对象进行管理，确保整个项目spring容器只有一个

2.5.1 导包



2.5.2 配置spring监听器

webx.xml Java

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">


    <!--配置spring容器的监听器-->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!--通过全局的初始化参数配置spring配置文件的位置-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <!--classpath: 代表走相对路径-->
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>


</web-app>
```

使用

```java
/**
 * Created by zxd on 2022/10/24 15:51
 */
@WebServlet("/helloServlet")
public class HelloServlet extends HttpServlet {


    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

        //获取spring容器
        WebApplicationContext applicationContext = WebApplicationContextUtils.getWebApplicationContext(getServletContext());

        CVN cvn = applicationContext.getBean(CVN.class);

        System.out.println(cvn);


    }
}

```

2.6 spring整合servlet的源码分析

ContextLoaderListener Java

```java
// 监听器 监听ServletContext的创建 创建spring容器，放入ServletContext域中
public class ContextLoaderListener extends ContextLoader implements ServletContextListener {


    //监听ServletContext的创建
    public void contextInitialized(ServletContextEvent event) {
        //创建spring容器，放入ServletContext域中
        this.initWebApplicationContext(event.getServletContext());
    }

    public void contextDestroyed(ServletContextEvent event) {
        this.closeWebApplicationContext(event.getServletContext());
        ContextCleanupListener.cleanupAttributes(event.getServletContext());
    }
}

```

ContextLoader Java

```java


//创建spring容器，放入ServletContext域中
public WebApplicationContext initWebApplicationContext(ServletContext servletContext) {
       //判断spring容器是否已经存在，存在就报错
       if (servletContext.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE) != null) {
            throw new IllegalStateException("Cannot initialize context because there is already a root application context present - check whether you have multiple ContextLoader* definitions in your web.xml!");
        } else {

                //如果不存在spring容器就创建spring容器
                if (this.context == null) {
                    this.context = this.createWebApplicationContext(servletContext);
                }

                if (this.context instanceof ConfigurableWebApplicationContext) {
                    ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext)this.context;
                     //加载配置文件，刷新spring容器进行IOC和DI
                    this.configureAndRefreshWebApplicationContext(cwac, servletContext);
                    }
                }

                 //把spring容器放入ServletContext域中
                servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.context);

        }
    }


////加载配置文件，刷新spring容器进行IOC和DI
 protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac, ServletContext sc) {
         //从ServletContext中获取全局的初始化参数，从而获取spring配置文件的位置
        configLocationParam = sc.getInitParameter("contextConfigLocation");
        if (configLocationParam != null) {
```

```
            //设置配置文件到spring容器中
            wac.setConfigLocation(configLocationParam);
        }
        //刷新spring容器进行IOC和DI
        wac.refresh();
    }
```

2.7 spring懒加载

```Java
    <bean id="cvn" class="com.gxa.bean.CVN" lazy-init="true" init-method="initMethod" destroy-method="destroyMethod">
        <property name="name" value="福建号"/>
        <property name="width" value="90"/>
        <property name="length" value="300"/>
        <property name="fighter" ref="fighter"/>
    </bean>
```

在单例的情况下，默认spring容器初始化的时候创建对象。如果配置了懒加载，在第一次使用的时候创建对象交给spring容器进行管理。

3.注解配置

3.1 IOC注解入门

xml配置

注解替代xml

编写接口

```java

/**
 * Created by zxd on 2022/10/24 16:37
 */
public interface UserService {

    void helloSpring();

}

```

编写实现类

```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
public class UserServiceImpl implements UserService {
    @Override
    public void helloSpring() {
        System.out.println("hello spring!");
    }
}

```

配置

```java
<?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">


     <bean id="userService" class="com.gxa.service.impl.UserServiceImpl"/>




</beans>
```

测试

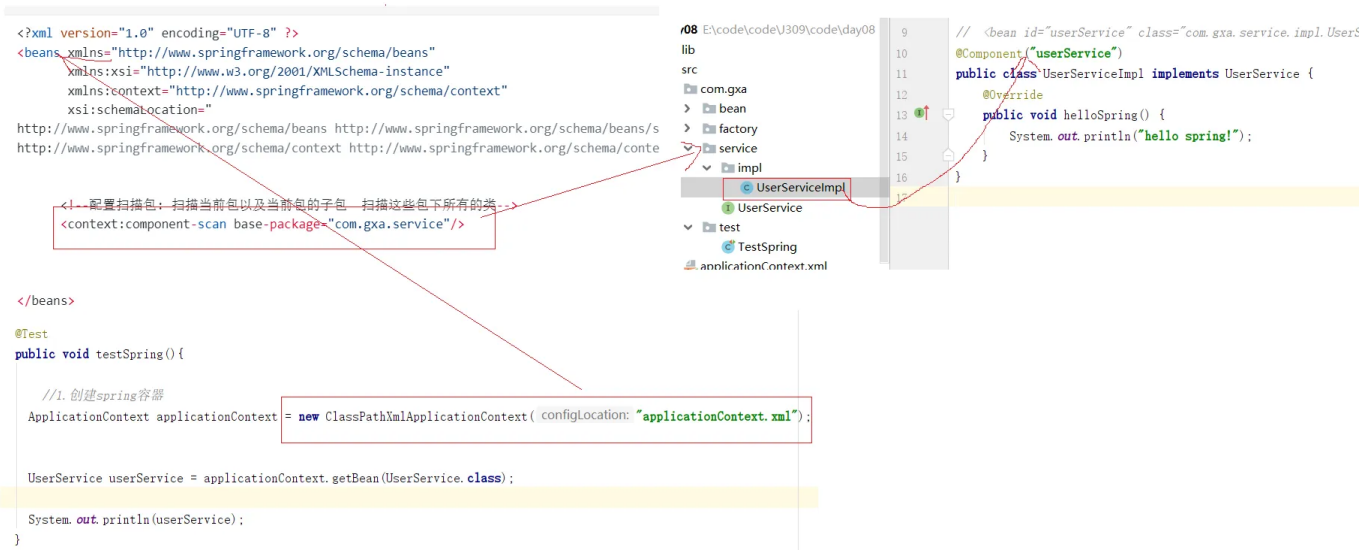
```java

    @Test
    public void testSpring(){

        //1.创建spring容器
      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");


      UserService userService = applicationContext.getBean(UserService.class);

      System.out.println(userService);
    }

```

使用注解配置

```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
//  <bean id="userService" class="com.gxa.service.impl.UserServiceImpl"/>
@Component("userService") //value属性指定bean的id
public class UserServiceImpl implements UserService {
    @Override
    public void helloSpring() {
        System.out.println("hello spring!");
    }
}
```

```java
<?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">


      <!--配置扫描包: 扫描当前包以及当前包的子包  扫描这些包下所有的类-->
      <context:component-scan base-package="com.gxa.service"/>



</beans>
```

```java
    @Test
    public void testSpring(){

        //1.创建spring容器
      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");


      UserService userService = applicationContext.getBean(UserService.class);

      System.out.println(userService);
    }

```



首先加载主配置文件，主配置文件中配置了扫描包，扫描对应的包以及所有的子包，反射加载所有的类，然后看这些类上面是否有Componen注解，如果有就反射创建对象，把对象放入spring容器，名字就用注解配置的名字。

3.2 IOC注解详解

3.2.1 等效注解

@Componen(重点) 替代bean标签的配置

提供了三个等效的注解

@Controller(重点)　web层　controller层

@Service(重点)　　service层

@Repository　dao层

@Component　哪一层都不属于用@Component注解

IOC三个等效注解的底层都是@Component注解

Java

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Service {
    String value() default "";
}

```

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Controller {
    String value() default "";
}

```

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Repository {
    String value() default "";
}

```

通过查看源码我们得知，@Controller、@Service、@Repository这三个注解只是在@Component注解的基础上起了三个新的名字。

对于Spring使用IOC容器管理这些组件来说没有区别。所以@Controller、@Service、@Repository这三个注解只是给开发人员看的，让我们能够便于分辨组件的作用。

3.2.2 默认的名字

默认情况:

类名首字母小写就是bean的id　例如UserServiceImpl对应的id就是userServiceImpl

```java
@Service
public class UserServiceImpl implements UserService {
    @Override
    public void helloSpring() {
        System.out.println("hello spring!");
    }
}
```

```java
    @Test
    public void testSpring(){

        //1.创建spring容器
      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");


      UserService userService = (UserService) applicationContext.getBean("userServiceImpl");

      System.out.println(userService);
    }
```

使用value属性指定名称

3.2.3 其它注解

@Scope("singleton")//prototype多例　singleton单例　配置bean的作用范围

@PostConstruct//配置初始化的方法

@PreDestroy//配置初始化的方法

@Lazy 默认值为true 懒加载

```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
@Service
@Scope("singleton")//prototype多例  singleton单例  配置bean的作用范围
public class UserServiceImpl implements UserService {


    @PostConstruct//配置初始化的方法
    public void initMethod(){
        System.out.println("初始化");
    }

    @PreDestroy//配置初始化的方法
    public void destroyMethod(){
        System.out.println("销毁..");
    }

    @Override
    public void helloSpring() {
        System.out.println("hello spring!");
    }
}
```

3.3 注解DI

3.3.1 注入普通属性

Java

```java
/**
 * Created by zxd on 2022/10/21 15:26
 */
@Component("fighter")
public class Fighter {

    //   <property name="name" value="歼15"/>
    @Value("歼15")
    private String name;

    @Value("银色")
    private String color;

    public Fighter(){

    }

    public Fighter(String name, String color) {
        this.name = name;
        this.color = color;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public String getName() {
        return name;
    }

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

    @Override
    public String toString() {
        return "Fighter{" +
                "name='" + name + '\'' +
                ", color='" + color + '\'' +
                '}';
    }
```

```
}
```

Java

```
<?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">


      <!--配置扫描包: 扫描当前包以及当前包的子包  扫描这些包下所有的类,扫描多个包用逗号隔开 -->
      <context:component-scan base-package="com.gxa.service,com.gxa.bean"/>


</beans>
```

```java
    @Test
    public void testSpring(){

        //1.创建spring容器
        ApplicationContext applicationContext = new ClassPathXmlApplicatio
nContext("applicationContext.xml");


        Fighter fighter = applicationContext.getBean(Fighter.class);

        System.out.println(fighter);

    }
```

3.3.2 注入对象的准备工作

编写dao接口

```java
/**
 * Created by zxd on 2022/10/25 10:36
 */
public interface UserDao {

    void addUser();

}
```

编写实现类

```java
/**
 * Created by zxd on 2022/10/25 10:37
 */
public class UserDaoImpl implements UserDao {
    @Override
    public void addUser() {
        System.out.println("添加用户");
    }
}

```

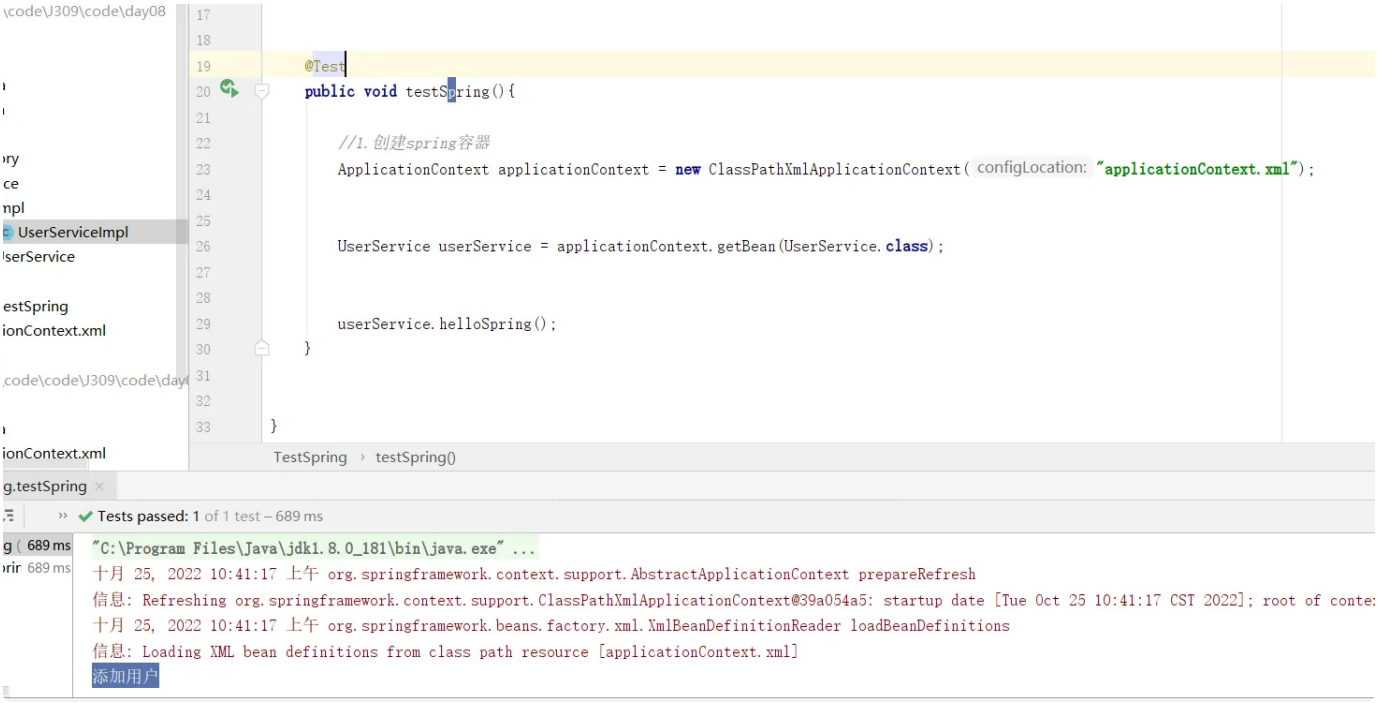
改造service

```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
public class UserServiceImpl implements UserService {

    //set方法注入
    private UserDao userDao;

    @Override
    public void helloSpring() {
        userDao.addUser();
    }

    public UserDao getUserDao() {
        return userDao;
    }

    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
}

```

使用项目xml进行配置

```java
<bean id="userDao" class="com.gxa.dao.impl.UserDaoImpl"/>

<bean id="userService" class="com.gxa.service.impl.UserServiceImpl">
      <property name="userDao" ref="userDao"/>
</bean>
```

测试



3.3.3 使用注解注入对象

改造dao

```java
/**
 * Created by zxd on 2022/10/25 10:37
 */
@Repository("userDao")
public class UserDaoImpl implements UserDao {
    @Override
    public void addUser() {
        System.out.println("添加用户");
    }
}

```

改造service

```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
@Service("userService")
public class UserServiceImpl implements UserService {

    //set方法注入
    //<property name="userDao" ref="userDao"/>
    //暴力反射
    @Resource(name = "userDao")
    private UserDao userDao;

    @Override
    public void helloSpring() {
        userDao.addUser();
    }


}

```

3.3.4 @Resource注解详解

@Resource 默认是按照名称进行注入

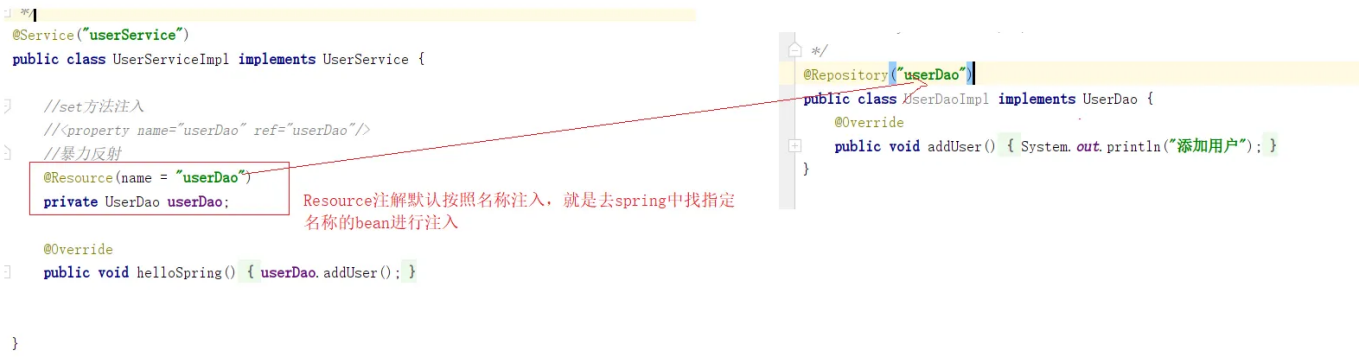
参与自动装配的组件（需要装配别人、被别人装配）全部都必须在IOC容器中。

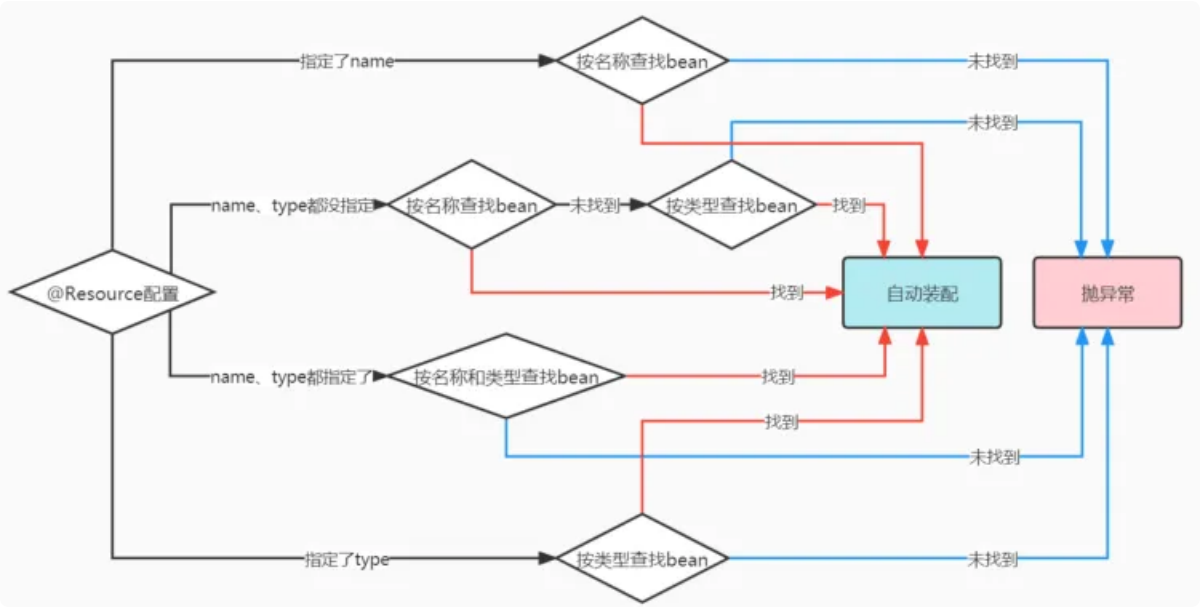
@Resource是JDK自带的注解，用于自动装配。 javax.annotation;

装配方式

@Resource默认按照名称自动注入。

如果名称找不到就会按照类型进行查找





Java

```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
@Service("userService")
public class UserServiceImpl implements UserService {

    //set方法注入
    //<property name="userDao" ref="userDao"/>
    //暴力反射
    @Resource//如果没有配置名称，按照类型查找
    private UserDao userDao;

    @Override
    public void helloSpring() {
        userDao.addUser();
    }


}
```

3.3.5 Autowired

3.3.5.1 和@Qualifier配置完成按照名称注入

@Qualifier的使用

通过@Autowired和@Qualifier的结合使用可以按名称装配。

```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
@Service("userService")
public class UserServiceImpl implements UserService {

    @Qualifier("userDao")
    @Autowired
    private UserDao userDao;

    @Override
    public void helloSpring() {
        userDao.addUser();
    }


}

```

3.3.5.2 按照类型注入

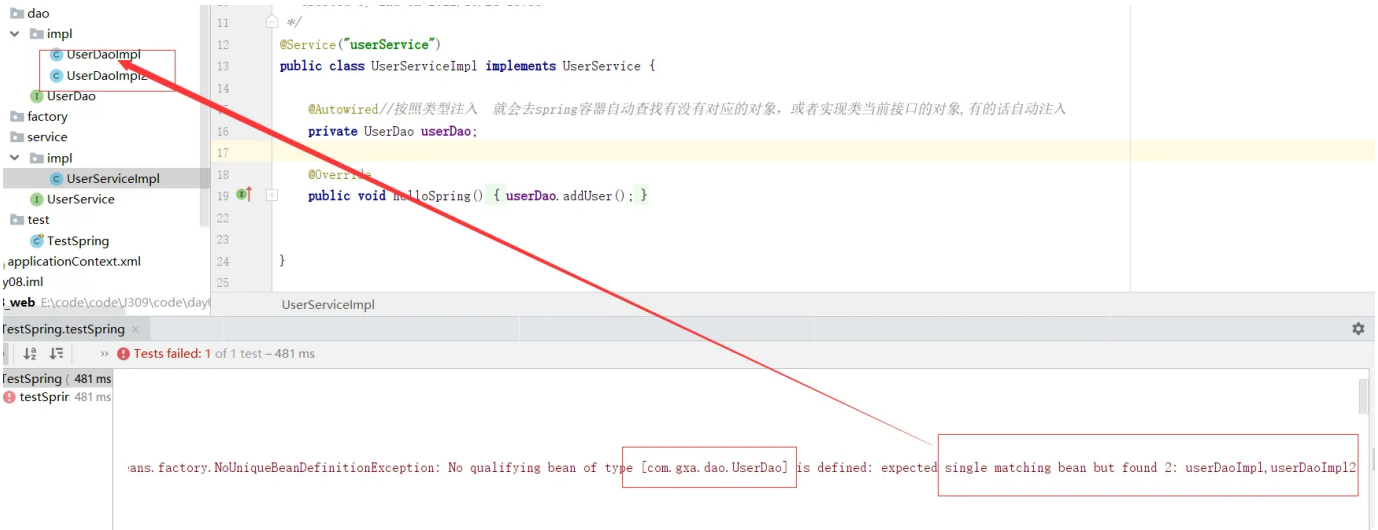
```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
@Service("userService")
public class UserServiceImpl implements UserService {

    @Autowired//按照类型注入  就会去spring容器自动查找有没有对应的对象，或者实现类当前接口的对象,有的话自动注入
    private UserDao userDao;

    @Override
    public void helloSpring() {
        userDao.addUser();
    }


}

```

3.3.6 按照类型注入的问题

新增dao实现类

```java
/**
 * Created by zxd on 2022/10/25 10:37
 */
@Repository
public class UserDaoImpl implements UserDao {
    @Override
    public void addUser() {
        System.out.println("添加用户");
    }
}

```

```java
/**
 * Created by zxd on 2022/10/25 11:22
 */
@Repository
public class UserDaoImpl2 implements UserDao {
    @Override
    public void addUser() {
        System.out.println("添加了");
    }
}
```



UserDao接口有两个实现类,我们现在注入的是接口，按照接口类型进行注入，spring就不知道注入哪一个

3.3.6.1 方案1

不注入接口，注入实现类

3.3.6.2 方案2

按照名称注入

Java

```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
@Service("userService")
public class UserServiceImpl implements UserService {

    @Autowired
    @Qualifier("userDaoImpl")
    private UserDao userDao;

    @Override
    public void helloSpring() {
        userDao.addUser();
    }


}
```

Java

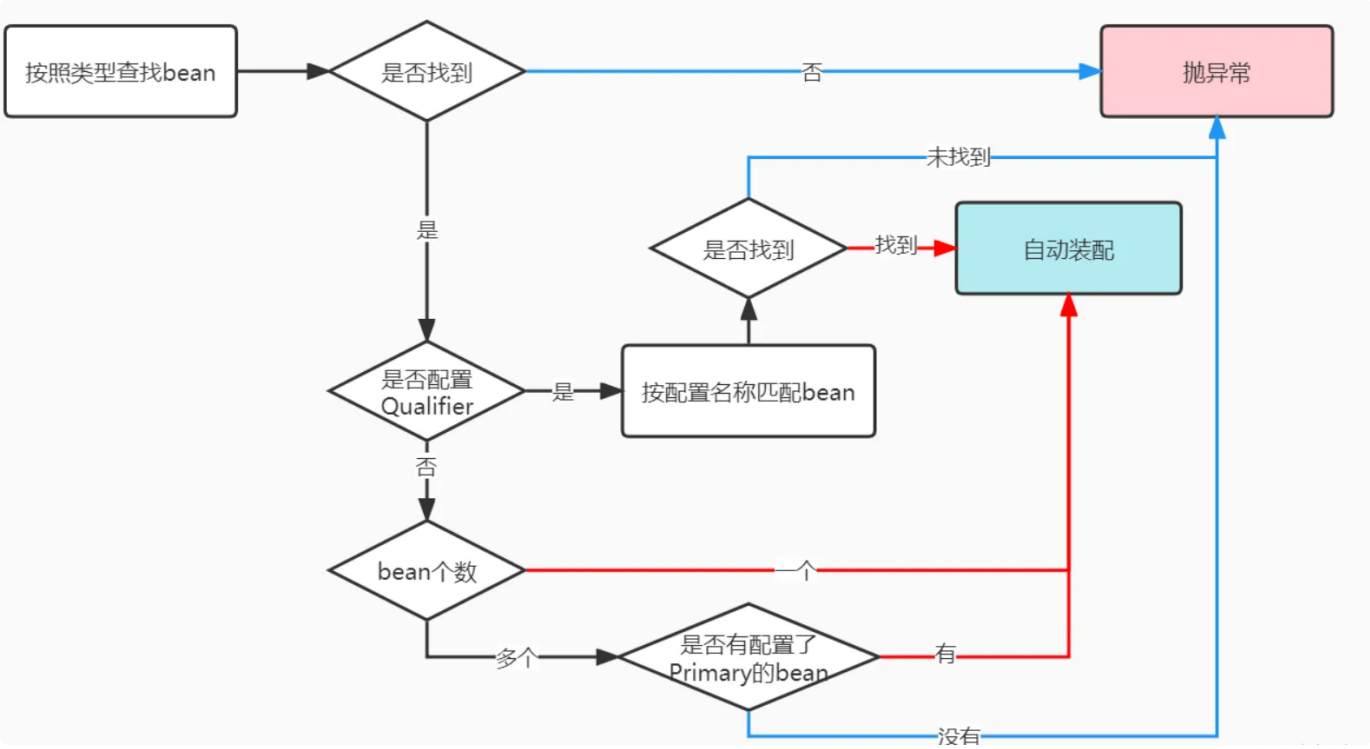
```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
@Service("userService")
public class UserServiceImpl implements UserService {

    @Resource(name = "userDaoImpl")
    private UserDao userDao;

    @Override
    public void helloSpring() {
        userDao.addUser();
    }


}
```

3.3.6.3 方案3

@Primary标记默认注入的对象，如果注入一个接口，这个接口存在对个实现类对象，注入@Primary标记类所创建的对象

```java
/**
 * Created by zxd on 2022/10/25 11:22
 */
@Repository
@Primary
public class UserDaoImpl2 implements UserDao {
    @Override
    public void addUser() {
        System.out.println("添加了2");
    }
}

```

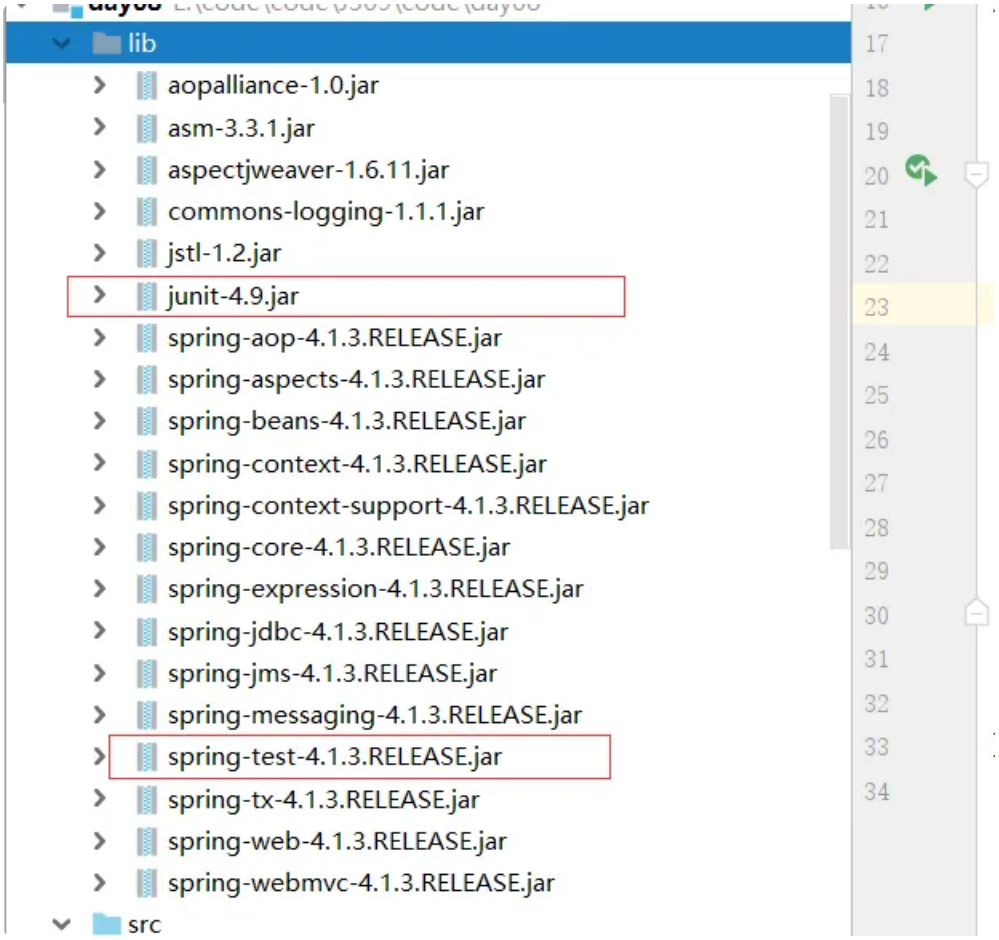
3.3.7 @Autowired流程



3.3.8 Resource和Autowired区别

@Autowired	@Resource
Spring定义的注解	JSR-250定义的注解
默认按类型自动装配	默认按名称自动装配
一个参数：required(默认true)，表示是否必须注入	七个参数：最重要的两个参数是name、type
默认按类型自动装配 如果要按名称自动装配，需要使用@Qualifier一起配合	默认按名称自动装配 如果指定了name，则按名称自动装配；如果指定了type，则按类型自动装配
作用范围：构造器、方法、参数、成员变量、注解	作用范围：类、成员变量、方法

3.4 spring整合junit


lib
aopalliance-1.0.jar
asm-3.3.1.jar
aspectjweaver-1.6.11.jar
commons-logging-1.1.1.jar
jstl-1.2.jar
junit-4.9.jar
spring-aop-4.1.3.RELEASE.jar
spring-aspects-4.1.3.RELEASE.jar
spring-beans-4.1.3.RELEASE.jar
spring-context-4.1.3.RELEASE.jar
spring-context-support-4.1.3.RELEASE.jar
spring-core-4.1.3.RELEASE.jar
spring-expression-4.1.3.RELEASE.jar
spring-jdbc-4.1.3.RELEASE.jar
spring-jms-4.1.3.RELEASE.jar
spring-messaging-4.1.3.RELEASE.jar
spring-test-4.1.3.RELEASE.jar
spring-tx-4.1.3.RELEASE.jar
spring-web-4.1.3.RELEASE.jar
spring-webmvc-4.1.3.RELEASE.jar
src

```java
/**
 * Created by zxd on 2022/10/24 10:47
 */
@RunWith(SpringJUnit4ClassRunner.class)//让spring来运行单元测试
@ContextConfiguration(locations = "classpath:applicationContext.xml")//配置spring配置文件的位置
public class TestSpring {

    @Autowired
    private UserService userService;

    @Test
    public void testSpring(){

        userService.helloSpring();
    }


}

```

3.5 我们到底用注解还是xml

项目中到底是用xml还是注解?

spring 1 2 xml

spring 2.5 3 4 注解 xml都可以用

ssm手动配置

注解挺好用的 自己写的代码用注解，别人写的用xml

@Controller

@Service

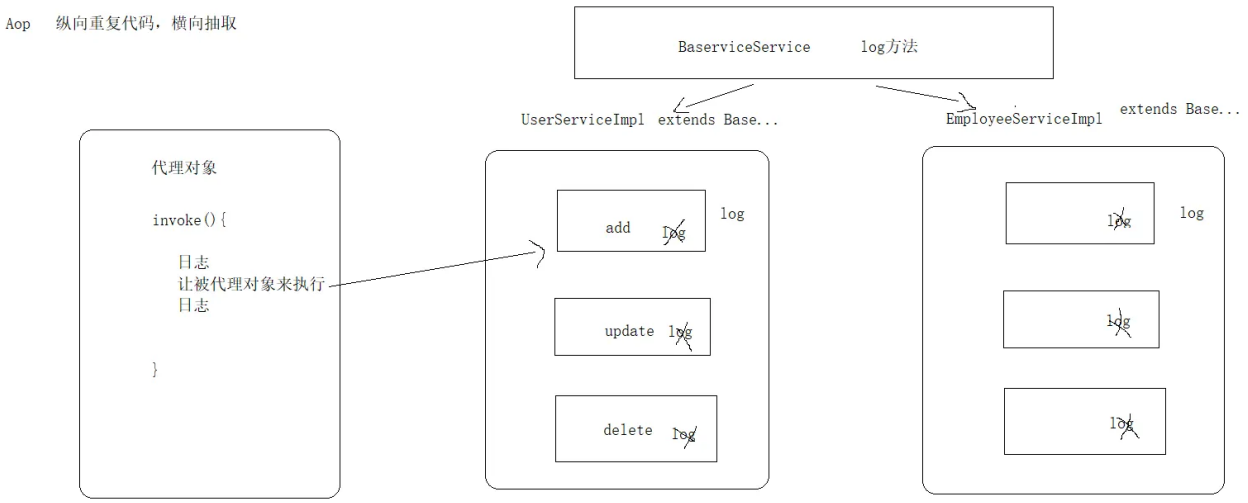
@Autowired

Java配置　别人写的东西都可以用注解来配置 没人用　难度太大了

springboot帮我进行配置

4.动态代理

4.1 aop思想

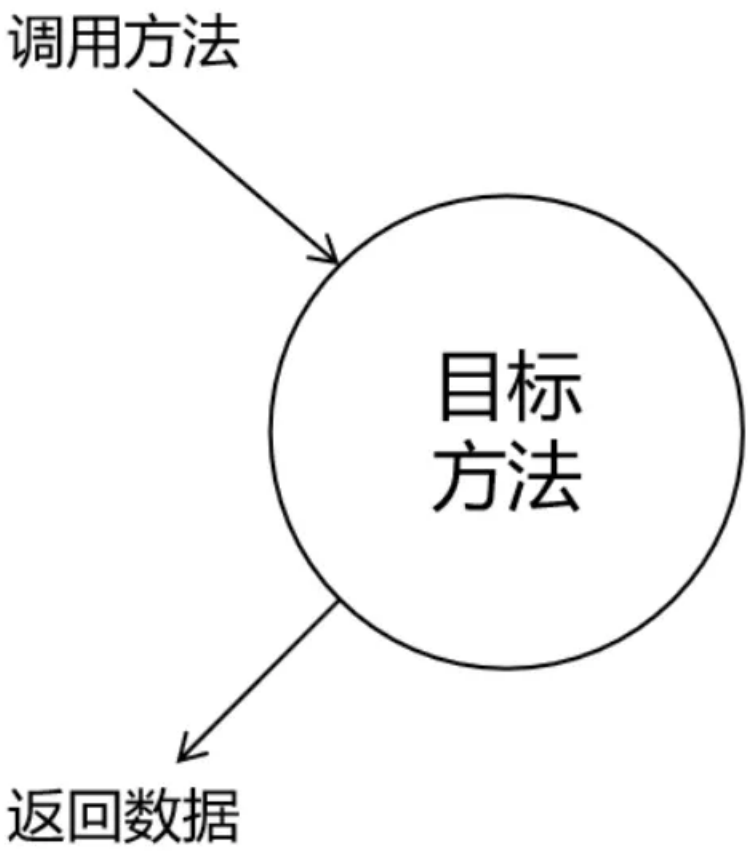


Spring Aop 封装了动态代理。 Jdk动态代理创建代理对象必须要有接口

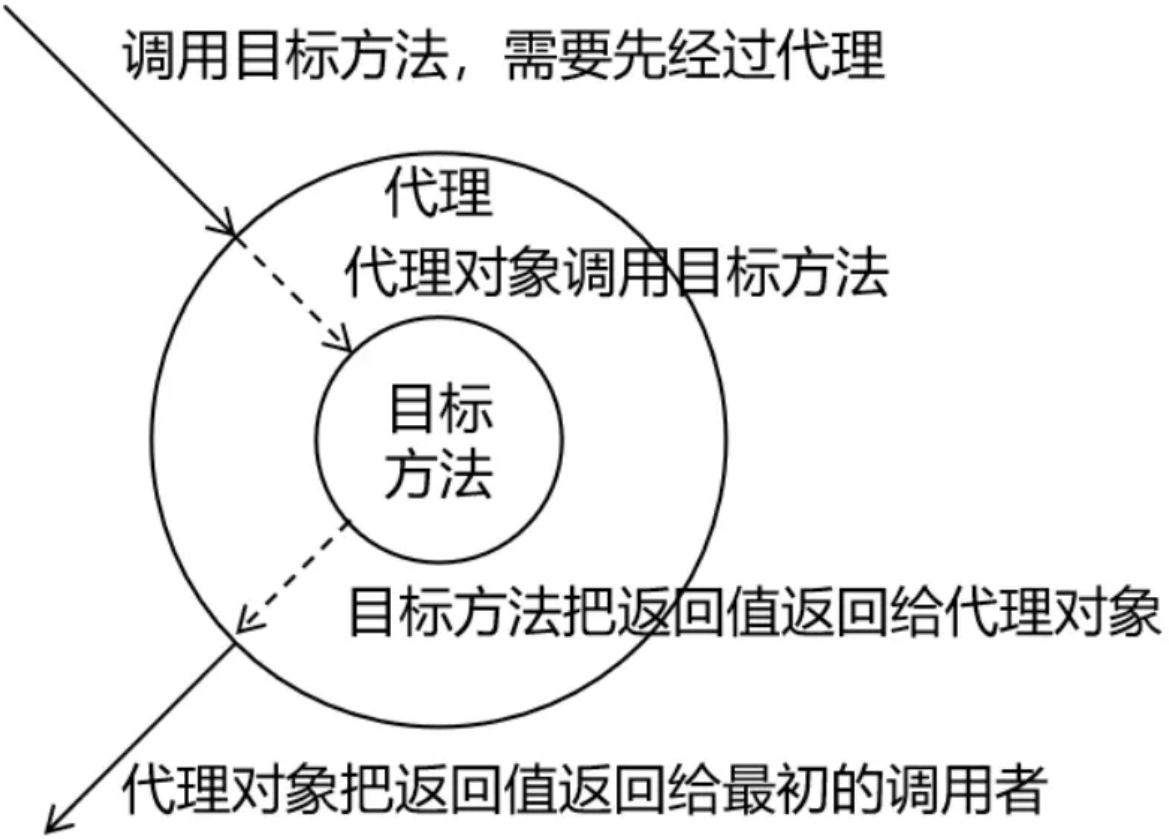
如果没有接口怎么办？ CGLIB动态代理

4.2 动态代理

它的作用就是通过提供一个代理类，让我们在调用目标方法的时候，不再是直接对目标方法进行调用，而是通过代理类**间接**调用。让不属于目标方法核心逻辑的代码从目标方法中剥离出来——**解耦**。调用目标方法时先调用代理对象的方法，减少对目标方法的调用和打扰，同时让附加功能能够集中在一起也有利于统一维护。



使用代理后：



4.3 SpringAop底层

SpringAop底层是使用了动态代理，如果实现了接口就会采用Jdk动态代理，如果没有实现接口就会采用Cglib动态代理。

Jdk动态代理 采用接口来实现动态代理

Cglib动态代理 子类继承的方式来实现动态代理

如果一个类既没有实现接口，并且被final修饰，spring怎么办？ 报错

4.4 Cglib动态代理

Clib是一个高效的，高质量的代码生成类库。 第三方

4.4.1 编写被代理对象

Java

```java
/**
 * Created by zxd on 2022/10/25 14:23
 * 被代理对象
 */
public class Person {


    public String sing(String name) {
        System.out.println("坤坤唱"+name+"歌了...");
        return "飞吻";
    }

    public String dance(String name) {
        System.out.println("坤坤跳"+name+"舞了...");
        return "脱衣服,裸奔...";
    }

}
```

4.4.2 编写拦截的方法

```java
/**
 * Created by zxd on 2022/10/25 14:30
 */
public class MyInterceptor implements MethodInterceptor {

    /***
     * @param proxy 代理对象
     * @param method  调用的方法
     * @param objects  调用的方法传递的参数
     * @param methodProxy  把method封装了一层，功能更强大
     * @return
     * @throws Throwable
     */
    @Override
    public Object intercept(Object proxy, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {

        System.out.println(method.getName()+"方法被调用了...");

        if(method.getName().equals("sing")){

            System.out.println("收300万");

            //invokeSuper让传入对象的父类对象来执行
            //代理对象的父类就是被代理对象
            Object result = methodProxy.invokeSuper(proxy, objects);

            System.out.println("谢谢");

            return result;
        }else if(method.getName().equals("dance")){

            System.out.println("收600万");

            //被代理对象去跳舞
            Object result = methodProxy.invokeSuper(proxy, objects);

            System.out.println("谢谢");

            return result;
        }else {
            System.out.println("无此功能");
            return null;
        }

```

```
45
46          }
47      }
48
```

4.4.3 编写创建代理对象的工厂

Java

```java
/**
 * Created by zxd on 2022/10/25 14:25
 */
public class ProxyFactory {

    //准备被代理对象
    private Person person=new Person();

    //创建代理对象
    public Person createProxy(){

        //1.准备cglib类库
        Enhancer enhancer=new Enhancer();

        //2.设置代理对象的父类
        enhancer.setSuperclass(person.getClass());

        //3.设置回调函数
        enhancer.setCallback(new MyInterceptor());

        //4.创建代理对象
        return (Person) enhancer.create();
    }


}


```

4.4.4 测试

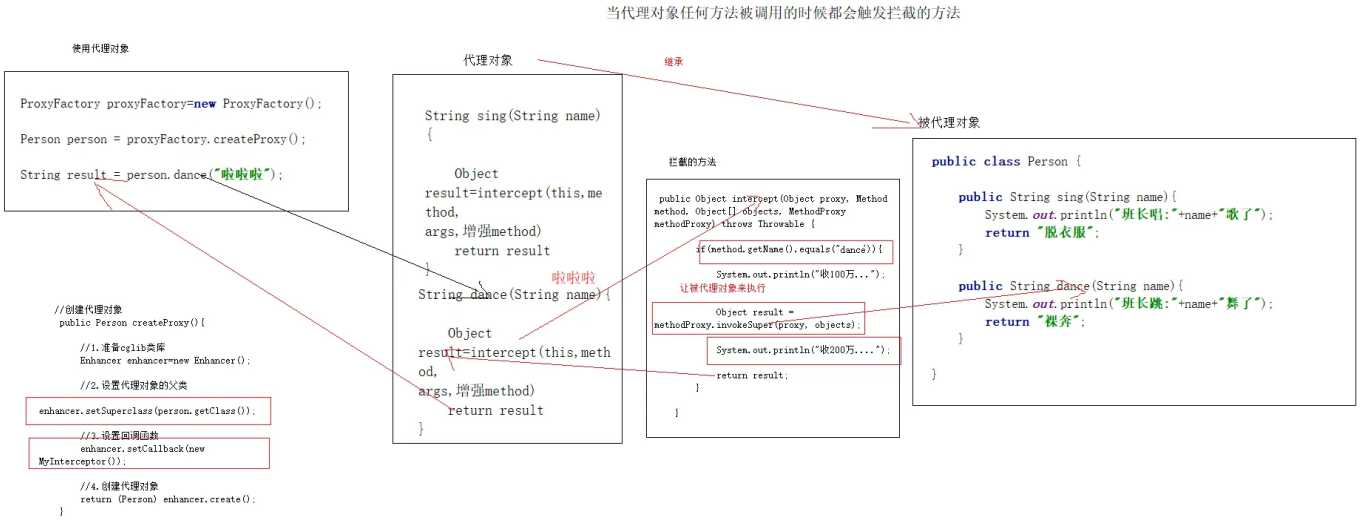
```java
@Test
  public void testProxy(){

      ProxyFactory proxyFactory=new ProxyFactory();
      Person person = proxyFactory.createProxy();

      String result = person.sing("aaa");

      System.out.println(result);

  }

```

4.4.5 简化写法

Java

```java
/**
 * Created by zxd on 2022/10/25 14:25
 */
public class ProxyFactory {

    //准备被代理对象
    private Person person=new Person();

    //创建代理对象
    public Person createProxy(){

        //1.准备cglib类库
        Enhancer enhancer=new Enhancer();

        //2.设置代理对象的父类
        enhancer.setSuperclass(person.getClass());

        //3.设置回调函数
        enhancer.setCallback(new MethodInterceptor() {
            @Override
            public Object intercept(Object proxy, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {

                System.out.println(method.getName()+"方法被调用了...");

                if(method.getName().equals("sing")){

                    System.out.println("收300万");

                    //invokeSuper让传入对象的父类对象来执行
                    //代理对象的父类就是被代理对象
                    //Object result = methodProxy.invokeSuper(proxy, objects);
                    Object result = method.invoke(person, objects);

                    System.out.println("谢谢");

                    return result;
                }else if(method.getName().equals("dance")){

                    System.out.println("收600万");

                    //被代理对象去跳舞
                    Object result = methodProxy.invokeSuper(proxy, objects);
```

```
43
44                      System.out.println("谢谢");
45
46
47                      return result;
48                  }else {
49                      System.out.println("无此功能");
50                      return null;
51                  }
52              }
53          });
54
55          //4.创建代理对象
56          return (Person) enhancer.create();
57      }
58
59  }
```



5.Aop

5.1 Aop介绍

5.1.1 AOP底层实现

AOP：Aspect Oriented Programming面向切面编程

- 动态代理（InvocationHandler）：JDK原生的实现方式，需要被代理的目标类必须实现接口。因为这个技术要求**代理对象和目标对象实现同样的接口**（兄弟两个拜把子模式）。
- cglib：通过**继承被代理的目标类**（认干爹模式）实现代理，所以不需要目标类实现接口。

5.1.2 AOP作用

- 简化代码：把方法中固定位置的重复的代码**抽取**出来，让被抽取的方法更专注于自己的核心功能，提高内聚性。
- 代码增强：把特定的功能封装到切面类中，看哪里有需要，就往上套，被**套用**了切面逻辑的方法就被切面给增强了。

5.1.3 Aop专业术语

AOP 相关概念

Spring 的 AOP 实现底层就是对上面的动态代理的代码进行了封装，封装后我们只需要对需要关注的部分进

行代码编写，并通过配置的方式完成指定目标的方法增强。

– Target（目标对象）：代理的目标对象

– Proxy （代理）： 一个类被 AOP 织入增强后，就产生一个结果代理类

– Joinpoint（连接点）： 所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法，因为

spring只支持方法类型的连接点

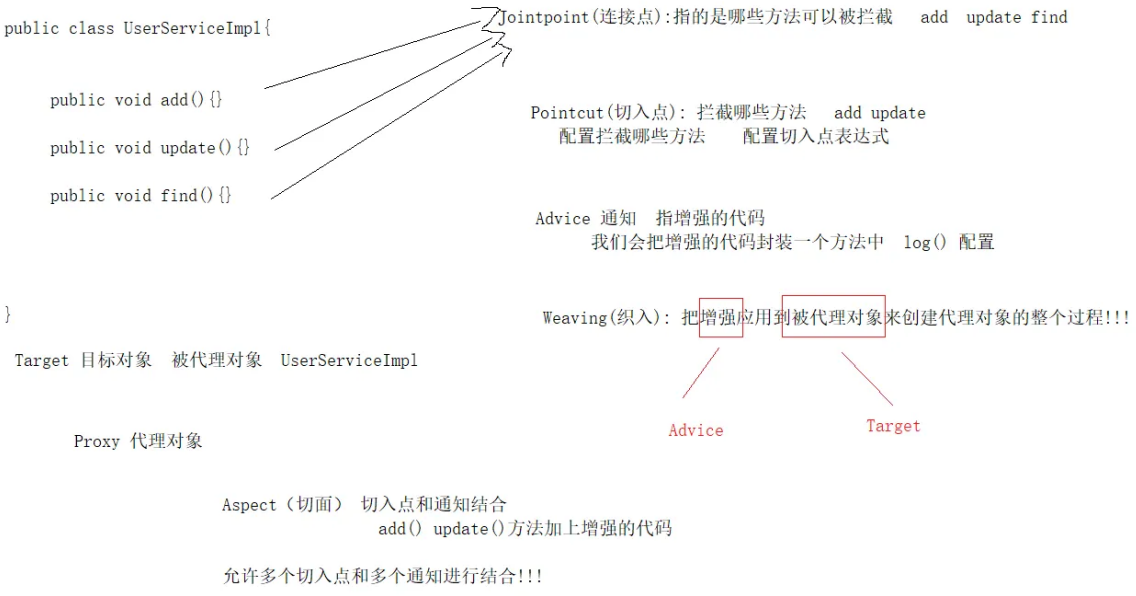
– Pointcut（切入点）： 所谓切入点是指我们要对哪些 Joinpoint 进行拦截的定义

– 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

– Aspect（切面）： 是切入点和通知（引介）的结合

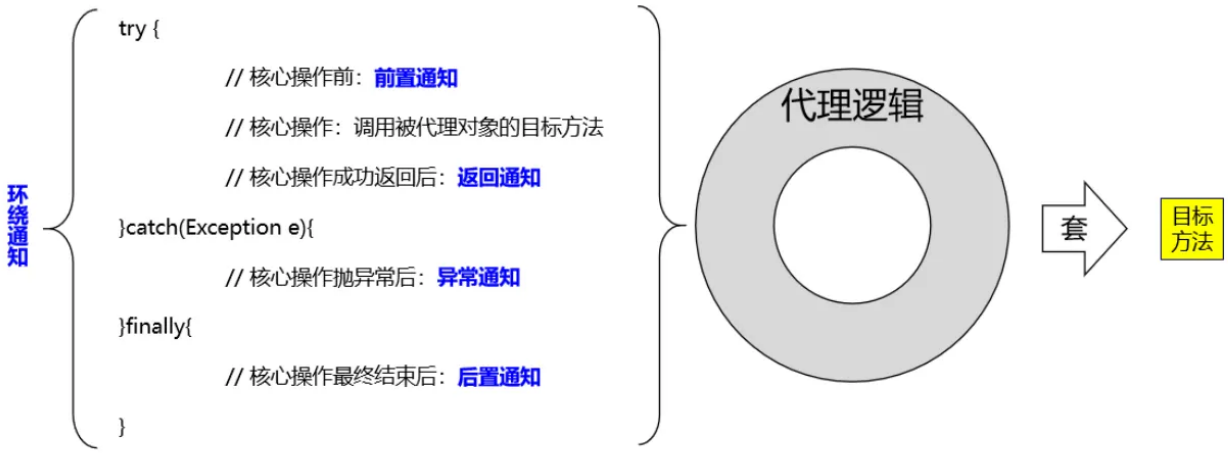
– Weaving（织入）： 是指把增强应用到目标对象来创建新的代理对象的过程。spring采用动态代理织入，

而AspectJ采用编译期织入和类装载期织入



事务 aop管理事务，aop实现注解权限 aop实现缓存的封装 redis aop实现读写分离....

5.1.4 Aop套路



Java

```java
try {
    //前置通知
    System.out.println("收300万");

    //在被代理对象之前和之后执行 环绕通知

    //invokeSuper让传入对象的父类对象来执行
    //代理对象的父类就是被代理对象
    Object result = methodProxy.invokeSuper(proxy, objects);

    //后置通知(出现异常不会调用)
    //返回通知
    System.out.println("谢谢");

    return result;
}catch (Exception e){
    //异常通知
    e.printStackTrace();
}finally {
    //后置通知(最终通知)  无论是否出现异常都会调用
}
```

5.2 aop入门

5.2.1 编写被代理对象

```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDao userDao;

    @Override
    public void helloSpring() {
        userDao.addUser();
    }


}

```

5.2.2 编写通知

前置通知　在目标方法执行之前执行

```java
/**
 * Created by zxd on 2022/10/25 15:56
 */
public class MyAdvice {


    //前置通知　在目标方法执行之前执行
    public void beforeMethod(){
        System.out.println("这里是前置通知....");
    }

}

```

5.2.3 配置

约束:

```Java
<?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">


</beans>
```

applicationContext-aop.xml Java

```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">


    <!--1.被代理对象  用注解进行了配置-->

    <!--2.配置通知-->
    <bean id="myAdvice" class="com.gxa.advice.MyAdvice"/>

    <!--配置织入-->
    <aop:config>

        <!--配置切入点表达式-->
        <aop:pointcut id="pc" expression="execution(void com.gxa.service.impl.UserServiceImpl.helloSpring())"/>

        <!--配置切面(切入点+通知)-->
        <aop:aspect ref="myAdvice">

            <!--配置切面:前置通知-->
            <aop:before method="beforeMethod" pointcut-ref="pc"/>

        </aop:aspect>

    </aop:config>


</beans>
```

applicationContext.xml Java

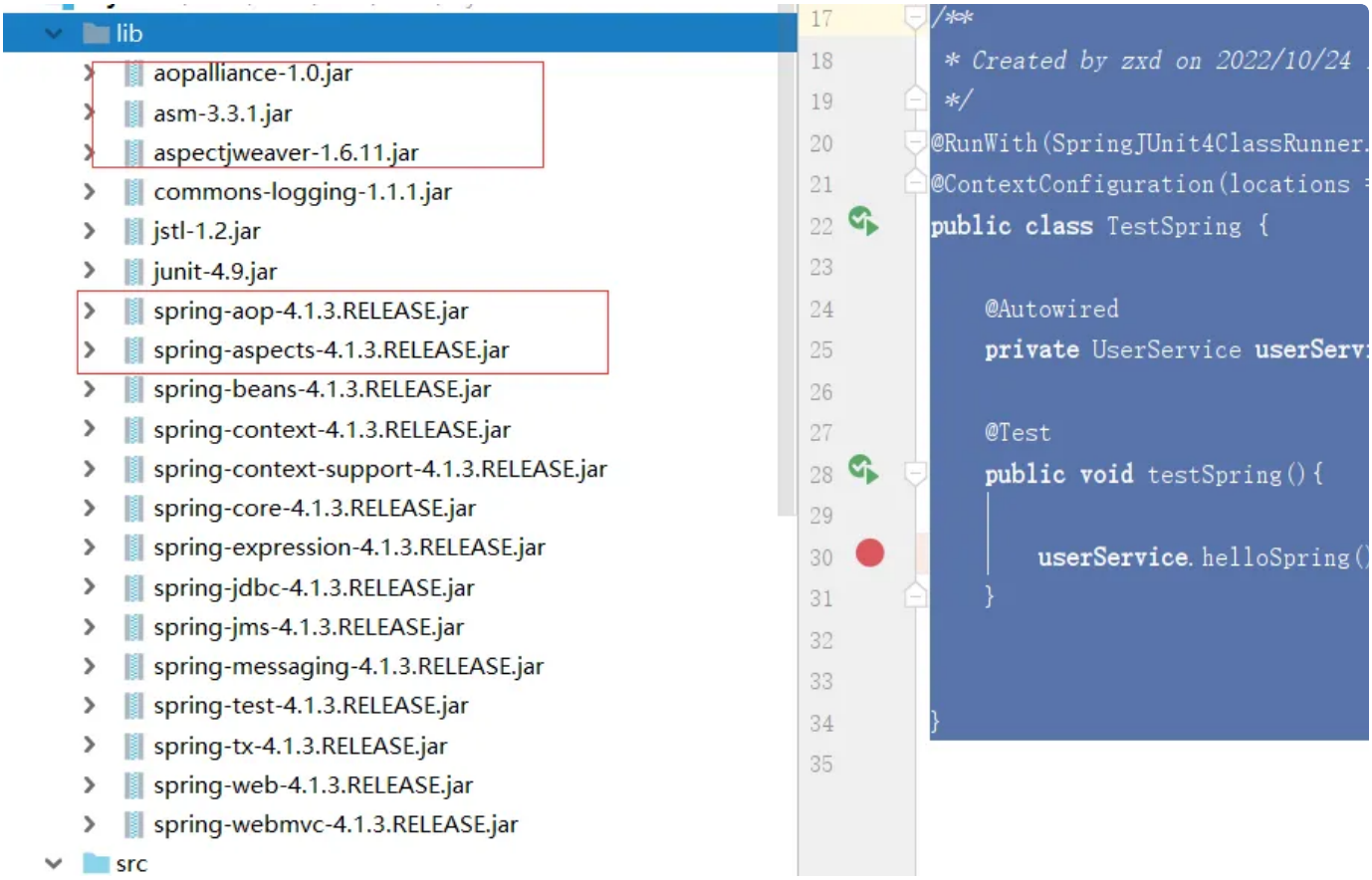
```xml
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans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


   <!--配置扫描包: 扫描当前包以及当前包的子包  扫描这些包下所有的类,扫描多个包用逗号隔开 -->
     <context:component-scan base-package="com.gxa.service,com.gxa.dao,com.gxa.bean"/>


    <import resource="applicationContext-aop.xml"/>


</beans>
```

测试

```java
/**
 * Created by zxd on 2022/10/24 10:47
 */
@RunWith(SpringJUnit4ClassRunner.class)//让spring来运行单元测试
@ContextConfiguration(locations = "classpath:applicationContext.xml")//配置spring配置文件的位置
public class TestSpring {

    @Autowired
    private UserService userService;

    @Test
    public void testSpring(){

        userService.helloSpring();
    }


}

```

5.3 AOP深入

5.3.1 jar包



5.3.2 编写被代理对象

编写pojo

Java

```java
/**
 * Created by zxd on 2022/10/25 16:24
 */
public class Employee implements Serializable {

    private Long id;
    private String name;
    private String address;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "Employee{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}

```

编写接口

```java
/**
 * Created by zxd on 2022/10/25 16:25
 */
public interface EmployeeService {


    Employee findEmployeeById(Long id);

    List<Employee> findEmployeeAll();

    void insertEmployee(Employee employee);

    void updateEmployee(Employee employee);

    void deleteEmployee(Long id);


}
```

```java
/**
 * Created by zxd on 2022/10/25 16:28
 */
@Service
public class EmployeeServiceImpl implements EmployeeService {
    @Override
    public Employee findEmployeeById(Long id) {
        System.out.println("根据id查找");
        return null;
    }

    @Override
    public List<Employee> findEmployeeAll() {
        System.out.println("查找所有");
        return null;
    }

    @Override
    public void insertEmployee(Employee employee) {
        System.out.println("插入");
}

    @Override
    public void updateEmployee(Employee employee) {
        System.out.println("修改");
    }

    @Override
    public void deleteEmployee(Long id) {
        System.out.println("删除");
    }
}

```

配置扫描包

applicationContext.xml Java

```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: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">


   <!--配置扫描包: 扫描当前包以及当前包的子包  扫描这些包下所有的类,扫描多个包用逗号隔开 -->
     <context:component-scan base-package="com.gxa.service,com.gxa.dao,com.gxa.bean"/>


    <import resource="applicationContext-aop.xml"/>


</beans>
```

5.3.3 编写通知

Java

```java
/**
 * Created by zxd on 2022/10/25 15:56
 */
public class MyAdvice {


    //前置通知    在目标方法执行之前执行
    public void beforeMethod(){
        System.out.println("这里是前置通知....");
    }

    //后置通知 最终通知(在目标方法执行之后执行，无论是否出现异常都会执行)
    public void afterMethod(){
        System.out.println("这里是后置通知   无论是否出现异常都会执行 ");
    }

    //返回通知 后置通知（出现异常不会调用）
    public void returningMethod(){
        System.out.println("返回通知 后置通知（出现异常不会调用）");
    }

    //环绕通知
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {

        System.out.println("这里是环绕通知，目标方法执行之前的部分...");

        //放行   执行目标方法
        Object result = joinPoint.proceed();

        System.out.println("这里是环绕通知，目标方法执行之后的部分...");

        return result;
    }

    //异常通知
    public void exceptionMethod(){
        System.out.println("这里是异常通知...");
    }

}

```

5.3.4 配置aop

Java

```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">


    <!--1.被代理对象  用注解进行了配置-->

    <!--2.配置通知-->
    <bean id="myAdvice" class="com.gxa.advice.MyAdvice"/>

    <!--3.配置织入 通知和被代理对象结合起来创建代理对象的整个过程-->
    <aop:config>

        <!--配置切入点表达式 拦截方法-->
        <!-- void com.gxa.service.impl.UserServiceImpl.helloSpring() -->
        <!-- * com.gxa.service.impl.UserServiceImpl.helloSpring() -->
        <!-- ..代表当前包以及当前包的子包 -->
        <!-- * com.gxa.service..UserServiceImpl.helloSpring() -->
        <!-- * com.gxa.service..*ServiceImpl.helloSpring() -->
        <!-- * com.gxa.service..*ServiceImpl.*() -->
        <!-- * com.gxa.service..*ServiceImpl.*(..) -->
        <aop:pointcut id="pc" expression="execution(* com.gxa.service..*ServiceImpl.*(..))"/>

        <!--配置切面(切入点+通知)-->
        <aop:aspect ref="myAdvice">

            <!--配置切面:前置通知-->
            <aop:before method="beforeMethod" pointcut-ref="pc"/>

            <!--返回通知 出现异常不会调用-->
            <aop:after-returning method="returningMethod" pointcut-ref="pc"/>

            <!--后置通知(最终通知) -->
            <aop:after method="afterMethod" pointcut-ref="pc"/>

            <!--环绕通知-->
            <aop:around method="around" pointcut-ref="pc"/>
```

```
            <!--异常通知-->
            <aop:after-throwing method="exceptionMethod" pointcut-ref="pc" />

        </aop:aspect>

    </aop:config>


</beans>
```

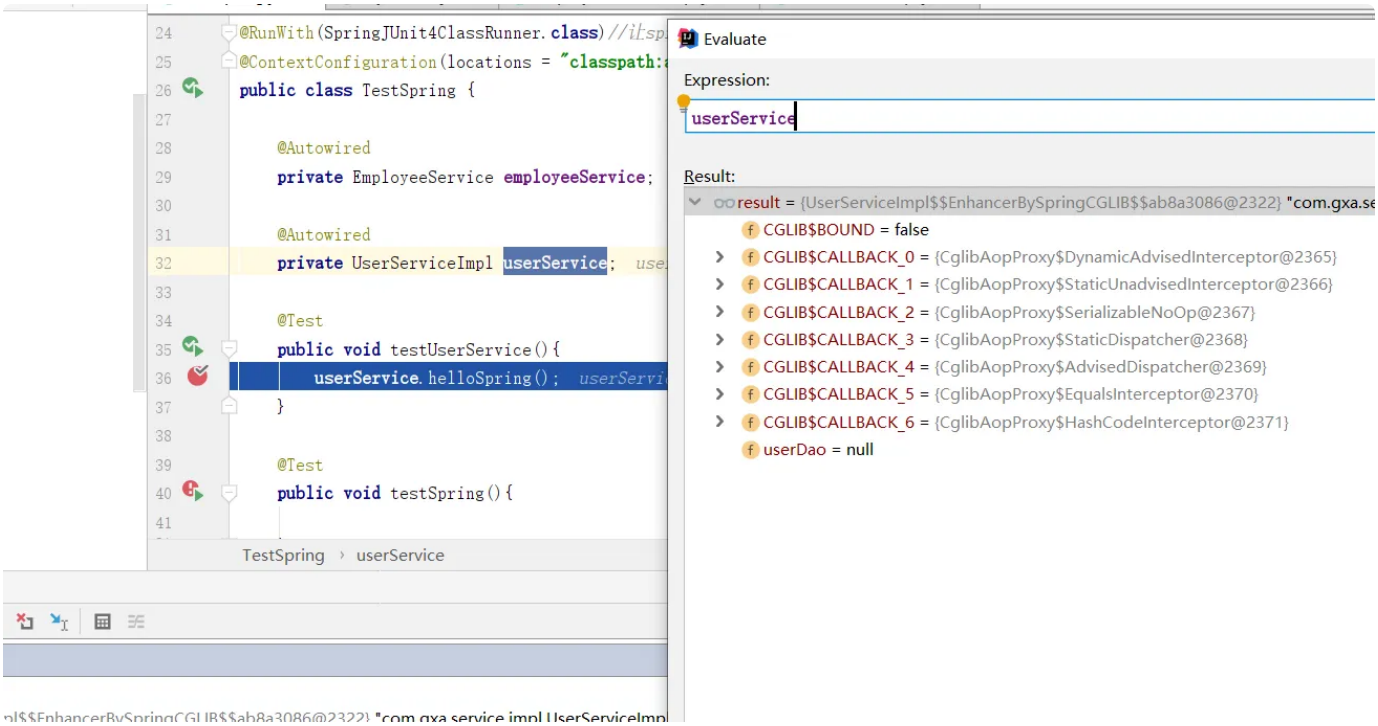
5.3.5 测试

Java

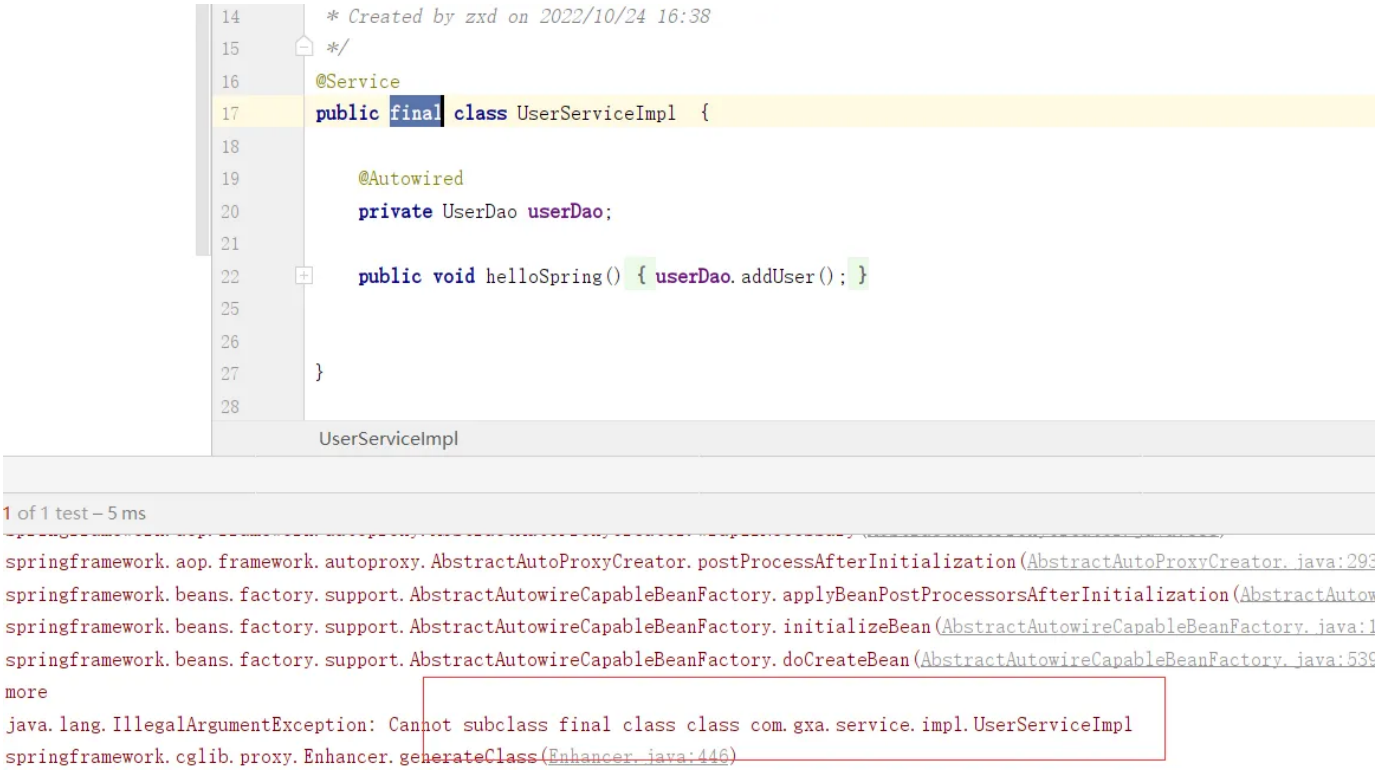
```java
/**
 * Created by zxd on 2022/10/24 10:47
 */
@RunWith(SpringJUnit4ClassRunner.class)//让spring来运行单元测试
@ContextConfiguration(locations = "classpath:applicationContext.xml")//配置spring配置文件的位置
public class TestSpring {

    @Autowired
    private EmployeeService employeeService;

    @Test
    public void testSpring(){

        List<Employee> employeeAll = employeeService.findEmployeeAll();
    }


}

```

5.4 测试没有接口的情况

```java
/**
 * Created by zxd on 2022/10/24 16:38
 */
@Service
public class UserServiceImpl  {

    @Autowired
    private UserDao userDao;

    public void helloSpring() {
        userDao.addUser();
    }


}
```



被final修饰 只能报错


14 * Created by zxd on 2022/10/24 16:38
15 */
16 @Service
17 public final class UserServiceImpl {
18
19 @Autowired
20 private UserDao userDao;
21
22 public void helloSpring() { userDao.addUser(); }
25
26
27 }
28
UserServiceImpl
1 of 1 test – 5 ms
springframework.aop.framework.autoproxy.AbstractAutoProxyCreator.postProcessAfterInitialization(AbstractAutoProxyCreator.java:29
springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.applyBeanPostProcessorsAfterInitialization(AbstractAutow
springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.initializeBean(AbstractAutowireCapableBeanFactory.java:1
springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanFactory.java:539
more
java.lang.IllegalArgumentException: Cannot subclass final class class com.gxa.service.impl.UserServiceImpl
springframework.cglib.proxy.Enhancer.generateClass(Enhancer.java:446)

5.5 使用注解aop

Java

```
<?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">


    <!--开启aop注解配置-->
    <aop:aspectj-autoproxy/>


</beans>
```

applicationContext.xml

Java

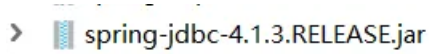
```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans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


   <!--配置扫描包: 扫描当前包以及当前包的子包  扫描这些包下所有的类,扫描多个包用逗号隔开 -->
     <context:component-scan base-package="com.gxa.service,com.gxa.dao,com.gxa.bean,com.gxa.advice"/>


    <import resource="applicationContext-aop.xml"/>


</beans>
```

```java
/**
 * Created by zxd on 2022/10/25 15:56
 */
@Component
@Aspect//代表当前类要配置切面(切入点+通知)
public class MyAdvice {


    //前置通知   在目标方法执行之前执行
    @Before("execution(* com.gxa.service..*ServiceImpl.*(..))")
    public void beforeMethod(){
        System.out.println("这里是前置通知....");
    }

    //...

}
```

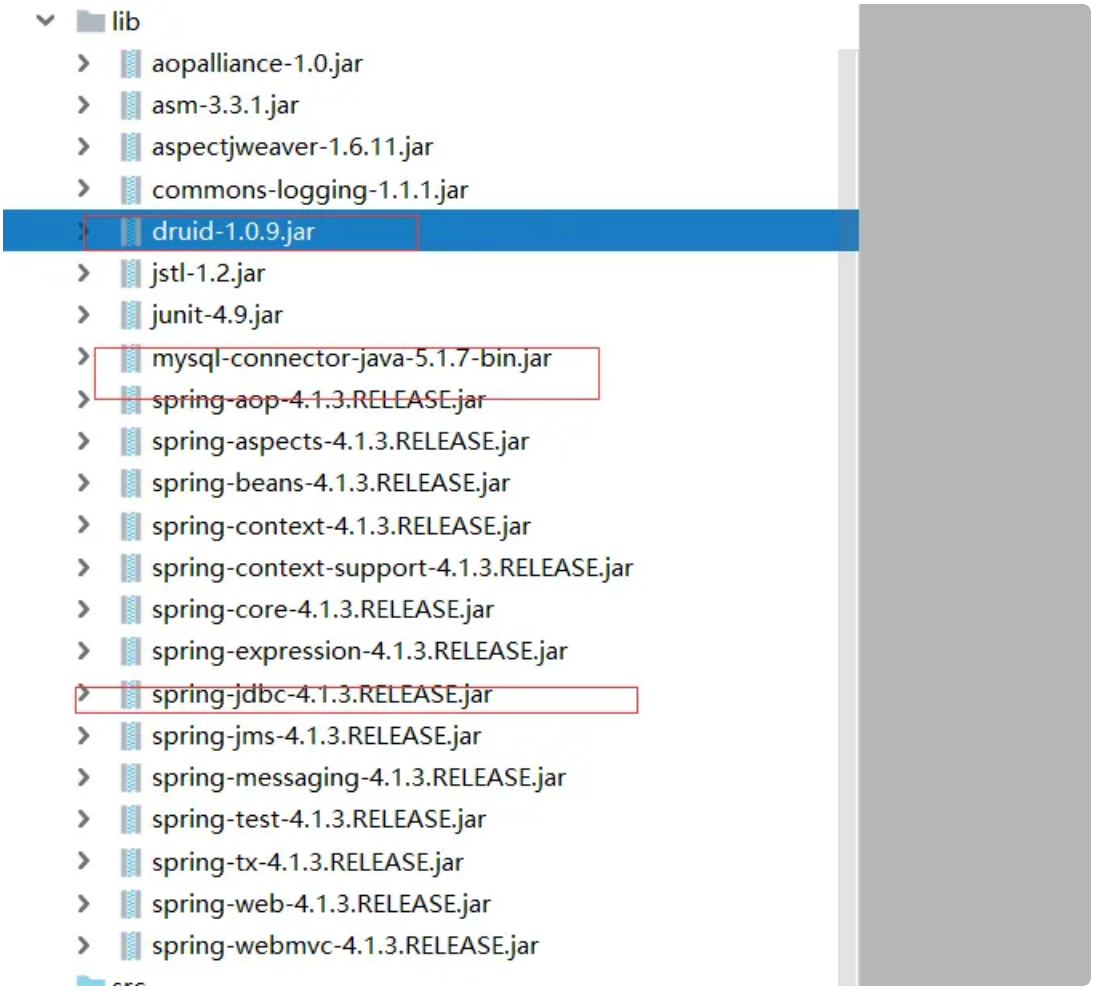
6.SpringJdbc

6.1 SpringJdbc入门

SpringJdbc针对jdbc的简单封装



导入数据库驱动和连接池



对象 gxa_employee @jdbc (localhost) - ...

保存 添加字段 插入字段 删除字段 主键 上移 下移

字段 索引 外键 触发器 选项 注释 SQL 预览

名	类型	长度	小数点	不是 null	键	注释
id	bigint	20		☑	1	
name	varchar	255		☐		
address	varchar	255		☐		

Java

```java
/**
 * Created by zxd on 2022/10/25 16:24
 */
public class Employee implements Serializable {

    private Long id;
    private String name;
    private String address;

    public Employee() {

    }

    public Employee(Long id, String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    @Override
    public String toString() {
```

```
46          return "Employee{" +
47                  "id=" + id +
48                  ", name='" + name + '\'' +
49                  ", address='" + address + '\'' +
50                  '}';
51      }
52  }
53
```

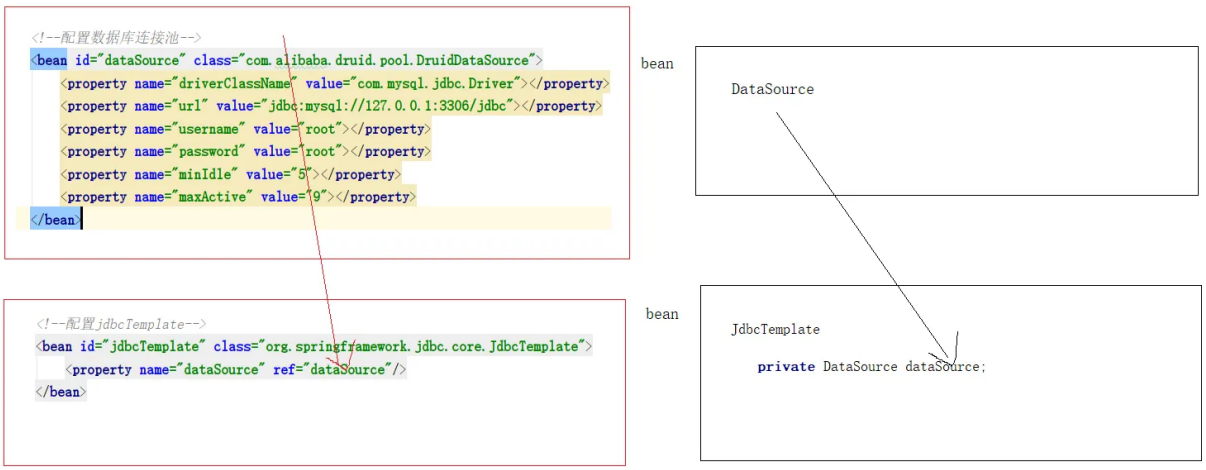
编写入门程序

```java
/**
 * Created by zxd on 2022/10/26 9:48
 */
public class TestSpringJdbc {


    @Test
    public void helloSpringJdbc(){


        //1.创建连接池
        DruidDataSource druidDataSource = new DruidDataSource();

        druidDataSource.setDriverClassName("com.mysql.jdbc.Driver");
        druidDataSource.setUrl("jdbc:mysql://127.0.0.1:3306/jdbc");
        druidDataSource.setUsername("root");
        druidDataSource.setPassword("root");
        druidDataSource.setInitialSize(5);
        druidDataSource.setMaxActive(9);


        //2.创建JdbcTemplate对象
        JdbcTemplate  jdbcTemplate=new JdbcTemplate();
        jdbcTemplate.setDataSource(druidDataSource);

        //3.执行sql
        jdbcTemplate.update("insert into gxa_employee(name,address) values
('龙龙','成都')");

    }

}

```

6.2 Spring整合数据库连接池和springjdbc

applicationContext-dao.xml Java

```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: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">


    <!--配置数据库连接池-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/jdbc"></property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
        <property name="minIdle" value="5"></property>
        <property name="maxActive" value="9"></property>
    </bean>

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


</beans>
```



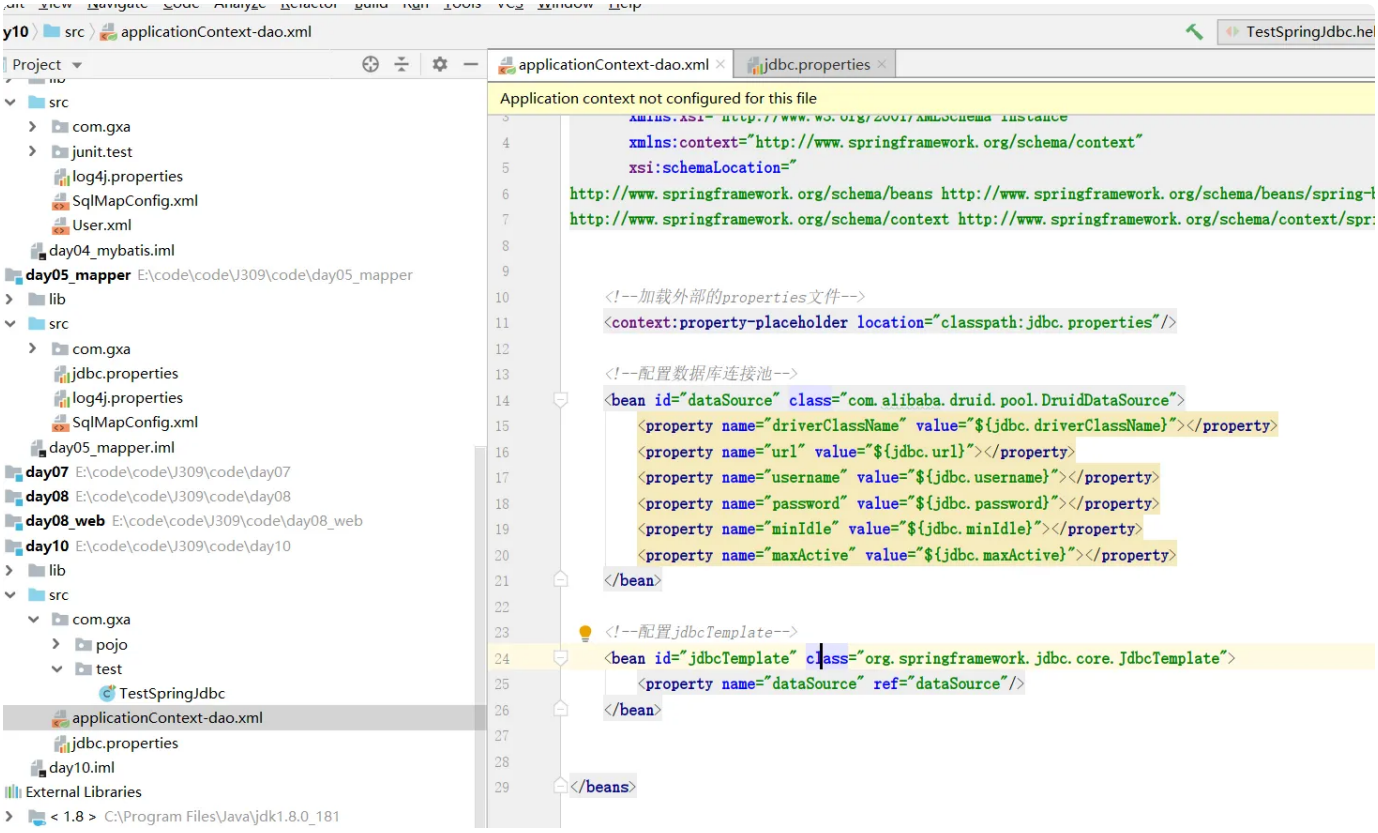
6.3 加载外部的properties文件

jdbc.properties Java

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://127.0.0.1:3306/jdbc
jdbc.username=root
jdbc.password=root
jdbc.minIdle=5
jdbc.maxActive=9
```

applicationContext-dao.xml Java

```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: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">


    <!--加载外部的properties文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!--配置数据库连接池-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${jdbc.driverClassName}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
        <property name="minIdle" value="${jdbc.minIdle}"></property>
        <property name="maxActive" value="${jdbc.maxActive}"></property>
    </bean>

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


</beans>
```



6.4 编写dao

编写dao接口

```java
/**
 * Created by zxd on 2022/10/26 10:24
 */
public interface EmployeeDao {

    Employee findEmployeeById(Long id);

    List<Employee> findEmployeeAll();

    void insertEmployee(Employee employee);

    void updateEmployee(Employee employee);

    void deleteEmployee(Long id);


}

```

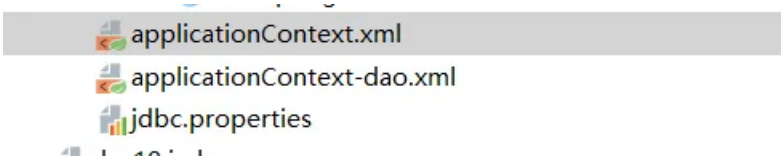
编写dao实现

Java

```java
/**
 * Created by zxd on 2022/10/26 10:25
 */
@Repository
public class EmployeeDaoImpl implements EmployeeDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public Employee findEmployeeById(Long id) {
        //帮我们把ResultSet的结果取出映射到指定的pojo中 前提是数据库的列名和pojo属性一致
        BeanPropertyRowMapper<Employee> beanPropertyRowMapper=new BeanPropertyRowMapper<>(Employee.class);
        return jdbcTemplate.queryForObject("select id,name,address from gxa_employee where id=?", beanPropertyRowMapper, id);
    }

    @Override
    public List<Employee> findEmployeeAll() {
        //帮我们把ResultSet的结果取出映射到指定的pojo中 前提是数据库的列名和pojo属性一致
        BeanPropertyRowMapper<Employee> beanPropertyRowMapper=new BeanPropertyRowMapper<>(Employee.class);
        return jdbcTemplate.query("select id,name,address from gxa_employee", beanPropertyRowMapper);
    }

    @Override
    public void insertEmployee(Employee employee) {
        jdbcTemplate.update("insert into gxa_employee (name,address) values(?,?)",employee.getName(),employee.getAddress());
    }

    @Override
    public void updateEmployee(Employee employee) {
        jdbcTemplate.update("update gxa_employee set name=?,address=? where id=?",employee.getName(),employee.getAddress(),employee.getId());
    }

    @Override
    public void deleteEmployee(Long id) {
        jdbcTemplate.update("delete from gxa_employee where id=?",id);
    }
```

```
38  }
39
```

配置扫描dao



applicationContext.xml Java

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:context="http://www.springframework.org/schema/context"
5         xsi:schemaLocation="
6  http://www.springframework.org/schema/beans http://www.springframework.or
   g/schema/beans/spring-beans.xsd
7  http://www.springframework.org/schema/context http://www.springframework.o
   rg/schema/context/spring-context.xsd">
8
9
10         <!--配置扫描包-->
11          <context:component-scan base-package="com.gxa.dao"/>
12
13
14         <import resource="applicationContext-dao.xml"/>
15
16
17 </beans>
```

6.5 测试

```java

/**
 * Created by zxd on 2022/10/26 10:41
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:applicationContext.xml")
public class TestEmployeeDao {

    @Autowired
    private EmployeeDao employeeDao;

    @Test
    public void testInsert(){

        Employee employee=new Employee(null,"坤坤","火星");
        employeeDao.insertEmployee(employee);
    }


    @Test
    public void testUpdate(){

        Employee employee=new Employee(2L,"坤坤","月球");
        employeeDao.updateEmployee(employee);
    }

    @Test
    public void testFind(){

        System.out.println(employeeDao.findEmployeeById(2L));

        System.out.println(employeeDao.findEmployeeAll());

    }


    @Test
    public void testDelete(){

        employeeDao.deleteEmployee(2L);

    }

}
```

7.Spring事务管理

7.1 编程式事务

7.1.1 编程式事务概念

事务功能的相关操作全部通过自己编写代码来实现：

```java
Connection conn = ...;

try {

    // 开启事务：关闭事务的自动提交
    conn.setAutoCommit(false);

    // 核心操作

    //1.插入

    //创建回滚点
    Savepoint savepoint;

    int x=1/0;

    //2.插入


    // 提交事务
    conn.commit();

}catch(Exception e){

    // 回滚事务
    conn.rollBack(savepoint);

}finally{

    // 释放数据库连接
    conn.close();

}
```

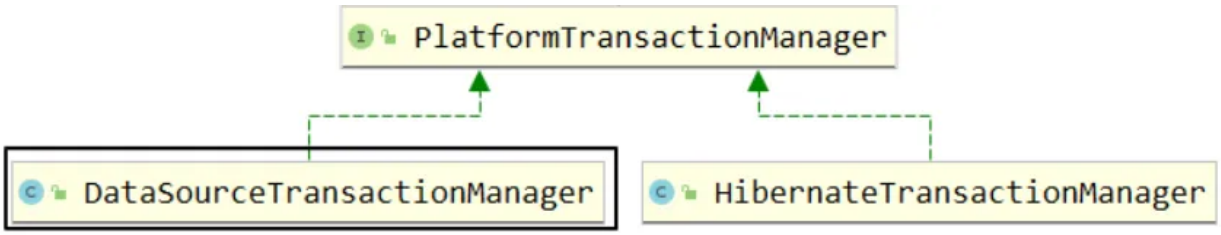
编程式的实现方式存在缺陷：

- 细节没有被屏蔽：具体操作过程中，所有细节都需要程序员自己来完成，比较繁琐。
- 代码复用性不高：如果没有有效抽取出来，每次实现功能都需要自己编写代码，代码就没有得到复用。

7.1.2 事务管理器

事务管理器是spring中专门管理事务的对象

```java

public interface PlatformTransactionManager {

    //获取事务状态
    TransactionStatus getTransaction(TransactionDefinition transactionDefinition) throws TransactionException;

    //提交事务
    void commit(TransactionStatus transactionStatus) throws TransactionException;

    //回滚事务
    void rollback(TransactionStatus transactionStatus) throws TransactionException;
}

```



PlatformTransactionManager 是接口类型，不同的 Dao 层技术则有不同的实现类，

例如：Dao 层技术

是Jdbc 或 mybatis 时：org.springframework.jdbc.datasource.DataSourceTransactionManager

Dao 层技术是hibernate时：org.springframework.orm.hibernate5.HibernateTransactionManager

Dao 层技术是springDataJpa时：JpaTransactionManager

7.1.3 TransactionDefinition

```java
public interface PlatformTransactionManager {

    //在获取事务的时候传入一个TransactionDefinition对象
    TransactionStatus getTransaction(TransactionDefinition transactionDefi
nition) throws TransactionException;

    void commit(TransactionStatus var1) throws TransactionException;

    void rollback(TransactionStatus var1) throws TransactionException;
}

```

TransactionDefinition 是事务的定义信息对象，里面有如下方法：

Java

```java
public interface TransactionDefinition {

    //事务的传播行为
    int PROPAGATION_REQUIRED = 0;
    int PROPAGATION_SUPPORTS = 1;
    int PROPAGATION_MANDATORY = 2;
    int PROPAGATION_REQUIRES_NEW = 3;
    int PROPAGATION_NOT_SUPPORTED = 4;
    int PROPAGATION_NEVER = 5;
    int PROPAGATION_NESTED = 6;

    //隔离级别
    int ISOLATION_DEFAULT = -1;
    int ISOLATION_READ_UNCOMMITTED = 1;
    int ISOLATION_READ_COMMITTED = 2;
    int ISOLATION_REPEATABLE_READ = 4;
    int ISOLATION_SERIALIZABLE = 8;

    int TIMEOUT_DEFAULT = -1;

    //获取传播行为
    int getPropagationBehavior();

    //获取隔离级别
    int getIsolationLevel();

    //获取超时时间
    int getTimeout();

    //是否只读
    boolean isReadOnly();

    String getName();
}

```

方法	说明
int getIsolationLevel()	获得事务的隔离级别
int getPropogationBehavior()	获得事务的传播行为
int getTimeout()	获得超时时间
boolean isReadOnly()	是否只读

7.1.3.1 事务隔离级别

设置隔离级别，可以解决事务并发产生的问题，如脏读、不可重复读和虚读。

ISOLATION_DEFAULT

ISOLATION_READ_UNCOMMITTED

ISOLATION_READ_COMMITTED

ISOLATION_REPEATABLE_READ

ISOLATION_SERIALIZABLE

7.1.3.2 传播行为

事务传播行为 决定事物在业务方法之间如何进行传递

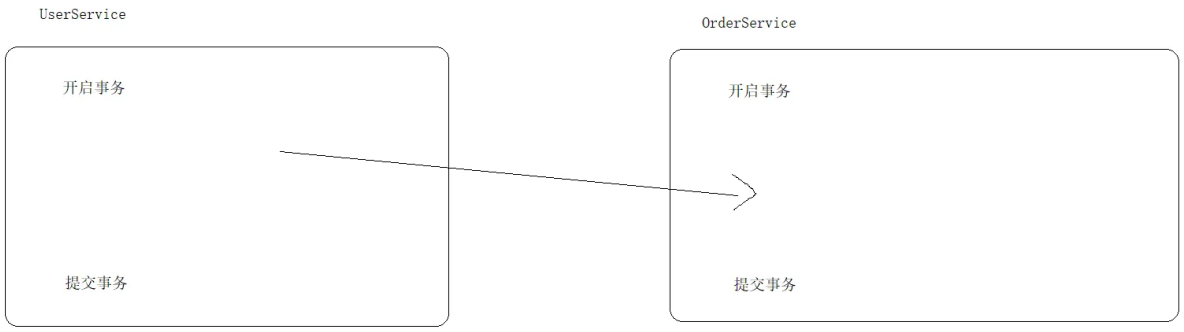
REQUIRED：如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。一般的选择（默认值） 最常见

REQUERS_NEW：新建事务，如果当前在事务中，把当前事务挂起。

传播行为:决定事务在方法中如何进行传递

REQUIRED：如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。一般的选择（默认值） 最常见

REQUERS_NEW：新建事务，如果当前在事务中，把当前事务挂起。



SUPPORTS：支持当前事务，如果当前没有事务，就以非事务方式执行（没有事务）

MANDATORY：使用当前的事务，如果当前没有事务，就抛出异常

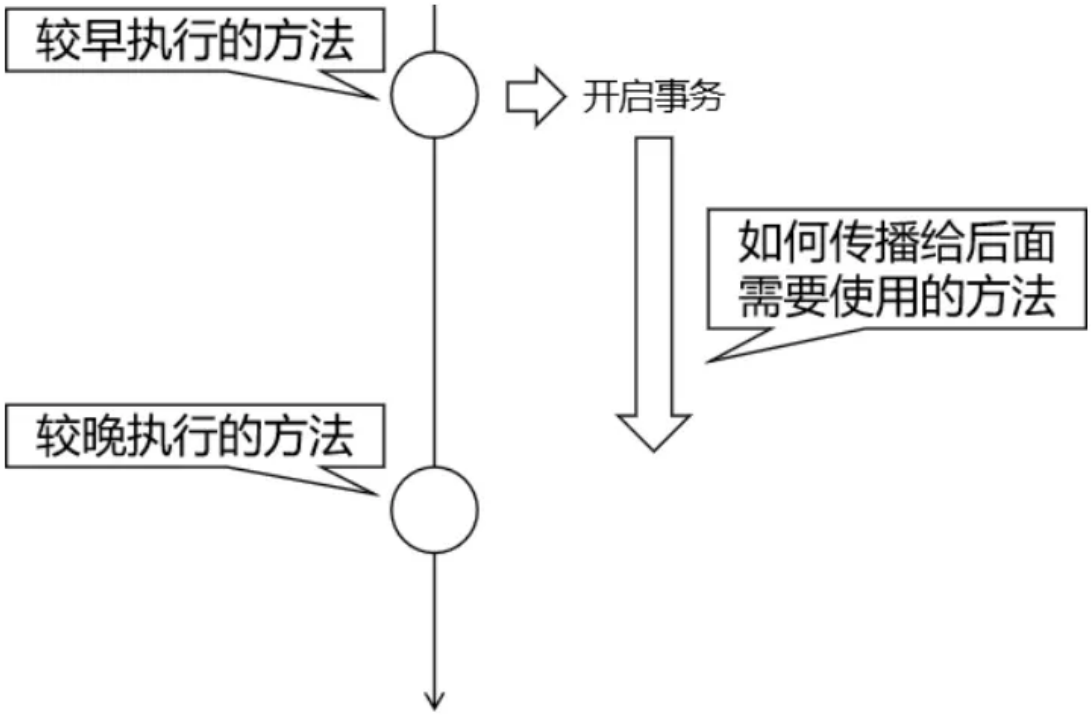
NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起

NEVER：以非事务方式运行，如果当前存在事务，抛出异常

NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行 REQUIRED 类似

的操作

事务的传播行为



propagation 属性的可选值由 org.springframework.transaction.annotation.Propagation 枚举类提供：

名称	含义
REQUIRED 默认值	当前方法必须工作在事务中 如果当前线程上有已经开启的事务可用，那么就在这个事务中运行 如果当千纤草上没有已经开启的事务，那么就自己开启新事务，在新事务中运行 所以当前方法有可能和其他方法共用事务 在共用事务的情况下：当前方法会因为其他方法回滚而受连累
REQUIRES_NEW	当前方法必须工作在事务中 不管当前线程上是否有已经开启的事务，都要开启新事务 在新事务中运行 不会和其他方法共用事务，避免被其他方法连累

7.1.3.3 只读事务

对一个查询操作来说，如果我们把它设置成只读，就能够明确告诉数据库，这个操作不涉及写操作。这样数据库就能够针对查询操作来进行优化

7.1.3.4 超时

事务在执行过程中，有可能因为遇到某些问题，导致程序卡住，从而长时间占用数据库资源。而长时间占用资源，大概率是因为程序运行出现了问题（可能是Java程序或MySQL数据库或网络连接等等）。

此时这个很可能出问题的程序应该被回滚，撤销它已做的操作，事务结束，把资源让出来，让其他正常程序可以执行。

概括来说就是一句话：超时回滚，释放资源。

7.1.4 TransactionStatus

在获取事务的时候返回事务的状态信息，在提交或者回滚事务的时候传入事务的状态信息

```java
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition var1) throws Tra
nsactionException;

    void commit(TransactionStatus var1) throws TransactionException;

    void rollback(TransactionStatus var1) throws TransactionException;
}

```

事务的状态信息

```java
public interface TransactionStatus extends SavepointManager, Flushable {

    //是否是新事务
    boolean isNewTransaction();

    //是否存储回滚点
    boolean hasSavepoint();

    //设置事务的回滚
    void setRollbackOnly();

    //事务是否回滚
    boolean isRollbackOnly();

    //刷新事务
    void flush();

    //是否是否完成
    boolean isCompleted();
}
```

7.1.5 配置事务管理器

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


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


</beans>
```

7.1.6 编写操作事务的工具类

```java
/**
 * Created by zxd on 2022/10/26 11:54
 */
public class TransactionUtils {

    @Autowired
    private PlatformTransactionManager transactionManager;

    //开启事务
    public TransactionStatus startTransaction(){
        TransactionStatus transactionStatus = transactionManager.getTransaction(new DefaultTransactionDefinition());
        return transactionStatus;
    }


    //提交事务
    public void commitTransaction(TransactionStatus transactionStatus){
        if(!transactionStatus.isCompleted()){
            transactionManager.commit(transactionStatus);
        }
    }


    //回滚事务
    public void rollbackTransaction(TransactionStatus transactionStatus){
        if(!transactionStatus.isRollbackOnly()){
            transactionManager.rollback(transactionStatus);
        }
    }

}

```

配置

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


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


    <!--配置操作事务的工具类-->
    <bean id="transactionUtils" class="com.gxa.utils.TransactionUtils"/>


</beans>
```

7.1.7 编写service并测试

编写service接口

```java
/**
 * Created by zxd on 2022/10/25 16:25
 */
public interface EmployeeService {


    Employee findEmployeeById(Long id);

    List<Employee> findEmployeeAll();

    void insertEmployee(Employee employee);

    void updateEmployee(Employee employee);

    void deleteEmployee(Long id);


}

```

编写service的实现

```java
/**
 * Created by zxd on 2022/10/25 16:28
 */
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeDao employeeDao;

    @Override
    public Employee findEmployeeById(Long id) {
        return employeeDao.findEmployeeById(id);
    }

    @Override
    public List<Employee> findEmployeeAll() {
        return employeeDao.findEmployeeAll();
    }

    @Override
    public void insertEmployee(Employee employee) {
        employeeDao.insertEmployee(employee);
        int x=1/0;
        employee.setName("班长");
        employeeDao.insertEmployee(employee);
    }

    @Override
    public void updateEmployee(Employee employee) {
        employeeDao.updateEmployee(employee);
    }

    @Override
    public void deleteEmployee(Long id) {
        employeeDao.deleteEmployee(id);
    }
}

```

测试没有事务控制的情况

```java
/**
 * Created by zxd on 2022/10/26 10:41
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:applicationContext.xml")
public class TestEmployeeService {

    @Autowired
    private EmployeeService employeeService;

    @Test
    public void testInsert(){

        Employee employee=new Employee(null,"坤坤","火星");
        employeeService.insertEmployee(employee);
    }



}

```

7.1.8 编码式事务(手动控制事务,细粒度事务)

```java
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeDao employeeDao;

    @Autowired
    private TransactionUtils transactionUtils;


    @Override
    public void insertEmployee(Employee employee) {
        TransactionStatus transactionStatus =null;
        try {
            //开启事务
            transactionStatus = transactionUtils.startTransaction();

            employeeDao.insertEmployee(employee);
            int x = 1 / 0;
            employee.setName("班长");
            employeeDao.insertEmployee(employee);

            //提交事务
            transactionUtils.commitTransaction(transactionStatus);

        }catch (Exception e){
            e.printStackTrace();
            transactionUtils.rollbackTransaction(transactionStatus);
        }
    }

}
```

7.1.9 使用spring官方提供的事务管理工具类

配置

Java

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


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


    <!--配置spring官方的事务管理的工具类-->
     <bean class="org.springframework.transaction.support.TransactionTemplate">
         <property name="transactionManager" ref="transactionManager"/>
     </bean>

</beans>
```

Java

```java
package com.gxa.service.impl;

import com.gxa.dao.EmployeeDao;
import com.gxa.pojo.Employee;
import com.gxa.service.EmployeeService;
import com.gxa.utils.TransactionUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionCallbackWithoutResult;
import org.springframework.transaction.support.TransactionTemplate;

import java.util.List;

/**
 * Created by zxd on 2022/10/25 16:28
 */
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeDao employeeDao;

    @Autowired
    private TransactionTemplate transactionTemplate;


    @Override
    public void insertEmployee(Employee employee) {

        //不带返回结果执行事务
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus transactionStatus) {
                employeeDao.insertEmployee(employee);
                int x = 1 / 0;
                employee.setName("班长");
                employeeDao.insertEmployee(employee);
            }
        });
    }
```

```
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
```

```java
    @Override
    public Employee findEmployeeById(Long id) {

      //带返回结果执行事务
      return  transactionTemplate.execute(new TransactionCallback<Employee>() {
            @Override
            public Employee doInTransaction(TransactionStatus transactionStatus) {
                return  employeeDao.findEmployeeById(id);
            }
        });
    }

    @Override
    public List<Employee> findEmployeeAll() {
        return employeeDao.findEmployeeAll();
    }


    @Override
    public void updateEmployee(Employee employee) {
        employeeDao.updateEmployee(employee);
    }

    @Override
    public void deleteEmployee(Long id) {
        employeeDao.deleteEmployee(id);
    }
}
```

源码分析:

TransactionCallback接口,doInTransaction用来编写业务代码

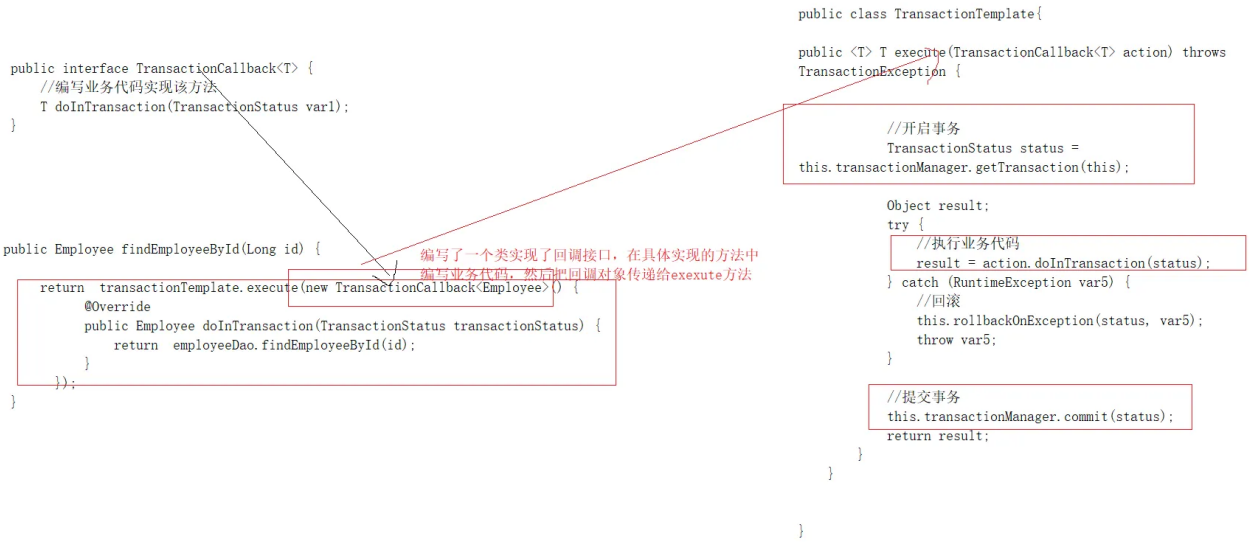
TransactionCallback Java

```java
public interface TransactionCallback<T> {
    //编写业务代码实现该方法
    T doInTransaction(TransactionStatus var1);
}

```

Java

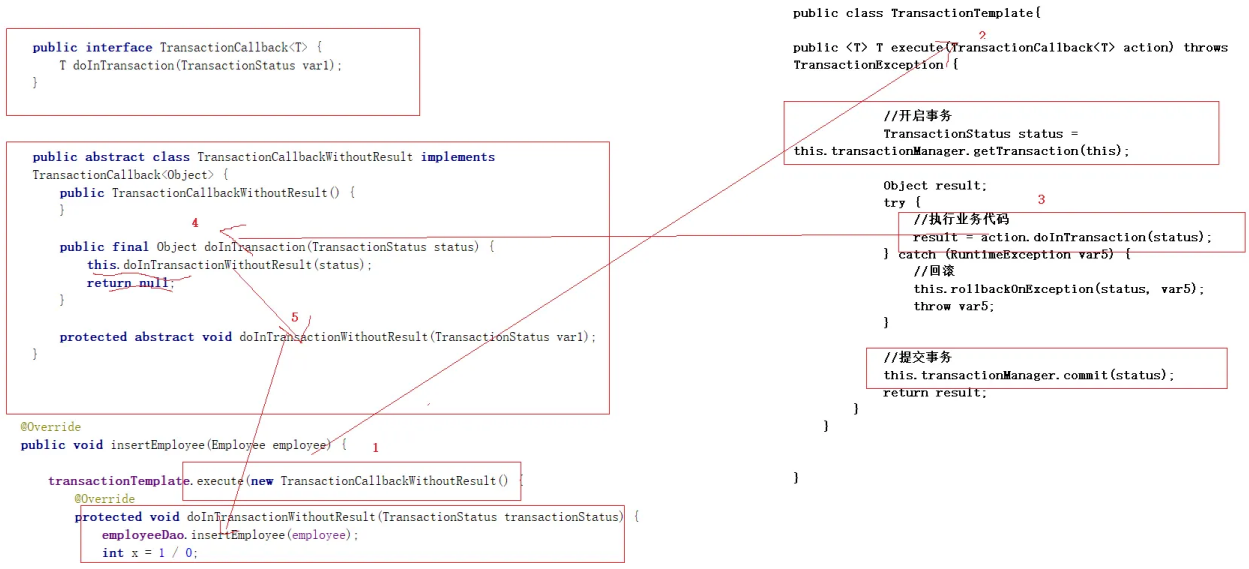
```java
public class TransactionTemplate{

public <T> T execute(TransactionCallback<T> action) throws TransactionExce
ption {


            //开启事务
            TransactionStatus status = this.transactionManager.getTransact
ion(this);

            Object result;
            try {
                //执行业务代码
                result = action.doInTransaction(status);
            } catch (RuntimeException var5) {
                //回滚
                this.rollbackOnException(status, var5);
                throw var5;
            }

            //提交事务
            this.transactionManager.commit(status);
            return result;
        }
    }


}
```

```java
 public Employee findEmployeeById(Long id) {

       return  transactionTemplate.execute(new TransactionCallback<Employee>() {
             @Override
             public Employee doInTransaction(TransactionStatus transactionStatus) {
                 return  employeeDao.findEmployeeById(id);
             }
         });
     }
```



不带返回的思想

```java
    @Override
    public void insertEmployee(Employee employee) {

        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus transactionStatus) {
                employeeDao.insertEmployee(employee);
                int x = 1 / 0;
                employee.setName("班长");
                employeeDao.insertEmployee(employee);
            }
        });
    }
```



7.2 声明式事务

既然事务控制的代码有规律可循，代码的结构基本是确定的，所以框架就可以将固定模式的代码抽取出来，进行相关的封装。

封装起来后，我们只需要在配置文件中进行简单的配置即可完成操作。

- 好处1：提高开发效率
- 好处2：消除了冗余的代码
- 好处3：框架会综合考虑相关领域中在实际开发环境下有可能遇到的各种问题，进行了健壮性、性能等各个方面的优化

所以，我们可以总结下面两个概念：

- **编程式**：**自己写代码**实现功能
- **声明式**：通过**配置**让**框架**实现功能

7.2.1 使用aop封装管理事务的代码

编写通知

```java
/**
 * Created by zxd on 2022/10/26 14:56
 */
@Component
@Aspect
public class TxAdvice {

    @Autowired
    private PlatformTransactionManager transactionManager;

    //事务控制?  用什么通知  环绕通知
    @Around("execution(* com.gxa.service..*ServiceImpl.*(..))")
    public Object aroundTransaction(ProceedingJoinPoint joinPoint){

        TransactionStatus transactionStatus=null;
        try{

            //1.开启事务
            transactionStatus = transactionManager.getTransaction(new DefaultTransactionDefinition());

            System.out.println("start transaction...");

            //2.执行目标方法
            Object result = joinPoint.proceed();

            System.out.println("commit transaction....");

            //3.提交事务
            transactionManager.commit(transactionStatus);

            return result;
        }catch (Throwable e){
            e.printStackTrace();
            System.out.println("rollback transaction....");
            transactionManager.rollback(transactionStatus);
            throw new RuntimeException("失败...");
        }
    }


}

```

配置开启aop的注解配置

applicationContext-tx.xml Java

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


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

    <!--开启aop注解配置-->
    <aop:aspectj-autoproxy/>



</beans>
```

主配置文件扫描通知所在的包

applicationContext.xml Java

```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: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">


        <!--配置扫描包-->
         <context:component-scan base-package="com.gxa.service,com.gxa.dao,com.gxa.advice"/>


        <import resource="applicationContext-dao.xml"/>
        <import resource="applicationContext-tx.xml"/>


</beans>
```

7.2.2 spring封装的aop事务(xml版本)

Java

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


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


    <!--2.配置通知-->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <!--定义spring操作事务方法的属性-->
        <tx:attributes>
            <tx:method name="insert*" isolation="REPEATABLE_READ" propagation="REQUIRED" read-only="false"/>
            <tx:method name="add*" isolation="REPEATABLE_READ" propagation="REQUIRED" read-only="false"/>
            <tx:method name="save*" isolation="REPEATABLE_READ" propagation="REQUIRED" read-only="false"/>
            <tx:method name="update*" isolation="REPEATABLE_READ" propagation="REQUIRED" read-only="false"/>
            <tx:method name="modify*" isolation="REPEATABLE_READ" propagation="REQUIRED" read-only="false"/>
            <tx:method name="delete*" isolation="REPEATABLE_READ" propagation="REQUIRED" read-only="false"/>
            <tx:method name="find*" isolation="REPEATABLE_READ" propagation="REQUIRED" read-only="true"/>
            <tx:method name="query*" isolation="REPEATABLE_READ" propagation="REQUIRED" read-only="true"/>
        </tx:attributes>

    </tx:advice>
```

```
    <!--3.配置织入-->
    <aop:config>

        <!--配置切入点表达式-->
        <aop:pointcut id="pc" expression="execution(* com.gxa.service..*ServiceImpl.*(..))"/>

        <!--切面(切入点表达式+通知)-->
        <aop:advisor advice-ref="txAdvice" pointcut-ref="pc"/>

    </aop:config>



</beans>
```

1.查看是否生成代理对象，如果没有生成代理对象切入点表达式，类名，包名找问题

2.如果有代理对象，事务没有生效，查看方法名是否编写符合规则

Java

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:applicationContext.xml")
public class TestEmployeeService {

    @Autowired
    private EmployeeService employeeService;

    @Test
    public void testInsert(){

        Employee employee=new Employee(null,"坤坤","火星");
        employeeService.insertEmployee(employee);
    }

}
```

7.2.3 spring封装的aop事务(注解版本)

applicationContext-tx.xml Java

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


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

    <!--开启注解事务-->
    <tx:annotation-driven/>


</beans>
```

Java

```java
/**
 * Created by zxd on 2022/10/25 16:28
 */
@Service
@Transactional(propagation = Propagation.REQUIRED,isolation = Isolation.REPEATABLE_READ,readOnly = false)
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeDao employeeDao;



    @Override
    public void insertEmployee(Employee employee) {
                employeeDao.insertEmployee(employee);
                employee.setName("班长");
                employeeDao.insertEmployee(employee);
    }

    @Transactional(propagation = Propagation.REQUIRED,isolation = Isolation.REPEATABLE_READ,readOnly = true)
    @Override
    public Employee findEmployeeById(Long id) {
     return  employeeDao.findEmployeeById(id);
    }

    @Override
    public List<Employee> findEmployeeAll() {
        return employeeDao.findEmployeeAll();
    }



    @Override
    public void updateEmployee(Employee employee) {
        employeeDao.updateEmployee(employee);
    }

    @Override
    public void deleteEmployee(Long id) {
        employeeDao.deleteEmployee(id);
    }
}

```